



---

# Newton Programmer's Guide: 2.1 OS Addendum



## **IMPORTANT**

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

4/22/97 Newton Technical Publications Team  
© Apple Computer, Inc. 1997

 Apple Computer, Inc.

© 1997 Apple Computer, Inc.  
All rights reserved.

No part of this publication or the software described in it may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., except in the normal use of the software or to make a backup copy of the software. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to another person. Under the law, copying includes translating into another language or format. You may use the software on any computer owned by you, but extra copies cannot be made for this purpose.

Printed in the United States of America.

The Apple logo is a registered trademark of Apple Computer, Inc. Use of the “keyboard” Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop

applications only for licensed Newton platforms.

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

Apple, the Apple logo, AppleTalk, eMate, Espy, LaserWriter, the light bulb logo, Macintosh, MessagePad, Newton, Newton Connection Kit, and New York are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Geneva, NewtonScript, Newton Toolkit, and QuickDraw are trademarks of Apple Computer, Inc. Acrobat, Adobe Illustrator, and PostScript are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

CompuServe is a registered service mark of CompuServe, Inc.

FrameMaker is a registered trademark of Frame Technology Corporation.

Helvetica and Palatino are registered trademarks of Linotype Company.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

Microsoft is a registered trademark of Microsoft Corporation.

Windows is a trademark of Microsoft Corporation.

QuickView™ is licensed from Altura Software, Inc.

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. 5/97

# Contents

Figures, Tables, and Listings      xv

## Preface

## About This Book      xix

---

Change History      xix  
Related Books      xix  
Sample Code      xx  
Conventions Used in This Book      xxi  
    Special Fonts      xxi  
Developer Products and Support      xxii

## Chapter 1

## Newton Works      1-1

---

About Newton Works      1-2  
    User Interface      1-2  
    Programming Interface Overview      1-4  
Using the Newton Works Interface      1-5  
    Registering Stationery      1-5  
    Creating the DataDef      1-6  
        Supporting Application-Defined Preferences      1-6  
        Adding Information to the Title Slip      1-7  
        Supporting Newton Find Operations      1-9  
    Creating the ViewDef      1-10  
        Supporting Document Find      1-11  
        Supporting Data Storage      1-13  
        Supporting Scrolling      1-14  
        Providing Status Bar Buttons      1-15  
        Providing Help      1-16  
        Notification of Changes      1-17  
Working With the Tools Picker      1-18

Newton Works Interface Reference	1-19
Newton Works Base Application Slots and Methods	1-19
Newton Works DataDef Slots and Methods	1-23
Newton Works Viewdef Slots and Methods	1-25
Summary of Newton Works	1-34
Newton Works Base Application	1-34
Newton Works Stationery DataDef	1-34
Newton Works Stationery ViewDef	1-34

---

Chapter 2	<b>Newton Works Draw Application</b>	2-1
-----------	--------------------------------------	-----

---

About the Draw Application	2-1
User Interface	2-2
Programmer's Overview	2-3
Using the Drawing Application Interface	2-3
Adding Custom Drawing Tools	2-3
Adding Patterns and Gray Tones to the Fill Tool	2-6
Adding Stamps to the Stamp Tool	2-7
Draw Application Methods	2-8
The Canvas and Its Methods	2-9
Draw Application Reference	2-10
Proto	2-10
protoDrawTool	2-10
Data Structures	2-17
The Canvas	2-17
Functions and Methods	2-20
Draw Application viewDef Methods	2-20
Summary	2-24
Proto	2-24
Data Structures	2-25

About protoTXView And the View System	3-1
Application-defined Methods	3-2
View Slots	3-3
Other View Features	3-3
About Paged and Non-paged Word-Processing Views	3-4
About Scrolling with protoTXView	3-6
About Storing protoTXView Documents	3-6
Using protoTXViewFinder to Search Documents	3-7
Word-Processing View User Interface	3-7
Terminology	3-8
Using Word Processing Views	3-8
Initializing Your Word-Processing View	3-9
Setting Up Your Word-Processing View	3-10
Scrolling the Word-processing View	3-11
Reading a Word-Processing Document From a Soup	3-14
Storing Documents In a Soup	3-15
Handling User Interactions	3-17
Changing the Font	3-17
Changing the Font Size	3-18
Replacing the Selected Text With a Graphic	3-19
Converting the Selected Text to Uppercase	3-20
Adding a Recognized Word to Your Word-Processing View	3-20
Word Processing View Reference	3-22
Common Parameter Descriptions	3-22
The Range Frame	3-22
The Graphics Specification Frame	3-22
The Ruler Information Frame	3-23
Tab Frames	3-23
Protos	3-24
protoTXView	3-24
protoTXViewFinder	3-45
Summary of Word Processing Views	3-47
Data Structures	3-47

protoTXView	3-47
protoTXViewFinder	3-50

## Chapter 4

## Keyboard Enhancements 4-1

About Keyboard Enhancements	4-1
Terminology	4-2
About Keystroke Handling	4-2
Keystroke Event Sequencing	4-3
Typing Without a Caret	4-4
About Command Key Handling	4-5
How Command Keys Are Found	4-5
About Displaying Command-Key Combinations in Menus	4-6
About Keyboard Support in Pickers	4-8
Calling a Key-Command Method From a Picker Script	4-8
Keyboard Enhancements User Interface	4-9
General Usage	4-9
Text entry and editing	4-10
Slips, windows, and buttons:	4-10
Menus	4-12
System and Built-in App Command Key Assignments	4-15
Compatibility	4-19
Default Buttons	4-19
Possible Key-view Compatibility Problem	4-20
Using the Keyboard Enhancements	4-20
Keystroke Handling	4-20
Intercepting Keystrokes Directly	4-21
Intercepting Individual Keystrokes	4-22
Intercepting Grouped Keystrokes	4-22
Text Flags and Keyboard Input	4-22
Handling Command Keys	4-24
Searching for Key Commands	4-24

Defining Key Commands	4-25
Displaying the Popup Command Key Help Slip	4-27
The Caret Stack and Caret Activation	4-27
Using Keys in Slips	4-28
Designating the Default Button In a Slip	4-29
Designating a Slip's Close Box	4-29
Default and Close Buttons in Confirm Slips	4-30
Keyboard Reference	4-31
Data Structures	4-31
The Command-Key Mapping Frame	4-31
Methods and Functions for Handling Keystrokes	4-34
Methods and Functions for Handling Command Keys	4-36
Application-Defined Methods for Keystroke Events	4-40
Summary of Keyboard Enhancements	4-44
Data Structures	4-44
Methods and Functions	4-44

## Chapter 5      **Spell Checker**      5-1

---

About the Spell Checker	5-1
Limitations	5-2
Using the Spell Checker	5-2
Processing of Words Passed to the Spell Checker	5-3
Use of Dictionaries by the Spell Checker	5-3
Spell Checker Reference	5-4
Functions	5-4
Summary of Spell Checker	5-10
Functions	5-10

## Chapter 6      **Drawing and Graphics 2.1**      6-1

---

About Drawing and Graphics in the Newton 2.1 OS	6-2
About Gray Tones and Patterns	6-2

About Gray Pictures	6-2
About Gray Bitmaps (Pix Families)	6-3
About Gray Extras Drawer Icons	6-5
About Ink Shapes	6-5
About Text Box Shapes	6-6
About Gray Text	6-6
About Selection Handles	6-6
About Anti-Aliasing	6-7
Compatibility	6-8
Using Drawing and Graphics in the Newton 2.1 OS	6-9
Specifying Shades of Gray	6-9
Specifying RGB Triplets	6-10
Using Patterns, Gray Patterns, and Dithered Patterns	6-11
Black and White Patterns	6-11
Gray Patterns	6-12
Dithered Patterns	6-12
Creating Gray Text	6-13
Importing Color PICTs from the Mac OS Version of NTK	6-14
Creating Graphic Shapes from Picture Objects	6-14
Using Pix Families	6-14
Creating Gray Extras Drawer Icons	6-15
Anti-Aliasing Monochrome Bitmaps	6-16
Gray Transfer Modes	6-17
How the System Scales Bitmaps	6-18
Using Selection Handles	6-19
Creating Ink and TextBox Shapes	6-19
New Graphic Shape Utility Functions	6-20
The FindShape Function	6-21
The GetPointsArrayXY Function	6-21
The MungeShape Function	6-22
The GetMaskedPixel Function	6-22
Changes to Existing Graphic Shape Functions	6-23
MakeBitmap Accepts a Depth Option	6-23
MakeShape Makes Bitmap Shapes With Masks	6-23



GetStrokePointsArray Filters More Points and Swaps	
Coordinates	6-23
Drawing and Graphics Reference	6-23
Constants	6-23
Gray Tone Constants	6-24
Transfer Mode Constants	6-24
Data Structures	6-25
Style Frame	6-25
Patterns	6-28
Functions and Methods	6-29
Summary	6-43
Constant	6-43
Data Structures	6-44
Functions and Methods	6-44

## Chapter 7

## Sound 7-1

---

About Sound	7-1
Terminology	7-2
Compatibility	7-2
Hardware Volume Support	7-2
User Interface	7-3
Sound Input	7-4
Sound Compression	7-4
Synthesized Sound	7-5
Devices and Channels	7-6
Sampling Rates	7-7
New NTK Sound Import Function	7-7
Using Sound	7-7
Using the protoRecorderView	7-8
Using the Built-in Sound Recorder Slip	7-9
Using the NewtonScript API to Record Sound	7-12
Setting the Input Gain	7-14
Compressing Sound	7-15

Using Codecs to Compress and Decompress Sound	7-15
Synthesizing Sound	7-17
Using Global Sound Preferences	7-23
Getting and Setting Input Gain Preference	7-23
Getting and Setting Default Input or Output Devices	7-24
PlaySound Errata	7-24
Using the Sound Registry	7-25
Sound Reference	7-26
Constants	7-26
Device Constants	7-26
Codec Constants	7-27
Compression Constants	7-27
Data Type Constants	7-28
Data Structures	7-28
Sound Frame	7-28
Sound Result Frame	7-31
User Configuration Variables	7-32
Synthesized Sound Data Format	7-32
soundRecorder Object	7-33
Protos	7-34
protoRecorderView	7-34
protoSoundChannel	7-36
protoSoundFrame	7-44
Functions	7-47
Sound Error Codes	7-52
Summary of Sound	7-53
Constants	7-53
Data Structures	7-53
Protos	7-55
Functions	7-56

## Chapter 8

## Dial-In Networks 8-1

---

Dial-in Networks Reference	8-2
----------------------------	-----

Data Structures	8-2
Access Frame	8-2
Network Frame	8-2
Functions	8-3
Dial-in Networks Summary	8-7
Data Structures	8-7
Functions	8-7

---

<b>Chapter 9</b>	<b>IrDA Communication Tool</b>	<b>9-1</b>
------------------	--------------------------------	------------

---

About the IrDA Communication Tool	9-1
Overview	9-1
Terminology	9-2
Using the IrDA Tool	9-2
Making a Connection	9-3
Getting IrDA Tool Information	9-6
Slow IR Connect Option	9-8
IrDA Tool Option Reference	9-9
Discovery Option	9-10
Connection Information Option	9-13
Receive Buffers Option	9-16
Link Disconnect Option	9-17
Connect User Data Option	9-18
Attribute Name Option	9-19
IrDA Tool Error Codes	9-20
Summary of IrDA Tool	9-21
IrDA Tool Service Option Label	9-21
IrDA Tool Options	9-21
Constants	9-21

---

<b>Chapter 10</b>	<b>eMate Multi-User Mode</b>	<b>10-1</b>
-------------------	------------------------------	-------------

---

Using Multi-user Mode	10-2
-----------------------	------

Reference	10-3
User Configuration Variables	10-3
Functions and Methods	10-4
Summary of Multi-User Mode	10-7
User Configuration Variables	10-7
Functions and Methods	10-7

## Chapter 11

## Miscellaneous 11-1

---

Reference	11-1
Data Structures	11-1
Views	11-1
Built-In Applications	11-4
Protos	11-8
protoPasswordSlip	11-8
protoBlindEntryLine	11-10
Constants	11-11
Views	11-11
Built-In Communication Tools	11-12
Functions and Methods	11-13
Views	11-14
Stationery	11-21
Text Input and Display	11-22
Recognition	11-24
System Services	11-25
Built-In Applications	11-27
Transports	11-33
Utility Functions	11-34
Summary	11-46
Data Structures	11-46
Protos	11-47
Constants	11-48
Functions and Methods	11-48

Editors     A-1

Picture Slot Editor     A-1

Application Icon Editor     A-3

Functions     A-5



# Figures, Tables, and Listings

Chapter 1	Newton Works	1-1
	<b>Figure 1-1</b>	Word processor display 1-3
	<b>Figure 1-2</b>	Info picker 1-7
	<b>Figure 1-3</b>	Title slip 1-8
	<b>Figure 1-4</b>	Find slip 1-12
	<b>Figure 1-5</b>	Status bar buttons 1-16
	<b>Figure 1-6</b>	Tools picker 1-18
	<b>Table 1-1</b>	FindChange parameters and actions 1-28
Chapter 2	Newton Works Draw Application	2-1
	<b>Figure 2-1</b>	The Newton Works Draw application 2-2
	<b>Listing 2-1</b>	Adding a tool to the Draw application's tool bar 2-4
	<b>Listing 2-2</b>	Adding to the Draw application's fill tool 2-6
	<b>Listing 2-3</b>	Adding stamps to the Draw application 2-7
Chapter 3	Word Processing Views	3-1
	<b>Figure 3-1</b>	The displayed ruler 3-8
	<b>Figure 3-2</b>	The TXWord button bar 3-17
	<b>Table 3-1</b>	Use of application-defined methods in protoTXView 3-2
	<b>Table 3-2</b>	Use of standard view system slots in protoTXView 3-3
	<b>Table 3-3</b>	Paged versus non-paged views 3-5
	<b>Table 3-4</b>	Scrolling methods of protoTXView 3-6
	<b>Listing 3-1</b>	Initializing a word-processing view 3-9
	<b>Listing 3-2</b>	Setting up a word-processing view 3-10

<b>Listing 3-3</b>	The SetScrollers method	3-11
<b>Listing 3-4</b>	The TXWord ViewUpdateScrollersScript method	3-11
<b>Listing 3-5</b>	The TXWord GetTextHeight method	3-13
<b>Listing 3-6</b>	The TXWord ViewScroll2DScript method	3-13
<b>Listing 3-7</b>	Reading a document from a soup	3-14
<b>Listing 3-8</b>	Closing the word-processing view	3-15
<b>Listing 3-9</b>	Storing a word-processing document	3-15
<b>Listing 3-10</b>	Changing the font in TXWord	3-17
<b>Listing 3-11</b>	Changing the font size in TXWord	3-18
<b>Listing 3-12</b>	Replacing the selcted text	3-19
<b>Listing 3-13</b>	Converting the selcted text to uppercase	3-20
<b>Listing 3-14</b>	Adding a recognized word to a word-processing view	3-21

## Chapter 4

### Keyboard Enhancements 4-1

---

<b>Figure 4-1</b>	The find slip when it is not the key view	4-10
<b>Figure 4-2</b>	The Find slip when it is the key view	4-11
<b>Figure 4-3</b>	A menu with and without its keyboard equivalents displayed	4-13
<b>Figure 4-4</b>	Command-key combination slip	4-14
<b>Table 4-1</b>	Command definition views	4-6
<b>Table 4-2</b>	System-level key assignments	4-15
<b>Table 4-3</b>	Notepad checklist and outline stationery command keys	4-17
<b>Table 4-4</b>	Names application command keys	4-17
<b>Table 4-5</b>	Dates application command keys	4-18
<b>Table 4-6</b>	In/Out box command keys	4-18
<b>Table 4-7</b>	Call log command keys	4-19
<b>Table 4-8</b>	BookPlayer command keys	4-19
<b>Table 4-9</b>	Summary of keystroke-handling methods and functions	4-20
<b>Table 4-10</b>	Text flags to specify the kind of keystrokes a view accepts	4-23
<b>Table 4-11</b>	Summary of command key methods and functions	4-24
<b>Table 4-12</b>	New default button lists	4-30



<b>Table 4-13</b>	Key codes for special keys	4-33
<b>Table 4-14</b>	Key event-processing script flags	4-41
<b>Listing 4-1</b>	Calling a key-command method from a picker	4-9
<b>Listing 4-2</b>	A key command array	4-25
<b>Listing 4-3</b>	Defining key-commands in the <code>ViewSetupFormScript</code> method	4-26
<b>Listing 4-4</b>	Removing key-commands	4-27
<b>Listing 4-5</b>	An example of a <code>ViewCaretActivateScript</code> method	4-28

## Chapter 6

### Drawing and Graphics 2.1 6-1

---

<b>Figure 6-1</b>	The effect of a mask for a pix family	6-4
<b>Figure 6-2</b>	An oval shape with selection handles	6-6
<b>Figure 6-3</b>	Four black and white pixels	6-7
<b>Figure 6-4</b>	The anti-aliasing effect on a bitmap that has been reduced by 50%	6-8
<b>Figure 6-5</b>	The 4-bit grayscale palette	6-9
<b>Figure 6-6</b>	Two bitmaps combined with the different transfer modes	6-17
<b>Figure 6-7</b>	A <code>textBox</code>	6-20
<b>Figure 6-8</b>	Overlapping ovals	6-22
<b>Table 6-1</b>	Truth table for <code>modeBic</code>	6-25
<b>Listing 6-1</b>	Code to add an <code>icon</code> and <code>iconPro</code> slot to a part frame	6-16
<b>Listing 6-2</b>	Function to retrieve ink shapes from a <code>clEditView</code>	6-20

## Chapter 7

### Sound 7-1

---

<b>Figure 7-1</b>	Sound stationery	7-3
<b>Figure 7-2</b>	Sound recorder slip	7-4
<b>Figure 7-3</b>	<code>protoRecorderView</code>	7-8
<b>Figure 7-4</b>	Tone envelope	7-22

<b>Table 7-1</b>	Sound recorder slots you can set	7-11
<b>Table 7-2</b>	Sound synthesis types	7-20
<b>Table 7-3</b>	Sound device constants	7-26
<b>Table 7-4</b>	Codec constants	7-27
<b>Table 7-5</b>	Compression constants	7-27
<b>Table 7-6</b>	Data type constants	7-28
<b>Table 7-7</b>	protoRecorderView state constants	7-36
<b>Table 7-8</b>	Sound interface error codes	7-52
<b>Listing 7-1</b>	Sound input	7-12

## Chapter 9

### IrDA Communication Tool 9-1

---

<b>Table 9-1</b>	Summary of IrDA tool options	9-9
<b>Table 9-2</b>	IrDA discovery option fields	9-11
<b>Table 9-3</b>	IrDA discovery option probe slots constants	9-12
<b>Table 9-4</b>	IrDA discovery option service hint constants	9-13
<b>Table 9-5</b>	IrDA connection information option fields	9-15
<b>Table 9-6</b>	Disconnect warning event values	9-18
<b>Table 9-7</b>	IrDA tool error codes	9-20

## Chapter 11

### Miscellaneous 11-1

---

<b>Figure 11-1</b>	A view created from protoPasswordSlip	11-8
<b>Figure 11-2</b>	A view based on protoBlindEntryLine	11-11
<b>Figure 11-3</b>	Screen orientation constants	11-12
<b>Table 11-1</b>	Clipboard data types accepted by the system	11-4

## Appendix A

### Newton Toolkit Enhancements A-1

---

<b>Figure A-1</b>	NTK's picture slot editor	A-2
<b>Figure A-2</b>	NTK's Application Icon pane of the Project Setting dialog	A-4

# 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, subject to change, and may not consistently reflect the latest technical information available. ♦

## Change History

---

Since the February 1997 release of this book, the following chapters are new or have been substantially revised and/or reviewed:

Chapter 2, "Newton Works Draw Application,"  
Chapter 5, "Spell Checker,"  
Chapter 7, "Sound,"  
Chapter 9, "IrDA Communication Tool,"  
Chapter 10, "eMate Multi-User Mode,"  
Chapter 11, "Miscellaneous,"  
Appendix A, "Newton Toolkit Enhancements."

## 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 xxii.

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

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>

# Newton Works

---

Newton Works is a new application for the Newton 2.1 OS that itself is based on the NewtApp framework. Newton Works is designed as a simple yet powerful shell for productivity applications, much like similar desktop “Works” products. “Applications” are installed into the Newton Works shell as stationery.

Initially, four applications are available in Newton Works: a word processor (based on the new proto `protoTXView`—see Chapter 3, “Word Processing Views,” for details), a drawing application, a spreadsheet, and a graphing calculator.

This chapter explains how to develop applications for Newton Works. It also describes the additional slots and methods that Newton Works adds to the standard NewtApp framework on which it is built.

To develop stationery for Newton Works you should have a basic understanding of the NewtApp framework and stationery. For general information on the NewtApp framework and standard NewtApp slots and methods, refer to the chapter “NewtApp Applications” in *Newton Programmer’s Guide for Newton 2.0*. For information on developing stationery, refer to the chapter “Stationery” in the same book.

## About Newton Works

---

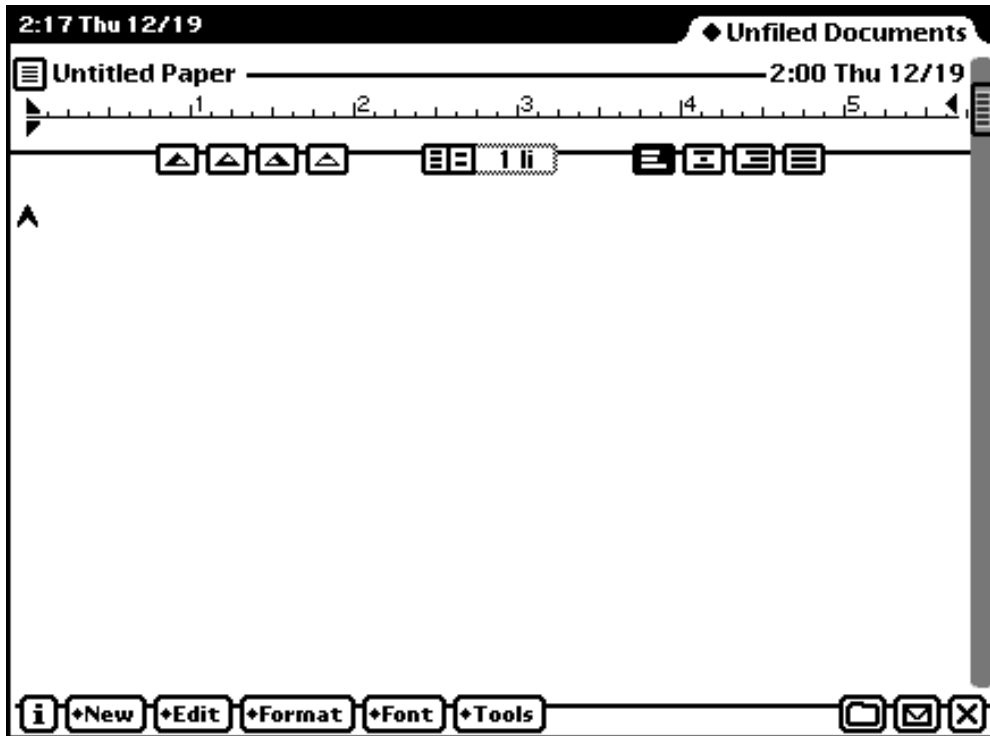
Newton Works was originally developed with the needs of a student in mind. It is an easy-to-use environment where a student can perform different tasks working on different documents. Individual projects are clearly separated as stand-alone documents of different types. However, Newton Works is not limited to use in the education setting. It is suitable as a general-purpose shell for any productivity applications, and is supported on all Newton 2.1 OS devices.

### User Interface

---

Newton Works looks like a typical NewtApp-based application. It has a large document work area under a title bar and a status bar at the bottom, with a New button, in addition to other application-specific buttons. The New button gives the user access to the different applications installed in the Newton Works shell. When tapped, the New button displays a picker listing the different types of documents that the user can create, corresponding to the stationery applications installed in Newton Works. Tapping on a document type switches to that application and creates a new document of that type. Figure 1-1 shows an example of the word processor application.



**Figure 1-1** Word processor display

When used on the eMate 300 device, the Newton Works user interface can look slightly different when the unit is in Classroom Mode. This is a simplified operating mode designed for student use. In this mode, the filing folder interface is hidden, to prevent work from appearing lost. The user sees all their documents together in the Newton Works overview. If the device is being shared among multiple students, each student logs into the unit under their name and sees only their documents in Newton Works. Filing isn't necessary in this mode, since most documents will be uploaded to a classroom server when finished, and only a few working documents will reside on the unit at a particular time.

## Newton Works

Note that this simplification of the Newton Works user interface in Classroom mode follows the standard recommendations for applications that support classroom mode on the eMate 300 device. This behavior is not special to Newton Works.

Like most Newton applications, work is saved as it is entered, so there is no “Save” function necessary for Newton Works users. All saved documents are accessed through the standard Newton Overview button (either on the floating button bar or on a real keyboard). Tapping the Overview button displays a list of existing Newton Works documents, from which the user can select one to work on.

Newton Works extends the NewtApp user interface in one notable way: with the addition of scroll bars to documents. Because documents are typically larger than the screen, scroll bars allow users to navigate their work. The support for scroll bars is provided by the Newton Works shell. For details on how scrolling is supported, see the section “Supporting Scrolling” (page 1-14).

## Programming Interface Overview

---

You can extend Newton Works in two ways: you can add tools to existing stationery, and you can add new stationery. You add new stationery by registering a new `dataDef` and `viewDef` with the system.

Remember that in the NewtApp framework, a single `dataDef` can have multiple associated `viewDefs`. However, in Newton Works, each `dataDef` (representing one type of stationery) can have only a single `viewDef`, named `'default`. Other `viewDefs` are allowed for routing formats, but not for use in displaying Newton Works data on the screen.

As usual for a NewtApp, the user accesses the different stationery types through the New button on the status bar. Each `dataDef` registered for Newton Works corresponds to one item in the New picker. The corresponding `'default` `viewDef` controls the status bar, scroll bars, and the document viewable area.

A good candidate application to add to Newton Works would include an editor that allows the user to view and modify some kind of document. Also, you’d want your application to fit with a general suite of productivity applications, such as those supplied with Newton Works. If your application

## Newton Works

is very specialized, perhaps it would be better to make it a stand-alone application.

You can also add tools to an existing application that supports this. For example, you might add a thesaurus to the word processor. The user accesses tools from the Tools button in the status bar. You register a new tool for an application by sending the `RegNewtWorksTool` message to the Newton Works base application.

## Using the Newton Works Interface

---

This section describes how to perform these tasks:

- register new Newton Works stationery
- support application-defined preferences for your `dataDef`
- add custom information to the title slip for your `dataDef`
- support Newton Find operations in your `dataDef` and `viewDef`
- support document Find operations in your `viewDef`
- support storage of permanent data in your `viewDef`
- support scrolling in your `viewDef`
- provide custom buttons on the status bar in your `viewDef`
- provide help information for the user in your `viewDef`
- handle notification of global preferences changes in your `viewDef`
- register, unregister, and work with tools for an installed Newton Works stationery application

## Registering Stationery

---

You register and unregister your stationery with the system by using the standard functions `RegDataDef`, `UnRegDataDef`, `RegisterViewDef` and `UnRegisterViewDef`. Typically, you would register your stationery in a

## Newton Works

package part `InstallScript` function, and unregister it in the part `RemoveScript` function.

## Creating the DataDef

---

You must use the standard stationery proto, `newtStationery`, to create a `dataDef` for Newton Works. The `superSymbol` slot of the `dataDef` must be set to the symbol `'newtWorks`. This associates the stationery with the Newton Works application.

Note that as in the `NewtApp` framework, the symbol of the `dataDef` is used as the `class` slot of soup entries created by that `dataDef`.

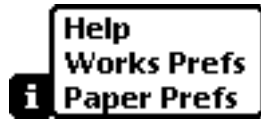
Certain slots are required for all `dataDefs`, such as `symbol`, `name`, `description`, `superSymbol`, `icon`, `StringExtract`, `TextScript`, `FillNewEntry`, etc. For details on these standard `NewtApp` slots and methods, refer to the “*NewtApp Reference*” chapter of *Newton Programmer's Reference*. The only `dataDef` slot unique to Newton Works is the `prefs` slot. See the next section for details on how it is used.

## Supporting Application-Defined Preferences

---

The `dataDef` can specify an application-specific preferences command that appears in the Info button picker when the user is viewing an entry of the `dataDef`. If this command is chosen, a preferences slip is displayed. To implement this feature, define a slot called `prefs` in the `dataDef`. This slot contains a frame defining the command name and icon to show in the Info picker and the frame contains a view template defining the actual preferences slip. Figure 1-2 shows an example of the Info picker with one application-specific preferences command added for the word processor stationery type (paper).

Note that there are also global preferences that apply to all Newton Works applications. These are accessed through the Works Prefs choice on the Info picker. For more information about handling global preferences, see the section “Notification of Changes” (page 1-17).

**Figure 1-2** Info picker

When the user chooses the application-specific preferences command from the Info button, Newton Works sets the slots `target` (the soup entry being viewed), `newtAppBase` (the Newton Works base view), and `viewDefView` (the current `viewDef`) appropriately in the preferences view template. Then it calls `BuildContext(prefsTemplate)` to create and display the preferences view.

The preferences view must read the appropriate preferences slots in its `ViewSetupFormScript` method (or other initialization method), and write the slots in its `ViewQuitScript` method (or before it closes). The preferences reside in the application preferences frame, which you can get using the `newtApplication` method `GetAppPreferences`. The preferences for the `dataDef` must reside in a subframe for that `dataDef` within the preferences frame. For example,

```
prefsFrame := newtAppBase:GetAppPreferences().(kDataSymbol);
```

should return the preferences for the `dataDef` identified by `kDataSymbol`.

To save the preferences, call `EntryChangeXmit(prefsFrame, kDataSymbol)`.

Note that the preferences view must be closed by calling `newtAppBase:RememberedClose(base)`, otherwise, Newton Works keeps a reference to the view and it uses up RAM. For example, here's how you would write the `ButtonClickScript` method for the close box of the preferences view:

```
ButtonClickScript: func() begin
    newtAppBase:RememberedClose(base);
end
```

## Adding Information to the Title Slip

The `dataDef` can specify extra information that appears in the title slip for documents. To do so, you must implement the `InfoBoxExtract` method in the

## Newton Works

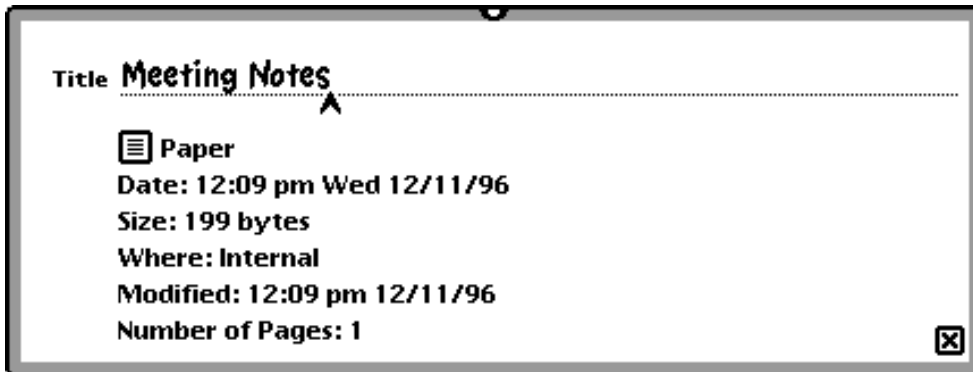
dataDef. This method is called conditionally by Newton Works whenever the title slip is opened.

You must return a shape from the `InfoBoxExtract` method. This shape is shown at the bottom of title slip, after the standard contents. You could return a text summary made into a shape, or a small sketch of a drawing, or anything else useful to show in the title slip. Figure 1-3 shows an example of the title slip for the Newton Works word processor. It has one extra line, showing the number of pages, added to the default slip. Here's an example of code that could be used to add that line:

```
InfoBoxExtract: func(target, maxSize, viewDefView) begin
    local numPages := if viewDefView then viewDefView:?GetCountPages();
    if numPages then
        [MakeText(ParamStr("Number of Pages: ^0, [numPages]),
            maxSize.left, maxSize.top, maxSize.right, maxSize.top+15)];
    end
```

If you don't want to add any extra information, don't implement the `InfoBoxExtract` method, or return `nil` from it.

**Figure 1-3** Title slip



## Newton Works

## Supporting Newton Find Operations

---

A Newton Find operation is a search initiated by the user tapping the main Find icon. Such a search can include all, some, or just one Newton application. In this kind of Find operation, Newton Works is considered a single application, including all of the individual stationery installed in it. Contrast this with a Newton Works document Find operation, which is initiated by the user tapping the Find choice in the Tools picker inside a particular document. This kind of Find operation applies only to the single Newton Works document that is open, and is described in the section “Supporting Document Find” (page 1-11).

To support a Newton Find operation that includes Newton Works, Newton Works first uses the `FindStringInFrame` function to search for a specified string in all soups. If this function does not find a match in a particular soup entry, Newton Works sends your `dataDef` the `FindFn` message to allow you to do your own search of that soup entry. Note that `FindFn` is called only if Newton Works doesn’t find a match by using `FindStringInFrame`.

The `FindFn` method gives you the opportunity to find data that may be stored in non-standard ways, that only your stationery knows how to decode. For example, if you compress entries, you might need to decompress them in order to do a search.

After the Find operation finishes, the system creates a Find overview that displays the found items. It uses the `FindSoupExcerpt` function to get a string for each item in the overview. The system sends the `FindSoupExcerpt` message to the `dataDef` for each item found in the Newton Works soups. You must supply this method in your `dataDef` if you want to specially construct the text that is shown for an item in the overview.

If the soup entries contain text information and you don’t want to do anything special to construct the overview text, you don’t need to implement the `FindSoupExcerpt` method. In this case, the system calls the root view method `FindSoupExcerpt` to obtain the overview text. This method calls the `StringExtract` method of the `dataDef` to obtain the overview text, or uses other means, if that method returns nothing.

In situations where the `FindStringInFrame` function is able to find a match, the root view method `FindSoupExcerpt` will also be able to display some text for the Find overview. If your Newton Works soup entries store string data

## Newton Works

with which these functions work, then you don't need to do anything else to support Newton Find operations.

In situations where the `FindStringInFrame` function does not work to search a soup entry, you'll need to implement both `FindFn` and `FindSoupExcerpt` methods in your `dataDef` if you want your data to be searchable in global Find operations.

Finally, if your soup is such that sometimes the `FindStringInFrame` function will find a match and sometimes the `FindFn` is needed, then you must implement a `FindSoupExcerpt` method that can handle both cases. In the case of special data found by your own `FindFn` method, you will need to return a string for the overview from `FindSoupExcerpt`. In the case of simple string data found by `FindStringInFrame`, your `FindSoupExcerpt` method can call the root method `FindSoupExcerpt` to get the overview string, like this:

```
GetRoot():FindSoupExcerpt(entry, resultFrame)
```

When the user taps an item in the Find overview, or if only a single item is found, the system sends the `ShowFoundItem` message to the application that owns that item. For a Newton Works soup item, this message is sent to your stationery viewDef. You must open the appropriate document, highlight the target text, and, if necessary, scroll it to display the text. Note that even though this message is sent to the viewDef, it is documented in this section for completeness.

Note that the `FindSoupExcerpt` and `ShowFoundItem` methods are documented in the *Newton Programmer's Reference*, since they are standard Newton 2.0 methods. Their use is also covered extensively in the chapter on Find in the *Newton Programmer's Guide*.

## Creating the ViewDef

---

Certain slots are required for all viewDefs, such as `symbol`, `type`, `protection`, etc. These slots are not documented in this chapter unless their values are Newton Works-specific. For details on the standard viewDef slots and methods, refer to the "Stationery Reference" chapter of *Newton Programmer's Reference*. For information regarding viewDefs for printing (print formats) refer to the "Routing Interface Reference" chapter of *Newton Programmer's Reference*. Newton Works does not expect any slots other than the standard ones required for all print formats.



## Newton Works

For the main viewDef, the methods and slots that Newton Works uses are explained in the following subsections according to their function.

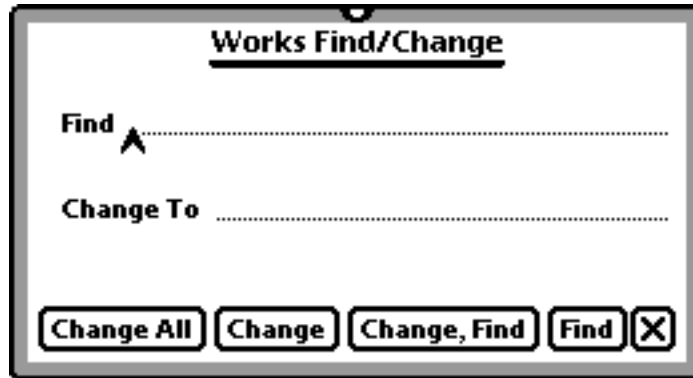
## Supporting Document Find

---

A Newton Works document Find operation is a search initiated by the user invoking the Find or Find Again commands in an open Newton Works document. Such a search operates only on the open document. Contrast this with a Newton Find operation, which applies to all or selected installed applications. Supporting the latter kind of Find operation is described in the section “Supporting Newton Find Operations” (page 1-9).

To support a document Find operation, your viewDef is responsible for putting the Find and Find Again commands into the appropriate button in the application status bar. To do this, add the items `{keyMessage: 'NewtworksFind}` and `{keyMessage: 'NewtworksFindAgain}` to the appropriate picker array. Typically, these commands are found in the picker displayed by the Tools button. The methods `NewtworksFind` and `NewtworksFindAgain` are defined in Newton Works, so the behavior will be correct as long as these methods are called. For example, the behavior for the Find command is to display the Find slip, shown in Figure 1-4.

For more information about adding items with command-key shortcuts to pickers, see the section “About Displaying Command-Key Combinations in Menus” (page 4-6). This section explains the usage of the `keyMessage` slot shown above.

**Figure 1-4** Find slip

Next, your viewDef must implement the `FindChange` method. This method is called when the user taps one of the action buttons (“Change All”, “Change,” “Change,Find,” or “Find”) in the Find slip. This method is passed two parameters: the first identifies the action requested by the user, and the second contains the string to find, and the string to replace it with, if appropriate (for a Change action). Your `FindChange` method must do the requested find or replace operation and update the view appropriately.

The recommended user interface guidelines for doing find or replace operations are as follows:

- Start the find or replace operation from the current highlight range, or from the insertion point if there is no highlight range.
- Wrap the search around to the beginning of the document if the end is reached, making a page flipping sound when the end of the document is passed.
- Stop when the entire document has been searched.
- For a find operation, if an item is found, display the portion of the document containing the item and highlight the item.

If you don’t implement the `FindChange` method, do not add the Find and Find Again commands to a picker, since Find can’t operate in your viewDef.

## Newton Works

## Supporting Data Storage

Newton Works automatically handles saving the current entry periodically (every 30 seconds or after 2 seconds of no system activity). Before saving the entry, Newton Works sends your `viewDef` the `SaveData` message. The `SaveData` method is passed the current entry. In this method, you can update any slots or data structures in the entry or do any other processing necessary.

You must return a value indicating if the entry should actually be saved.

Return `nil` if it does not need to be saved or `true` to save the entry.

Alternatively, you can return the symbol `'NoRealChange`. This causes the entry to be saved, but the modification time stamp is not updated. You might want to do this if the change was insignificant, such as a change in the highlight location but not a change in the data itself.

**Note**

Even if you return `nil` from `SaveData`, the entry still might be saved, if Newton Works detects that it has been modified, for example by one of the `NewtApp` framework protos.

Note that your `SaveData` method does not actually do the saving operation. That is performed by Newton Works after `SaveData` returns. Also, your `SaveData` method does not need to check all the `viewDef` fields to see if they have changed, if your fields are based on the `NewtApp` slot-view protos. That is because these slot-view protos handle saving their own data when it is changed.

In addition to the automatic periodic saving that Newton Works performs, you can manually invoke a save operation on the current entry by calling the `newtEntryView` method `StartFlush`. Just send this message to `self`, like this:

```
:StartFlush();
```

The `StartFlush` method starts the flush timer that calls the method `EndFlush` after 5 seconds. `EndFlush` sends your `viewDef` the `SaveData` message, as described previously, and saves the entry, if appropriate. This is the standard way of saving stationary data outside the automatic saving mechanism described previously.

It is possible for your `SaveData` method to be passed an entry that is invalid or read-only. In the `SaveData` method, before you do begin any operations that write to the soup entry, you should perform this check:

## Newton Works

```
If EntryValid(entry) and not EntryStore(entry):IsReadOnly()
    then // do the operation
```

## Supporting Scrolling

---

If the documents created by your stationery can be larger than what can be displayed on the screen at one time, then you'll want to provide a mechanism for the user to scroll the view. Newton Works can display scroll bars for you and can facilitate scrolling with some support from you in your viewDef.

First, you must supply the method `GetScrollableRect` in your viewDef. This method must return a bounds frame that describes the rectangle enclosing the scrollable part of the view (usually the currently visible part of the data, not including rulers or other tools). Newton Works automatically displays scroll bars at the right side and bottom of the view, as needed. The scroll bars are displayed in the stationery's topmost view, not inside the scrollable view.

If you return `nil` from `GetScrollableRect`, Newton Works does not display any scroll bars and scrolling functionality is not supplied by Newton Works. You can return `nil` if you want to provide your own scrolling functionality or if you don't need scrolling.

You also must supply a group of methods in the viewDef that return status information. `GetScrollValues` must return a frame with `x` and `y` slots containing the current scroll thumb positions as integers. `GetTotalHeight` and `GetTotalWidth` must each return integers, corresponding respectively to the total height and width of the object to be scrolled (not just the visible area). The values returned by `GetScrollValues` are relative to the values returned by `GetTotalHeight` and `GetTotalWidth`. So if the total height and width of the object were each 1000, and each scroll thumb was positioned in the middle of its scroll bar, then `GetScrollValues` should return `{x: 500, y: 500}`. If the thumbs were positioned at the maximum positions, then `GetScrollValues` should return `{x: (1000 - width), y: (1000 - height)}`.

Next, you may want to supply methods that update your view in response to a scroll request. Two methods, `ViewScrollDownScript` and `ViewScrollUpScript` must update the view in response to a Down or Up arrow key, respectively. If you don't define these methods, then Newton Works performs the default action of scrolling down or up one screenful.

## Newton Works

Finally, the `Scroll` method is the key method in which you must do the appropriate work to scroll the document. Your `Scroll` method is called by Newton Works as a result of the user manipulating the scroll bar controls. The `Scroll` method is passed a frame with `x` and `y` slots, describing how far to scroll in each direction.

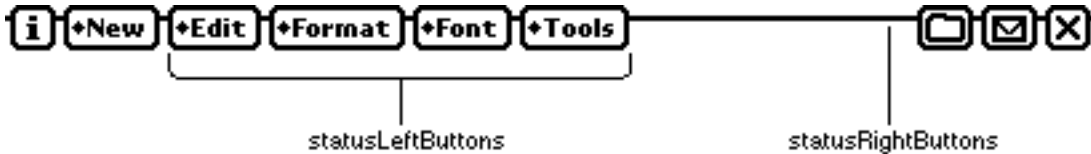
Note that you must call the Newton Works method `UpdateAllScrollers` from your `Scroll` method. `UpdateAllScrollers` updates the scroll bar controls to reflect the new scrolled position of the view. You should call this method when you open the view and anytime you make a change that affects the size of the view (for example, changing margins or displaying rulers or other control elements), the size of the document (for example, adding or removing information), or the scrolled position of the document. You pass it the view and four Boolean values that indicate: if the total height changed, if the vertical scroller thumb needs to be updated, if the total width changed, and if the horizontal scroller thumb needs to be updated. `UpdateAllScrollers` calls your methods `GetScrollValues`, `GetTotalHeight`, and `GetTotalWidth` as needed to recalculate the internal scroller data structures.

## Providing Status Bar Buttons

---

To provide any custom buttons in the status bar, you can specify button view templates in one of two slots in your `viewDef`: `statusLeftButtons` and `statusRightButtons`. Specify an array of one or more button templates in the `statusLeftButtons` slot to include them at the left side of the status bar, but to the right of the New button, as shown in Figure 1-5. They are laid out from left to right, beginning with the first button in the array.

Specify an array of one or more button frames in the `statusRightButtons` slot to include them at the right side of the status bar, but to the left of the Routing and Filing buttons. They are laid out from right to left, beginning with the first button in the array.

**Figure 1-5** Status bar buttons

If you are registering tools for your Newton Works stationery or documenting your tool format for other developers to use, you should include a Tools button on the status bar that displays a picker listing the installed tools. You can use the Newton Works method `GetNewtWorksTools` to return an array of tools registered for your stationery.

There is a notification method that you can supply in your viewDef that is called if some other package installs or removes an auxiliary button in the Newton Works status bar. This method is `UpdateStatusBar`. If you have dynamic buttons in the status bar, you should provide this method to update the status bar button arrays. Note that you don't need to do anything to add auxiliary buttons; Newton Works does all the work. The `UpdateStatusBar` message simply allows you to update your own buttons.

This viewDef method is also called anytime the status bar is redrawn by the `newtApp` base method `UpdateStatusBar`. For example, this can happen when the screen is rotated, and you might want to show a different number of buttons depending on how much screen width is available.

The `newtApp` base `UpdateStatusBar` method causes the status bar view to regenerate its child views, calling its `ViewSetupChildrenScript` method, which in turn calls your viewDef `UpdateStatusBar` method. Whatever you set up in the two button arrays in your method is added to the existing status bar buttons when the status bar is redrawn.

## Providing Help

The user accesses Help by tapping the Help command in the Information picker ("i" button on the status bar). When this command is chosen, you can either do a custom operation for your stationery, or open a help book, or both. When this command is chosen, Newton Works first conditionally sends

## Newton Works

your `viewDef` the `DoHelp` message. You can do any kind of special processing in this method. If you also want to open a help book, you must return the symbol `'loadHelp` from the `DoHelp` method.

If `DoHelp` is not implemented, or if it returns `'loadHelp`, then Newton Works opens a help book. Two slots in the `viewDef` are used to identify the help book and location to which it is opened: `helpManual` and `viewHelpTopic`. The `helpManual` slot must contain a help book frame (the book frame produced by Newton Book Maker). The `viewHelpTopic` slot sets the topic to which that help book should open. This must be a string that matches one of the `.subject` lines in the help book. Refer to *Newton Book Maker User's Guide* and the sample code available on help books for more details about help book frames and topics.

## Notification of Changes

---

There are global preferences that apply to all Newton Works applications. These are accessed by the user through the Works Prefs choice on the Info picker. Currently, the global preferences contain two items: a choice of using metric measurement units (rather than English) and a setting to always store new documents internally (rather than on a storage card).

If the global preferences for Newton Works are changed, Newton Works sends the current `viewDef` the `PrefsChanged` message, passing the global preferences frame. This allows you to respond appropriately, if necessary, to the user's wishes.

If the view bounds of your view change, Newton Works sends the `viewDef` the `ViewChangedScript` message. This could happen if the horizontal scroller is no longer needed, or if the icon bar is moved from the side to the bottom on a MessagePad 2000 unit, and so forth. This `viewDef` method is a standard view method that is also called anytime a view is changed as a result of a `SetValue` call.

The `ViewChangedScript` method is passed two parameters, the slot that changed and the view that changed. For more information, see the *Newton Programmer's Reference*.

## Working With the Tools Picker

You can add tools to the Tools picker of any Newton Works stationery that supports additional tools. For example, you might add a thesaurus to the word processor. The user accesses such tools from the Tools button in the status bar. When the user taps the Tools button, a picker of available tools is displayed, as shown in Figure 1-6. The application viewDef is responsible for displaying the tools installed for that viewDef, and for calling the tool correctly when it's chosen from the Tools picker.

**Figure 1-6** Tools picker



Note that the Tools button already exists in the built-in Newton Works stationery, but if you are creating your own stationery, you must add the Tools button to the status bar yourself. For details on how to do this, see the section “Providing Status Bar Buttons” (page 1-15). If you are adding a tool to a different type of stationery that supports tools, you may need to add the Tools button if there are no other tools registered. Don’t assume that the button already exists.

You register a tool for an application by calling the `RegNewtWorksTool` method. To remove a tool you’ve added, call `UnregNewtWorksTool`. You can call `GetNewtWorksTools` to return an array of registered tools for a particular application, or you can call `GetNewtWorksTool` to return a particular tool.

Within the viewDef or status bar context, you can use the syntax `newAppBase:MethodName` to call these methods. For example: `newAppBase:GetNewtWorksTools(sym)`. Outside the viewDef, you’ll need to call these methods like this:

```
GetRoot().newWorks:RegNewtWorksTool(toolSym, toolFrame);
```



## Newton Works

Whenever a tool is installed or removed for the current document, the system sends the message `ToolsChanged` to the `viewDef`. If you choose to implement it, this method allows you to update the tools picker, if necessary, or do other work. Note that an even better way of making sure the tools picker is current is to generate it dynamically when the user taps the Tools button.

## Newton Works Interface Reference

---

This section describes the slots and methods in the Newton Works base application and in `dataDefs` and `viewDefs` that you intend to register for use in Newton Works.

### Newton Works Base Application Slots and Methods

---

The following slots and methods reside in the Newton Works base application.

#### Slot descriptions

<code>newAppBase</code>	Contains the Newton Works base view.
<code>viewDefView</code>	Contains the current <code>viewDef</code> view. Can be <code>nil</code> if no <code>viewDef</code> is currently active, as in the case when the current layout is the overview.

The following methods are also provided.

#### GetNewtWorksTool

---

`newAppBase: GetNewtWorksTool(toolSym)`

Returns a tool frame.

<i>toolSym</i>	The tool symbol to find among the registered tools.
return value	Returns the frame corresponding to the tool identified by <i>toolSym</i> . If no tool with that symbol is found, <code>nil</code> is returned.

## Newton Works

**GetNewtWorksTools**

---

*newtAppBase*:GetNewtWorksTools(*dataTypeSym*)

Returns an array of tools registered under a particular symbol.

*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.

return value      Returns an array containing the tools identified by *dataTypeSym*. The array contains the frames passed in the *toolFrame* parameter to the *RegNewtWorksTool* method. If no tools were found, an empty array is returned.

**DISCUSSION**

Use this method to return all tools registered for a particular Newton Works stationery application. For example, *GetNewtWorksTools('paper')* returns an array of all the tools registered for the Word Processor, since 'paper' is its *dataDef* symbol.

**RegNewtWorksTool**

---

*newtAppBase*:RegNewtWorksTool(*toolSym*, *toolFrame*)

Registers a tool for the viewDef identified by the *dataTypeSymbol* slot in the *toolFrame* frame.

*toolSym*      A unique symbol under which to register the tool.

*toolFrame*      A frame describing the command to appear in the Tools picker. Each command is 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 (such as *item*, *icon*, and all other slots supported by *PopupMenu*) define the appearance of the command in the viewDef's

## Newton Works

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` for which it's registered. For example, tools for the word processor have the symbol `'paper`.

**return value** Returns `non-nil` if the tool was successfully registered, or `nil` if it was not.

## DISCUSSION

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 Newton Works application. The return value of this method is not used.

`keyCommand`

Optional, used by Word Processor stationery only. A frame with command key information, containing the same slots as the frames used to register command keys, as described in “The Command-Key Mapping Frame” (page 4-31). To register a command key that activates an item on the Tools picker, the `keyCommand` frame must contain the method `KeyFn(keyView)`. This method is called when the command key for this item is pressed. It is passed the key view, which will be the `viewDefView` or a child of it. This function should perform the action as if the command were chosen from the Tools picker. The return value of this function is not used.

## Newton Works

**Note**

The `keyCommand` frame has a `keyMessage` slot that contains a symbol identifying a function. You should specify a symbol that contains your developer signature appended to it in order to uniquely identify it. This is because the Word Processor creates a slot of this name in its base view, and you don't want to conflict with any other tools that might specify a function with the same name. ♦

If you are registering a tool for the Word Processor and including the `keyCommand` frame in the `toolFrame`, and thus are specifying a `KeyFn` method, you don't need to supply a `CmdFunc` method. This is because if a `CmdFunc` method does not exist, then Newton Works calls the `KeyFn` method in the `keyCommand` frame when a tool is chosen.

Normally you would want the `KeyFn` method in the `keyCommand` frame to do the same thing as the `CmdFunc` method in `toolFrame` anyway. The `KeyFn` method is called if the user presses the command-key combination for the tool and the `CmdFunc` method is ordinarily called if the tool is chosen from the Tools picker directly.

A tool can be registered for only one `dataDef` with each call to `RegNewtWorksTool`; to register a tool for more than one `dataDef`, call `RegNewtWorksTool` again with the new `dataTypeSymbol` and a different `toolSym`.

**UnregNewtWorksTool**


---

```
newtAppBase:UnregNewtWorksTool(toolSym)
```

Unregisters a tool registered by `RegNewtWorksTool`.

<i>toolSym</i>	The symbol under which the tool was originally registered.
----------------	--

return value	Undefined; do not rely on it.
--------------	-------------------------------

**DISCUSSION**

Note that if you pass a symbol for a tool that is not registered, nothing happens and no error occurs.

## Newton Works

**UpdateStatusBar**

---

*newtAppBase*:UpdateStatusBar()

Recreates and redraws the status bar.

return value            Undefined; do not rely on it.

**DISCUSSION**

This method can be called to recreate the status bar, for example, if you have changed the button arrays. This method calls the `ViewSetupChildrenScript` method of the status bar, which in turn calls the `UpdateStatusBar` method of the view identified in the `viewDefView` slot, if there is a currently active `viewDef`.

**Newton Works DataDef Slots and Methods**

---

The slots in the `dataDef` that are used by Newton Works are as follows.

**Slot descriptions***superSymbol*

Required. Must be the symbol 'newtWorks.

*prefs*

Optional. Defines an application-specific preferences command that appears in the Info button picker when the user is viewing an entry of this `dataDef`. This slot holds 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 picker. 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 preferences.

The following methods are also optional in the `dataDef`.

## Newton Works

**FindFn**

---

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

Called by Newton Works when a global (Newton-wide) Find is performed, and a search using the function FindStringInFrame finds nothing in a Newton Works soup entry.

<i>entry</i>	The soup entry to search.
<i>what</i>	The string to search for.
<i>offset</i>	An integer specifying an offset, in characters. Begin searching in <i>entry</i> starting at this offset.
return value	Return non- <i>nil</i> from this method if the string <i>what</i> is found in the entry anywhere after <i>offset</i> . Return <i>nil</i> if the string is not found.

**DISCUSSION**

This method should search the soup entry for the *what* string.

Note that if the string is in the soup entry in plain NewtonScript string form (if FindStringInFrame returns true) FindFn will not be called because Newton Works has already identified the entry as matching the Find criteria. The FindFn method is called only when FindStringInFrame finds nothing, to let you do any custom processing that might be needed in order to do a string search.

## Newton Works

**InfoBoxExtract**

---

*dataDef*: InfoBoxExtract(*target*, *maxSize*, *viewDefView*)

Called conditionally by Newton Works when the user opens the title slip, to get extra information to add to the title slip.

<i>target</i>	The current entry in Newton Works.
<i>maxSize</i>	A bounds frame defining the maximum size of the shape you can return. It contains the slots <code>left</code> , <code>top</code> , <code>right</code> , and <code>bottom</code> .
<i>viewDefView</i>	The viewDef for the target.
return value	Return an array of one or more shapes, which are added to the bottom of the title slip, or <code>nil</code> if you don't want to add any extra information.

**DISCUSSION**

The shapes you return must fit inside the rectangle described by *maxSize*. Typically you don't need the maximum size, so just return a shape that is smaller than this size. Newton Works puts the shape you return at the bottom of the Title slip, making the slip just big enough to hold the shape you return.

This method is optional. If you don't supply it, no extra information is added to the title slip.

**Newton Works Viewdef Slots and Methods**

---

The slots in the viewDef that are used by Newton Works are as follows.

**Slot descriptions**

<code>symbol</code>	Must be the symbol <code>'default</code> for the main viewDef.
<code>statusLeftButtons</code>	Optional. 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

## Newton Works

	array, see the <code>menuLeftButtons</code> slot in the <code>newStatusBar</code> proto in <i>Newton Programmer's Reference</i> .
<code>statusRightButtons</code>	Optional. 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 <code>viewDef</code> appears. For more details on this array, see the <code>menuRightButtons</code> slot in the <code>newStatusBar</code> proto in <i>Newton Programmer's Reference</i> .
<code>helpManual</code>	Optional. A frame defining the help book to open. This is the same as the <code>newApplication</code> <code>helpManual</code> slot. This frame must be a book frame as produced by Newton Book Maker.
<code>viewHelpTopic</code>	Optional. A string specifying the location to which the help book is to open. This string must match one of the <code>.subject</code> lines in the help book.

The following methods are also expected or optional in the `viewDef`.

**DoHelp**


---

*viewDef*: `DoHelp(entry)`

Called conditionally when the user chooses Help from the information picker.

<i>entry</i>	The soup entry currently being displayed in the view.
return value	If the default behavior of opening the help book is desired, return <code>'loadHelp</code> from this method; otherwise, return any other value.

**DISCUSSION**

If this method exists, it will be called when the user chooses Help from the information picker when the `viewDef` is visible, instead of the default behavior of using the slots `viewHelpTopic` and `helpManual` to open a help book. However, you can still open a help book by returning the symbol `'loadHelp` from this method.



Newton Works

SEE ALSO

For more information, see the section “Providing Help” (page 1-16).

**FindChange**

---

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

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

<i>action</i>	A symbol indicating the action the user requested: 'find', 'change, or 'changeAll.
<i>data</i>	Varies depending on the value of <i>action</i> . See Table 1-1 for details.
return value	Varies depending on the value of <i>action</i> . See Table 1-1 for details.

DISCUSSION

Table 1-1 explains what you should do in this method, the value of the *data* parameter, and what you should return. This method also is responsible for

Newton Works

updating the view appropriately. For example, if *action* is 'find, you must scroll to the first found item and highlight it.

**Table 1-1** FindChange parameters and actions

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

**GetScrollableRect**

*viewDef*:GetScrollableRect()

Called to get a rectangle that describes the scrollable area of the view.

**return value** Return a bounds frame of integers (*top*, *left*, *bottom*, *right*) describing the rectangle enclosing the visible region of the view. If *nil* is returned, Newton Works removes the scroll bars, and no other scroll methods are required.

**DISCUSSION**

Return *nil* from this method if you want to implement scrolling entirely on your own, or if scrolling is not needed.

## Newton Works

Note that if the scroll bars are displayed and you want to remove them by returning `nil` from this method, you must also call `UpdateAllScrollers`, as follows:

```
:UpdateAllScrollers(self, true, nil, true, nil);
```

---

**GetScrollValues**

---

*viewDef*: `GetScrollValues()`

Called to get the current scroll values.

return value	Return a frame with <i>x</i> and <i>y</i> slots containing the horizontal and vertical scroll values, as integers.
--------------	--

**DISCUSSION**

The values you return are the coordinates of the current origin of the view; that is, the point at the upper-left corner of the view.

---

**GetTotalHeight**

---

*viewDef*: `GetTotalHeight()`

Called to get the total height of the document.

return value	Return an integer that is the total height of the document in pixels.
--------------	---

---

**GetTotalWidth**

---

*viewDef*: `GetTotalWidth()`

Called to get the total width of the document.

return value	Return an integer that is the total width of the document in pixels.
--------------	--

## Newton Works

**PrefsChanged**

---

*viewDef:* PrefsChanged(*prefsFrame*)

Called conditionally when the global preferences for Newton Works are changed.

<i>prefsFrame</i>	A frame containing the following slots:
<i>metricUnits</i>	True if the measurement unit is set to centimeters; nil if inches.
<i>internalStore</i>	True if all items are to be stored on the internal store; nil if they can be stored anywhere.
return value	You can return anything; it is ignored.

**SaveData**

---

*viewDef:* SaveData(*entry*)

Called when the current entry is about to be saved to the soup.

<i>entry</i>	The target entry to be saved. Note that the entry could be invalid or read-only; see the discussion.
return value	Return true (if you want the entry saved), nil (if you haven't changed the entry), or the symbol 'NoRealChange (if you want the data saved, but the modification time not updated).

**DISCUSSION**

If there is data to be saved, modify *entry* to hold the new or changed data, and return true. Note that if you return nil, the soup entry may still be saved if it has otherwise been modified, for example, by the NewtApp framework. Returning nil does not prevent the entry from being saved, it just notifies Newton Works that you didn't change it.

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.

## Newton Works

This tells Newton Works to save the changes to the soup entry, but not to update the modification time of the entry (as displayed in the title slip).

It is possible for your `SaveData` method to be passed an entry that is invalid or read-only. In the `SaveData` method, before you do begin any operations that write to the soup entry, you should perform this check:

```
If EntryValid(entry) and not EntryStore(entry):IsReadOnly()
    then // do the operation
```

**Scroll**

---

*viewDef*: `Scroll(scrollValues)`

Called to scroll the view contents horizontally and/or vertically.

*scrollValues*            A frame with *x* and *y* slots. You must scroll the view contents by *x* pixels horizontally and *y* pixels vertically.

return value            You can return anything; it is ignored.

**DISCUSSION**

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

**ToolsChanged**

---

*viewDef*: `ToolsChanged(actionSym, toolSym)`

Called conditionally when a tool is installed or removed for the current document.

*actionSym*            The symbol 'install if a tool was installed, or 'remove if it was removed.

*toolSym*            The unique symbol identifying the installed or removed tool.

return value            You can return anything; it is ignored.

**DISCUSSION**

This method allows the *viewDef* to update its tools menu, if necessary.

## Newton Works

**UpdateAllScrollers**

---

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

Updates the scroll bar controls.

<i>view</i>	The view ( <i>self</i> ).
<i>totalHeightChanged</i>	A Boolean value indicating whether or not the height of the document has changed.
<i>scrolledV</i>	A Boolean value indicating whether or not the vertical scroller thumb needs to be updated.
<i>totalWidthChanged</i>	A Boolean value indicating whether or not the width of the document has changed.
<i>scrolledH</i>	A Boolean value indicating whether or not the horizontal scroller thumb needs to be updated.
return value	Undefined; don't rely on it.

**DISCUSSION**

You call this Newton Works method to update the scroll bar controls to reflect the new scrolled position or size of the view. You should call this method when you open the view and anytime you make a change that affects the size of the view (for example, changing margins or displaying rulers or other control elements), the size of the document (for example, adding or removing information), or the scrolled position of the document.

UpdateAllScrollers calls your methods *GetScrollValues*, *GetTotalHeight*, and *GetTotalWidth* as needed to recalculate the internal scroller data structures.

Note that UpdateAllScrollers needs to be called from your *Scroll* method if the scroll bar controls need visual updating.

**UpdateStatusBar**

---

*viewDef:*UpdateStatusBar()

Allows the view to update the status bar button arrays before the status bar is redrawn.

return value	You can return anything; it is ignored.
--------------	---

Newton Works

**DISCUSSION**

This message is sent to the view identified by the `viewDefView` slot, if Newton Works receives notification of an auxiliary button change. This can occur if a package installs or removes an auxiliary button for Newton Works. When this message is called, the view should update its `statusLeftButtons` and `statusRightButtons` slots. This message is also sent anytime the button array is recreated and redrawn.

Note that this method is optional.

## Summary of Newton Works

---

### Newton Works Base Application

---

```
newtonWorksBaseView := {
newtAppBase: view, // Newton Works base view
viewDefView: view, // current viewDef; nil if overview is current
GetNewtWorksTool: func(toolSym), // gets a specific tool
GetNewtWorksTools: func(dataTypeSym), // gets all tools for an app
RegNewtWorksTool: func(toolSym, toolFrame), // registers tool
UnregNewtWorksTool: func(toolSym), // unregisters tool
UpdateStatusBar: func(), // redraws status bar
...
}
```

### Newton Works Stationery DataDef

---

```
myNewtonWorksDataDef := {
superSymbol: 'newtWorks, // required
prefs: { // app-specific preferences info
    prefsTemplate: viewTemplate, // preferences slip
    item: string, // preferences command name for picker
    icon: frame, // bitmap frame holding icon for picker (optional)
    indent: integer, // indent for name in picker (optional)
},
FindFn: func(entry, what, offset) ..., // searches soup entry for what
InfoBoxExtract: // called when title slip is opened
    func(target, maxSize, viewDefView) ...,
...
}
```

### Newton Works Stationery ViewDef

---

```
myNewtonWorksViewDef := { // viewDef for screen (not printing)
symbol: 'default, // required
statusLeftButtons: array, // left button array
```



## Newton Works

```

statusRightButtons: array, // right button array
helpManual: frame, // book frame identifying help book
viewHelpTopic: string, // location to open help book to
DoHelp: func(entry) ..., // called when user chooses Help command
FindChange: // called when user performs a document Find operation
    func(action, data) ...,
GetScrollableRect: func() ..., // called to get the scrollable rect
GetScrollValues: func() ..., // called to get the current scroll values
GetTotalHeight: func() ..., // called to get the total height
GetTotalWidth: func() ..., // called to get the total width
PrefsChanged: func(prefsFrame) ..., // called when global prefs change
SaveData: func(entry) ..., // called when entry is about to be saved
Scroll: func(scrollValues) ..., // called to scroll the view
ToolsChanged: // called when tool is installed or removed for app
    func(actionSym, toolSym) ...,
UpdateAllScrollers: // updates scroll bar controls
    func(view, totalHeightChanged, scrolledV, totalWidthChanged, scrolledH),
UpdateStatusBar: // called so view can update status bar button arrays
    func() ...,
...
}

```

## CHAPTER 1

### Newton Works

# Newton Works Draw Application

---

The Draw application is implemented as stationery in the Newton Works framework. It is accessed by choosing “Drawing” from the Newton Works New picker. This chapter describes how you can programmatically create and edit the graphic objects in the current drawing, and add tools, stamps, and patterns to this application.

This chapter assumes familiarity with the Newton object-based graphic system, as defined in Chapter 6, “Drawing and Graphics 2.1,” of this document, and Chapter 13, “Drawing and Graphics,” in *Newton Programmer’s Guide*.

You may wish to read Chapter 1, “Newton Works,” before reading this chapter.

## About the Draw Application

---

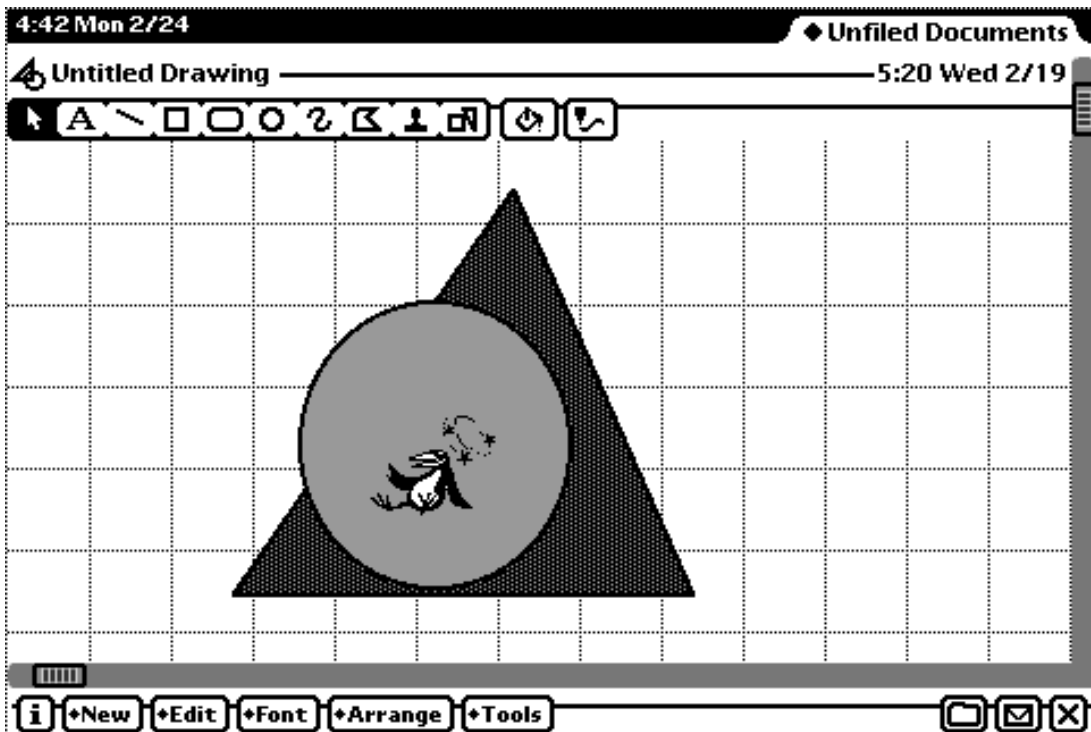
The Draw application is one of the built-in Newton Works stationeries.

## Newton Works Draw Application

## User Interface

The Draw application has a typical Newton Works appearance. It contains a status bar, a title bar, and both horizontal and vertical scroll bars. In addition, there is a tool bar and a large editable view called the **canvas**. All drawing takes place in the canvas. The tool bar contains the tools that create graphical objects and set the fill and pen styles.

**Figure 2-1** The Newton Works Draw application



This application does not include folder tabs if it is in Classroom Mode. For more information on Classroom Mode, see “User Interface” (page 2-2) in Chapter 2, “Newton Works Draw Application.”

## Programmer's Overview

---

You can add tools, stamps, and patterns to this application, and manipulate the contents of the canvas. The patterns are used in the fill pattern tool's palette (the paint bucket). The stamps are used by the stamp tool; they add bitmaps or picture shapes to the canvas. The tools are added to the tool bar, and should create a graphical shape of some sort.

If you wish to add bitmaps to the Draw application, you can do this through either the tool or the stamp interface. In general, you should include bitmaps as stamps; this is what the stamp tool is designed to do. It is also easier to programmatically add stamps, than to add tools.

As with other Newton Works stationery, you can add tools to the Tools picker. This is not discussed in this chapter, this chapter only describes how to add tools to the tool bar. For information on adding to the Tools picker, see "Working With the Tools Picker" (page 1-18).

## Using the Drawing Application Interface

---

This section describes the NewtonScript interface to the Draw application.

### Adding Custom Drawing Tools

---

You can add tools to the Draw application. These tools allow the user to create new types of draw objects. Note that in many cases, it makes more sense to create new graphic objects as bitmaps via the stamp interface; see "Adding Stamps to the Stamp Tool" (page 2-7).

Tools are added by calling the Draw application `viewDef.RegTool` method. Each new tool is added to the end of the toolbar. The toolbar grows off the right end of the screen, if enough tools are added.

Tools are based on the `protoDrawTool` frame. You are required to supply the following:

- an icon to display in the toolbar
- a class symbol that identifies the tool and the object it creates

## Newton Works Draw Application

- a `MakeObject` method that returns a graphic shape to add to the canvas
- a `SetAttribute` method to respond to the user selecting, and perhaps modifying the object

The object created should be a shape array with the same class as the 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 rectangle shape. The rectangle should be the size of the shape you are returning, and the style frame should be:

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

This rectangle at the end of your shapes array is for the selection handles.

The default behavior of `protoDrawTool` is to draw a bounding box as the user drags the pen across the screen. This creates a bounds box which is passed to the tool’s `MakeObject` method. You can override this behavior in a number of ways.

If it makes more sense to create the object with a tap, instead of by dragging a bounding box, set the tool’s `createByTapping` slot to `true`, and its `createByDragging` slot to `nil`. The stamp tool, for example, creates its objects with pen taps.

You may also draw something other than a rectangle as the user drags the bounding box. The oval tool, for example, draws an oval as the bounding box is being created. To draw your own object, set your tool’s `dragARectangle` slot to `nil`, and optionally provide a `MakeDragObject` method. The `MakeDragObject` method is called to return a shape as the bounding box is being dragged. The default version of this method simply sends your tool a `MakeObject` message. If the objects your tools creates are very complicated, you may want to create a simpler object here. Your `MakeDragObject` method could return the outline of your object, for example.

The example below creates a simple tool that makes a shape comprised of an equal size oval and rectangle.

---

**Listing 2-1** Adding a tool to the Draw application's tool bar

```
OpenResFile(Home & "resources");
DefineGlobalConstant ('kToolIcon ,
```

## Newton Works Draw Application

```

        MakePixFamily ( nil, nil, {rsrcSpec:"toolIcon", bitdepth:1}) );
CloseResFile();

DefineGlobalConstant('kMyDrawToolTemplate,
{
    _proto : UR('|Draw:NEWTON|, 'protoDrawTool), // get proto from unit
    class  : '|OvalRect:MySig|,
    icon   : kToolIcon,
    MakeObject : func (left,top,right,bottom,style)
    begin
        local l := Min(left,right), r := Max(left,right),
              t := Min(top,bottom), b := Max(top,bottom);
        local oval := MakeOval (l,t,r,b);
        local rect := MakeRect (l,t,r,b);
        local mainStyle := Clone(style);
        mainStyle.selection := nil;

        [|OvalRect:MySig|:
            mainStyle,
            rect,
            oval,
            { penSize:0, fillPattern:vfNone, selection:nil },
            rect,
        ];
    end,

    SetAttribute : func ( shape, attribute, newValue )
    begin
        //selection handles go on the invisible rect.
        if attribute = 'selection then
            shape[3] . selection := newValue;
        else shape[0].(attribute) := newValue;
        end
    end;

    InstallScript := func(partFrame, removeFrame)
    begin
        local viewDef := GetViewDefs('drawPaper).default;
        local sym := EnsureInternal(kAppSymbol);
        if viewDef then
            viewDef:RegTool(sym, kMyDrawToolTemplate);
        end;

        RemoveScript := func (removeFrame)
        begin
            local viewDef := GetViewDefs('drawPaper).default;
            if viewDef then

```

## Newton Works Draw Application

```
viewDef:UnRegPatterns(kAppSymbol);
end;
```

## Adding Patterns and Gray Tones to the Fill Tool

---

To add patterns or gray tones to the fill pattern palette, use the `RegPatterns` method of the Draw application's `viewDef`. This method takes two arguments, a uniquely identifying symbol and an array of patterns. This array can contain any of the `kRGB_GrayXX` gray tone constants, a packed RGB integer (as returned by `PackRGB`), or a pattern, gray pattern, or dithered pattern. For information on these values see "Specifying Shades of Gray" (page 6-9) and "Using Patterns, Gray Patterns, and Dithered Patterns" (page 6-11). The `viewDef` contains a `UnRegPatterns` method to remove patterns from the fill palette.

Patterns are usually added in an auto part package. The sample code below shows the full content of a text file that creates such an auto part.

---

### Listing 2-2 Adding to the Draw application's fill tool

```
DefineGlobalConstant( 'kCheckerPattern,
    MakeBinaryFromHex ( "F0F0F0F0F0F0F0F", 'pattern)

InstallScript := func (partFrame, removeFrame)
begin
    local viewDef := GetViewDefs('drawPaper).default;
    local sym     := EnsureInternal(kAppSymbol);
    local pats    := [    kRGB_Gray6,
                          vfGray,
                          kCheckerPattern,
                          {    class      : 'ditherPattern,
                              pattern    : vfLtGray,
                              foreground : kRGB_Gray2,
                              background : kRGB_GrayD,
                              }
                        ];
    if viewDef then
        viewDef:RegPatterns(sym, pats)
    end;

    RemoveScript := func (removeFrame)
begin
```



## Newton Works Draw Application

```

local viewDef := GetViewDefs('drawPaper').default;
if viewDef then
    viewDef:UnRegPatterns(kAppSymbol);
end;

```

## Adding Stamps to the Stamp Tool

---

To add stamps to the stamp palette, use the `RegStamps` method of the Draw application's `viewDef`. This method takes two arguments, a uniquely identifying symbol and an array of stamps. The stamp array should contain no more than 24 stamps, since the stamp palette only displays 24 stamps at a time. If you wish to add more than 24 stamps, make more than one array, and call `RegStamps` separately for each array.

The stamps can be pix families or picture objects, and should be less than 1 KB. For information on pix families and picture objects, see Chapter 6, "Drawing and Graphics 2.1." There is no limit on the size of the image, but you must be aware of the memory consumption.

### Note

The image is stored in a soup, so you are ultimately limited by the maximum size of a soup entry (currently 64 KB). Soups are described in Chapter 11, "Data Storage and Retrieval," in *Newton Programmer's Guide*. ♦

The Draw application's `viewDef` contains an `UnRegStamps` method to remove stamps added with `RegStamps`.

Stamps are usually added in an auto part package. The sample code below shows the full content of a text file that creates such an auto part.

---

### Listing 2-3 Adding stamps to the Draw application

```

OpenResFile( Home & "myResourceFile" );

//Note that only stamp6 really needs 16 grays, and some stamps are OK
//using only 4 grays. This saves a lot of space over importing all the
//stamps at a bit depth of 4.
DefineGlobalConstant('kStampArray', [
    MakePixFamily( nil, nil, {rsrSpec: "stamp1", bitdepth: 1} ),

```

## Newton Works Draw Application

```

        MakePixFamily (nil,nil,{rsrcSpec : "stamp2", bitdepth : 1} ),
        MakePixFamily (nil,nil,{rsrcSpec : "stamp3", bitdepth : 1} ),
        MakePixFamily (nil,nil,{rsrcSpec : "stamp4", bitdepth : 2} ),
        MakePixFamily (nil,nil,{rsrcSpec : "stamp5", bitdepth : 2} ),
        MakePixFamily (nil,nil,{rsrcSpec : "stamp6", bitdepth : 4} ),
    ] );
    CloseResFile();

    InstallScript := func(partFrame, removeFrame)
    begin
        local viewDef := GetViewDefs('drawPaper').default;
        local sym := EnsureInternal(kAppSymbol);

        if viewDef then
            viewDef:RegStamps(sym, kStampArray);
        end;

    RemoveScript := func(removeFrame)
    begin
        local sym := EnsureInternal(kAppSymbol);
        local viewDef := GetViewDefs('drawPaper').default;

        if viewDef then
            viewDef:UnRegStamps(sym);
        end;
    end;

```

## Draw Application Methods

---

In addition to the `RegTool`, `RegPatterns`, and `RegStamps` methods, and their unregistering counterparts discussed elsewhere in this chapter, the Draw application's `viewDef` contains three other methods of interest: `GetContents`, `SetContents`, and `GetCanvas`. The `GetContents` and `SetContents` methods are used to manipulate the entire contents of the current drawing.

The `GetCanvas` method returns a reference to the canvas view; a number of methods are defined in this view which are available to you. For more information on the canvas view, see “The Canvas and Its Methods” (page 2-9).

The way you get a reference to the Draw application `viewDef`, to send these methods to, depends on the environment in which your code is executing.

- If you are writing code for a tool, this code executes as a child of the canvas. In this case you can send these methods to `self`. The Draw

## Newton Works Draw Application

application is found via parent inheritance, as in the following sample calls:

```
:GetContents();
:GetCanvas();
```

- If you are writing code for a tool to be added to the Tools picker, with the Newton Works `RegNewtWorksTool` method, this code executes in the environment of Newton Works, not its Draw application `viewDef`. In this case, send these messages to the `viewDefView` slot, as in the following sample calls:

```
viewDefView:GetContents();
viewDefView:GetCanvas();
```

- If your code is executing outside these environments, you can get a reference to the Draw application by explicitly accessing Newton Work's `viewDefView` slot. However, in this case it is not guaranteed that the Draw application is the current displaying `viewDef`; you must check for this before sending these messages, as in the following code:

```
if GetRoot().newtWorks AND Visible(GetRoot().newtWorks) then
begin
  local curNWApp := GetRoot().newtWorks.viewDefView;
  if GetVariable(curNWApp, 'currentdatatype') = 'drawPaper then
  begin
    local cont := curNWApp:GetContents();
    local can := curNWApp:GetCanvas();
    ...
  end;
end;
```

## The Canvas and Its Methods

The canvas view provides methods to programmatically add a shape to it, and a number of methods that manipulate the set of currently selected shapes. If you are writing code for a tool in the tool bar, simply send these methods to `self`. Otherwise, use the `GetCanvas` method described in “Draw Application Methods” (page 2-8).

The current drawing style is stored in the canvas' `currentDrawStyle` slot. You may change the values in this style frame programmatically; the user can also set some of these slot through the user interface.

## Newton Works Draw Application

The `AddShape` method adds a new shape to the canvas. The canvas provides a `DirtyShape` method to redraw the part of the canvas where the new shape was added. Simply pass the same shape to both functions, as in:

```
local shape := MakeRect (0,0,100,100);  
:AddShape(shape,nil,nil);  
:DirtyShape(shape);
```

A number of functions exist to select shapes, and manipulate them. You can programmatically call many of the methods the user accesses through the Edit and Arrange pickers. See “The Canvas” (page 2-17).

## Draw Application Reference

---

This section describes the `protoDrawTool`, the canvas’ slots and methods, and the Draw application’s methods.

### Proto

---

The `protoDrawTool` is described below. Tool templates passed to `RegTool` should be based on this prototype.

### protoDrawTool

---

This proto provides the basic functionality of a tool template. You must override the `class`, `icon`, `MakeObject`, and `SetAttribute` slots.

## Newton Works Draw Application

**slot description**

<code>class</code>	Symbol that uniquely identifies the tool and the graphic objects created by this tool. This symbol should include your developer signature.
<code>icon</code>	Pix family that represents this tool in the toolbar. Note that the tool border is drawn for you—this image should supply only the interior bits.
<code>createByTapping</code>	The value <code>true</code> specifies that the user can tap to create an object. The default value is <code>nil</code> .
<code>createByDragging</code>	The value <code>true</code> specifies that the user can drag out bounds for an object. That is, the first tap specifies the <code>top</code> and <code>left</code> slots of the object's bounds, and where the pen is lifted specifies the <code>right</code> and <code>bottom</code> slots. The default value is <code>true</code> .
<code>dragARectangle</code>	The value <code>true</code> specifies that the user drags out a plain rectangle when creating an object. If the value of this slot is <code>nil</code> , the template's <code>MakeDragObject</code> method is invoked. The default value is <code>true</code> .

The methods of `protoDrawTool` are described in the following sections.

## Newton Works Draw Application

**MakeObject**


---

*protoDrawTool*:MakeObject(*left*, *top*, *right*, *bottom*, *style*)

Called to create a new graphic object when the user taps or drags in the canvas with the tool.

<i>left</i>	The x-coordinate of the start point of the rectangle the user dragged with this tool.
<i>top</i>	The y-coordinate of the start point of the rectangle the user dragged with this tool.
<i>right</i>	The x-coordinate of the end point of the rectangle the user dragged with this tool.
<i>bottom</i>	The y-coordinate of the start point of the rectangle the user dragged with this tool.
<i>style</i>	A style frame with the current user defaults for <i>penSize</i> , <i>penPattern</i> , <i>fillPattern</i> , and <i>font</i> .
return value	An array of shapes with the same class as the tool.

**DISCUSSION**

You must provide your own version of this method in your tool template.

The array returned 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 rectangle shape. The rectangle should be the size of the shape you are returning, and the style frame should be:

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

**SPECIAL CONSIDERATIONS**

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. The *left* slot in your shape’s bounds should be `Min(left, right)`, the *top* slot should be `Min(top, bottom)`, and so on.

## Newton Works Draw Application

**SetAttribute**

---

*protoDrawTool: SetAttribute(shape, attribute, newValue)*

Called when a shape is selected or draw attributes are changed.

<i>shape</i>	The shape that has been selected or altered.
<i>attribute</i>	A symbol, the name of one of the following style frame slots: selection, penSize, penPattern, fillPattern, or font.
<i>newValue</i>	The new value of the style frame slot specified in <i>attribute</i> .
return value	You can return anything; it is ignored.

**MakeDragObject**

---

*protoDrawTool: MakeDragObject(left, top, right, bottom, style)*

Called to create a shape to show while the user drags out a bounding box.

<i>left</i>	The x-coordinate of the start point of the rectangle the user dragged with this tool.
<i>top</i>	The y-coordinate of the start point of the rectangle the user dragged with this tool.
<i>right</i>	The x-coordinate of the end point of the rectangle the user dragged with this tool.
<i>bottom</i>	The y-coordinate of the start point of the rectangle the user dragged with this tool.
<i>style</i>	A style frame with the current user defaults for penSize, penPattern, fillPattern, and font.
return value	A shape array.

**DISCUSSION**

The default version of this method calls your template's `MakeObject` method. This message is only sent if the tool template's `dragARectangle` slot is `nil`.

## Newton Works Draw Application

**AdjustBounds**

---

*protoDrawTool*:AdjustBounds(*shape*, *style*, *left*, *top*, *right*, *bottom*, *constrain*)

Called to limit the bounds of the object as the user is dragging out a new bounding box or resizing an existing object.

<i>shape</i>	Either a graphic object, or <code>nil</code> if a new object is being created and <code>dragARectangle</code> is set to <code>true</code> .
<i>style</i>	The current user defaults for <code>penSize</code> , <code>penPattern</code> , <code>fillPattern</code> , and <code>font</code> .
<i>left</i>	The x-coordinate of the start point of the rectangle the user dragged with this tool.
<i>top</i>	The y-coordinate of the start point of the rectangle the user dragged with this tool.
<i>right</i>	The x-coordinate of the end point of the rectangle the user dragged with this tool.
<i>bottom</i>	The y-coordinate of the start point of the rectangle the user dragged with this tool.
<i>constrain</i>	The value <code>true</code> indicates that the shift key is pressed.
return value	A bounds frame.

**DISCUSSION**

The default version of this method limits the bounding box to always be square when *constrain* is `true`. If you override this method, you should do something similar.

**SPECIAL CONSIDERATIONS**

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. The `left` slot in bounds frame you return should be `Min(left, right)`, the `top` slot should be `Min(top, bottom)`, and so on.



## Newton Works Draw Application

**ScaleShape**

---

*protoDrawTool*:ScaleShape(*shape*, *oldBounds*, *newBounds*)

Called when user is resizing an existing object; this method actually resizes the object.

return value                      Either *shape* with the bounds altered or a new shape.

**DISCUSSION**

The default version of this function calls the global function `ScaleShape` and flips the shape with `MungeShape` as necessary.

**CanvasClickScript**

---

*protoDrawTool*:CanvasClickScript(*unit*)

Called when user taps in the canvas and your tool is selected.

*unit*                                  Stroke unit passed to this method by the recognition system. For information on stroke units, see Chapter 9, “Recognition,” in *Newton Programmer's Guide*.

return value                      Your must return one of the following values:

`true`                                  Tool has handled click completely.

`nil`                                    Tool has failed, click will turn into tap or highlight.

`'continue`                      Tool wants default shape creation behavior. That is, tapping or dragging out a shape. All of the tool methods (such as `MakeObject`, `AdjustBounds`, etc.) are called normally.

*graphicalObject*

Tool has created a graphical object to be added to the canvas. This object should be of the same type as returned by the tool's `MakeObject` method.

## Newton Works Draw Application

## DISCUSSION

This function is intended to provide additional control over the creation of shapes.

**ToolClickScript**

---

*protoDrawTool*:ToolClickScript(*unit*)

Called when user taps on this template's tool in the tool bar.

*unit*                      Stroke unit passed to this method by the recognition system. For information on stroke units, see Chapter 9, "Recognition," in *Newton Programmer's Guide*.

return value              You can return anything; it is ignored.

## DISCUSSION

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. If you override this method, call the inherited ToolClickScript method.

**ToolBegin**

---

*protoDrawTool*:ToolBegin()

Called before the ToolClickScript method.

return value              You can return anything; it is ignored.

## DISCUSSION

Your tool can use this method to perform setup tasks before creating shapes.

**ToolEnd**

---

*protoDrawTool*:ToolEnd()

Called when the user selects a tool other than the currently active one.

return value              You can return anything; it is ignored.

## Newton Works Draw Application

## DISCUSSION

Your tool can use this method to perform housekeeping tasks.

## Data Structures

---

This section describes the canvas view.

### The Canvas

---

The canvas is the view in which all drawing takes place. You can get a reference to the canvas with the Draw application's `viewDef GetCanvas` method. It has one slot of interest:

#### Slot Descriptions

`currentDrawStyle`      The current (default) drawing style.

The canvas's methods are described in the following sections.

#### AddShape

---

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

Adds *shape* to the document.

*shape*                      The shape to add.

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

`nil`                        The third argument must be `nil`.

return value              Undefined; do not rely on it.

#### AddShapeToSelection

---

`canvas:AddShapeToSelection(shape)`

Adds shape to the current selection.

*shape*                      An existing shape in the current document.

return value              Undefined; do not rely on it.

## Newton Works Draw Application

**ClearSelection**

---

*canvas*:ClearSelection()

Deselects the current selection.

return value            Undefined; do not rely on it.

**SelectAll**

---

*canvas*:SelectAll()

Selects all shapes.

return value            Undefined; do not rely on it.

**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.

return value            A shape array.

◆ **WARNING**

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

**DirtyShape**

---

*canvas*:DirtyShape(*shape*)Dirties the canvas just enough to draw the area around *shape*.

return value            Undefined; do not rely on it.

**EditGroup**

---

*canvas*:EditGroup()

Groups the current selection.

return value            Undefined; do not rely on it.

## Newton Works Draw Application

**EditUnGroup**

---

*canvas*:EditUnGroup()

Ungroups the current selection.

return value            Undefined; do not rely on it.

**EditCopy**

---

*canvas*:EditCopy()

Copies the current selection to the clipboard.

return value            Undefined; do not rely on it.

**EditCut**

---

*canvas*:EditCut()

Cuts the current selection and places it on the clipboard.

return value            Undefined; do not rely on it.

**EditPaste**

---

*canvas*:EditPaste()

Replaces the current selection with the contents of the clipboard.

return value            Undefined; do not rely on it.

**EditDelete**

---

*canvas*:EditDelete()

Deletes the current selection.

return value            Undefined; do not rely on it.

**EditDuplicate**

---

*canvas*:EditDuplicate()

Duplicates the current selection.

return value            Undefined; do not rely on it.

## Newton Works Draw Application

**EditUndo**

---

*canvas*:EditUndo()

Reverses the effects of the most recent editing operation.

return value            Undefined; do not rely on it.

**Functions and Methods**

---

This section describes methods provided by the Draw application's viewDef.

**Draw Application viewDef Methods**

---

This section describes methods provided by the Draw application's viewDef.

**GetCanvas**

---

*viewDefView*:GetCanvas()

Returns the canvas view, which contains the actual style/shape pairs in the document.

return value            The canvas view.

**GetContents**

---

*viewDefView*:GetContents()

Returns an array with all the shapes in the current document.

return value            A shape array.

**DISCUSSION**

This array is always an even length of style/shape pairs. Every shape is preceded by a style.

## Newton Works Draw Application

**SetContents**

---

*viewDefView*:SetContents(*newShapes*)

Replaces the contents of the current drawing with *newShapes*, an array of shapes.

*newShapes*                      An array containing any combination of styles and shapes.

return value                  Undefined; do not rely on it.

**RegPatterns**

---

*viewDefView*:RegPatterns(*sym*, *arrayOfPatterns*)

Registers a set of gray tones or patterns to be used with the fill settings tool.

*sym*                              A symbol uniquely identifying your set of patterns, this symbol should include your developer signature.

*arrayOfPatterns*              An array of patterns or gray tones. Possible values are the `kRGB_GrayXX` constants, RGB values (as returned by `PackRGB`), or a pattern, gray pattern, or dithered pattern. For more information on these values, see “Specifying Shades of Gray” (page 6-9) and “Using Patterns, Gray Patterns, and Dithered Patterns” (page 6-11).

return value                  Undefined; do not rely on it.

**SEE ALSO**

For an example of using this method, see Listing 2-2 (page 2-6).

**UnRegPatterns**

---

*viewDefView*:UnRegPatterns(*sym*)

Unregisters a set of patterns registered with `RegPatterns`.

*sym*                              The symbol used in the call to `RegPatterns`.

return value                  Undefined; do not rely on it.

## Newton Works Draw Application

## SEE ALSO

For an example of using this method, see Listing 2-2 (page 2-6).

**RegStamps**

---

*viewDefView*: `RegStamps(sym, arrayOfStamps)`

Registers an array of stamps to be used with in the stamp tool.

<i>sym</i>	A symbol uniquely identifying your set of patterns. This symbol should include your developer signature.
<i>arrayOfStamps</i>	An array of pix families or picture object. This array should contain no more than 24 items. For more details, see “Adding Stamps to the Stamp Tool” (page 2-7).
return value	Undefined; do not rely on it.

## SEE ALSO

For an example of using this method, see Listing 2-3 (page 2-7).

**UnRegStamps**

---

*viewDefView*: `UnRegStamps(sym)`

Unregisters a set of stamps registered with `RegStamps`.

<i>sym</i>	The symbol used in the call to <code>RegStamps</code> .
return value	Undefined; do not rely on it.

## SEE ALSO

For an example of using this method, see Listing 2-3 (page 2-7).



## Newton Works Draw Application

**RegTool**

---

*viewDefView*:RegTool(*sym*, *toolTemplate*)

Registers a set of stamps to be used by the stamp tool.

<i>sym</i>	A symbol uniquely identifying your tool, this symbol should include your developer signature.
------------	---

<i>toolTemplate</i>	A frame based on <code>protoToolTemplate</code> .
---------------------	---

return value	Undefined; do not rely on it.
--------------	-------------------------------

**SEE ALSO**

For an example of using this method, see Listing 2-1 (page 2-4).

**UnRegTool**

---

*viewDefView*:UnRegTool(*sym*)

Unregisters a set of stamps registered with RegTool.

<i>sym</i>	The symbol used in the call to RegTool.
------------	---

return value	Undefined; do not rely on it.
--------------	-------------------------------

**SEE ALSO**

For an example of using this method, see Listing 2-1 (page 2-4).

## Summary

---

### Proto

---

#### protoDrawTool

---

```
myToolTemplate := {
  _proto : protoDrawTool,
  class : classSymbol, // this tool's symbol
  icon : pixFamily, //icon for tool bar
  createByTapping : Boolean , //create object with pen tap?
  createByDragging : Boolean , //create object by dragging bounds?
  dragARectangle: Boolean , //create object by dragging a rectangle?
  MakeObject: //make a new object
    func(left, top, right, bottom, style) ...
  SetAttribute: //set an attribute
    func(shape, attribute, newValue) ...
  MakeDragObject: // make an object to show user when dragging a new obj.
    func(left, top, right, bottom, style) ...
  AdjustBounds: // constrain the bounding rect a user drags
    func(shape, style, left, top, right, bottom, constrain) ...
  ScaleShape: // scale an object
    func(shape, oldBounds, newBounds) ...
  CanvasClickScript : // called when user touches canvas w/your tool
    func(unit)...
  ToolClickScript : // called when user taps your tool in the tool bar
    func(unit) ...
  ToolBegin : // called before ToolClickScript to set up too
    func() ...
  ToolEnd : // called when another tool is selected, to clean up
    func() ...
  ...
}
```

## Newton Works Draw Application

## Data Structures

---

### The Canvas

---

```

canvasView := {
currentDrawStyle :
AddShape : func(shape, style, nil) ... //add a shape to the canvas
AddShapeToSelection : func(shape) ... //selects a shape
ClearSelection : func() ... //clears selected shapes
SelectAll : func() ... // selects all shapes on the canvas
GetSelectedShapes : func(makeCopy) ... //returns selected shapes
DirtyShape : func(shape) ... //marks as dirty area under a shape
EditGroup : func() ... //groups selection
EditUnGroup : func() ... //ungroups selection
EditCopy : func() ... //copies selection
EditCut : func() ... //cuts selection
EditPaste : func() ... //pastes clipboard contents
EditDelete : func() ... //deletes selection
EditDuplicate : func() ... //duplicates selection
EditUndo : func() ... //undos last undoable action
...
}

```

### Draw Application viewDef

---

```

drawApplicationViewDef := {
GetCanvas : func() ... //returns canvas view
GetContents : func() ... //returns contents of canvas
SetContents : func(newShapes) ... //sets contents of canvas
RegPatterns : func(sym, arrayOfPatterns) ... //registers new patterns
UnRegPatterns : func(sym) ... //unregisters new patterns
RegStamps : func(sym, arrayOfStamps) ... //registers new stamps
UnRegStamps : func(sym) ... //unregisters new stamps
RegTool : func(sym, toolTemplate) ... //registers new tool
UnRegTool : func(sym) ... //unregisters new tool
...
}

```

Newton Works Draw Application

# Word Processing Views

---

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 Newton Works 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`.

## Application-defined Methods

The `protoTXView` proto supports some, but not all, of the standard Newton view system application-defined methods. Table 3-1 shows the status of application-defined methods with `protoTXView`.

**Table 3-1** Use of application-defined methods in `protoTXView`

Method	Use in <code>protoTXView</code>
<code>ViewGestureScript</code>	Supported.
<code>ViewClickScript</code>	Supported.
<code>ViewStrokeScript</code>	Supported.
<code>ViewWordScript</code>	Supported.
<code>ViewKeyDownScript</code>	Called if you specify <code>vSingleKeystrokes</code> in the <code>textFlags</code> slot.
<code>ViewKeyUpScript</code>	Called if you specify <code>vSingleKeystrokes</code> in the <code>textFlags</code> slot.
<code>ViewScrollUpScript</code>	Use <code>protoTXView:Scroll</code> for scrolling. Note, however, that the view system does send this message to <code>protoTXView</code> .
<code>ViewScrollDownScript</code>	Use <code>protoTXView:Scroll</code> for scrolling. Note, however, that the view system does send this message to <code>protoTXView</code> .
<code>ViewOverviewScript</code>	Not supported.
<code>ViewGetDropTypesScript</code>	Supported.
<code>ViewGetDropDataScript</code>	Supported.
<code>ViewHiliteScript</code>	Not supported. Use the <code>protoTXView</code> <code>hilite</code> methods instead. See “Highlighting Methods” (page 3-40).

Word Processing Views

# View Slots

Some of the standard view slots are used differently for `protoTXView`, as shown in Table 3-2.

**Table 3-2** Use of standard view system slots in `protoTXView`

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

# Other View Features

This section describes several characteristics of `protoTXView`.

- No child views are allowed within `protoTXView`. 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 `editView`, `protoTXView` supports the movement of 'text', 'ink', 'picture and 'polygon content; in addition, `protoTXView` also supports 'shape typesl.

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

## Word Processing Views

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`.

## About Paged and Non-paged Word-Processing Views

---

You can use either paged or non-paged views with `protoTXView`. The text in a paged view is laid out in many pages, with text flowing from one page to another; pages are shown separated by a dotted line. The text in a non-paged view is all contained in one box. You specify whether a word-processing view is paged or non-paged when you set up its characteristics in the



Word Processing Views

`protoTXView:SetGeometry` method. Paged versus non-paged views have a few other implications, as shown in Table 3-3.

**Table 3-3** Paged versus non-paged views

Issue	Paged views	Non-paged views
Text layout	Text flows from one page to another, as required.	All text is contained within one bounding box.
Text height computation	Text height is automatically adjusted as you add or remove text.	<p>Text height is not automatically adjusted, which means that newly added text might not appear in the text region.</p> <p>As a result, you might want to define the text height as an arbitrarily large number in your call to <code>SetGeometry</code>.</p> <p>Note that this is only an issue for read-write views.</p>
Scrolling	Default scrolling behavior is to scroll to the bottom of a page, even if the text does not fill the entire page.	<p>For proper scrolling, you need to write a function to determine the actual text height in your view.</p> <p>Note that this is related to text height computation (see the previous table entry) and is only an issue for read-write views.</p> <p>See Listing 3-5 for an example.</p>

**Note**

The sample code in this chapter for the TXWord program uses a paged word-processing view. ♦

## About Scrolling with protoTXView

---

Your word processing views automatically scroll when the user is entering text with the keyboard or pen. `protoTXView` also provides a number of scrolling methods that you can use if you have attached scrollers (vertical or horizontal) to the view that encloses your `protoTXView`.

Table 3-4 summarizes the `protoTXView` scrolling methods, which are described in more detail in “Scrolling Methods” (page 3-38).

**Table 3-4**      Scrolling methods of `protoTXView`

---

Method	Description
<code>Scroll</code>	Scrolls the contents of the word-processing view horizontally and/or vertically.
<code>GetScrollValues</code>	Returns the current scroll values, which you can use to adjust the thumbbars of attached scrollers
<code>GetTotalHeight</code>	Returns the current total text height, which you can use to set the maximum value of a vertical scroller.
<code>GetTotalWidth</code>	Returns the total text (or page) width, which you can use to set the maximum value of a horizontal scroller.
<code>GetScrollableRect</code>	Returns a frame describing the coordinates of the text display rectangle.
<code>ViewUpdateScrollersScript</code>	Called by <code>protoTXView</code> when the text is scrolled or when the text height changes. Used to notify you that any attached scrollers might need updating.

## About Storing protoTXView Documents

---

You can store `protoTXView` word-processing documents in RAM-based soup objects, or you can store a word-processing document as a virtual binary

## Word Processing Views

object (VBO). The VBO is a “black-box” representation of the data in your view. You can perform a limited set of operations on these objects, as follows:

- You can use the `protoTXView:Externalize` method to save the contents of a word-processing view to a VBO.
- You can use the `protoTXView:Internalize` method to replace the contents of a word-processing view with the data stored in a VBO that was previously stored with a call to `protoTXView:Externalize`.
- You can call `protoTXView:IsModified` to determine if your view has changed since the most recent call to `protoTXView:Internalize` or `protoTXView:Externalize`.
- You can use `protoTXViewFinder` methods to search for text in a VBO without creating a word-processing view.

## Using protoTXViewFinder to Search Documents

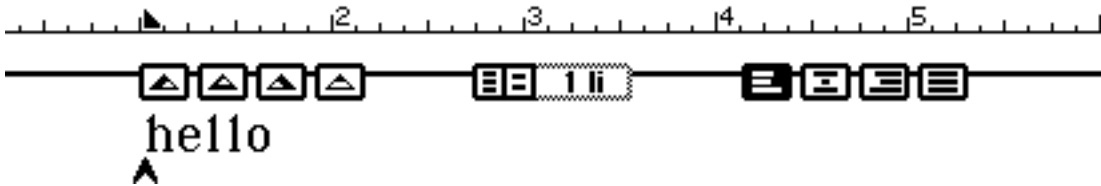
---

The Newton System Software provides a second proto that you can use with word-processing views: `protoTXViewFinder`. This proto allows you to search for text in a word-processing document without incurring the overhead of creating a word-processing view. For more information see “protoTXViewFinder” (page 3-45).

## Word-Processing View User Interface

---

Word-processing views can display the ruler, which shows the user the current formatting settings. When the ruler is displayed, it always appears at the top of the view, as shown in Figure 3-1.

**Figure 3-1** The displayed ruler

Note that the ruler does use screen space, which means that it reduces the size of the scrollable rectangle in the view. The size adjustments are made automatically when you display or hide the ruler with the `protoTXView:ShowRuler` and `protoTXView:HideRuler` methods.

## 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.

## Using Word Processing Views

This section describes how to use word-processing views in your applications. The code examples used in this section are taken from a sample program named `TXWord`, which demonstrates the basic features of `protoTXView`. The full source code for `TXWord` is available from Newton Developer Technical Support.

## Word Processing Views

## Initializing Your Word-Processing View

---

You can initialize your word-processing view in the `ViewSetupFormScript` method. The `TXWord` implementation of this method calls the inherited `ViewSetupFormScript` method and then initializes the store, as shown in Listing 3-1.

---

**Listing 3-1**     Initializing a word-processing view

```
ViewSetupFormScript:
func()
    begin
        inherited:?ViewSetupFormScript();
        // store data in a VBO, not in NewtonScript memory
        :SetStore(GetDefaultStore());
        // cache box top value for use in GetTextHeight method
        self.ourGlobalBoxTopCoordinate := :GlobalBox().top;

        constant kMaxScrollHeight := 32767;
        :SetGeometry(true, :localbox().right, kMaxScrollHeight,
                        Relbounds(0,0,0,0));
    end
```

The `protoTXView:SetStore` method, which is called in the `TXWord` program's `ViewSetupFormScript` method, specifies which store is used to contain the text edited in your word-processing view. This is normally the same as your soup entry's store. You can (as `TXWord` does) call the `GetDefaultStore` function to retrieve the default store and use that.

The `TXWord ViewSetupFormScript` method sets the scroll height to a very large number (32767), which ensures that the user can see all of the text. Note that this is used for a non-paged view; if you specify a paged view in your `SetGeometry` call (true as the value of the first argument), you must specify the actual size of an individual page as the scroll height value.

Although `TXWord` does not use margins, it still must provide a margins rectangle frame as the last parameter in the call to `SetGeometry`. The call to `Relbounds` returns a rectangle frame.

## Word Processing Views

**Note**

You can call the `SetStore` method of your word-processing view only once, and you must call it while the view is being created (before the `ViewSetupDoneScript` method is called). If you do not call `SetStore`, your text defaults to being RAM based. ♦

## Setting Up Your Word-Processing View

---

You can initialize your word-processing view with the current document by reading the document data from your soup after the view has been completely set up. The `TXWord` program initializes its view in the `ViewSetupDoneScript` method, as shown in Listing 3-2.

---

**Listing 3-2**     Setting up a word-processing view

```
ViewSetupDoneScript:
func()
begin
    :SetupScrollers();

    :GetSoupData();
    SetKeyView(self, 0);
    inherited:?ViewSetupDoneScript();
end
```

The `TXWord` implementation of `ViewSetupDoneScript` calls the following four methods:

- the `editor:SetupScrollers` method to set up the scrollers. This method, which you must call if you have scrollers, is shown in Listing 3-3.
- the `editor:GetSoupData` method to read the document most recently written by `TXWord` from the current store, and then display that document. The `GetSoupData` method is described in “Reading a Word-Processing Document From a Soup” (page 3-14).
- the `SetKeyView` function to establish the word-processing view as the key view, which means that the view will receive keystrokes from the user. For more information about key views and the `SetKeyView` function, see Chapter 4, “Keyboard Enhancements.”

## Word Processing Views

- the inherited `ViewSetupDoneScript` method.

## Scrolling the Word-processing View

---

This section describes how to handle scrolling in your word-processing views. You must first set up your scrollers. The TXWord program uses the `SetupScrollers` method for this purpose. This method, which is shown in Listing 3-3, is called after the editor view and scroll arrows have been created.

---

### Listing 3-3 The `SetScrollers` method

```
SetupScrollers:
    func()
    begin
        scroller.scrollview    := editor;
        scroller.viewRect      := editor:GetScrollableRect();

        // note: GetScrollableRect returns the scroll area in
        // global coordinate values, omitting the ruler area
        local visualArea      := :GetScrollableRect();

        scroller.viewRect      := clone(:LocalBox());
        scroller.viewRect.bottom:= visualArea.bottom - visualArea.top;
        scroller.scrollRec     := RelBounds(0,0,
                                           scroller.viewRect.right, :GetTotalHeight());
    end,
```

The `protoTXView:ViewUpdateScrollersScript` method is called when something happens that might affect the the scroll arrows; for example, if more text is added, which causes the maximum height or thumb position to change. The TXWord version of this method is shown in Listing 3-4.

---

### Listing 3-4 The TXWord `ViewUpdateScrollersScript` method

```
ViewUpdateScrollersScript:
    func(updateMaxValue, scrolled)
    begin
        if updateMaxValue then
            begin
```

## Word Processing Views

```

        // use editor's GetTextHeight method to get the height
        scroller.scrollRect.bottom := :GetTextHeight();
        scroller.dataRect := scroller.scrollRect;
    end;

    if scrolled then
        scroller.yPos := :GetScrollValues().y;

    if call kViewIsOpenFunc with (scroller) then
        scroller:AdjustArrows();
    end,

```

The `TXWord` implementation of `ViewUpdateScrollersScript` calls a local method, `GetTextHeight`, to return the actual text height in the view. Although `TXWord` uses a non-paged word-processing view, the `GetTextHeight` method is included to illustrate how you would implement scrolling in a non-paged word-processing view.

You need to use a method such as `GetTextHeight` if you are implementing a read-write, non-paged view.

**Note**

Recall that when you initialize a read-write, non-paged view, you need to specify the text height as your view height in your call to the `SetGeometry` method. ♦

When the user adds text to such a view, the new text can appear outside of the view's text region. To get around this, you typically create the view with an arbitrarily large height and write a function such as `GetTextHeight` to return the actual text height. The `TXWord:GetTextHeight` method is shown in Listing 3-5.

**Note**

You do not need to be concerned with this situation if your word-processing view is a read-only view, because the text height never changes in such a view. ♦

The `GetTextHeight` method returns the actual text height in the view. This method is not required if your word-processing view is paged, because paged views automatically adjust their height as the user adds text.



## Word Processing Views

**Listing 3-5** The TXWord `GetTextHeight` method

---

```

GetTextHeight:
    func()
    begin
        local chars      := :GetCountCharacters();
        local maxInfo     := :CharToPoint(chars);

        local globalY     := maxInfo.y + maxInfo.lineHeight;

        // take the scroll position into consideration...
        globalY           := globalY + :GetScrollValues().y;
        return globalY - ourGlobalBoxTopCoordinate;
    end,

```

The `GetTextHeight` method uses an application variable, `ourGlobalBoxTopCoordinate`, in its computation of the text height. This value is set in the `ViewSetupFormScript` method.

The final scrolling method, `ViewScroll2DScript`, method is called when the user taps the scroll arrows. You only need to implement this method if you are using the `protoXXXScroller` methods. The TXWord version of this method is shown in Listing 3-6.

**Listing 3-6** The TXWord `ViewScroll2DScript` method

---

```

ViewScroll2DScript:
    func(direction, extras)
    begin
        if direction = 'up then
            :Scroll({x:0, y: - kScrollDist});
        else
            // direction = 'down
            :Scroll({x:0, y: kScrollDist });

        scroller.yPos := :GetScrollValues().y;
        if call kViewIsOpenFunc with (scroller) then
            scroller:AdjustArrows(); // this is a protoXXXScroller method

        RefreshViews();
    end,

```

The TXWord implementation of `ViewScroll2DScript` scrolls the view up or down and then calls the `RefreshViews` function to redraw the view.

## Word Processing Views

**Note**

The call to the `RefreshViews` function in Listing 3-6 is required for the scroller to update. The `protoTXView` automatically updates whenever it receives a scroll message.

## Reading a Word-Processing Document From a Soup

---

To read a word-processing document that you have previously created in your application, you can access the document in a soup and then call the `protoTXView:Internalize` method, which replaces the view's contents with the document data. The TXWord program implements this in the `editor:GetSoupData` method, which is shown in Listing 3-7.

---

**Listing 3-7**      Reading a document from a soup

```
editor:GetSoupData:
func()
    begin
        local soup := GetUnionSoup(kSoupName);
        if soup then
            begin
                local query := soup:Query(nil);
                // set our slot with the entry, if there
                theSoupEntry := query:entry();

                if theSoupEntry then
                    self:Internalize(theSoupEntry.EditorData);
                end;
            end
        end
    end
```

The TXWord program assumes that there should only be one word-processing entry in the soup. During program debugging, the `editor:GetSoupData` method verifies this by checking for duplicate entries; if a duplicate entry is found, `editor:GetSoupData` generates an exception.

## Word Processing Views

**Note**

The TXWord model of reading a document from a soup is not necessarily the correct approach for your application. It works for TXWord’s relatively simple style of managing its documents. ♦

Note that the `editor:GetSoupData` method stores a reference to the soup entry in the slot `editor:theSoupEntry`. This slot is initialized to `nil` when the TXWord program starts and is also used by the `editor:PutSoupData` method.

## Storing Documents In a Soup

---

After the user has finished modifying a word-processing document, you need to save it in your soup. The TXWord program does this at view closing time, in its implementation of the `ViewQuitScript` method, which is shown in Listing 3-8.

---

**Listing 3-8** Closing the word-processing view

```
ViewQuitScript:
func()
    begin
        :SetSoupData();

        inherited:?ViewQuitScript();
    end
```

The TXWord implementation of the `ViewQuitScript` method calls the `editor:SetSoupData` method to store the modified data and then calls the inherited `ViewQuitScript` method. The `editor:SetSoupData` method is shown in Listing 3-9.

---

**Listing 3-9** Storing a word-processing document

```
editor:SetSoupData:
func()
    begin
        // only save if something changed
```

## Word Processing Views

```

    if not :IsModified() then
        return;

    local externalData := self:Externalize();

        // change existing soup entry if there is one
    if theSoupEntry then
        begin
            theSoupEntry.EditorData := externalData;
            EntryChangeXmit(theSoupEntry, nil);
        end;
    else
        begin
            local soup := GetUnionSoupAlways(kSoupName);
            if soup then
                theSoupEntry := soup:AddToDefaultStoreXmit(
                    {EditorData: externalData}, nil);
            else
                begin
                    soup := GetDefaultStore():CreateSoup(kSoupName, []);
                    theSoupEntry := soup:AddXmit(
                        {EditorData: externalData}, nil);
                end;
            end;
        end;
    end
end

```

The `editor:SetSoupData` method first determines if the view has been modified by the user by calling the `protoTXView:IsModified` method. If not, `SetSoupData` simply returns, rather than spending time saving unchanged data.

If the view has been modified, `SetSoupData` calls the `protoTXView:Externalize` method to convert the word-processing data in the view into a format that can be stored in your soup. `SetSoupData` then updates the soup as follows:

- If there already is a soup entry for the view, `SetSoupData` updates that entry. `SetSoupData` knows that a soup entry already exists if `theSoupEntry` slot has a value. This is the case if the `GetSoupData` method found a soup entry at view setup time, as described in “Reading a Word-Processing Document From a Soup” (page 3-14).
- If there is not already a soup entry for the view, `SetSoupData` creates the soup in the default store and adds the data to the soup.

## Handling User Interactions

The TXWord program allows the user to perform several different operations by picking buttons from a button bar. The supported user actions are:

- changing the font used in the word-processing view
- changing the font size used in the word-processing view
- capitalizing the selected text range
- censoring the selected text range by replacing the text with a graphic image

Figure 3-2 shows the TXWord button bar. For more information about button bars, see the `newtStatusBar` proto in *Newton Programmer's Reference*. The remainder of this section describes the code used to implement each button bar action in the TXWord program.

**Figure 3-2** The TXWord button bar



## Changing the Font

The TXWord program allows the user to replace the font used to display the selected range of text. This is handled by the `ButtonClickScript` and `PickActionScript` methods of the “Font” button displayed in the TXWord button bar.

**Listing 3-10** Changing the font in TXWord

```
{_proto: protoPopupButton,
  text: "  Font",
  ButtonClickScript: func()
begin
  self.font := editor:GetContinuousRun();
                // display only fonts
```

## Word Processing Views

```

popup := MakeFontMenu(font, nil, 'none, 'none);
inherited:?ButtonClickScript();
end,

PickActionScript: func(index)
begin
local range := editor:GetHiliteRange();
editor:ChangeRangeRuns(range,
                        {family: popup[index].family}, nil, true);
inherited:?PickActionScript(index);
end;
},

```

The `ButtonClickScript` for the “Font” button gets the style run for the current selection and displays a font menu with the style run’s font as its initial value. When the user selects a font from the menu, the `PickActionScript` changes the font attribute of the currently selected text range.

## Changing the Font Size

---

The `TXWord` program allows the user to replace the font size used to display the selected range of text. This is handled by the `ButtonClickScript` and `PickActionScript` methods of the “Size” button displayed in the `TXWord` button bar.

---

### Listing 3-11 Changing the font size in `TXWord`

```

{ _proto: protoTextButton,
  text: "    Size",
  ButtonClickScript: func()
begin
  self.font := editor:GetContinuousRun();
                // display only sizes
  popup := MakeFontMenu(font, 'none, nil, 'none);
  inherited:?ButtonClickScript();
end,

  PickActionScript: func(index)
begin
  local range := editor:GetHiliteRange();
  editor:ChangeRangeRuns(range,
                        {size: popup[index].size}, nil, true );

```

## Word Processing Views

```

        inherited:?PickActionScript(index);
    end;
}

```

The `ButtonClickScript` for the “Size” button gets the style run for the current selection and displays a font size menu with the style run’s font size as its initial value. When the user selects a size from the menu, the `PickActionScript` changes the font attribute of the currently selected text range.

## Replacing the Selected Text With a Graphic

---

The `TXWord` program allows the user to replace the selected text with a shape that indicates the text is censored. The graphic used to indicate that text is censored is stored in a shape object. The `Censor` method, which is shown in Listing 3-12, is called by the `ButtonClickScript` of the “Censor” button when the user taps the button.

---

### Listing 3-12 Replacing the selected text

```

Censor:
func()
    begin
        local graphicSpecForTXView := {
            class: 'graphics',
            shape: MakeShape(kJustSayNoPict)
        };

        local range := editor:GetHiliteRange();

        editor:Replace(range, graphicSpecForTXView, true);
    end,

```

The `Censor` function uses two of the `protoTXView` methods to replace the selected text with a shape:

- it calls the `GetHiliteRange` method to retrieve the currently selected text range
- it calls the `Replace` method to replace that range with a shape. The shape, `graphicSpecForTXView`, is a graphic shape that `TXWord` creates from a picture that it reads from its resources at program initialization time.

## Converting the Selected Text to Uppercase

---

The TXWord program allows the user to convert the selected text into all uppercase letters. This is handled by the `ButtonClickScript` method of the “UpperCase” button displayed in the TXWord button bar.

---

### Listing 3-13 Converting the selected text to uppercase

```
{_proto: protoTextButton,
  text: "UpperCase",
  ButtonClickScript: func()
    begin
      local range := editor:GetHiliteRange();
      local theText := editor:GetRangeData(range, 'text');
      local theStyles:= editor:GetRangeData(range, 'styles');
      Uppcase(theText);

      // if we left the styles slot nil, the styles would be reset
      // to match the style at the beginning of the run.
      editor:Replace(range, {text: theText, styles: theStyles}, true);
      // Ensure the new text is visible and selected
      editor:SetHiliteRange(range, true, true);
      inherited:?ButtonClickScript();
    end,
}
```

The “UpperCase” button’s `ButtonClickScript` converts the text using the following steps:

- It retrieves the range and styles for the selected text.
- It applies the `Uppcase` function to the text range.
- It resets the styles slot of the selected range.
- It calls the `SetHiliteRange` method to ensure that the new text is visible and selected.

## Adding a Recognized Word to Your Word-Processing View

---

Although `protoTXView` does not handle handwriting recognition, you can use the `ViewWordScript` to recognize a word and add it to your view. The TXWord implementation of this method is shown in Listing 3-14.



## Word Processing Views

**Listing 3-14** Adding a recognized word to a word-processing view

```

ViewWordScript:
func(unit)
begin
    local words := GetWordArray(unit);
    local range := editor:GetHiliteRange();

    // determine if we need to add a space before or after the word
    local first := range.first - 1;
    local last := range.last + 1;
    local totalChars := :GetCountCharacters();
    local textToAdd := "";

    if first >= 0 and totalChars > 0 then
        if :GetRangeData({first: first, last: first + 1}, 'text')[0]
            <> $ then
            textToAdd := textToAdd & " "; // insert space before word

        // add first (most likely) word returned by recognition
        textToAdd := textToAdd & words[0];

    if last < totalChars then
        if :GetRangeData({first: last, last: last + 1}, 'text')[0]
            <> $ then
            textToAdd := textToAdd & " "; // add space after word

    editor:Replace(range, {text: textToAdd}, true);

    true; // Return true if input has been completely handled
end,

```

The TXWord implementation of `ViewWordScript` calls the global function `GetWordArray` to recognize the input word (`unit`) and then adds the first word in the returned array—the most likely match—to the word-processing view.

**Note**

`protoTXView` does not provide the same recognition hooks as does `clParagraphView`. For example, the user does not get the correction picker if he or she double-taps on a word. ♦

## Word Processing View Reference

---

This section provides reference information for all of the data structures, methods, and functions that you can use with word-processing views.

### 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 non-negative integer value.

#### 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 “Drawing and Graphics” (page 13-1) in *Newton Programmer's Guide*.

## Word Processing Views

## The Ruler Information Frame

---

Several of the word-processing methods use a ruler information parameter to specify style information for a paragraph.

<i>rulerInfo</i>	A frame describing the attributes of a ruler. This frame contains the following slots:
<i>justification</i>	Optional. One of the symbols: 'left, 'right, 'center or 'full.
<i>indent</i>	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.
<i>leftMargin</i>	Optional. The left margin of the paragraph, expressed as an integer number of pixels measured from the left edge of the text bounds.
<i>rightMargin</i>	Optional. The right margin of the paragraph, expressed as an integer number of pixels measured from the right edge of the text bounds.
<i>lineSpacing</i>	Optional. The spacing for text lines in the paragraph, expressed as an integer number of text lines. A value of 2 indicates double spacing.
<i>tabs</i>	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:

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

## Word Processing Views

## Protos

---

This section describes the slots, and functions of the `protoTXView` and `protoTXViewFinder` word-processing protos.

### protoTXView

---

This section describes the slots and methods of `protoTXView`.

#### Initialization Methods

---

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). You only need to call this method if you want your view stored as a VBO; the default is to store the text as a RAM-based object.

<i>store</i>	The store object to contain the text. This is usually the same as your soup entry's store.
return value	An error code, or <code>nil</code> if the operation was successful.

#### DISCUSSION

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. ♦

## Word Processing Views

**SetGeometry**


---

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

Changes the geometrical characteristics of the view.

*isPaged*

Specifies whether the text is laid out in many pages, with text flowing from one page to another (with dotted lines visible between pages), 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 pasted into a paged view does cause a page break.

*width*

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*

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

## Word Processing Views

	margins. The default value is the height of the view (as specified in the <code>viewBounds</code> rectangle).								
<i>margins</i>	A rectangle that specifies the margins of the page or text box. The rectangle is specified as a frame with four slots: <table> <tr> <td><i>top</i></td><td>The indent from the top edge of the view rectangle, expressed as a number of pixels.</td></tr> <tr> <td><i>left</i></td><td>The indent from the left edge of the view rectangle, expressed as a number of pixels.</td></tr> <tr> <td><i>bottom</i></td><td>The indent from the bottom edge of the view rectangle, expressed as a number of pixels.</td></tr> <tr> <td><i>right</i></td><td>The indent from the right edge of the view rectangle, expressed as a number of pixels.</td></tr> </table> <p>To specify no margins at all, use a rectangle in which all four values are 0.</p>	<i>top</i>	The indent from the top edge of the view rectangle, expressed as a number of pixels.	<i>left</i>	The indent from the left edge of the view rectangle, expressed as a number of pixels.	<i>bottom</i>	The indent from the bottom edge of the view rectangle, expressed as a number of pixels.	<i>right</i>	The indent from the right edge of the view rectangle, expressed as a number of pixels.
<i>top</i>	The indent from the top edge of the view rectangle, expressed as a number of pixels.								
<i>left</i>	The indent from the left edge of the view rectangle, expressed as a number of pixels.								
<i>bottom</i>	The indent from the bottom edge of the view rectangle, expressed as a number of pixels.								
<i>right</i>	The indent from the right edge of the view rectangle, expressed as a number of pixels.								
return value	An error code, or <code>nil</code> if the operation worked.								

**DISCUSSION**

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. ♦

**Methods for Getting Information**

---

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

Word Processing Views

**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:

Symbol	Returned data
'text	A string allocated from the NewtonScript 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 font specification frame or a graphics specification frame. The run length is always 1 for graphics specification frames.  For a description of the font specification frame, see the section "Font Frame" (8-18) in <i>Newton Programmer's Guide</i> . See "The Graphics Specification Frame" (page 3-22) for a description of the graphics specification frame.
'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 ruler information frame.

See "The Ruler Information Frame"

Word Processing Views

	(page 3-23) for a description of the ruler information frame.
'all	Returns all information in a frame that contains three slots: 'text, 'styles, and 'rulers.
return value	A frame or array containing the requested data.

DISCUSSION

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

**GetCountCharacters**

---

*protoTXView*:GetCountCharacters()

return value	The integer number of characters in the specified word-processing view.
--------------	---

DISCUSSION

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

This cannot be called before `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.

<i>str</i>	The string to be searched.
<i>startOffset</i>	The offset at which the search should start.
<i>options</i>	Must be <code>nil</code> . Currently the search is not case sensitive.
return value	The offset of the matching string in the view. If no match is found, <code>FindString</code> returns <code>nil</code> .



## Word Processing Views

## DISCUSSION

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 is made or the end of the text is reached.

Note that `FindString` does not offer wrap-around searching of the text in your view.

**GetWordRange**

---

*protoTXView*:`GetWordRange(offset)`

Finds the first and last characters of a word.

<i>offset</i>	The offset from the beginning of the text in the word-processing view.
return value	A range frame that specifies the starting and ending offsets of the word that follows the specified <i>offset</i> in a word-processing view. The <code>GetWordRange</code> method returns a range frame, as described in “The Range Frame” (page 3-22). The <code>GetWordRange</code> method returns <code>nil</code> if it does not find a word after <i>offset</i> .

## DISCUSSION

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.

**CharToPoint**

---

*protoTXView*:`CharToPoint(offset)`

Returns a frame that specifies the coordinates of a character in a word-processing view.

<i>offset</i>	The offset from the beginning of the text in the word-processing view. This offset specifies the caret location: a value of 0 indicates before the first character,
---------------	---

Word Processing Views

	and a value of 1 indicates between the first and second characters.	
return value	The <code>CharToPoint</code> method returns a frame with three slots:	
	<i>x</i>	The horizontal coordinate of the top-left corner of the rectangle enclosing the character at <i>offset</i> .
	<i>y</i>	The vertical coordinate of the top-left corner of the rectangle enclosing the character at <i>offset</i> .
	<i>lineHeight</i>	The line height of the line that contains the character.

DISCUSSION

The value of each of the slots in the returned frame 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.

<i>point</i>	A frame containing two slots:	
	<i>x</i>	The horizontal coordinate, as a global coordinate value, relative to the top-left of the screen.
	<i>y</i>	The vertical coordinate, as a global coordinate value, relative to the top-left of the screen.
return value	The <code>PointToChar</code> method returns a range frame for the character at the specified point. Range frames are described in “The Range Frame” (page 3-22).	

DISCUSSION

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

## Word Processing Views

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

**GetLineRange**

---

*protoTXView*:GetLineRange(*offset*)

Returns a range frame for the text line in the view that contains the specified *offset*.

<i>offset</i>	The offset from the beginning of the text in the word-processing view. This offset specifies the caret location: a value of 0 indicates before the first character, and a value of 1 indicates between the first and second characters.
---------------	---

return value	A range frame corresponding to the text line that contains the specified <i>offset</i> .
--------------	--

**DISCUSSION**

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 break characters are treated the same as carriage returns.

**Editing Functions and Methods**

---

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.

## Word Processing Views

**Cut**

---

*protoTXView*:Cut()

Removes the highlighted range and copies it to the clipboard.

return value            An error code, or *nil* if the operation worked.**Copy**

---

*protoTXView*:Copy()

Copies the highlighted range to the clipboard.

return value            An error code, or *nil* if the operation worked.**Paste**

---

*protoTXView*:Paste()

Replaces the highlighted range with the clipboard content.

return value            An error code, or *nil* if the operation worked.**Clear**

---

*protoTXView*:Clear()

Removes the highlighted range. The clipboard is not changed.

return value            An error code, or *nil* if the operation worked.**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 3-22).*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. For a description of the font specification frame, see the

## Word Processing Views

	section “Font Frame” (8-18) in <i>Newton Programmer’s Guide</i> .
<i>toggleFace</i>	<p>A Boolean value that specifies the font face attribute to use for all text in the range.</p> <p>If <i>toggleFace</i> is <code>nil</code>, the font face is changed to the value of the <code>face</code> slot in <i>fontSpec</i>. If that value is <code>nil</code>, the font face is not change.</p> <p>If <i>toggleFace</i> is non-<code>nil</code>, the font face is toggled: if one of the font face values specified in the <code>face</code> slot in <i>fontSpec</i> is used across the entire range, <i>ChangeRangeRuns</i> turns off that attribute. Otherwise, that attribute is applied to all of the text in <i>range</i>.</p>
<i>undoable</i>	If the value of this slot is non- <code>nil</code> , the operation can be undone. If the value of this slot is <code>nil</code> , the operation cannot be undone.
return value	An error code, or <code>nil</code> if the operation worked.

**ChangeRangeRulers**


---

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

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

<i>range</i>	The rulers in this range are changed. Note that the range is grown to enclose entire paragraphs.
<i>ruler</i>	A ruler information frame. Any non- <code>nil</code> slots in this frame indicate new ruler attributes for the <i>range</i> . Use <code>nil</code> slots to indicate that the corresponding attribute is not to change. See “The Ruler Information Frame” (page 3-23).
<i>undoable</i>	If non- <code>nil</code> , the operation is undoable.
return value	An error code, or <code>nil</code> if the operation worked.

## Word Processing Views

**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.

<i>range</i>	A range frame defining the text range that you want to change. See “The Range Frame” (page 3-22).				
<i>data</i>	<p>A frame describing the new data. This can be a graphics specification frame, as described in “The Graphics Specification Frame” (page 3-22). Or <i>data</i> can be a frame with the following slots:</p> <table> <tr> <td><i>text</i></td><td>If this slot is <i>nil</i>, style runs are replaced, but the text remains the same. If non-<i>nil</i>, this is the new text string.</td></tr> <tr> <td><i>styles</i></td><td> <p>If this slot is <i>nil</i>, the new text uses the style attributes at the start of the <i>range</i>.</p> <p>If non-<i>nil</i>, 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 font specification frame or a graphics specification frame. The run length is always 1 for graphics specification objects.</p> <p>For a description of the font specification frame, see the section “Font Frame” (8-18) in <i>Newton Programmer’s Guide</i>. See “The Graphics Specification Frame” (page 3-22) for a description of the graphics specification frame.</p> </td></tr> </table>	<i>text</i>	If this slot is <i>nil</i> , style runs are replaced, but the text remains the same. If non- <i>nil</i> , this is the new text string.	<i>styles</i>	<p>If this slot is <i>nil</i>, the new text uses the style attributes at the start of the <i>range</i>.</p> <p>If non-<i>nil</i>, 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 font specification frame or a graphics specification frame. The run length is always 1 for graphics specification objects.</p> <p>For a description of the font specification frame, see the section “Font Frame” (8-18) in <i>Newton Programmer’s Guide</i>. See “The Graphics Specification Frame” (page 3-22) for a description of the graphics specification frame.</p>
<i>text</i>	If this slot is <i>nil</i> , style runs are replaced, but the text remains the same. If non- <i>nil</i> , this is the new text string.				
<i>styles</i>	<p>If this slot is <i>nil</i>, the new text uses the style attributes at the start of the <i>range</i>.</p> <p>If non-<i>nil</i>, 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 font specification frame or a graphics specification frame. The run length is always 1 for graphics specification objects.</p> <p>For a description of the font specification frame, see the section “Font Frame” (8-18) in <i>Newton Programmer’s Guide</i>. See “The Graphics Specification Frame” (page 3-22) for a description of the graphics specification frame.</p>				
<i>undoable</i>	If non- <i>nil</i> , the operation is undoable.				
return value	An error code, or <i>nil</i> if the operation worked.				

## Word Processing Views

## EXAMPLE

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 offset, and replaces all occurrences of a string with other data.

<i>str</i>	The string to be replaced				
<i>startOffset</i>	The starting offset of the search in the text, specified as a number of characters.				
<i>options</i>	Must be <code>nil</code> .				
<i>data</i>	A frame describing the new data. This can be a graphics specification frame, as described in “The Graphics Specification Frame” (page 3-22). Or <i>data</i> can be a frame with the following slots: <table> <tr> <td><i>text</i></td><td>If this slot is <code>nil</code>, style runs are replaced, but the text remains the same. If non-<code>nil</code>, this is the new text.</td></tr> <tr> <td><i>styles</i></td><td>If this slot is <code>nil</code>, the new text uses the style attributes at the start of the <i>range</i>.  If non-<code>nil</code>, 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 font specification frame or a graphics</td></tr> </table>	<i>text</i>	If this slot is <code>nil</code> , style runs are replaced, but the text remains the same. If non- <code>nil</code> , this is the new text.	<i>styles</i>	If this slot is <code>nil</code> , the new text uses the style attributes at the start of the <i>range</i> .  If non- <code>nil</code> , 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 font specification frame or a graphics
<i>text</i>	If this slot is <code>nil</code> , style runs are replaced, but the text remains the same. If non- <code>nil</code> , this is the new text.				
<i>styles</i>	If this slot is <code>nil</code> , the new text uses the style attributes at the start of the <i>range</i> .  If non- <code>nil</code> , 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 font specification frame or a graphics				

## Word Processing Views

specification frame. The run length is always 1 for graphics specification objects.

For a description of the font specification frame, see the section “Font Frame” (8-18) in *Newton Programmer’s Guide*. See “The Graphics Specification Frame” (page 3-22) for a description of the graphics specification frame.

return value      The number of replacements made by `ReplaceAll`.

**WARNING**

This operation can not be undone. ♦

**Storage Methods**

---

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()`

Creates an object that contains all of the data for the document in the view, including the document’s text, style runs, and rulers.

return value      A reference to the object that was created for the document. The object is allocated from the NewtonScript heap.

If you used `SetStore` to store your document in a virtual binary object (VBO), your document is referenced from this frame.



## Word Processing Views

**Note**

The object referenced by this method is intended for use as a black box, which means that you must use it for only 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.

return value                An error code, or `nil` if the operation worked.

**DISCUSSION**

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()`

Determines if the view has changed since the last call to either the `Externalize` or `Internalize` methods.

return value                A non-`nil` value if the contents of the view have changed since the last call to `Externalize` or `Internalize`. Returns `nil` if the contents have not been changed.

**Note**

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

## Scrolling Methods

---

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

### Note

The `protoTXView` has built-in scrolling that happens automatically when the user is entering text with the keyboard or pen. You usually only have to call the methods described in this section if you have attached scroller protos to the view that encloses your `protoTXView`. ♦

## 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.

return value                      An error code, or `nil` if the operation worked.

### DISCUSSION

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

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

## GetScrollValues

---

*protoTXView*:GetScrollValues()

Returns the current scroll values.

return value                      A frame with *x* and *y* slots containing the current scroll values, expressed in pixels.

### DISCUSSION

You can call this method and then use the returned values to update your scrollers.

## Word Processing Views

**GetTotalHeight**

---

*protoTXView*:GetTotalHeight()

Determines the current text height of the view.

return value            The current total text height of the view.

**DISCUSSION**

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 page (as specified in `SetGeometry`) multiplied by the total number of pages.

**GetTotalWidth**

---

*protoTXView*:GetTotalWidth()

Returns the current text width of the view.

return value            The current total text width of the view.

**DISCUSSION**

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 the scrollable area (the visible region) of the view.

return value            A rectangle frame describing the global coordinates of the rectangle in which text is displayed.

## Word Processing Views

## DISCUSSION

The rectangle returned by `GetScrollableRect` 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.

<i>updateMaxVal</i>	A Boolean value indicating whether you need to update the maximum value of the vertical scroller.
<i>scrolled</i>	A Boolean value indicating whether you need to update the scroller (of both the horizontal and vertical scrollers).
return value	Not used.

The `ViewUpdateScrollersScript` method is called when something happens that might affect the the scroll arrows; for example, if more text is added, which causes the maximum height or thumb position to change.

**Highlighting Methods**

---

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 discontinuous highlights.

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

**GetHiliteRange**

---

*protoTXView*:GetHiliteRange()

Returns the current highlight range.

return value	A range frame describing the current highlight range. Range frames are described in “The Range Frame” (page 3-22).
--------------	--

## Word Processing Views

**SetHiliteRange**

---

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

Changes the current highlight range.

<i>newRange</i>	A range frame that specifies the new highlight range. Range frames are described in “The Range Frame” (page 3-22).
<i>showHilite</i>	A Boolean value. If this is <code>true</code> , 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.
<i>setKeyView</i>	A Boolean value. If this is <code>true</code> , the view becomes the current key view, which activates the view for keyboard input. If this is <code>nil</code> and there already is a key view, the highlight range is shown as an inactive selection, which has a different on-screen appearance than a selection in the key view.
return value	An error code, or <code>nil</code> if the operation worked.

**GetContinuousRun**

---

*protoTXView: GetContinuousRun()*

Determines the style run for the currently highlighted range.

return value	A frame that specifies the style run containing the current highlight range.
--------------	--

**DISCUSSION**

If the current highlight range contains only one graphics object, `GetContinuousRun` returns a graphics specification frame, as described in “The Graphics Specification Frame” (page 3-22).

Otherwise, `GetContinuousRun` returns a font specification frame, as described in the section “Font Frame” (8-18) in *Newton Programmer’s Guide*. Any non-`nil` values in the font specification frame indicate values that are continuous for the highlight range.

## Word Processing Views

**Note**

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

**Ruler Methods**

This section describes the methods you can use to work with the ruler user interface that is built into `protoTXView`. The ruler is shown in Figure 3-1 (page 3-8).

**Note**

The ruler occupies screen space, which means that the text view rectangle is smaller when the ruler is displayed. This causes the `GetScrollableRect` method to return different values when the ruler is hidden than it does when the ruler is displayed. ♦

**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.

return value            An error code, or `nil` if the operation worked.

**HideRuler**


---

`protoTXView:HideRuler()`

Hides the ruler if it is currently shown.

return value            An error code, or `nil` if the operation worked.

## Word Processing Views

**IsRulerShown**

---

*protoTXView*:IsRulerShown()

Determines if the ruler is currently shown or hidden.

return value	Returns a non- <i>nil</i> value if the ruler is currently visible and <i>nil</i> if the ruler is currently hidden.
--------------	--

**UpdateRulerInfo**

---

*protoTXView*:UpdateRulerInfo(*rulerSettings*)

Changes the ruler display settings and updates the ruler display.

<i>rulerSettings</i>	A frame with one slot, <i>type</i> . The value of this slot can be one of the following symbols: <i>metric</i> or <i>inches</i> .
----------------------	---

return value	An error code, or <i>nil</i> if the operation worked.
--------------	---

**Page-Handling Methods**

---

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

**GetCountPages**

---

*protoTXView*:GetCountPages()

Determines the number of pages in the view.

return value	The number of pages. <i>GetCountPages</i> 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.

<i>range</i>	A range frame, as described in “The Range Frame” (page 3-22).
--------------	---

return value	An error code, or <i>nil</i> if the operation worked.
--------------	---

## Word Processing Views

## DISCUSSION

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

## Printing Methods

---

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

### SetDrawOrigin

---

*protoTXView*: `SetDrawOrigin(origin)`

Reconfigures a view for printing.

<i>origin</i>	A point frame with the following two slots:
x	The horizontal margin, in pixels.
y	The vertical margin, in pixels.
return value	An error code, or <code>nil</code> if the operation worked.

## DISCUSSION

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 then draw the view:

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



## Word Processing Views

**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`.

**FindString**

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

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

<i>object</i>	A data object returned from the <code>Externalize</code> method.
<i>str</i>	The string to find.
<i>startOffset</i>	The offset at which the search should start, expressed as a number of characters from the start of the data.
<i>options</i>	Must be <code>nil</code> .
return value	The offset of the matching string in the object, or <code>nil</code> if no match is found.

**DISCUSSION**

The `FindString` method searches in a word-processing data object for a sequence of characters that matches *str*. The search is not case sensitive. The search begins at *startOffset* from the beginning of the object and continues until a match is 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 works the same as the `protoTXView:FindString` method, except that the `protoTXViewFinder` version has an additional parameter: *object*. ♦

## Word Processing Views

**GetCountCharacters**

---

*protoTXViewFinder*:GetCountCharacters()

Determines the number of characters in the view object.

return value            The total number of characters.

**IMPORTANT**

You must call the *protoTXViewFinder:FindString* method to select a range of text in the data object before calling *protoTXViewFinder:GetCountCharacters*. ♦

**GetRangeText**

---

*protoTXViewFinder*:GetRangeText(*range*)Creates and returns a string that contains the characters in the specified *range* in the data object.

*range*                    A range frame, as described in “The Range Frame” (page 3-22).

return value            A string containing the characters in *range*.**IMPORTANT**

You must call the *protoTXViewFinder:FindString* method to select a range of text in the data object before calling *protoTXViewFinder:GetRangeText*. ♦

## Summary of Word Processing Views

---

### Data Structures

---

#### Ruler Information Frame

---

```
{
justification: int,           // 'left', 'right', 'center', or 'full'
indent:       int,           // first line indentation in pixels
leftMargin:   int,           // left margin in pixels
rightMargin:  int,           // right margin in pixels
lineSpacing:  int,           // paragraph line spacing, in lines
tabs:         [t1, t2, ... tN] // array of tab frames
}
```

#### Tab Frame

---

```
{
kind:          int,           // 'left', 'right', 'center', or 'decimalPoint'
value:         int,           // number of pixels from left edge
}
```

### protoTXView

---

```
{
// tells protoTXView to store view as VBO
SetStore:      func(store),

// changes view geometry
SetGeometry:   func(isPaged, width, height, margins),

// returns data for specified range in view
GetRangeData:  func(range, which),

// returns the number of chars in the view
}
```

## Word Processing Views

```

GetCountCharacters:func(),

// searches for matching text in a view
FindString:      func(str, startOffset, options),

// finds first and last characters of a word
GetWordRange:    func(offset),

// returns coordinates of a character
CharToPoint:     func(offset),

// returns the character at a poing
PointToChar:     func(point),

// returns range spec for a text line
GetLineRange:    func(offset),

// cuts selection to clipboard
Cut:             func(),

// copies selection to clipboard
Copy:           func(),

// replaces selection with clipboard contents
Paste:          func(),

// removes selection from view
Clear:          func(),

// changes font attributes for a range
ChangeRangeRuns: func(range, fontSpec, toggleFace, undoable),

// changes ruler attributes for a range
ChangeRangeRulers: func(range, ruler, undoable),

// replaces text in range with different data
Replace:         func(range, data, undoable),

// replaces all occurrences in a view
ReplaceAll:      func(str, startOffset, options, data),

// creates a VBO for the data in view
Externalize:     func(),

// replaces view contents with data from VBO
Internalize:     func(object),

// determines if view has changed
IsModified:      func(object),

```

## Word Processing Views

```

// scrolls the view horizontally or vertically
Scroll:      func(scrollValues),

// returns current scroll values
GetScrollValues: func(),

// determines current text height of view
GetTotalHeight: func(),

// determines current text width of view
GetTotalWidth: func(),

// determines scrollable area of view
GetScrollableRect: func(),

// notifies you that scrollers need updating
ViewUpdateScrollersScript:
    func(updateMaxVal, scrolled),

// returns current highlight range
GetHiliteRange: func(),

// sets the current highlight range
SetHiliteRange: func(newRange, showHilite, setKeyView),

// gets style run for current highlight range
GetContinuousRun: func(),

// displays the ruler
ShowRuler:    func(rulerSettings),

// hides the ruler
HideRuler:    func(),

// determines if ruler is currently shown
IsRulerShown: func(),

// changes ruler display settings & redisplay it
UpdateRulerInfo: func(rulerSettings),

// returns number of pages in view
GetCountPages: func(),

// replaces text range with a page break
InsertPageBreak: func(range),

// reconfigures view for printing
SetDrawOrigin:  func(origin)
}

```

## Word Processing Views

## protoTXViewFinder

---

```
{  
  
    // finds a string in a VBO  
FindString:      func(object, str, startOffset, options),  
  
    // returns the number of characters in a VBO  
GetCountCharacters:func(),  
  
    // returns characters for a text range in a VBO  
GetRangeText:    func(range)  
}
```

# Keyboard Enhancements

---

This chapter describes the expanded toolbox support that is provided in the Newton 2.1 Operating System (OS). This chapter provides you with information about the new software facilities for defining, processing, and displaying keyboard commands and shortcuts, including the following:

- command keys
- keyboard-based selection of 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

The features described in this chapter apply to all forthcoming Newton-based devices.

## About Keyboard Enhancements

---

This section provides general information about the keyboard enhancements provided by the Newton 2.1 OS.

## Terminology

---

The following terms are used to describe keys and keyboard actions in this chapter:

- **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.
- **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.
- **Keystrokes** are groups of individual key presses that the system has collected together for batch processing.

## About Keystroke Handling

---

This section provides general information about how keystrokes are handled by the system software and how your application can intercept keystrokes, including the following techniques:

- 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 keystrokes to improve performance. You can also intercept these strings, which are used to improve user response time. Note that rapid successions of 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



## Keyboard Enhancements

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.

- You can define keyboard commands (command keys) that the system will match and execute where appropriate.

## 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` method in the key-view (proto inheritance only) and calls it. Note that the `ViewKeyDownScript` method is also called when there are no other pending, unprocessed keystrokes.
2. If the `ViewKeyDownScript` method 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 “Handling Command Keys” (page 4-24).
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. Note that if the user is typing in an existing paragraph in an edit view, the paragraph receives the key strokes.

### Key-repeat Events

---

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

1. After a brief delay, the system starts issuing key-repeat events.

## Keyboard Enhancements

2. For each key-repeat event, if `vSingleKeystrokes` is set in the key-view, the system calls `ViewKeyRepeatScript` (proto inheritance only) is called.
3. If the view does not have a `ViewKeyRepeatScript`, the system calls `ViewKeyDownScript` instead.
4. If this method 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 view class 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`.

### 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.

## Keyboard Enhancements

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.

## About Command Key Handling

---

Each view in your application has a set of key commands associated with it. This section describes how key commands are defined and associated with each view.

To define a key command for one of your views, you need to define a `keyCommand` frame, as described in “The Command-Key Mapping Frame” (page 4-31).

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.

## How Command Keys Are Found

---

The system software searches for command keys when the user presses one of the following keys:

- 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 key commands 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.
2. The system looks through the key commands registered in the current view for a `keyCommand` that matches the pressed key.
3. If the system finds a match, the search is complete.

Keyboard Enhancements

4. Otherwise, the system looks for a slot named `_nextKeyView` in the current view. This slot contains a reference to another view. If the `_nextKeyView` 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 4-1 shows how commands are associated with views.

**Table 4-1** 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

About Displaying Command-Key Combinations in Menus

To display a command-key combination for an item in a popup menu, you must define a `keyMessage` frame for the item.

For example, you can define a popup menu without command-key equivalents by passing an array to the `PopupMenu` view method. The following array creates a menu with three items:

[

## Keyboard Enhancements

```

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

```

The second item in the above menu contains an icon and the third item contains an icon and a mark.

To create the above menu with command-key combinations, you specify `keyMessage` slots in the item frames. When the menu is displayed, the system software searches for a matching command key and then uses the name, character, and modifiers defined in the found `keyCommand` frame. For example, the menu could be defined with the following array:

```

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

```

The names for the second and third items in the menu array above are found in the corresponding `keyCommand` frames.

**Note**

To find a `keyCommand` frame that matches a menu item, the system looks for a `keyCommand` frame whose `keyMessage` slot has the same value as the `keyMessage` slot in the menu item frame. ♦

If you want to display a name in the menu that is not the same as the name specified in the `keyCommand` frame, you can use a `keyMessage` slot and an item slot in the item frame. In this case, the item name is used instead of the name in the matching `keyCommand` frame. For example:

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

In the above example, the name “Something” is displayed in the menu.

**Note**

An item defined with a `keyMessage` slot can also have icon and mark slots.

## About Keyboard Support in Pickers

---

In most Newton applications, when the user selects an item in a picker, the picker's `PickActionScript` is called, and when the user taps outside of the picker, the picker's `PickCancelledScript` is called. When you have key commands defined in a picker view, this changes.

In the Newton 2.1 OS, when the user selects an item in a picker, the system first determines if the item has a key-command associated with it. If so, the method associated with that key-command (its `keyMessage`) is called instead of the `PickActionScript`.

You can override this by adding the `alwaysCallPickActionScript` flag to your picker. When this flag is on (set to `true`), the system software always calls the `PickActionScript`, regardless of whether or not there's a key-command for the item. If the `alwaysCallPickActionScript` flag is off (set to `nil`), the system software calls the key-command method for the item if there is one, and calls the `PickActionScript` method for the item if there is not a key-command associated with it.

### WARNING

You must always set the `alwaysCallPickActionScript` flag to `true` in the `protoLabelPicker` and `protoLabelInputLine` protos; otherwise, these protos do not function properly. Note that you can still call the key-command method for an item in one of these pickers, as shown in “Calling a Key-Command Method From a Picker Script” (page 4-8). ♦

## Calling a Key-Command Method From a Picker Script

---

As described in “About Keyboard Support in Pickers” (page 4-8), your key-command methods can be automatically called when a user selects an item in a picker. This happens when the `alwaysCallPickActionScript` flag is set to `nil` in a picker.

There are two picker protos that do not work properly when the `alwaysCallPickActionScript` flag is set to `nil`: `protoLabelPicker` and `protoLabelInputLine`. You must set the `alwaysCallPickActionScript` flag to `true` in these protos.

## Keyboard Enhancements

If you want to call a key-command method when the user selects an item in a `protoLabelInputLine` or `protoLabelPicker`, you need to call the key-command method from the `PickActionScript` for that proto.

Listing 4-1 shows an example of calling a key-command method from a picker. In this sample code, the method is called from the `labelActionScript`, which is called by the proto's `pickActionScript`.

---

**Listing 4-1**      Calling a key-command method from a picker

```
labelActionScript: func(cmd)
    begin
    local item := labelCommands[cmd];
    if item.keyMessage then
        SendKeyMessage( self, item.keyMessage );
    else
        // do normal item processing here
    end;
```

## Keyboard Enhancements User Interface

---

This section describes the user interface characteristics of the keyboard enhancements, including specific command-key combinations used for various built-in 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. All built-in applications that use paragraph views ignore the control keys during text input, which means that users cannot insert control characters into paragraph views. However, applications such as a terminal emulator can make use of control keys. Note that the control key is not used for command-key combinations.

## Keyboard Enhancements

- Pressing the Command key in combination with an alphanumeric key can be used to invoke 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

Users can apply the following keyboard actions during text entry and text editing:

- Pressing the arrow and tab keys to move between fields.
- Pressing the standard Macintosh clipboard key combinations produces the same actions as on the Macintosh: Cmd-X to cut the selection to the clipboard, Cmd-C to copy the selection to the clipboard, and Cmd-V to paste from the clipboard.
- Pressing the Cmd-A key combination selects all text in the view.

## Slips, windows, and buttons:

The key-view is generally the view that contains the caret and receives and processes keyboard commands. A slip is displayed differently when it is the key-view, to indicate to the user that key presses are directed to the slip.

Figure 4-1 shows how the find slip looks when it is not the key-view.

**Figure 4-1** The find slip when it is not the key view

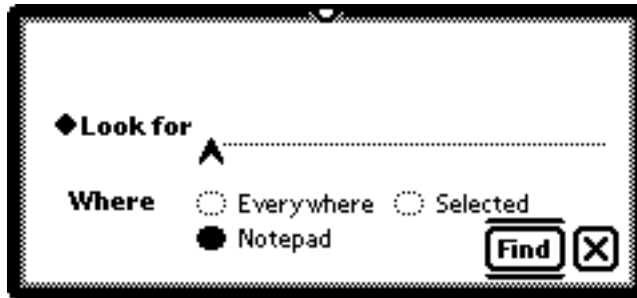




## Keyboard Enhancements

Figure 4-2 shows how the Find slip looks when it is the key view.

**Figure 4-2** The Find slip when it is the key view



When a slip is the key-view, its border is thicker, and the default button is marked with lines above and below the button. The user can select the default button by pressing the Return key. 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.

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-view 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.

## Keyboard Enhancements

## Menus

---

This section describes keyboard usage with Newton menus.

Application-defined keys or key combinations can be used to display popup menus. Some system-wide standards are defined (such as Cmd-N for New and Cmd-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.

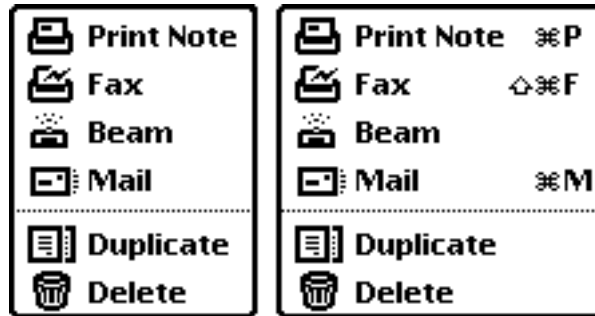
Menu items can have keyboard equivalents that are displayed to the right of the item. These are only displayed when a keyboard is actually connected.

**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 4-3 shows the basic appearance of a menu with and without its keyboard equivalents displayed.

## Keyboard Enhancements

**Figure 4-3** A menu with and without its keyboard equivalents displayed**Note**

The keyboard equivalents shown in Figure 4-3 are not the actual key-combinations for these commands. ♦

**The Command-key Combination popup Help Slip**

Keyboard equivalents are not displayed for buttons. Instead, the user can display a popup help slip by holding down the command key for approximately 1.5 seconds. The popup help slip displays all of the valid command keys for the current context.

The system automatically produces the popup help slip. You register your application's command-key combinations, as described in "The Command-Key Mapping Frame" (page 4-31), and the system constructs the help slip based on which command keys are available.

The popup help slip displays the available commands ordered alphabetically by category. Your application can add to the standard categories or can define new categories, as described in "Handling Command Keys" (page 4-24).

Figure 4-4 shows a version of the command-key combination popup slip.

## Keyboard Enhancements

**Figure 4-4** Command-key combination slip

<b>Creating</b>			
New	⌘N	Title	⌘T
<b>Editing</b>			
Copy	⌘C	Paste Copy	⇧⌘V
Cut	⌘X	Select All	⌘A
Paste	⌘V	Styles	⇧⌘S
<b>Filing</b>			
File	⇧⌘F	Folder	⇧⌘T
<b>General</b>			
Assist	⌘`	Help	⌘/
Close	⌘.	Undo	⌘Z
Find	⌘F		
<b>Routing</b>			
Print	⌘P	Routing	⌘R
<b>Viewing</b>			
Close	⌘W	Show	⌘S
Overview	⌘O		

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

The popup help slip closes automatically when the user releases the command key, or when the user presses any key on the keyboard.

If the number of commands defined exceeds the maximum number that can be displayed at once, the system adds a scroller to the popup help slip. The user can scroll the view with the up-arrow and down-arrow keys (while holding down the command key), or by tapping on the scroller's buttons. The user must continue to press the command key while scrolling the view.

If a command lacks a name, the command does not appear in the popup help slip. This is true of the standard command-arrow combinations for

Keyboard Enhancements

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 slip, or can use the information presented in the default slip in their own way, as described in “Handling Command Keys” (page 4-24).

System and Built-in App Command Key Assignments

---

This section describes the command-key combinations for system-level and built-in application operations. Table 4-2 shows the system-level command-key combinations.

**Table 4-2**      System-level key assignments

---

<b>Command-key combination</b>	<b>Behavior</b>
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 picker
Cmd-O	Opens the overview, toggles to close overview
Cmd-P	Opens the print slip
Cmd-R	Opens the Routing button picker, with first item hilited
Cmd-S	Opens the Show button picker
Cmd-V	Pastes selected text at cursor
Cmd-W	Closes the open window / slip
Cmd-X	Cuts selected text
Cmd-Z	Undo/redo
Cmd-return	Opens popup

Keyboard Enhancements

**Table 4-2** System-level key assignments (continued)

<b>Command-key combination</b>	<b>Behavior</b>
Cmd-` (tilde)	Opens assist
Cmd-. (period)	Cancels action, closes window or slip
Cmd-=	Activates spellcheck
Cmd-shift-F	Opens the Filing button picker, 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 Newton Works 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 picker
Return	Selects highlighted item from overview or picker

**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 affect the local scrollers.



Keyboard Enhancements

**Command-key Assignments for the NotePad Application**

---

Table 4-3 shows the command-key assignments for the Notepad checklist and outline stationery.

**Table 4-3** 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 are no keyboard commands for demoting, promoting, expanding, or collapsing already created items. The user can only perform these operations with the pen. ♦

**Command-Key Assignments for The Names Application**

---

Table 4-4 shows the command-key assignments for the Names application.

**Table 4-4** Names application command keys

---

Command Keys	Behavior
Cmd- +	Opens the Add picker. The user can choose the 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. ♦

Keyboard Enhancements

**Command-Key Assignments for The Dates Application**

---

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

**Table 4-5**      Dates application command keys

---

<b>Command Keys</b>	<b>Behavior</b>
Cmd-+	Opens the Add picker. The user can choose the item with the arrow and Return keys.

**Command-Key Assignments for The In/Out Box**

---

Table 4-6 shows the command key assignments for the In/Out box

**Table 4-6**      In/Out box command keys

---

<b>Command Keys</b>	<b>Behavior</b>
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



Keyboard Enhancements

Command-Key Assignments for The Call Log

---

Table 4-7 shows the command-key assignments for the call log.

**Table 4-7** 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

Command-Key Assignments for the BookPlayer

---

Table 4-8 shows the command-key assignments for the BookPlayer application.

**Table 4-8** 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

Compatibility

---

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

Default Buttons

---

Default buttons in slips appear and function only in applications that were designed with them in mind. This is described in “Designating the Default Button In a Slip” (page 4-29).

Keyboard Enhancements

Possible Key-view Compatibility Problem

---

Prior to Newton 2.1 OS, you could not set the key-view to anything other than a `clParagraphView` (using `Setkeyview`) or a `clEditView` (using `SetCaretInfo`). In Newton 2.1 OS, you can designate any view as a key-view. This could be a problem for some older applications.

Using the Keyboard Enhancements

---

This section describes how you can use the keyboard enhancements in your 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” (page 4-20).
- How to work with command keys, as described in “Handling Command Keys” (page 4-24).

Keystroke Handling

---

This section describes how to handle keystrokes in your Newton applications. Table 4-9 shows the functions and methods that you can use to handle keystrokes.

**Table 4-9** Summary of keystroke-handling methods and functions

---

Function/Method	Description
<code>IsCommandKeystroke</code>	Determines if a keystroke is a command key combination.
<code>ViewKeyDownScript</code>	Sent to the key-view when the user presses a key.

## Keyboard Enhancements

**Table 4-9** Summary of keystroke-handling methods and functions

Function/Method	Description
<code>ViewKeyUpScript</code>	Sent to the key-view when the user releases a key.
<code>ViewKeyStringScript</code>	Sent to the key-view when a group of keystrokes needs to be processed. This can occur when single keystroke handling is not applied to the key-view (when the <code>vSingleKeystrokes</code> flag is not set).
<code>ViewKeyRepeatScript</code>	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. If you set the `vSingleKeystrokes` text flag for your view, you are guaranteed that the scripts for these key events are called for every keystroke. If you do not set the `vSingleKeystrokes` text flag for your view, the scripts are called only under certain circumstances.

#### 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.

The system software calls a method for each of the keyboard events. For an overview of the sequencing of actions when the user presses keys, see “Keystroke Event Sequencing” (page 4-3). If the method returns `nil` (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.

The functions and event scripts for handling keystrokes are shown in Table 4-9; their use is described in the remainder of this section.

## Keyboard Enhancements

## Intercepting Individual Keystrokes

---

Here is an example of code that processes individual keystrokes.

```

ViewKeyDownScript: func( char, flags )
    begin
        if char = unicodeCR then
            begin
                :OutputTextLine( self.text );
                SetValue( self, 'text', "" );
                return true;
            end;

            // Explicitly return nil just for clarity
            nil;
        end;
    end;

```

## Intercepting Grouped Keystrokes

---

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

**Note**

Although the system can group keystrokes into keystrings, this only happens when the user is typing at a very high speed. If the system can keep up with individual keystrokes, the keystrokes are not grouped into a keystring. ♦

## Text Flags and Keyboard Input

---

Paragraph views, edit views, and text editing views all accept both command keys and normal keys (for insertion). For other views, there are

## Keyboard Enhancements

two additional `textFlags` that you can use to specify the kinds of keystrokes you want to handle, as shown in Table 4-10.

**Table 4-10** Text flags to specify the kind of keystrokes a view accepts

Text Flag	Description
<code>vTakesCommandKeys</code>	The view accepts command keys.
<code>vTakesAllKeys</code>	The view accepts all keys, including command keys.

**Note:**

Do not use the text flags shown in Table 4-10 for edit views, paragraph views, or word-processing (`protoTXView`) 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);
```

Keyboard Enhancements

# Handling Command Keys

Each view in your application can have a set of key commands associated with it. You can use the functions and methods shown in Table 4-11 to work with key commands.

**Table 4-11** Summary of command key methods and functions

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

## Searching for Key Commands

If your `ViewKeyDownScript` or `ViewKeyRepeatScript` methods return `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 user presses a function key, the escape key, or any key in combination with the command key. You can also force the system to search for key commands after each keystroke by setting the `vAlwaysTryKeyCommands` text flag in the key view.

## Keyboard Enhancements

## Defining Key Commands

---

To define a set of key commands, you need to create an array of `keyCommand` frames, each of which defines an individual key command. See “The Command-Key Mapping Frame” (page 4-31) for a full description of the `keyCommand` frame slots. Listing 4-2 shows an example of a key command array.

---

**Listing 4-2**     A key command array

```
keyCommandArray:
[
    {
        char: $g,
        modifiers:kCommandModifier,
        keyMessage:'DoGKeyCommand,
        name: "Do G Key",
    },
    {
        char: $b,
        modifiers:kCommandModifier,
        keyMessage:'DoBKeyCommand,
        name: "Do B Key",
    },
],
```

Each `keyCommand` frame specifies the keys that the user presses to invoke the command, the name to display for the command, and the message that the system sends when the command is invoked.

## Adding the Key-Commands

---

You can use the `AddKeyCommand` method of a view to add a single key-command, or you can use the `AddKeyCommands` method to add an array of key-commands to the view. You typically add your key-commands during view setup. Listing 4-3 shows an example of setting up key-commands in the `ViewSetupFormScript` method.

## Keyboard Enhancements

**Listing 4-3** Defining key-commands in the `ViewSetupFormScript` method

---

```
ViewSetupFormScript: func()
    begin
        :AddKeyCommands( keyCommandArray );
    end
```

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

**Invoking the Command-Key Method**


---

When the user presses a command-key combination, the system software sends the message associated with the command key (as defined in its `keyCommand` frame). The method that is invoked for the command-key message need not be implemented in the same view as is the command key.

Once the system software matches a `keyCommand`, the system searches the same chain (starting at the key-view and following 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, if the key commands array shown in Listing 4-2 is in use and the user presses the Cmd-g key-combination, the following call is made:

```
view:DoGKeyCommand( currentKeyView)
```

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

**Removing Key-Commands**


---

It is good practice to remove your key-commands when your view is closing. To do so, you can call the `ClearKeyCommands` method in the `ViewQuitScript` method of your view. Listing 4-4 shows an example.



## Keyboard Enhancements

**Listing 4-4** Removing key-commands

---

```
ViewQuitScript: func()
    begin
        :ClearKeyCommands();
        inherited:?ViewQuitScript();
    end
```

**Note**

If you do not remove key commands when your view is closing, the key commands can waste valuable heap space. ♦

## Displaying the Popup 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 root view, dynamically builds and displays the standard “Keyboard Commands” popup help slip, according to the available keyCommands. You can provide your own versions of these scripts to modify or override the default popup help slip.

## 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.

## Keyboard Enhancements

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

Listing 4-5 shows an example of a `ViewCaretActivateScript` method. This implementation plays a sound at caret activation time: if a keyboard is connected, it beeps; if not, it clicks.

---

### **Listing 4-5**     An example of a `ViewCaretActivateScript` method

```
ViewCaretActivateScript: func( active )
begin
  if KeyboardConnected() then
    :SysBeep();
  else
    PlaySound(ROM_Click);
  nil;
end
```

## Using Keys in Slips

---

This section describes how to use key in your slips.

## Keyboard Enhancements

## Designating the Default Button In a Slip

---

To designate a button as a slip's default button, you need to create a `_defaultButton` slot in the view. This slot must contain a reference to the view that is the default button. 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`, `protoFloater`, `protoFloatNGo`, and many other built-in protos are based on `protoContainerView`.

Paragraph views that have the `oneLineOnly` view justification flag automatically send the key message `_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).

Keyboard Enhancements

Default and Close Buttons in Confirm Slips

Four new default button lists are now available for use in confirm slips. This makes it easy to add keyboard support to all of your confirm slips. Table 4-12 shows the new default button lists.

**Table 4-12**     New default button lists

Button list	Description
'okCancel	The user can select either Ok or Cancel. The system closes the slip when the user taps Cancel. The default value is Ok.
'okCancelDefaultCancel	The user can select either Ok or Cancel. The system does not provide a keyboard equivalent for the Ok button. The default value is Cancel.
'yesNoDefaultYes	The user can select Yes or No. The system closes the slip when the user taps No. The default value is Yes.
'yesNoDefaultNo	The user can select Yes or No. The system does not provide a keyboard equivalent for Yes. The default value is No.

If you are creating your own button list, you can add a slot to the `buttonFrame` named `keyValue`. The value of this slot can be `'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 view version of the `Confirm` method (`:Confirm()`) previously used the `okCancel` button list; it now uses the `okCancelDefaultOk` button list instead.

## Keyboard Reference

---

This section describes the functions, methods, and data structures for keyboard handling in Newton applications.

### Data Structures

---

This section describes the data structures that you use with the keyboard enhancement methods and functions.

### 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 six slots, as shown here:

```
keyCommand := {  
    char:      $a,  
    modifiers: kCommandModifier,  
    keyMessage: '_SelectAll',  
    name:      "Select All",  
    category:  "Editing"  
    showChar:  $a  
};
```

## Keyboard Enhancements

**Slot descriptions**

<code>char</code>	The unmodified character of the keypress. Required.
<code>modifiers</code>	The required modifiers. This slot can be absent or <code>nil</code> , in which case no modifiers are required.  This slot can also be used to specify other flags related to the command:
<code>kRepeatable</code>	The command is to be executed on key-repeat events as well as key-down events.
<code>kWorksInAllModals</code>	Applies only to system-wide (root view) commands. When set, the command is available in modal dialogs.
<code>kWorksInAppModals</code>	Applies only to system-wide (root view) commands. When set, the command is available in modal dialogs whose <code>vApplication</code> bit is set.
<code>keyMessage</code>	A symbol. Required. This is the message that is sent when the <code>keyCommand</code> 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.
<code>name</code>	A string. The name of the command that appears on menus and the command key popup help slip. If this slot is <code>nil</code> or absent, the key equivalent is not displayed on the popup help slip.
<code>category</code>	A string. The name of the category to which the command belongs on the command key popup help slip. If this slot is absent or <code>nil</code> , but there is a name slot, the command is placed in the “Other” category on the command key popup help slip.
<code>showChar</code>	A character. Optional. If present, this character is shown on popup menus and in the popup help slip instead of the character in the <code>char</code> slot. This is useful for presenting a more user-friendly key combination to the user than the actual combination. For example, you can

Keyboard Enhancements

define the Cmd-/ combination and present it as Cmd-?  
by defining this slot with the '?' character.

Table 4-13 shows the key codes for special (non-printing) keyboard keys. You can use these values in the char slot of your keyCommand frame.

**Table 4-13**     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
kF6Key	\$\uF726
kF7Key	\$\uF727
kF8Key	\$\uF728
kF9Key	\$\uF729

Keyboard Enhancements

**Table 4-13** Key codes for special keys (continued)

Constant	Value
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. ♦

## Methods and Functions for Handling Keystrokes

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

### HandleKeyEvents

`HandleKeyEvents (keyEvents)`

Posts key events as if they were typed on a hardware keyboard. You can use this function for testing purposes or to play back keyboard macros.

<i>keyEvents</i>	An array of integers. Each integer specifies a single key-down or key-up event. The least significant seven bits of each integer specify a key code 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.
return value	Undefined; do not rely on it.



## Keyboard Enhancements

## DISCUSSION

The state of the hardware keyboard (its keymap) is saved and restored before and after the events are handled so that inconsistencies are avoided (that is, 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. ♦

**IsCommandKeystroke**


---

`IsCommandKeystroke(char, flags)`

Determines if the specified keystroke is a command-key combination.

<i>char</i>	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.
<i>flags</i>	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 4-14.
return value	Returns <code>true</code> if the keystroke is a command-key combination, and <code>nil</code> if not.

**IsKeyDown**


---

`IsKeyDown(keyCode, isHardKeyboard)`

Determines if the specified key is currently down.

<i>keyCode</i>	The keycode that you want to test.
<i>isHardKeyboard</i>	True if you want the hardware keyboard tested. A value of <code>nil</code> means that the on-screen keyboard is tested.
return value	Returns <code>true</code> if the specified key is down on the keyboard.

## Keyboard Enhancements

## DISCUSSION

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.

## Methods and Functions for Handling Command Keys

---

This section describes the methods and functions that you can use to work with command keys in your Newton applications. You define key commands in `keyCommand` frames, which are described in “The Command-Key Mapping Frame” (page 4-31). Each `keyCommand` frame associates a key combination with a message and other information.

### 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 4-31).

return value            Undefined; do not rely on it.

## DISCUSSION

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 4-31).

return value            Undefined; do not rely on it.

## Keyboard Enhancements

## DISCUSSION

You can call the `AddKeyCommands` method from your `ViewSetupDoneScript`.

This method is efficient for adding an array of key commands 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.

return value      Undefined; do not rely on it.

## DISCUSSION

The `BlockKeyCommand` method makes any key command that matches *keyMessageSymbol* unavailable from the view. The key command no longer appears on the command key popup help slip in the view.

## IMPORTANT

The `AddKeyCommand` and `BlockKeyCommand` methods grow a RAM-based array, so you must be careful to not overuse these methods. ♦

**CategorizeKeyCommands**

---

*CategorizeKeyCommands (keyCommandArray)*

Categorizes an array of key command frames.

*keyCommandArray*      An array of `keyCommand` frames.

return value      Undefined; do not rely on it.

## Keyboard Enhancements

## DISCUSSION

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.

**RemoveKeyCommandFrame**

---

*view*: `RemoveKeyCommandFrame (keyCommand)`

Removes the specified key command frame from the view.

<code>keyCommand</code>	The <code>keyCommand</code> frame to remove from the registry for the view. This frame must match the frame used in a previous call to the <code>AddKeyCommand</code> or <code>AddKeyCommands</code> methods.
-------------------------	---

<code>return value</code>	Undefined; do not rely on it.
---------------------------	-------------------------------

## DISCUSSION

The `RemoveKeyCommandFrame` method removes a specific key-command frame from view. Not that `RemoveKeyCommandFrame` actually removes the RAM-based frame from the registry.

**ClearKeyCommands**

---

*view*: `ClearKeyCommands ()`

Removes all key commands from the view.

<code>return value</code>	Undefined; do not rely on it.
---------------------------	-------------------------------

## DISCUSSION

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.

## Keyboard Enhancements

**FindKeyCommand**

---

`FindKeyCommand (startView, char, flags)`

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

<i>startView</i>	The view in which to start searching for the command key.
<i>char</i>	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.
<i>flags</i>	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 4-14.
return value	The matching <code>keyCommand</code> or <code>nil</code> if none is found.

**DISCUSSION**

The `FindKeyCommand` function starts at the view *startView* and looks for a `keyCommand` frame that matches the keypress described by *char* and *flags*.

**GatherKeyCommands**

---

`GatherKeyCommands (startView)`

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

<i>startView</i>	The view in which you are interested.
return value	An array of all the key commands available to the view <i>startView</i> .

**PressButton**

---

`PressButton (buttonView)`

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

<i>buttonView</i>	The button you want tapped.
return value	Undefined; do not rely on it.

## Keyboard Enhancements

## DISCUSSION

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.

**SendKeyMessage**

---

`SendKeyMessage (keyView, keyMessage)`

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

<i>keyView</i>	The view to which the message gets sent.
<i>keyMessage</i>	A symbol that names the command message. This must be the same message as you specified in the <code>keyMessage</code> slot of the <code>keyCommand</code> frame.
return value	Undefined; do not rely on it.

## DISCUSSION

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

**Application-Defined Methods for Keystroke Events**

---

This section describes the methods that you can define in your application to intercept keystroke events.

Keyboard Enhancements

ViewKeyDownScript

*view*:ViewKeyDownScript(*char*, *flags*)

Sent by the system when a user presses down a keyboard key.

<i>char</i>	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.
<i>flags</i>	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 4-14.
return value	Your implementation must return <code>nil</code> if you want the system to continue processing the keystroke.

DISCUSSION

Table 4-14 shows how the bits in the *flags* parameter are used for the key event scripts.

**Table 4-14** 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. ( <code>kIsSoftKeyboard</code> )
25	Indicates that the Command key was down. ( <code>kCommandModifier</code> )
26	Indicates that the Shift key was down. ( <code>kShiftModifier</code> )

Keyboard Enhancements

**Table 4-14** Key event-processing script flags (continued)

Bits	Description
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**

*view*:ViewKeyUpScript(*char*, *flags*)

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

<i>char</i>	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.
<i>flags</i>	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 4-14.
return value	Your implementation must return <code>nil</code> if you want the system to continue processing the keystroke.



## Keyboard Enhancements

**ViewKeyRepeatScript**

---

*view*:ViewKeyRepeatScript(*char*, *flags*)

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 4-14.

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

**ViewKeyStringScript**

---

*view*:ViewKeyStringScript(*string*)

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.

return value             Your implementation must return `true` if your method handles the *string* and `nil` if not.

**DISCUSSION**

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

## Summary of Keyboard Enhancements

---

### Data Structures

---

#### Command-Key MappingFrame

---

```
{
char:          char,           // unmodified keypress
character
modifiers:    int,            // modifiers
keyMessage:    symbol,         // message to send for
key-command
name:          string,         // command name for menus
category:      string,         // command category
showChar:      char,          // char to display in menus
}
```

### Methods and Functions

---

```
// Posts key events as if they were typed on a
hardware keyboard
HandleKeyEvents(keyEvents)
```

```
// Determines if the keystroke is a command-key
combination.
IsCommandKeystroke(char, flags)
```

```
// Determines if a key is down
IsKeyDown(keyCode, isHardKeyboard)
```

```
// Associates key-command with view
view: AddKeyCommand(keyCommandFrame)
```

## Keyboard Enhancements

```

// Associates an array of key-commands with a view
view:AddKeyCommands(arrayOfKeyCommandFrames)

// Hides a key-command in a view
view:BlockKeyCommand(keyMessageSymbol)

// Categorizes an array of key-command frames
CategorizeKeyCommands(keyCommandArray)

// Removes all key-commands from a view
view:ClearKeyCommands()

// Finds key-command that matches key combination
FindKeyCommand(startView, char, flags)

// Returns array of all key commands in a view
GatherKeyCommands(startView)

// Causes a button to act as if it was tapped
PressButton(buttonView)

// Sends a message to a view as if the user typed a
key-command
SendMessage(keyView, keyMessage)

// Sent to view when user presses down on key
view:ViewKeyDownScript(char, flags)

// Sent to view when user releases a key
view:ViewKeyUpScript(char, flags)

// Sent repeatedly to view while user holds a key down
view:ViewKeyRepeatScript(char, flags)

// Sent to view when a batch of keystrokes needs
processing
view:ViewKeyStringScript(string)

```

## Keyboard Enhancements

# Spell Checker

---

This chapter documents the built-in spell checker available in Newton 2.1 OS.

## About the Spell Checker

---

The built-in spell checker is a fast, dictionary-based spell checker. It supports letter insertion, letter deletion, letter transposition, letter substitution, and phonetic substitution. It can also split run-together words. It can remember words that have been skipped in the current spell-check session, so it won't flag them as incorrect again, and it supports an interface for learning words.

It is also smart about locales. Locale-specific spellings (color versus colour) are provided by locale-specific dictionaries, so these spellings are allowed only in the proper locale.

Currently, the spell checker supports English only.

The spell checker has no built-in user interface. It exists as a set of global function calls. The Newton Works word processor implements a user interface for it, but you must create your own user interface if you want to add the spell checker capabilities to your own application.

## Spell Checker

If you need to test for the existence of the spell checker in the system (for an application that runs on all 2.x systems), you can use the following test:

```
if GlobalFnExists('SpellDocBegin') then ...
```

## Limitations

---

There are some limitations to the spell checker.

- There's a maximum character length of 50 for both input words and returned guesses. If a word is longer than 50 characters, it won't be flagged as containing a spelling error.  
Note that there are other word length limitations within the system (usually 30 characters) preventing long words from being added to the personal word list, for example.
- The spell checker doesn't allow random insertion, deletion, or substitution during the phonetic substitution. This means, for example, that it won't correct the word "hexllow", because that would require both removing the "x" and substituting "o" for "ow".

## Using the Spell Checker

---

You tell the spell checker that you're about to start spell-checking a document by calling this function:

```
local speller := SpellDocBegin();
```

Then for each word in the document, you call the function `SpellCheck`. The `SpellCheck` function returns non-nil if the word appears to need correction.

For words needing correction, you call the function `SpellCorrect`. `SpellCorrect` returns a list of possible alternate words. If you want the spell checker to temporarily remember words that have been skipped, call the function `SpellSkip`. `SpellSkip` remembers words for the currently spell-check session only. If you want a word to be permanently learned, call the function `SpellLearn`, which adds a word to the user's personal word list.

## Spell Checker

When you have finished processing all the words in the document, call the function `SpellDocEnd`. This clears out the list of words that have been skipped, and it causes the learned words to be saved to the personal word list on the user store.

One way of using the spell checker is to call `SpellDocBegin` when you open a document, and `SpellDocEnd` when you close it. This preserves the skipped-word list for as long as the document is open.

Note that the spell checker can be used on multiple documents simultaneously.

## Processing of Words Passed to the Spell Checker

---

Many of the spell-checking functions accept a word as a parameter. This parameter is processed similarly by these functions. First, leading and trailing punctuation are stripped from the word. Next the spell checker remembers whether the word was capitalized or all caps, and converts the word to lower-case. If the word contains a curly apostrophe (’), it is converted to a normal apostrophe (because that’s what’s in the dictionary).

Note that a hyphen is not considered to be a valid symbol by the spell checker. The application must split hyphenated words before calling the spell checker.

## Use of Dictionaries by the Spell Checker

---

The spell checker assembles a list of dictionaries to use for both checking and correction. A word is considered to be valid if it is in one of the built-in dictionaries (with appropriate locale-specific words), the user’s personal word list, or the list of skipped words.

The dictionaries used are the same as the dictionaries that are used by the cursive recognizer in a view with the `vAnythingAllowed` view flag set. For checking, the spell checker scans through the list of dictionaries looking for dictionaries that have any of the following flags set: `vCharsAllowed`, `vDateField`, `vTimeField`, `vPhoneField`, `vNumbersAllowed`. This means that dates, times, phone numbers, and numbers are “spell-checked” against the built-in lexical dictionaries.

## Spell Checker

For correction, the spell-checker uses the dictionaries that have the `vCharsAllowed` flag set, which includes third party dictionaries that have been installed as default dictionaries. The lexical dictionaries are not used for correction because they generate too many correct alternatives.

## Spell Checker Reference

---

### Functions

---

This section describes the spell checker global functions.

#### **SpellDocBegin**

---

`SpellDocBegin()`

Initializes the spell checker.

return value	A frame referencing a data structure for the current spell-checking session.
--------------	--

#### **DISCUSSION**

Call this function before you begin to spell check a document. Among other things, this function initializes the list of skipped words to be empty (see `SpellSkip`).

When you are done spell checking, you must call `SpellDocEnd`.

#### **SpellDocEnd**

---

`SpellDocEnd(speller)`

Frees the data structures allocated for a spell-checking session and performs other clean-up functions.

<i>speller</i>	The frame returned by the <code>SpellDocBegin</code> function.
return value	Undefined; do not rely on it.



## Spell Checker

## DISCUSSION

You must call this function when you have finished spell checking your document. This function causes the user's personal word list to be saved if the `SpellLearn` function had been called, deletes the list of skipped words, and then deallocates the spell checker data structures. Once you have called `SpellDocEnd`, the *speller* frame cannot be used in subsequent calls. You must call `SpellDocBegin` to start another session.

**SpellCheck**


---

`SpellCheck(speller, word)`

Checks the spelling of a word.

<i>speller</i>	The frame returned by the <code>SpellDocBegin</code> function.
<i>word</i>	A string containing a word to spell check.
return value	A <code>nil</code> value indicates that the word or number is correct. A non- <code>nil</code> value indicates that the word may need to be corrected.

## DISCUSSION

This function first processes the word as described in “Processing of Words Passed to the Spell Checker” (page 5-3). It then looks to see if the word is in one of the dictionaries or word lists. If so, `SpellCheck` returns `nil`. If the word is not in a dictionary, or if its capitalization is not correct, then `SpellCheck` returns non-`nil`.

Here are some examples:

```
SpellCheck(s, "and") => nil // word is spelled correctly
SpellCheck(s, "And") => nil // word is spelled correctly
SpellCheck(s, "AND") => nil // word is spelled correctly
SpellCheck(s, "ernie") => non-nil // capitalization is wrong
SpellCheck(s, "Ernie") => nil // word is spelled correctly
SpellCheck(s, "ERNIE") => nil // word is spelled correctly
SpellCheck(s, "irs") => non-nil // word needs all caps
SpellCheck(s, "Irs") => non-nil // word needs all caps
SpellCheck(s, "IRS") => nil // word is spelled correctly
SpellCheck(s, "(and.)") => nil // word is spelled correctly
SpellCheck(s, "(bxnd.)") => non-nil // word is not spelled correctly
SpellCheck(s, "isn't") => nil // word is spelled correctly
```

## Spell Checker

```

SpellCheck(s, "isn't") => nil // word is spelled correctly
SpellCheck(s, "so-so") => non-nil // word is not spelled correctly
SpellCheck(s, "ab#de") => non-nil // word is not spelled correctly
SpellCheck(s, "so") => nil // word is spelled correctly

```

**SpellCorrect**


---

```
SpellCorrect(speller, word)
```

Returns a list of correct alternates for a word.

<i>speller</i>	The frame returned by the <code>SpellDocBegin</code> function.
<i>word</i>	A string containing a word to correct.
return value	An array of strings containing correct alternates for <i>word</i> . The value <code>nil</code> or an empty array might also be returned if there are no alternates found.

**DISCUSSION**

This function first processes the word as described in “Processing of Words Passed to the Spell Checker” (page 5-3). It then uses the resulting word to generate a list of possible correct alternates for the word. If it finds no alternates, it returns either `nil` or an empty array.

The returned list contains up to 7 alternates, ordered by their similarity to the original word. Each alternate is punctuated as was the original word, and it is also capitalized like the original (unless fixing the capitalization was part of the problem), so the returned alternates can be directly substituted for the original word.

Note that `SpellCorrect` can suggest alternates for correctly spelled words, even though it is normally used only for words that `SpellCheck` flagged as incorrect.

The following examples provide an indication of the types of corrections that `SpellCorrect` makes:

```

SpellCorrect(s, "bxnd") => ["band", "bend"...] // letter substitution
SpellCorrect(s, "baxnd") => ["band"] // letter deletion
SpellCorrect(s, "bnd") => ["bond", "band"...] // letter insertion
SpellCorrect(s, "ernie") => ["Ernie", "Renie"...] // capitalization
SpellCorrect(s, "hasn't") => ["hasn't", "isn't"] // curly apostrophe preserved
SpellCorrect(s, "(and.)") => ["(and.)", "(end.)"...] // punctuation preserved

```

## Spell Checker

```
SpellCorrect(s, "looseends") => ["loose ends"] // words split
SpellCorrect(s, "unfourtaneatly") => ["unfortunately"] // phonetic substitution
SpellCorrect(s, "Shes") => ["She's", "Shoes"...] // case preserved
SpellCorrect(s, "SHEP") => ["SHEEP", "SHIP"...] // case preserved
```

**SpellSkip**

---

`SpellSkip(speller, word)`

Adds a word to the list of words that should be skipped (not checked) in this spelling session.

<i>speller</i>	The frame returned by the <code>SpellDocBegin</code> function.
<i>word</i>	A string containing a word to skip.
return value	Undefined; do not rely on it.

**DISCUSSION**

`SpellSkip` is used to add a word to a list of words that should be skipped during the course of spell checking a document, but that should not be added permanently to the user's personal word list. `SpellSkip` processes the word as described in "Processing of Words Passed to the Spell Checker" (page 5-3). It then adds the word to the list of skipped words.

With regard to capitalization, `SpellSkip` stores the word exactly as written.

**SpellLearn**

---

`SpellLearn(speller, word)`

Adds a word to the user's personal word list.

<i>speller</i>	The frame returned by the <code>SpellDocBegin</code> function.
<i>word</i>	A string containing a word to add to the word list.
return value	Returns the word that was learned, so that you can pass it to <code>SpellUnlearn</code> for implementing undo behavior. The value <code>nil</code> is returned if the word was not added to the personal word list because the list is full or some other error occurred.

## Spell Checker

## DISCUSSION

`SpellLearn` processes the word as described in “Processing of Words Passed to the Spell Checker” (page 5-3). It then adds the word to the personal word list. This will cause the word to be recognized as a correctly spelled word in subsequent spell-check sessions.

When you pass this function an unknown capitalized word, or one that is all uppercase letters, then `SpellLearn` displays a slip asking the user to confirm that the word should be stored capitalized (or all uppercase). If the user taps Yes, the word is stored as written. If the user taps No, the word is converted to lowercase before being stored.

This notification slip can be disabled by setting the *speller* frame slot `dialogInhibit` to a non-`nil` value before calling `SpellLearn`. In this case, `SpellLearn` stores the word exactly as it is passed to it.

The personal word list has a limit of 1000 words. Once this limit is reached, `SpellLearn` won’t store any more words and instead displays a notification slip telling the user that the list is full and asking if they want to open the personal word list in order to remove some words. The user can tap Yes to open the word list. You can disable the display of this notification slip by setting the *speller* frame slot `dialogInhibit` to a non-`nil` value before calling `SpellLearn`. In this case, `SpellLearn` simply returns `nil` if the list is full.

**SpellUnlearn**


---

`SpellUnlearn(speller, learnedword)`

Removes a word from the user’s personal word list.

<i>speller</i>	The frame returned by the <code>SpellDocBegin</code> function.
<i>learnedword</i>	A string containing a word to remove from the word list.
return value	Undefined; do not rely on it.

## DISCUSSION

`SpellUnlearn` is typically used to implement undo operations. The following example shows how `SpellUnlearn` is typically used:

```
learnedWord := SpellLearn(sp, word);
SpellUnlearn(sp, learnedWord);
```

## Spell Checker

`SpellUnlearn` should be passed the word returned by the `SpellLearn` call, not the original word that was passed to `SpellCheck` or `SpellLearn`.

If you pass `SpellUnlearn` a word that is not in the personal word list, there is no effect.

## Summary of Spell Checker

---

### Functions

---

SpellDocBegin()  
SpellDocEnd(*speller*)  
SpellCheck(*speller*, *word*)  
SpellCorrect(*speller*, *word*)  
SpellSkip(*speller*, *word*)  
SpellLearn(*speller*, *word*)  
SpellUnlearn(*speller*, *learnedword*)

# Drawing and Graphics 2.1

---

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.

This document only describes changes to the graphics model that existed in the Newton 2.0 OS as described in Chapter 13, “Drawing and Graphics,” in *Newton Programmer’s Guide*.

The following changes have made to this graphics model:

- Support has been added for the new grayscale screens.
- Color PICTs can now be rendered.
- There are two new graphic shapes: ink and text box shapes.
- Graphic shapes can include resize handles.
- Bitmap shapes can now include masks.
- Support has been added to anti-alias reduced black and white bitmaps.
- There are a number of new utility functions, and a number of functions have been altered.

## About Drawing and Graphics in the Newton 2.1 OS

---

This section provides an overview of the graphics capabilities introduced by the Newton 2.1 OS.

### About Gray Tones and Patterns

---

The Newton 2.1 OS provides support for the sixteen shades of gray available on the eMate 300 and the MessagePad 2000 screens. There are constants defined for these 16 gray tones. RGB (red-green-blue) values can also be used to specify a color. The system maps these RGB values to gray tones when needed.

You can also create patterns using these 16 grays. Two new types of patterns have been added, gray patterns and dithered patterns, in addition to the black and white patterns available on earlier systems. A **dithered pattern** is a 1-bit pattern with an associated foreground and background color. It is like an old black and white pattern except that any two gray tones can be used. A **gray pattern** is a pattern containing any number of gray tones.

These gray tones and patterns can be used in graphic shapes, text, pictures, and as a view's fill, frame, or line pattern. Ink and ink text however are always black.

### About Gray Pictures

---

The Newton 2.1 OS can render color PICTs in grayscale. Previous versions could render only black and white PICTs. While the Newton 2.1 OS can process PICTs in up to 32-bit colors, these objects are unnecessarily large. You can save package space by using a graphics program on the desktop machine to convert the PICT to 16 grays using the standard 4-bit palette with.

A PICT specifies its colors as indices to a color table. When NTK or the Newton OS creates a picture, it uses a default color table. By default the indices are used as the values for the gray tones. The value 0 is white,



## Drawing and Graphics 2.1

subsequent values are evenly spaced up to black. Including a color table makes the picture larger, and draw slower.

**Note**

The RGB values in a color PICT are not modified when rendering it in grays. However, the system only writes and creates PICTs in gray tones. This means that if the user creates a new picture from a color one, the new picture will be defined in gray tones. For example, if a color picture is downloaded from a desktop machine to a Newton device, resized or otherwise edited, and then uploaded to the desktop, it will appear in gray on the desktop machine. ♦

## About Gray Bitmaps (Pix Families)

---

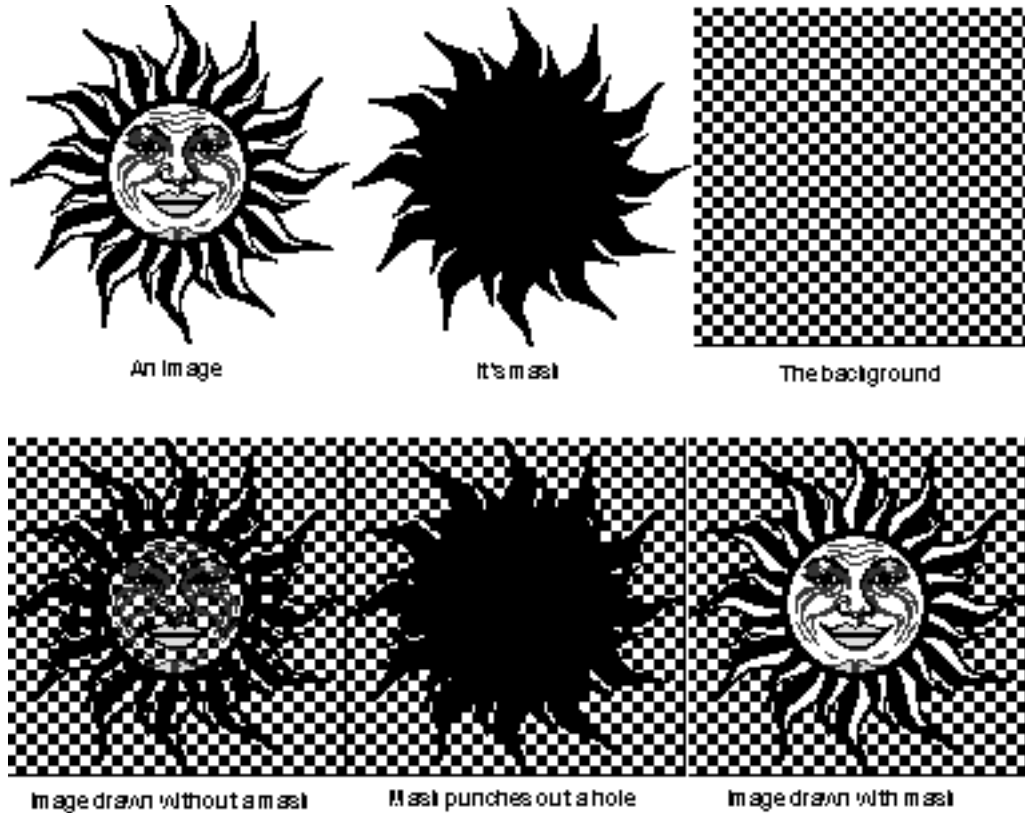
A new data structure called a pix family has been introduced in the Newton 2.1 OS. A **pix family** is a group of one or more bitmaps at various bit depths. The system picks the most appropriate image to display on the current hardware. A pix family can be backwards compatible if 1-bit data is included.

**Note**

Currently support exists for creating a pix family only from the Mac OS version of NTK. Future versions of WinNTK will make pix families. ♦

The term “bitmap” is still used however to refer to the graphic shape produced from a pix family with `MakeShape`. Bitmaps of a 1, 2, an 4-bit depths can also be created with the `MakeBitmap` function.

Pix families and bitmap shapes can include a mask. The mask is used when the bitmap is rendered in the `modeMask` transfer mode. The mask is a 1-bit image used to “punch a hole” in the background before the regular image is rendered. Figure 6-1 illustrates the how a mask is used to punch a hole in the background.

**Figure 6-1** The effect of a mask for a pix family**WARNING**

You must not use the `MungeBitmap` function with the `'rotateLeft'`, `'rotateRight'`, or `'flipHorizontal'` options, the source bitmap is destructively replaced with garbage. You can still use the `'rotate180'` and `'flipVertical'`, however. ♦

## About Gray Extras Drawer Icons

---

Gray icons are added to a part in the new part frame slot `iconPro`. Newton 1.x and 2.0 OS use the `icon` slot. The 2.1 OS displays the icon in the `iconPro` slot, if there is one, otherwise it displays the icon in the `icon` slot. The format of the icon in the `iconPro` slot differs from that of the icon in the `icon` slot. The `iconPro` slot contains two pix families, for normal and highlighted versions of the icon. The highlighted version of the icon is shown when the icon is selected.

For parts without an `iconPro` slot, the Newton 2.1 OS uses the `icon` slot. These icons are highlighted by xor'ing the mask as in Newton 1.x and 2.0 OS, since these icons do not have a highlighted version. However, when displaying old icons in the button bar, the highlighting effect is simply to invert the text label. The xor'ing doesn't work properly over the non-white background of the button bar. Extras Drawer icons should contain a mask for this reason. The mask should be slightly larger than the icon image to provide a vignette effect when on a non-white background.

NTK 1.6.4 provides a special editor for creating gray form part (application) icons. You must do this programmatically for icons of other parts with the NTK 1.6.4 function `MakeExtrasIcons`. See "Creating Gray Extras Drawer Icons" (page 6-15).

### Note

Currently support exists for creating gray Extras Drawer icons only from the Mac OS version of NTK. Future versions of WinNTK will make gray icons. ♦

## About Ink Shapes

---

The new ink shapes, created by `MakeInk`, allow you to treat ink as you would any other graphic shape in the system. Ink objects can now be drawn, stored in a shape array, hit tested, drawn in different styles, and so on, with all the functions that manipulate graphic shapes.

## About Text Box Shapes

---

The new text box shape, created by `MakeTextBox`, provide support for multi-line text shapes. Single line text shapes have been available since the original version of the Newton OS. With the new text box shapes you specify a bounding box and a string, and the system wraps the text at word boundaries, clipping the text if it spills out of the bounding box. An example of a text box shape is shown in Figure 6-7 on page 6-20.

## About Gray Text

---

Support has been added for gray text in two ways. There is a new font spec frame slot named `color` which specifies the gray tone to use when drawing text. For more information on font spec frames, see “Using Fonts for Text and Ink Display” (page 8-17) in *Newton Programmer’s Guide*.

There is also a new style frame slot `textPattern` that is used to draw text within a graphic shape. If the style frame does not contain a `textPattern` slot, text is drawn in the tone specified by the `fillPattern` slot.

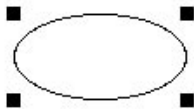
## About Selection Handles

---

**Selection handles** are small squares on each corners of a graphic shape’s bounding box, these are added by including a selection slot in the style frame. These are commonly used in graphics software packages; see Figure 6-2. The system can draw these handles, and supports hit testing of taps on these handles. The system does not, however, perform any action in response to the user’s tap on a selection handle; responding to such a tap is the application’s responsibility.

---

**Figure 6-2** An oval shape with selection handles



## About Anti-Aliasing

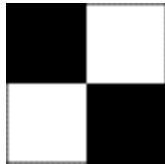
---

When a black and white picture is reduced in size, the resulting image will often have a jagged look. This jagged effect is called aliasing. **Anti-aliasing** is a technique to overcome this effect, by rendering a reduced picture in gray tones.

For example, consider a picture that is being reduced to half its size. The four pixels at the top left corner of the picture are now going to be represented by a single pixel. These four pixels could look like those shown in Figure 6-3.

---

**Figure 6-3** Four black and white pixels



If these pixels had been all black, it would make sense to render them with a single black pixel. But since these four pixels are not all black (or all white), rendering them as a single black or a single white pixel, creates a picture with a jagged, aliased, look. This can be overcome by rendering these four pixels as a single pixel in a tone of gray.

Support has been added to `protoImageView` to automatically anti-alias reduced black and white bitmaps. The built-in fax viewer, for example, uses `protoImageView`.

You may also anti-alias monochrome bitmaps with the new view method `GrayShrink`. The `GrayShrink` method was used to anti-alias the text on Figure 6-4 (page 6-8).

## Drawing and Graphics 2.1

**Figure 6-4** The anti-aliasing effect on a bitmap that has been reduced by 50%

**The Newton platform incorporates a sophisticated preemptive, multitasking operating system. The operating system is a modular set of tasks performing functions such as memory management, task management, scheduling, task to task communications, input and output, power management, and other low-level functions. The operating system manages and interacts directly with the hardware.**

The Newton platform incorporates a sophisticated preemptive, multitasking operating system. The operating system is a modular set of tasks performing functions such as memory management, task management, scheduling, task to task communications, input and output, power management, and other low-level functions. The operating system manages and interacts directly with the hardware.

## Compatibility

The PostScript driver in Newton OS versions prior to 2.1 substituted actual gray tones for dithered patterns such as `vfGray`, `vfLtGray`, etc. This substitution is no longer performed, so gray views that previously printed fine might now look awkward in black and white dots.

Text is always drawn in black in Newton 1.x and 2.0 OS. A new style frame slot `textPattern` has been added. Furthermore, if `DrawShape` draws text without a `textPattern` slot, it is drawn in the tone specified by `fillPattern`. This second change can cause compatibility problems, text in a shape array drawn with a `vfWhite` `fillPattern`, for example, is displayed in Newton 1.x and 2.1 OS, but the text is not shown in Newton 2.1 OS.

The following functions are new to Newton 2.1 OS: `FindShape`, `GetMaskedPixel`, `GetBlue`, `GetGreen`, `GetPointsArrayXY`, `GetRed`, `GetTone`, `view:GrayShrink`, `IsEqualTone`, `MakeInk`, `MakeTextBox`, `MungeShape`, `PackRGB`, and `PictToShape`.

The following functions have been changed in Newton 2.1 OS: `GetStrokePointsArray`, `MakeBitmap`, and `MakeShape`.

## Using Drawing and Graphics in the Newton 2.1 OS

---

This section describes how to use the programmer's interface to the Newton drawing engine.

### Specifying Shades of Gray

---

The following constants are defined which specify 4-bit grayscale values:

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

Black is `kRGB_Gray15`, and can also be referred to by the constant `kRGB_Black`. White is `kRGB_Gray0`, for which the constant `kRGB_White` is also defined. These values can be used anywhere a color needs specifying, in a graphic shape, in a font spec frame, and as a view's fill, frame, or line pattern. The sixteen gray tones are shown in Figure 6-5.

**Figure 6-5** The 4-bit grayscale palette



The constant `kRGB_16GrayIncrement` is also quite useful. This constant equals the difference between two gray tones. For example, the following expression evaluates to true:

```
kRGB_Gray3 = (3 * kRGB_16GrayIncrement) + kRGB_Black;
```

## Drawing and Graphics 2.1

## Specifying RGB Triplets

---

You can specify a gray tone as an RGB (red-green-blue) triplet. These are mapped to gray tones at run time. They can be used anywhere the `kRGB_GrayXX` values are used. RGB triplets are represented as packed integers.

There are a number of utility functions provided to deal with packed RGB integers:

- **PackRGB** takes three 16-bit integers (that is, integers in the range [0,65535]) specifying the red, green, and blue components, and returns a packed RGB integer. The following example uses this function:

```
myView.viewFillPattern := PackRGB (0x8888, 0xFFFF, 0x21AA);
```

- **GetTone** takes a packed RGB integer and returns the tone of gray the RGB triplet maps to. It is this gray tone that is displayed on the screen. For example, the following two expressions evaluate to `true`:

```
0 = GetTone( PackRGB (0xFFFF, 0xFFFF, 0xFFFF) );    //white
1 = GetTone( PackRGB (0xFFFF, 0xCCCC, 0xFFFF) );    //a very light gray
```

- **IsEqualTone** takes two packed RGB integers and returns `true` if they map to the same gray tone. The following code illustrates the use of this function:

```
IsEqualTone( PackRGB(0,0,0), PackRGB(2,7,88) );      // returns true
IsEqualTone( PackRGB(0,0,0), PackRGB(2000,7000,8800) );// returns nil
```

- **GetRed**, **GetGreen**, and **GetBlue** take a packed RGB triplet and return the relevant color component as an integer in the range [0,65535]. The following example illustrates the use of one of these functions:

```
local thePackedInt := PackRGB( 0, 0x1111, 0xFFFF);
GetBlue (thePackedInt); // this returns an integer close to 0xFFFF
```

### Note

`GetRed(PackRGB(r,g,b))` might not return `r`. All that is guaranteed is that the return value of this function call is an integer close to `r`. ♦

- **UnPackRGB** takes a packed RGB triplet and returns a frame with red, green, and blue slots. This function is provided by the NTK environment, and is



## Drawing and Graphics 2.1

thus available at build time only. The following example illustrates the use of this function:

```
UnPackRGB(PackRGB(r,g,b)); //returns integers close to r, g, and b
```

## Using Patterns, Gray Patterns, and Dithered Patterns

---

The following sections provide information about the three types of patterns that can be drawn. You should read “Black and White Patterns” even if you want to create one of the other patterns.

### Black and White Patterns

---

A black and white pattern is specified as a 8-byte binary object of class 'pattern, representing an 8x8 bitmap. The system has five built-in patterns, which you can reference through the constants `vfWhite`, `vfLtGray`, `vfGray`, `vfDarkGray`, and `vfBlack`. You may also define your own patterns.

To create a pattern, use the NTK function `MakeBinaryFromHex`. It takes a class symbol and a sting with an even number of hex digits, each set of two digits defining a byte in the binary object.

The following example creates a simple striped pattern, and stores it in a constant, since `MakeBinaryFromHex` is available at build-time, but not at run-time):

```
DefineGlobalConstant ( 'kMyBlackAndWhitePattern,
    MakeBinaryFromHex ("AAAAAAAAAAAAAAAA", 'pattern) );
```

Each A has the binary representation 1010, making for the following 8x8 bitmap:

```
10101010
10101010
10101010
10101010
10101010
10101010
10101010
10101010
```

## Gray Patterns

---

Gray patterns are binary objects with the class `'grayPattern`. A gray pattern consists of an 8x8 pattern of pixels, each of which is specified as an RGB triplet. Each color component is specified with two bytes, making for 6 bytes per pixel. You do not, however, need to specify all 64 RGB triplets. The following rules are used when a gray pattern has less than 64 pixels:

- If there are less than 8 pixels: the defined pixels are repeated until an 8 pixel line has been completed. This line is repeated 8 times.
- Otherwise, the pixels are divided into 8 pixel lines, discarding any left over pixels. These lines are repeated as needed to create an 8 line pattern.

The following example creates a 1-pixel pattern in a dark tone of gray:

```
DefineGlobalConstant ( 'kMyOnePixelGrayPattern,
    MakeBinaryFromHex ("AAAAFFFF6666" , 'grayPattern) );
```

This next example creates a striped pattern as in “Black and White Patterns” (page 6-11), using 2 pixels in different tones of gray:

```
DefineGlobalConstant ( 'kMyTwoPixelGrayPattern,
    MakeBinaryFromHex ("999999999999555555555555", 'grayPattern) );
```

### Note

A gray pattern can be a very large object. Specify only as much of a pattern as you need, and try to keep your patterns simple. With a simple pattern you frequently can take advantage of the duplication done by the system when less than 64 pixels are defined. Also consider using a dithered pattern if you only need two tones of gray. A dithered pattern requires about as much memory as a gray pattern with five pixels defined. ♦

## Dithered Patterns

---

If you need a two-toned pattern, you can use the `'ditherPattern` class to create a “black and white” pattern, and assign one tone of gray to the “black” pixels, and another tone to the “white” pixels. A dithered pattern is defined as a frame of the following format:

## Drawing and Graphics 2.1

```
{
    class: 'ditherPattern,
    pattern: aBlackAndWhitePattern, // a 'pattern (e.g. vfGray)
    foreground: kRGB_Gray0,           // kRGB_Gray0 through kRGB_Gray15
    background: kRGB_Gray15,         // kRGB_Gray0 through kRGB_Gray15
}
```

The `pattern` slot contains a black and white pattern object as described in “Black and White Patterns” (page 6-11); this includes the built-in black and white patterns of the `vfGray` family. The `foreground` slot defines the tone of gray of the black pixels (the 1’s), and the `background` slot defines the tone of the white pixels (the 0’s).

The `MakeDitheredPattern` function creates frames of this format. You should use this function instead of creating your own frame, as this ensures that the frame map is shared.

The following example creates the striped black and white pattern with two shades of gray; this pattern has fatter stripes than the one in “Black and White Patterns” (page 6-11):

```
DefineGlobalConstant ( 'kMyStripedBWPattern,
    MakeBinaryFromHex ("F0F0F0F0F0F0F0", 'pattern) );

DefineGlobalConstant ( 'kMyDitheredPattern ,
    MakeDitheredPattern(kMyStripedBWPattern, kRGB_Gray3, kRGB_Gray9) );
```

## Creating Gray Text

---

There are a few ways in which text can be drawn in gray tones. A new font spec frame slot named `color` has been added. This slot can be set to any gray tone, and used anywhere a font spec frames are used, including within a graphic shape’s style frame’s `font` slot.

A graphic shape’s style frame can also contain a new slot, `textPattern`. If this slot is present, text in text shapes will be drawn in the specified tone or pattern. If this slot is not present, text is drawn in the tone specified by the `fillPattern` slot. (Note that it is the `fillPattern` slot, and not the `penPattern` slot that is used.)

Text shapes are always drawn in black on Newton 1.x and 2.0 OS.

## Importing Color PICTs from the Mac OS Version of NTK

---

You can use the Mac OS version of NTK to import PICTs into your package as either a picture object or as a pix family. The following sections describe what you can do with a picture object or pix families.

### Creating Graphic Shapes from Picture Objects

---

A picture object, sometimes called a Newton PICT, is an object that consists of primitive drawing commands. It is basically the same data structure as a Mac OS PICT. In most cases, a picture object is smaller than a pix family, but may draw slower.

You can incorporate a picture object in your package using the existing NTK functions `GetResource` and `GetNamedResource`. Once a picture object has been included in a package you can render it by converting it to a graphic shape. 16 and 32 bit PICTs can only be imported with these two functions. You cannot create a pix family from such a PICT.

There are two ways to transform the picture object into a graphic shape. You can use the existing function `MakeShape`, or the new function `PictToShape`. These functions use different algorithms to transform the picture object to a shape. `MakeShape` returns a single picture shape, which is basically a wrapper around the PICT. `PictToShape` on the other hand, returns an array of shapes derived from the PICT's own drawing commands.

#### Note

The PICT itself could contain either drawing commands, such as “draw a circle here,” “fill this rectangle in black,” or it could contain bitmap-based data. If the PICT contains drawing commands, `PictToShape` creates an array of shapes based on those commands, which is generally a smaller object. However if the PICT contains bitmap-based information, use of these two functions is basically equivalent. ♦

### Using Pix Families

---

You can create a pix family from a PICT by using either the picture slot editor in NTK, or programmatically with the NTK function `MakePixFamily`.

## Drawing and Graphics 2.1

`MakePixFamily` replaces the deprecated function `GetPictAsBits`. Pix families can be used anywhere bitmaps are allowed. They can be rendered when used as the value of the `icon` slot of a `clPictureView`, or in the `pickItems` array of a popup menu item, etc. A pix family can be drawn directly on the screen with the `CopyBits` function. You can create a bitmap shape from the pix family with `MakeShape`.

The picture slot editor is provided for the `icon` slot of `clPictureViews`. It creates a pix family. For more information on this slot editor, see “Picture Slot Editor” (page A-1). The existing NTK function `GetPictAsBits`, has not been modified, and returns an old-style black and white bitmap. If you want to create a pix family programmatically, you must use the new NTK function `MakePixFamily`.

## Creating Gray Extras Drawer Icons

---

Gray icons are created as a pair of normal and a highlighted icons. Each of these icons is a pix family. Gray icons are stored in a part frame’s `iconPro` slot. To remain compatible with previous system versions, the part must also contain an `icon` slot with black and white data.

Note that icons in the button bar are clipped if they are too large. Also, since the button bar is gray, you must include a mask with your icon. The mask should be lightly larger than the icon. The size of the mask must be no larger than 29x32, and the size of the regular icon must be less than 27x30. For more information on how to create icons, see Chapter 5, “Icons,” in *Newton 2.0 User Interface Guidelines*.

### Note

The size of all PICTs in a pix family must be identical. That is, they must be selected with the same size selection rectangle. An icon’s mask should be larger in the sense of having a larger image, that is the mask should have more bits than the icon has non-white pixels. ♦

Icons for form parts (application parts) can be created with NTK’s application icon editor. For information on this editor, see “Application Icon Editor” (page A-3). For other kind of parts (auto parts, store parts, etc.) you must create them programmatically with the `MakeExtrasIcons` function, and add them to your part frame with the `SetPartFrameSlot` function. The code

## Drawing and Graphics 2.1

example in Listing 6-1 shows how to assign an icon to a part programmatically.

---

**Listing 6-1** Code to add an icon and iconPro slot to a part frame

```
// example giving an icon to a non-form part - works for 2.0 and
// later devices (NTK's Project Setting handles this for form parts)

OpenResFile(HOME & "foo.rsrc");

extrasIcons := MakeExtrasIcons([
    {unhilitedRsrcSpec: "foo.bw",           //b&w normal icon
     hilitedRsrcSpec: "foo.bw.hilited",    //b&w hilited icon
     bitDepth: 1},
    {unhilitedRsrcSpec: "foo.gray",        //gray normal icon
     hilitedRsrcSpec: "foo.gray.hilited",  //gray hilited icon
     bitDepth: 4},
    ],
    "foo.mask",                          //mask for normal icons
    "foo.hilited.mask"                   //mask for hilited icons
);

SetPartFrameSlot('iconPro, extrasIcons.iconPro);
SetPartFrameSlot('icon, extrasIcons.icon);
SetPartFrameSlot('text, "foo"); // must have a text slot or the part
                                //is ignored by the Extras Drawer

CloseResFile();
```

## Anti-Aliasing Monochrome Bitmaps

---

There are two ways in which black and white bitmaps can be anti-aliased when being reduce in size. Bitmaps in `protoImageView` views are anti-aliased automatically if the view contains a `drawGrayScaled` slot set to true. You can also anti-alias bitmaps programmatically with the new `GrayShrink` function.

**IMPORTANT**

The image slot of a `protoImageView` can contain a variety of objects. However, you may only set the `drawGrayScaled` slot to true if the image slot contains a bitmap shape. ♦

## Drawing and Graphics 2.1

`GrayShrink` accepts two arguments, the bitmap to draw and a style frame. The style frame must contain a `transform` slot, and the transformation must represent a reduction in size either horizontally or vertically. `GrayShrink` renders the bitmap on the screen anti-aliased. You use it instead of using a combination of `MakeShape` and `DrawShape`. If `GrayShrink` is not passed a 1-bit bitmap, and a style frame with a `transform` slot representing a reduction on either axis, the bitmap is not anti-aliased, but still drawn on the screen.

**Note**

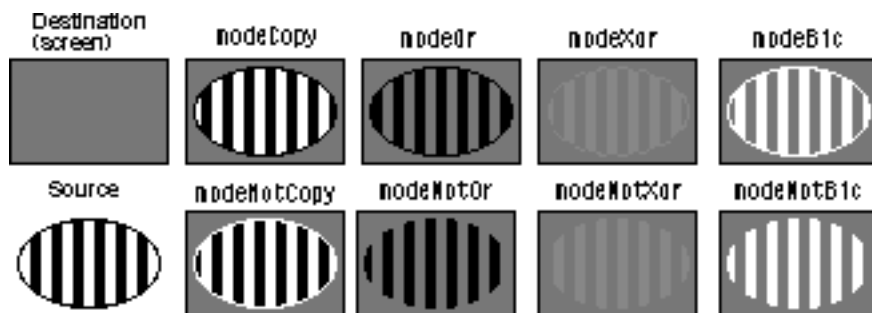
The anti-aliasing algorithm is somewhat time expensive. You should not use `GrayShrink` indiscriminately. ♦

## Gray Transfer Modes

The transfer modes are used to specify how a graphic object is drawn onto an existing bitmap, usually the screen. The system combines the two images at the pixel level, combining the images with the operation specified in the transfer mode. The transfer mode constants are described in “Transfer Mode Constants” (page 6-24).

The effect of the different transfer modes is illustrated in Figure 6-6.

**Figure 6-6** Two bitmaps combined with the different transfer modes



The value of the individual pixels is determined by first looking up the value in the color lookup table. If the default color table is used, no lookup is

## Drawing and Graphics 2.1

required as the indices and the tones are identical. Secondly, the source bitmap must be changed to the bit depth of the destination bitmap. Then the source bitmap is scaled, if necessary. Finally the individual pixels are combined as indicated by the transfer mode.

In most cases, you do not need to worry about how it is that the source bitmap is scaled (in both size and bit depth). Some of the transfer modes, `modeXor` and `modeBic`, operate by a bitwise combination of the pixel values. These transfer modes are most useful when using black and white sources, and have unexpected results when combining grays. Using `modeBic` is especially useful for drawing white text over a black or gray background.

If you do wish to use these transfer modes with two shades of gray, and you want to understand what the result will be, you must consider the effect of scaling the source bitmap. This information is provided in “How the System Scales Bitmaps.”

### How the System Scales Bitmaps

---

Before two bitmaps can be merged, the two bitmaps must be made to agree in bit depth, and the source bitmap may need to be scaled. The source bitmap is always the one to be shrunk or enlarged to match the destination bit depth.

If the source bitmap needs to be shrunk in size, it is partitioned into as many groups of pixels as are desired, and the darkest pixel in each source group is used. For example, if a 4-bit scan line contains the indices `0x56 34 A2 C8...`, and is being shrunk 50%, the indices `0x64AC...` are used. The darkest of each 2 pixel group is used. Since the following scan line of source data will also be combined with the current line, this algorithm is repeated, and the darkest of each 2x2 block of source pixels is mapped to each destination pixels.

When the source bitmap is expanded to fit a larger destination bitmap, pixels are repeated as necessary. For example if the same `0x56 34 A2 C8...` scan line is being increased horizontally by 100%, the individual pixels are simply repeated to produce `0x55663344AA22CC88...`

When the bit depth of the source bitmap is reduced or increased, it is necessarily changed by a power of 2. If the bitmap is being expanded, the pixel's bit patterns are simply repeated. For example if a pixel with the 2-bit binary value `01` is being increase to 4-bits, the zero is repeated and the one is



## Drawing and Graphics 2.1

repeated to produce the binary value 0011. The 2-bit pixel 01 is expanded to the 8-bit pixel 00001111.

If a bitmap is being reduced in bit depth, the high bits of each pixel is used. For example, an 8-bit scan line 0x08 17 28 A5... is reduced to the four-bit scan line 0x012A... As a further example, the 8-bit pixel 0xA2 (binary 10100010) is reduced to the 4-bit pixel 1010 and to the 2-bit pixel 10.

Once the two bitmaps have been made to agree in bit depth and size, the affect of the transfer modes can be established. These are described in “Transfer Mode Constants” (page 6-24).

## Using Selection Handles

---

The new style frame slot `selection` signals that a shape contains selection handles. This slot must contain an integer, which is the size in pixels of the resize handles. It is recommended that the `selection` slot be set to an even integer, it tends to look better. `DrawShape` draws the selection handles around shapes, and `FindShape` can be used for hit testing on these handles.

## Creating Ink and TextBox Shapes

---

Two functions have been added to create ink and text box shapes: `MakeInk` and `MakeTextBox`. `MakeInk` takes five parameters, an ink data object and four integer coordinates for the bounding box. `MakeTextBox` also takes five parameters, a string and four integers for the bounding box.

The following code example illustrates the use of `MakeTextBox`, producing the shape displayed in Figure 6-7:

```
local tb := MakeTextBox ("Note how text is automatically wrapped at
word boundaries, and clipped at shape bounds.", 0, 0, 150, 123);
:DrawShape (tb,nil);
```

**Figure 6-7** A textBox

**Note how text is  
automatically  
wrapped at word  
boundaries, and  
clipped at shape  
bounds**

The code example in Listing 6-2 illustrates the use of the `MakeInk` function. This code sample is a method of a `clEditView` that creates a shape array with all the ink children of the view.

**Listing 6-2** Function to retrieve ink shapes from a `clEditView`

```
myEditView.CollectInkShapes := func()
begin
    local kids := :ChildViewFrames();
    local inkShapes := [];

    foreach child in kids do
        if child.ink then
            begin
                local bounds := child.viewBounds;
                AddArraySlot( inkShapes,
                            MakeInk(child.ink,bounds.left,bounds.top,
                                    bounds.right, bounds.bottom));
            end;
        inkShapes;
    end;
```

## New Graphic Shape Utility Functions

---

This section defines a few new utility functions.

## The FindShape Function

---

The new `FindShape` function has been provided to overcome some of the deficiencies of `HitShape`. Since changing `HitShape` would break existing code, this new function has been introduced. `HitShape` is now a deprecated function, use `FindShape` from now on.

This new function differs from `HitShape` in the following ways:

- Shapes are found from front to back instead of back to front. If two shapes overlap and you call `HitShape` with a point in their intersection, the backmost shape was returned. `FindShape` returns the frontmost shape.
- Support is provided for hit testing resize handles.
- `FindShape` takes a style frame as an argument so you can use it the same way you do `DrawShape`. If you draw a rectangle on the screen with a style frame that moves it 20 pixels to the right, you are probably interested in whether a pen tap is within the rectangle where it is drawn on the screen, not in its original position.
- If a shape has no fill pattern the tap “falls through” and will miss, or hit a shape below it.
- There is “slop” built into the hit testing so taps a few pixels from a shape still hit the shape.
- The new ink and text box shapes are supported.

## The GetPointsArrayXY Function

---

The `GetPointsArray` function has existed since the Newton 1.0 OS. It accepts a unit, which is passed in to a `ViewWordScript`, `ViewStrokeScript`, or `ViewGestureScript` method, and returns an array of points in that unit. It returns the points in (y,x) order. That is, the first array element is the first point’s y coordinate, the second is its x coordinate, the third is the next point’s y coordinate, and so on. The new function `GetPointsArrayXY` does the same thing, only that the coordinates are in (x,y) order.

## The MungeShape Function

---

The `MungeShape` function can flip a shape or rotate it 90° to the right. The `MungeBitmap` function has existed since Newton 2.0 OS, it similarly flips and rotates bitmaps. The following code example illustrates the use of the `MungeShape` function:

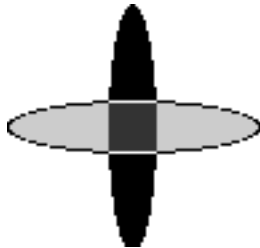
```
//this is in a ViewDrawScript
local oval := MakeOval (5,5,100,25); //a short and fat oval
:DrawShape(oval, {fillPattern: kRGB_Gray3});

//now draw it tall and skinny on top of the original oval
MungeShape(oval, 'rotateRight, nil);
:DrawShape(oval, {fillPattern: kRGB_Black, transferMode: modeXor});
```

The above code generated Figure 6-8 on page 6-22.

---

**Figure 6-8**      Overlapping ovals



## The GetMaskedPixel Function

---

`GetMaskedPixel` augments the existing function `PtInPicture`. `PtInPicture` determines whether a specific point is a black pixel in a black and white bitmap. `GetMaskedPixel` returns the value of a pixel in a color picture. This function considers the effect of a mask.

## Changes to Existing Graphic Shape Functions

---

### MakeBitmap Accepts a Depth Option

---

The third parameter to the `MakeBitmap` function is an options frame. This frame may now include a `depth` slot containing an integer. This integer represents the bit depth of the bitmap; allowable values are 1, 2, and 4.

### MakeShape Makes Bitmap Shapes With Masks

---

In previous version of the Newton OS, bitmap objects could contain masks, but not bitmap graphic shapes. You can now create a bitmap graphic shape with a mask with the `MakeShape` function.

An additional slot, `mask`, is added to the bitmap shape if `MakeShape` is passed a `pix` family which contains a mask. This mask is used when the bitmap shape is drawn with the `transferMode` set to `modeMask`.

### GetStrokePointsArray Filters More Points and Swaps Coordinates

---

The `GetStrokePointsArray` function has been modified in two ways. It can now filter more points by requiring a minimum distance between adjacent points. You can now also specify whether points' coordinates are returned in (y,x) order or (x,y) order.

## Drawing and Graphics Reference

---

### Constants

---

Constants have been added as values for the sixteen supported gray tones. The existing constants used for transfer modes have new interpretations.

## Drawing and Graphics 2.1

## Gray Tone Constants

---

The following gray tone constants are defined:

`kRGB_Gray0`, `kRGB_Gray1`, ..., `kRGB_Gray15`

These values define the 16 supported gray tones shown in Figure 6-5 (page 6-9). The constants `kRGB_Black` and `kRGB_White` are also defined and equal `kRGB_Gray15` and `kRGB_Gray0`, respectively.

The constant `kRGB_16GrayIncrement` equals the difference between two gray tones. For example, the following expression evaluates to `true`:

```
kRGB_Gray3 = (3 * kRGB_16GrayIncrement) + kRGB_Black;
```

## Transfer Mode Constants

---

The transfer mode constants are used as values to the style frame slot `transferMode`, and the view slot `viewTransferMode`. For more information on how these are used, see “Gray Transfer Modes” (page 6-17).

<code>modeCopy</code>	The source pixel is drawn over the destination pixel.
<code>modeOr</code>	Non-white pixels are drawn over the destination pixels, but the destination pixels under white pixels are left untouched.
<code>modeXor</code>	The pixels are combined with a bitwise XOR, exclusive or, operation. This can create unexpected results when used with mid-level grays. For example, the result of combining a <code>0xA</code> pixel (75% gray) with a <code>0x7</code> pixel (about 50% gray) is <code>Bxor(0xA, 0x7) = 0xD</code> which is a dark gray. This mode is most often used with a black and white source image. The black pixels invert the destination pixels and the white pixels have no effect on the destination bitmap.
<code>modeBic</code>	<p>The pixels are combined with a bitwise BIC, bit clear, operation. The BIC operation’s truth table is show in Table 6-1 (page 6-25).</p> <p>This mode is most useful when using a black source image to erase the destination bitmap. It is also useful when drawing white text on a dark background, set the</p>

Drawing and Graphics 2.1

	<code>textPattern</code> or <code>fillPattern</code> slots for the text to black. It can have strange effects when combining two gray pixels. For example, source <code>0xA</code> and destination <code>0xB</code> produce <code>0x1</code> , but source <code>0xA</code> and destination <code>0x5</code> produce <code>0x5</code> .
<code>modeNotCopy</code>	The source image is inverted and then merged with destination bitmap using <code>modeCopy</code> .
<code>modeNotOr</code>	The source image is inverted and then merged with destination bitmap using <code>modeOr</code> .
<code>modeNotXor</code>	The source image is inverted and then merged with destination bitmap using <code>modeXor</code> .
<code>modeNotBic</code>	The source image is inverted and then merged with destination bitmap using <code>modeBic</code> .
<code>modeMask</code>	The bitmap's mask is drawn with <code>modeBic</code> , then the "normal" bitmap image is drawn in <code>modeOr</code> .

**Table 6-1** Truth table for `modeBic`

Source	Destination	BIC
0	0	0
0	1	1
1	0	0
1	1	0

## Data Structures

The style frame has changed in Newton 2.1 OS.

### Style Frame

Style frames, as used by the `DrawShape` and other functions, may contain the following slots (new or changed slots are listed first):

## Drawing and Graphics 2.1

**Slot descriptions**

<code>textPattern</code>	New slot. The shade of gray to render text in. 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. Possible values for this slot are the constants specified in “Gray Tone Constants” (page 6-24), or the return value of <code>PackRGB</code> .
<code>selection</code>	New slot. The size of the selection handles in pixels. If this slot is present selection handles are 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 <code>FindShape</code> function supports hit testing of these handles.
<code>penPattern</code>	<p>Changed slot. The pen pattern. Possible values for this slot are the constants specified in “Gray Tone Constants” (page 6-24), the return value of <code>PackRGB</code>, or a pattern, gray pattern, or dithered pattern.</p> <p>The Newton 2.0 OS draws text in a dithered pattern if a gray pattern (such as <code>vfGGray</code>) is specified. The Newton 2.1 OS draws the text in the proper tone of gray when <code>kRGB_GrayXX</code> constants, or a gray or dithered pattern are used.</p>
<code>fillPattern</code>	<p>Changed slot. The Newton 2.0 fills OS areas in a dithered pattern if a gray pattern is specified. The 2.1 OS fills the area in the proper tone of gray when <code>kRGB_GrayXX</code> constants, or a gray or dithered pattern are used. Possible values for this slot are the constants specified in “Gray Tone Constants” (page 6-24), the return value of <code>PackRGB</code>, or a pattern as defined in “Patterns” (page 6-28).</p>
<code>font</code>	<p>Changed slot. The font to use for drawing text. The default is the font selected by the user in the Styles palette. In Newton 2.1 OS, this slot can also contain a <code>color</code> slot, which is ignored by earlier systems. The color slot can contain a <code>kRGB_GrayXX</code> constant, or a packed RGB integer as returned by <code>PackRGB</code>. See “Fonts</p>



## Drawing and Graphics 2.1

	for Text and Ink Display” (page 8-3) in <i>Newton Programmer’s Guide</i> for details on specifying a font.
<code>transferMode</code>	Changed slot. The way a drawing is merged with the existing background. Specify one of these constants listed in “Transfer Mode Constants” (page 6-24). The default transfer mode is a split state: bitmap shapes and text are drawn with a <code>modeOr</code> transfer mode, but other items (geometric shapes, pens, and fill patterns) are drawn with a <code>modeCopy</code> transfer mode.
<code>penSize</code>	Exiting slot. The size of the pen in pixels. You can specify a single integer to indicate a square pen of the specified size, or you can specify an array giving the pen width and height (for example, <code>[1, 2]</code> ). This value is not used for drawing text. The minimum and default pen size is 1. However, no frame will be drawn for a shape if <code>penPattern</code> is set to <code>vfNone</code> (the default <code>penPattern</code> is <code>vfBlack</code> ).
<code>justification</code>	Exiting slot. The alignment of text in the rectangle specified for it. Specify one of the following symbols: <code>'left'</code> , <code>'right'</code> , <code>'center'</code> . The default value is <code>'left'</code> .
<code>clipping</code>	Exiting slot. Specifies a clipping region to which all drawing is clipped in addition to the default clipping. The value of this slot can be a primitive shape, a region, or an array of shapes (from which a new clipping region is constructed automatically by the system). For more information see “Controlling Clipping” (page 13-12) in the <i>Newton Programmer’s Guide</i> .
<code>transform</code>	Exiting slot. Used to offset or scale the shape. The value of this slot is an array that can hold a coordinate pair or a pair of source and destination rectangles. For more information, see “Transforming a Shape” (page 13-13) in the <i>Newton Programmer’s Guide</i> .

## Drawing and Graphics 2.1

## Patterns

---

Three types of patterns are supported: patterns, gray patterns, and dithered patterns.

### Pattern

---

A pattern is a binary object containing an 8x8 bitmap of class 'pattern. The constants `vfWhite`, `vfLtGray`, `vfGray`, `vfDkGray`, and `vfBlack` are patterns in the ROM. For more information, see “Black and White Patterns” (page 6-11).

### Gray Pattern

---

A gray pattern is a binary object of class 'grayPattern containing 1 or more pixels in an 8x8 pixel map of class 'pattern. Each pixel is specified as an RGB triplet. Two bytes are used per color component, making for 6 bytes per pixel. For more information, see “Gray Patterns” (page 6-12).

### Dithered Pattern

---

A dithered pattern is a frame with the following slots:

**Slot description**

<code>class</code>	The symbol 'ditherPattern.
<code>pattern</code>	A black and white pattern, see “Pattern” (page 6-28).
<code>foreground</code>	A gray tone to use wherever the pattern in the <code>pattern</code> slot contains an on pixel, such as one of the constants listed in “Gray Tone Constants” (page 6-24).
<code>background</code>	A gray tone to use wherever the pattern in the <code>pattern</code> slot contains an off pixel, such as one of the constants listed in “Gray Tone Constants” (page 6-24).

You should use the `MakeDitheredPattern` function to create dithered patterns. This ensures that dithered pattern does not have its own frame map. For more information, see “Dithered Patterns” (page 6-12).

## Functions and Methods

---

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

### FindShape

---

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

Indicates whether the point (x,y) lies in the specified shape if it were drawn in the specified style.

<i>shape</i>	The shape or array of shapes to test.				
<i>x</i>	The <i>x</i> coordinate of the point to be tested, in local (view) coordinates. Note that <code>GetPoints</code> returns global coordinates.				
<i>y</i>	The <i>y</i> coordinate of the point to be tested, in local (view) coordinates. Note that <code>GetPoints</code> returns global coordinates.				
<i>style</i>	A style frame to be applied to <i>shape</i> .				
return value	Returns <code>nil</code> if no shape is found, otherwise a frame with the following slots: <table> <tr> <td><i>path</i></td><td>If <i>shape</i> is a single shape, then the Boolean <code>true</code>. If <i>shape</i> is an array of shapes, then a path expression to the shape within the array.</td></tr> <tr> <td><i>vertex</i></td><td>If the hit was not on a resize handle, this slot is set to <code>nil</code>. If a handle was hit, this slot contains an integer with the following meaning:               <ul style="list-style-type: none"> <li>0 = top-left corner</li> <li>1 = top-right corner</li> <li>2 = bottom-right corner</li> <li>3 = bottom-left corner</li> </ul> </td></tr> </table>	<i>path</i>	If <i>shape</i> is a single shape, then the Boolean <code>true</code> . If <i>shape</i> is an array of shapes, then a path expression to the shape within the array.	<i>vertex</i>	If the hit was not on a resize handle, this slot is set to <code>nil</code> . If a handle was hit, this slot contains an integer with the following meaning: <ul style="list-style-type: none"> <li>0 = top-left corner</li> <li>1 = top-right corner</li> <li>2 = bottom-right corner</li> <li>3 = bottom-left corner</li> </ul>
<i>path</i>	If <i>shape</i> is a single shape, then the Boolean <code>true</code> . If <i>shape</i> is an array of shapes, then a path expression to the shape within the array.				
<i>vertex</i>	If the hit was not on a resize handle, this slot is set to <code>nil</code> . If a handle was hit, this slot contains an integer with the following meaning: <ul style="list-style-type: none"> <li>0 = top-left corner</li> <li>1 = top-right corner</li> <li>2 = bottom-right corner</li> <li>3 = bottom-left corner</li> </ul>				

## Drawing and Graphics 2.1

## DISCUSSION

This function replaces the `HitShape` function. While `HitShape` is still defined in the ROM for compatibility reasons, you should use `FindShape` from now on.

The *shape* and *style* parameters are intended to be used in the same manner as in the `DrawShape` method. Style frames within a shape array in the *shape* parameter are also taken into effect. This allows you to use the same code to create shapes and style frames in both your `ViewDrawScript` (to pass to `DrawShape`) as well as in your `ViewClickScript` (or wherever you call `FindShape` from).

More than one shape in a shape array could encompass a point. In this case the frontmost shape is returned. The frontmost shape is the one in a later position in the shape array.

If a shape has no fill pattern, the hit “falls through” and will miss, or hit a shape below it. Also, there is “slop” built into the hit testing; taps a few pixels from a shape still hit the shape.

**GetBlue**


---

`GetBlue (packedRGB)`

Returns the value of the *packedRGB* argument’s blue component.

*packedRGB*                      A packed integer representation of an RGB color.

return value                    An integer in the range [0,65535].

## SPECIAL CONSIDERATIONS

`GetBlue(PackRGB(r,g,b))` might not return *b*. It is only guaranteed that this function call will return an integer close to *b*.

**GetGreen**


---

`GetGreen (packedRGB)`

Returns the value of the *packedRGB* argument’s green component.

*packedRGB*                      A packed integer representation of an RGB color.

return value                    An integer in the range [0,65535].

## Drawing and Graphics 2.1

**SPECIAL CONSIDERATIONS**

`GetGreen(PackRGB(r,g,b))` might not return `g`. It is only guaranteed that this function call will return an integer close to `g`.

**GetMaskedPixel**


---

`GetMaskedPixel(x, y, pixFamily)` //Platform file function

Retrieves the value of a specific pixel within a pix family, taking into account its mask.

<i>x</i>	The x coordinate of the point to be tested, in local (view) coordinates.
<i>y</i>	The y coordinate of the point to be tested, in local (view) coordinates.
<i>pixFamily</i>	The pix family to test.
return value	An integer, -1 if the (x,y) pixel location lies outside the bounds of the pix family or if the mask is off at this position, otherwise the integer value of the specified pixel is returned (see below).

**IMPORTANT**

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

```
call kGetMaskedPixelFunc with (x, y, pixFamily);
```

**DISCUSSION**

This function is similar to the existing `PtInPicture` function.

The value returned for a pixel that is actually within the pix family's bounds (and at an on position in the mask) depends on the bit depth of the pix family image. For images with a bit depth of 1, 2, 4, and 8, the pixel will be an index in the range  $[0, 2^{\text{bit depth}} - 1]$ . For example, if the image has a bit depth of 4, the value returned by the function would range from 0 to 15. If the image has a bit depth of 16 or 32, the pixels will have a direct format, and the function will return the direct RGB pixel value.

## Drawing and Graphics 2.1

**GetPointsArrayXY**

---

`GetPointsArrayXY(unit)`

Returns an array of points extracted from the specified unit.

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

return value            An array of points.

**DISCUSSION**

The array that this function returns consists of coordinate pairs describing the points. The first element contains the x coordinate of the first point, the second element contains the y coordinate, and so on. The existing function `GetPointsArray` returns a similar array with the points in y,x order.

Coordinates are global; that is, they are relative to the upper-left corner (0, 0) of the screen.

If the unit encapsulates multiple strokes, this function returns points from the first stroke.

**GetRed**

---

`GetRed (packedRGB)`

Returns the value of the *packedRGB* argument's red component.

*packedRGB*              A packed integer representation of an RGB color.

return value            An integer in the range [0,65535].

**SPECIAL CONSIDERATIONS**

`GetRed(PackRGB(r,g,b))` might not return *r*. It is only guaranteed that this function call will return an integer close to *r*.

## Drawing and Graphics 2.1

**GetStrokePointsArray**

---

`GetStrokePointsArray(stroke, format)`

Copies the data for all the points in an ink stroke into an array.

<i>stroke</i>	A binary object representing an ink stroke; see “Stroke, Word, and Gesture Units” (page 8-29) in <i>Newton Programmer’s Reference</i> .								
<i>format</i>	An integer or frame specifying how to format the output Possible integer values are: <table> <tr> <td>0</td><td>Data in screen resolution. Filter out duplicate points.</td></tr> <tr> <td>1</td><td>Data in screen resolution. Duplicate points are allowed.</td></tr> <tr> <td>2</td><td>Data in tablet resolution. Filter out duplicate points.</td></tr> <tr> <td>3</td><td>Data in tablet resolution. Duplicate points are allowed.</td></tr> </table>	0	Data in screen resolution. Filter out duplicate points.	1	Data in screen resolution. Duplicate points are allowed.	2	Data in tablet resolution. Filter out duplicate points.	3	Data in tablet resolution. Duplicate points are allowed.
0	Data in screen resolution. Filter out duplicate points.								
1	Data in screen resolution. Duplicate points are allowed.								
2	Data in tablet resolution. Filter out duplicate points.								
3	Data in tablet resolution. Duplicate points are allowed.								

For information on the screen vs. tablet resolution distinction, see “Using Stroke Bundles” (page 10-42) in *Newton Programmer’s Guide*.

If this parameter is a frame, it may contain the following slots:

<i>format</i>	Required. An integer, any of the integers this parameter can accept if not supplying a frame (listed above).
<i>distance</i>	Optional. An integer, the minimum distance between points. The default is 0. Using a value of just 1 or 2 significantly reduces the size of the array returned by this function. This is handy, as you don't have to munge through so much data if you want to analyze a stroke yourself. You can also create polygon views with reasonable fidelity using significantly fewer points. With values greater than 4,

## Drawing and Graphics 2.1

	the resulting polygon looks noticeably different.
<code>order</code>	Optional. One of the following symbols: <code>'xy</code> or <code>'yx</code> . The default is <code>'yx</code> . This slot controls the xy ordering of the points. Normally, <code>GetStrokePointsArray</code> returns the points in (y, x) order. If this slot has the value <code>'xy</code> , the points are returned in (x, y) order.
<code>return value</code>	An array of points. This array contains an even number of elements, containing the x and y coordinates. Normally the coordinates are returned in (y,x) order, however you can control this through the <code>order</code> slot of the <i>format</i> parameter.

**COMPATIBILITY**

The version of this function on Newton OS versions prior to 2.1 accepts only an integer for the *format* parameter.

**GetTone**


---

`GetTone(packedRGB)`

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

<i>packedRGB</i>	A packed integer representation of an RGB color.
<code>return value</code>	An integer in the range [0,15]; 0 is white and 15 is black.

**GrayShrink**


---

`view:GrayShrink(bitmap, style)`

Anti-aliases a 1-bit pix family and renders it on the screen.

<i>bitmap</i>	A 1-bit pix family or bitmap graphic shape.
<i>style</i>	A style frame, as used by <code>DrawShape</code> . This frame should contain a <code>transform</code> slot representing a reduction in size, either horizontally, vertically, or both. The <code>transform</code>



## Drawing and Graphics 2.1

slot should contain two bounds frames, i.e. not two integers. You may pass `nil` for the source bounds frame, in which case, *bitmap*'s bounds are used. You may also pass `nil` for the destination bounds, in which case the `viewBounds` slot is used.

return value      Undefined, do not rely on what this method returns.

**DISCUSSION**

If *bitmap* is not 1-bit, or if *style* does not have a transform slot representing a reduction in size, *bitmap* is still rendered on the screen, but not anti-aliased.

**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.

return value      Either `true` or `nil`.

**MakeBitmap**


---

`MakeBitmap(widthInPixels, heightInPixels, optionsFrame)`

Returns a blank (white) bitmap shape of the specified size.

*widthInPixels*      Width of the bitmap shape.

*heightInPixels*      Height of the bitmap shape.

*optionsFrame*      An optional frame specifying additional characteristics of the bitmap shape created by this method. It can contain any of the slots specified here. If this frame is not used, the value of the *optionsFrame* parameter must be `nil`.

*rowBytes*      Specifies the number of bytes per row of the bitmap; use only for a data source that creates scan lines longer than the default value. An `exMakeBitmapBadArgs` exception is thrown if the value of *rowBytes* is not a

## Drawing and Graphics 2.1

	multiple of 32 bits or is too narrow for the bitmap's width as specified by the <i>widthInPixels</i> parameter. When no other value is specified, this slot has the default value $\text{BAND}(\text{widthInPixels} + 31, -32) / 8$ .
depth	Specifies a bit depth for the creation of grayscale or color images. The value of <i>depth</i> must be an integer describing the number of bits per pixel, and it must be one of the values 1, 2, or 4. This slot's default value is 1 when no other value is specified.
resolution	Specifies high- or low-resolution images. Like a pen size, the value of the <i>resolution</i> slot may be an array or a single value. If this value is an array, the elements of the array specify the <i>x</i> and <i>y</i> dimensions of the pixels comprising the bitmap. If this slot stores a single value, it specifies that the pixels are square, having equal values for their <i>x</i> and <i>y</i> dimensions. Applications that display or otherwise manipulate bitmap documents (for example, fax pages) need to use this slot to control scaling functionality. This slot's default value is [72,72] when no other value is specified.
store	By specifying a store, the bitmap is created as a VBO (virtual binary object). To applications, VBOs appear to be NewtonScript binaries, but they are actually handled directly by the system, using automatic compression and decompression to allow these objects to be much larger than the available heap space. If you are going to create a bitmap, and you know that it will ultimately wind

## Drawing and Graphics 2.1

up in a soup on a particular store, you can increase the system efficiency by using this slot to specify the store on which to create the object.

If this slot is `nil`, the NewtonScript heap is used, and the bitmap will not be a VBO.

You must limit the use of the NewtonScript heap to small bitmaps only.

An exception occurs in the event the NewtonScript heap or store does not have enough space for the bitmap.

`companderName`

When a VBO is written to the store, the system uses a compander, or compression-decompression utility. This slot is a string that represents the name of the compander to use when writing or reading this bitmap from the store.

The default compander is

`TPixelMapCompander`, which is efficient for monochrome images.

You can supply your own compander as a protocol. If you don't want to compress the data when written out to the store you would need to supply an appropriate protocol. If you are not writing to the store (default), then there is no compression, no VBO, and the data is written out to the frames heap.

`companderData`

This slot is intended for optional arguments that would be passed to the compander. The default is `nil`.

`return value`

A bitmap shape.

## Drawing and Graphics 2.1

## DISCUSSION

The origin of the bitmap returned is at (0,0); however, you can subsequently use the `OffsetShape` function to modify the returned bitmap's origin.

## COMPATIBILITY

Versions of this function prior to Newton 2.1 OS ignore a the depth slot in *optionsFrame*.

**MakeInk**


---

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

Creates an ink shape in the specified bounds box.

<i>inkdata</i>	The ink data object.
<i>left</i>	The left boundary.
<i>top</i>	The top boundary.
<i>right</i>	The right boundary.
<i>bottom</i>	The bottom boundary.
return value	An ink shape.

## DISCUSSION

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. You must use the new function `FindShape` (page 6-29) on these objects.

The bounds slot doesn't scale the ink, it just tells `DrawShape` where to place the upper left corner. The shape is not clipped at its bounds.

## SEE ALSO

For an example of using this function see "Creating Ink and TextBox Shapes" beginning on page 6-19.

## Drawing and Graphics 2.1

**MakeShape**

---

`MakeShape(object)`Creates and returns a graphic shape based on *object*.

<i>object</i>	A bounds frame, points array, pix family or bitmap, picture, or view to convert to a graphic shape.
return value	The following kinds of shapes are created, depending on what kind of object is passed in <i>object</i> :
bounds frame	A rectangle shape is returned.
points array	A polygon shape is returned. You can pass in the value stored in the <code>points</code> slot in a view of class <code>clPolygonView</code> . This is a binary data structure that has a class of 'polygonShape and contains data describing a polygon shape.

**Note**

This option is intended to create a shape from data you retrieve from a `clPolygonView`. However, you can manually create the points data structure by using the `ArrayToPoints` routine. ♦

pix family or bitmap

A bitmap shape is created and returned. If the pix family or bitmaps has a mask, it is contained in an extra slot `mask`.

picture

A picture shape is returned.

view

A picture shape is returned.

**Note**

`MakeShape` may return a shape that uses less memory than what you would need if you did the equivalent capture of a view into a bitmap with `ViewIntoBitmap`. ♦

**COMPATIBILITY**

Versions of this function prior to Newton 2.1 OS ignore a bitmap's mask.

**MakeTextBox**

---

`MakeTextBox(text, left, top, right, bottom)`

Creates a text box shape in the specified bounds box.

<i>text</i>	A string used to create the text box shape.
<i>left</i>	The left boundary.
<i>top</i>	The top boundary.
<i>right</i>	The right boundary.
<i>bottom</i>	The bottom boundary.
return value	A text box shape.

**DISCUSSION**

The text is drawn in the font specified in the style frame. The text is clipped if it spills out of the bounds box.

**SEE ALSO**

For an example of using this function see “Creating Ink and TextBox Shapes” beginning on page 6-19.

**MungeShape**

---

`MungeShape(shape, action, style)`

Flips a shape, or rotates it to the right.

<i>shape</i>	A shape or shape array.
<i>action</i>	On of the following symbols: 'rotateRight, 'flipHorizontal, or 'flipVertical.
<i>style</i>	Style frame to use if <i>shape</i> must be converted to a bitmap.
return value	A shape or shape array.

## Drawing and Graphics 2.1

## DISCUSSION

This function 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.

## SPECIAL CONSIDERATIONS

You cannot use the `'rotateRight'` or `'flipHorizontal'` options, with bitmap shapes of bit depth greater than 1.

## SEE ALSO

For an example use of this function, see “The MungeShape Function” (page 6-22).

**PackRGB**


---

`PackRGB(red, green, blue)`

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

<i>red</i>	A 16-bit integer, that is in the range [0,65535].
<i>green</i>	A 16-bit integer, that is in the range [0,65535].
<i>blue</i>	A 16-bit integer, that is in the range [0,65535].
return value	A packed integer representing the color.

**PictToShape**


---

`PictToShape(pict, bounds)`

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

<i>pict</i>	The PICT binary object to convert.
<i>bounds</i>	A bounds frame for the rectangle you would like <i>pict</i> to be imaged into. If you pass <code>nil</code> , it uses the bounds stored in the PICT (which is the normal thing to do).
return value	A shape or shape array.

Drawing and Graphics 2.1

**DISCUSSION**

Unlike `MakeShape`, this function creates an array of shapes for the individual drawing commands that comprise the PICT object. For a discussion of this, see the note on page 6-14.



## Summary

---

### Constant

---

#### Gray Tone Constants

---

kRGB\_Gray0  
kRGB\_Gray1  
kRGB\_Gray2  
kRGB\_Gray3  
kRGB\_Gray4  
kRGB\_Gray5  
kRGB\_Gray6  
kRGB\_Gray7  
kRGB\_Gray8  
kRGB\_Gray9  
kRGB\_Gray10  
kRGB\_Gray11  
kRGB\_Gray12  
kRGB\_Gray13  
kRGB\_Gray14  
kRGB\_Gray15  
kRGB\_White  
kRGB\_Black  
kRGB\_16GrayIncrement

#### Transfer Mode Constants

---

modeCopy  
modeOr  
modeXor  
modeBic  
modeNotCopy  
modeNotOr  
modeNotXor  
modeNotBic  
modeMask

## Data Structures

---

### Style Frame

---

```
aStyleFrame := {
textPattern: toneOrPattern,      //tone to display text in
selection: integerOrNil,         //size of selection handles
penPattern: toneOrPattern,       //tone for text and frame of shapes
fillPattern: toneOrPattern,      //tone to fill shapes with
...
}
```

### Patterns

---

pattern

gray pattern

dithered pattern

## Functions and Methods

---

```
FindShape(shape, x, y, style) //is point (x,y) in shape?
GetMaskedPixel(x, y, bitmap) //returns value of pixel in a bitmap
GetBlue(packedRGB) //returns blue value of packed RGB integer
GetGreen(packedRGB) //returns green value of packed RGB integer
GetPointsArrayXY(unit) //returns points array in (x,y) order
GetRed(packedRGB) //returns red value of packed RGB integer
GetStrokePointsArray(stroke, format) //get points array from ink stroke
GetTone(packedRGB) //returns value of packed RGB int at curr bit depth
view:GrayShrink(bitmap, style) //renders pix family anti-aliased
IsEqualTone(packedRGB1, packedRGB2) // are these two RGB ints close?
MakeBitmap(widthInPixels, heightInPixels, optionsFrame) // makes bitmap
MakeInk(inkdata, left, top, right, bottom) // creates an ink shape
MakeShape(object) //creates a shape
MakeTextBox(text, left, top, right, bottom) // creates a text box shape
MungeShape(shape, action, style) //flips or rotates a shape
PackRGB(red, green, blue) //creates a packed RGB integer
PictToShape(pict, bounds) //makes picture shape from picture object
```

# Sound

---

This chapter describes the enhancements to the sound interface for the Newton 2.1 OS, including the sound input interface, and improvements to playback and sound compression.

## About Sound

---

The sound interface has been improved in the Newton 2.1 OS with a number of enhancements. The most notable include:

- support for sound recording in `protoSoundChannel`, with appropriate sound input hardware
- new objects to support sound recording: `soundRecorder` and `protoRecorderView`
- a new sound frame object, `protoSoundFrame`
- support for 16-bit sound samples, which are generated by the MessagePad 2000 and eMate 300 sound recording hardware
- support for recording and playback of compressed sounds via codecs (compressor-decompressors); built-in codecs include `muLaw`, `IMA`, `GSM`, and a synthesizer codec for sound generation

## Sound

Terminology

---

channel	A virtual connection to specific piece of sound hardware.
codec	Compressor-decompressor.
sample	A binary object consisting of sound samples (supplied by an analog to digital converter). The 1.x and 2.0 Newton devices use the same sample format as Macintosh (8 bit unsigned). Newton devices with the 2.1 OS can also use 16-bit linear (signed) samples.
volume	A value used to specify the loudness for a sound. The 1.x interface supports integer volumes 0 through 4, (0 = quiet; 4 = maximum volume). The 2.0 interface supports real volumes in the <code>volume</code> slot of a sound frame, which correspond to dB (decibel) levels. A large negative value corresponds to silence, and 0.0 corresponds to full volume. The Newton 2.1 OS interfaces add a decibel-based system volume.

Compatibility

---

All 1.x and 2.0 functions, protos, and methods work the same under the Newton 2.1 OS. Newton 2.1 OS sound frames are backwards compatible with 1.x and 2.0 systems, although sounds created under the 2.1 OS that use codecs or the new 16-bit data format cannot be played on 1.x or 2.0 systems.

Hardware Volume Support

---

Support for hardware volume control has been added to the Newton 2.1 OS. If a particular system has hardware volume control (for example, the eMate 300), the hardware volume control is used to determine the default volume for sounds. This is overridden when you specify the volume in sound functions.

## Sound

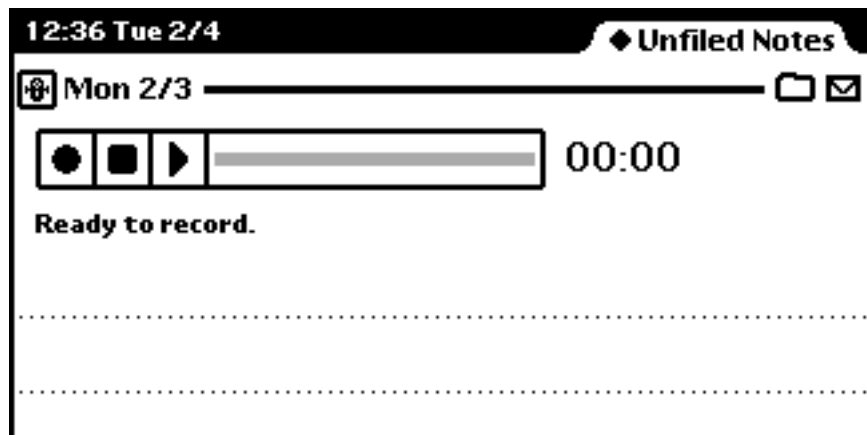
## User Interface

Other than support for hardware volume control, there is no direct user interface for sounds, apart from the Sound, Recording, and Alarm preferences panels, and the volume control in the Extras Drawer. The system does include three new user interface objects that support sound input: sound stationery, the `soundRecorder` slip, and `protoRecorderView`.

Sound stationery is a new type of Notepad stationery that allows users to record and play back sounds. Users can tap the New button and choose the Recording item to create a new recording sheet, as shown in Figure 7-1. The recording sheet contains controls the user can tap to record and play sounds. In addition, it contains lined “paper” like a regular note, on which the user can write notes.

There’s no developer interface to the sound stationery.

**Figure 7-1** Sound stationery



The `soundRecorder` slip is a root view slip (Figure 7-2) that allows you to easily include sound recording and playback in any application. This slip incorporates all the code necessary to record and playback sounds. For

## Sound

information on how to use this slip in your applications, see “Using the Built-in Sound Recorder Slip” (page 7-9).

**Figure 7-2** Sound recorder slip



## Sound Input

There are four main components to the new sound input capabilities in the Newton 2.1 OS: the NewtonScript API for recording and storing sound; `protoRecorderView` (a proto) and `soundRecorder` (a built-in sound recorder slip) that let you add sound recording and playback capabilities to an application; and sound stationery, a new type of Notepad stationery that lets users record and organize sounds.

For details on using the `protoRecorderView` in your applications, see “Using the `protoRecorderView`” (page 7-8). For details on using the `soundRecorder` object in your applications, see “Using the Built-in Sound Recorder Slip” (page 7-9). For details on how to use the NewtonScript API to record sound, see “Using the NewtonScript API to Record Sound” (page 7-12).

Sound stationery has no developer interface and is discussed in the previous section, “User Interface.”

## Sound Compression

Under previous versions of the Newton OS, sound was not compressed. The Newton 2.1 OS now supports recording and playback of compressed sounds. The following types of sound compression are supported:

## Sound

- **muLaw.** This is a standard compression technique that is often used to compress voice data. It compresses each 16-bit sample into 8-bits. Rather than simply throwing away the least significant 8 bits, it instead saves fewer bits from each sample, but it shifts them to save the important ones. This tends to preserve the dynamic range better than truncation.
- **IMA (Interactive Multimedia Association).** This is another popular compression technique that is especially good for voice data. It converts a frame of 64 16-bit samples to a 34-byte frame. It compresses each 16-bit sample to 4 bits, and uses 2 additional bytes as decompression information. This technique works better for music than muLaw or GSM.
- **GSM (Global System for Mobile-Communications).** This is a standard compression used for cellular phone data in Europe. It converts a frame of 160 16-bit samples to a 33-byte frame. It is a mathematically intensive compression technique that works effectively only on devices containing a fast CPU, such as the MessagePad 2000. GSM compressed data doesn't sound quite as good as IMA, but it is less than half the size.

These types of sound compression are implemented by built-in codecs (compressor-decompressors). Using a codec, the system plays a compressed sound using a two-step process. The sound data is decompressed into an intermediate buffer by the codec, then the buffer is scheduled and played like an uncompressed sound. Similarly, during recording, sound sample data is recorded into a temporary buffer first and then compressed into the binary object where it is to be stored.

With codecs, intermediate buffers are used to avoid skips in the sound that might be caused by the system trying to access the binary object, which is typically a virtual binary object (VBO). This is in contrast to uncompressed sounds which are played directly from the sound object, or recorded directly to the binary object, with no intermediate buffering.

For details on how to compress and uncompress sounds, see “Compressing Sound” (page 7-15).

## Synthesized Sound

---

The Newton 2.1 OS contains a built-in synthesizer codec. This codec can be used to generate tones based on sine wave addition (like telephone tones) using up to 12 sine waves, and tones produced by several variants of FM

## Sound

synthesis using up to 4 sine waves to describe a sound. The following attributes can be specified for each sine wave: sustain amplitude, peak amplitude, sustain time, release time, attack time, decay time, lead silence, trail silence, and frequency.

Synthesized sounds are typically much smaller than sampled sounds, so if your application plays custom sounds, you may want to synthesize them rather than using a recorded sample. For details on using the synthesizer codec, see “Synthesizing Sound” (page 7-17).

## Devices and Channels

---

The Newton 2.1 OS now supports multiple output devices for playback and multiple input devices for recording. The system uses a default output device which can be overridden by the sound channel. The system also uses a default input device which can similarly be overridden by the channel.

Despite this flexibility, current 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 is routed to the external speaker instead. The same is true of input: all simultaneously active input channels have the same source.

The number of simultaneous sounds that can be played is limited by the processing power of the CPU, the codecs in use, and memory constraints. On the eMate 300 and MessagePad 2000, the number is about 4, depending on the exact nature of the sounds being played.

The Newton 2.1 OS supports the following output devices:

kDefaultDevice	0x00	// default output device
kInternalSpeaker	0x01	// the internal speaker
kLineOut	0x08	// line out on the interconnect

and the following input devices:

kDefaultDevice	0x00	// default input device
kInternalMic	0x04	// the internal microphone
kLineIn	0x10	// line in on the interconnect bus



## Sound

Notice that each device is represented by a single bit, so that the devices can theoretically be OR'ed together. In the 2.1 OS, however, more than one input or output device at a time is not supported.

It is possible to have multiple recording sessions running simultaneously, if you want to record the same input on multiple channels.

## Sampling Rates

---

The hardware sampling rate of all Newton devices based on the 2.0 and earlier operating system is 22026.43172. On these devices, sampling rates of 22026.43172 and 11013.21586 have been commonly used for sounds.

On Newton 2.1-based devices, the hardware sampling rate is 21600, so you should use that as your base rate on such devices. On the MessagePad 2000 unit, a powerful algorithm resamples sounds recorded at any rate, so they can be played without significant degradation of quality. On the eMate 300 unit, the best sampling rate multiple to use for best quality sounds is 10800.

## New NTK Sound Import Function

---

There is a new NTK compile-time function, `GetSoundFrame`, that retrieves a sound from an open Macintosh sound resource at compile time. This function works just like the older functions, `GetSound` and `GetSound11`, but it can import a sound recorded at any sampling rate. Note that this function is available in the Mac OS version of NTK 1.6.4. The Windows version of NTK will implement this functionality differently.

## Using Sound

---

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

- record sound using the `protoRecorderView`
- record sound using the built-in recorder slip
- record sound programmatically using the NewtonScript API

## Sound

- using sound compression
- synthesizing sounds
- set and get sound-related user preference data

This section also gives new information about using the existing `PlaySound` function.

## Using the `protoRecorderView`

The system includes a new proto, `protoRecorderView`, that implements a simple user interface object to record and play sounds. The proto incorporates all the code necessary to record and play sounds. You can use this proto to add sound recording and playback capabilities to any application by embedding it in a view in the application. The proto is shown in Figure 7-3.

**Figure 7-3** `protoRecorderView`



You retrieve the sound data recorded in a `protoRecorderView` by sending the view the `GetSounds` message. This method returns the array of sound frames recorded.

The recorder (`protoRecorderView`) has an internal state that changes when the user taps one of its buttons. Each time there's an event that causes a state change, the recorder sends a `SetState` message to the `statusText` slot in the view. By default the `statusText` slot is `nil`. You can catch state changes by implementing a `statusText` slot that contains a frame that contains a `SetState` method. For example, here is how you could implement such a frame to handle the `SetState` message.

```
myProtoRecorderView.ViewSetupFormScript := func()
begin
self.statusText := {
    _parent: self, // only if you want to call inherited
```

## Sound

```

SetState: func(oldState, newState, hasSound)
begin
    // do your work here
    // for example, catch a transition to the stopped state
    // and call GetSounds to retrieve the recorded sounds
end,
}
end;

```

The `SetState` method is called with three parameters. The first, (`oldState` in the example) is an integer representing the recorder's previous state; the second (`newState` in the example) is an integer representing the recorder's new state; and the third (`hasSound` in the example) is a Boolean that is `true` if the recorder currently has some sound frames defined. This last parameter allows you to determine if the recorder view has any data to play (Boolean is `true`), or if it has none (Boolean is `nil`).

The first two parameters to `SetState` (`oldState` and `newState` in the example) can have the following values:

```

kInactive := 1;           // default state (stopped)
kRecording := 2;          // sent before recording is started
kPlaying := 4;            // sent before playing is started
kPlayPaused := 8;         // sent before playing is paused
kRecordPaused := 16;      // sent before recording is paused
kStopping := 32;          // sent before the sound channel is stopped
kSetupStore := 64;        // sent before recording is started

```

The `kSetupStore` state indicates that recording is about to start. After the `kSetupStore` message is sent, the state is immediately set back to `kInactive`. The `kStopping` value indicates that recording or playback is about to stop. After it has completely stopped, the state is set to `kInactive`.

If you want to explicitly set the store where recorded sounds are stored by the recorder view, you can set the slot `protoRecorderView.RecordEngine.fStore` to the destination store. Otherwise the default store is used for recorded sounds (in VBOs).

## Using the Built-in Sound Recorder Slip

The system includes a new built-in user interface object that lets the user control sound recording and playback. This floating slip (Figure 7-2 on

## Sound

page 7-4) exists as a child of the root view, and is named `soundRecorder`. This slip incorporates all the code necessary to record and play sounds. You can use this slip to add sound recording and playback capabilities to any application.

Note that because the `soundRecorder` slip is an independent floating view, you don't embed it in your application's view hierarchy, you simply send it a message to open it.

You can access the built-in sound recorder with code like this:

```
sr := GetRoot().soundRecorder;
```

To open this view to allow recording, use this call:

```
if sr then
    sr:OpenRecord(callback);
```

This sets up the recorder slip to do recording and opens it on the screen. The *callback* parameter is a function that you supply and is called when the user closes the slip. When the user taps the close box to close the recorder, the callback function is called with a single argument, the array of sound frames that were allocated as a result of recording. You can store them or do whatever processing is required.

To play a sound using the sound recorder slip, use this call:

```
sr:OpenPlay(soundFrame);
```

This sets up the recorder to play the sound frame passed in *soundFrame*, and opens it on the screen.

After opening the recorder slip with either `OpenPlay` or `OpenRecord`, you can set several slots in the sound recorder base view to override values in the

Sound

associated sound channel and sound frames created by it. The slots you can set are shown in Table 7-1.

**Table 7-1**      Sound recorder slots you can set

Slot	Description
fSoundFrameSlots	A frame that contains slots to be copied to the sound frames used for recording. This frame overrides the default values of the slots in any sound frames created by <code>soundRecorder</code> . This frame can contain any slots that you want copied to the sound frames created by <code>soundRecorder</code> .
fInputGain	Sets the <code>inputGain</code> slot in the sound channel.
fInputDevice	Sets the <code>inputDevice</code> slot in the sound channel.
fOutputDevice	Sets the <code>outputDevice</code> slot in the sound channel.
fCallback	A function that is called when the user taps the close box to close the recorder. It is passed a single argument, the array of sound frames that were allocated as a result of recording. Note that this slot is set by the <code>OpenRecord</code> method (it's the argument you passed to <code>OpenRecord</code> ), so if you open the recorder by calling <code>OpenRecord</code> , then you don't need to also set this slot.

Here's an example of setting the `fSoundFrameSlots` slot in the sound recorder base view to specify that IMA compression be used for recording:

```
sr.fSoundFrameSlots :=
{sndFrameType: 'codec, // use a codec
codecName: "TIMACodec", // select IMA codec
bufferSize: 12500, // size of codec buffers
bufferCount: 4, // # of codec buffers
compressionType: kSampleLinear, // for playback
dataType: k16bit, // for playback
samplingRate: 10000, // samples per second
compressionRatio: 64/34, // for IMA
}
```

## Sound

## Using the NewtonScript API to Record Sound

---

Because sound recording requires some user interface controls to start and stop the recording process, there is no single programmatic interface for recording equivalent to the `PlaySound` function. However, there is a ready-made sound recording proto and slip that you can use to quickly add sound recording and playback capabilities to any application; for details, see the previous subsections, “Using the `protoRecorderView`” and “Using the Built-in Sound Recorder Slip.”

If you want to create your own sound recording interface and programmatically control everything, then you’ll need to use the NewtonScript API directly, as described in this subsection.

To record sound programmatically, you must do these steps:

1. Create a new sound channel.
2. Open the sound channel.
3. Allocate a block of memory to hold the recorded sound samples.
4. Schedule the recording.
5. Start the recording.
6. Finally, stop recording.

This whole process is shown in Listing 7-1.

---

### Listing 7-1      Sound input

```
// callback function passed to NewInputBlock below
MyCallback: func(state, result) begin
if self.notdone then begin // test a flag to see if we're done
    // if so, continue with another input block
    local anotherSound := myChannel:NewInputBlock(MyCallback);
    myChannel:Schedule(anotherSound);
end
else
    // if we're done, then close the channel
    myChannel:Close();
end
```

## Sound

```
// sound input sample code; first initialize and open the channel
myChannel := protoSoundChannel:NewRecording();
myChannel:Open();

// good idea to create and schedule 2 input blocks initially
local mySound := myChannel:NewInputBlock(MyCallback);
local anotherSound := myChannel:NewInputBlock(MyCallback)
myChannel:Schedule(mySound);
myChannel:Schedule(anotherSound);
myChannel:Start(true);
```

You first create a new sound channel using the method

`protoSoundChannel:NewRecording`. This creates a sound channel that is properly initialized for recording. Next, open the sound channel by sending it the `Open` message. Then, create a new block of memory to hold the recorded samples by sending it the `NewInputBlock` message, which returns a sound frame. The `samples` slot in this sound frame holds a virtual binary object (VBO) that is allocated for the sample data. This size of the VBO is determined by the value of the `inputBlockSize` slot in the sound channel, which defaults to 65536 bytes.

If you might need to record more than one buffer's worth of input data, it's a good idea to allocate and schedule two input sound frames initially, to provide double-buffering. This way, the next buffer is always ready immediately, if needed.

Next, schedule the recording by sending the sound channel the `Schedule` message, passing the new sound frame as a parameter. Finally, you can start recording by sending the `Start` message to the sound channel. After you send the `Start` message, there will be about a half-second delay while the sound hardware powers up before recording starts.

The Sound Manager then records sound into the scheduled sound frame. Once the `samples` VBO is full, it calls the callback function object provided by the sound frame. Then, the Sound Manager looks to see if there is another scheduled sound frame to continue recording into, and if so, it continues recording into the next sound frame. It is your responsibility to ensure that enough sound frames are scheduled to keep recording.

It is also your responsibility to keep track of which sound frames have been created and scheduled, and to update the size of the final input block (`samples` VBO) when the stop button is pressed. The return value of the `Stop` method is a result frame (page 7-31) that indicates where in the sample data

## Sound

recording was stopped, so you can adjust the size of the VBO in the last sound frame to be just large enough to hold the recorded samples. Here is an example of how to use the `SetRecordingLength` method do this:

```
// stop recording and trim last VBO size
local result := myChannel:Stop();
local sound := result.sound;
local numSamples := result.index;
sound:SetRecordingLength(numSamples, nil);
```

## Setting the Input Gain

---

For 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 is not amplified at all; if the input gain is 255, then the signal is 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.

You can set the initial input gain by setting the `inputGain` slot of the `protoSoundChannel`. After the channel is open, you can change the input gain by using the method `SetInputGain`. The method `GetInputGain` returns the current input gain of the channel.

The behavior of the input gain can seem non-intuitive. The signal that comes from the internal microphone on the MessagePad 2000 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.

The system does not support a self-adjusting input gain.

In the Recording slip of the Prefs application, there’s a slider that lets the user adjust the input gain (Recording volume). If the `inputGain` slot of the sound channel is `nil`, this user preference setting is used (which is stored in the `inputGain` user configuration variable).



## Sound

## Compressing Sound

---

On the Newton 2.1 OS, you can play compressed sounds and record sound, compressing it on the fly. You control sound compression by setting slots in the sound frame; for a complete description of sound frame slots, see “Sound Frame” (page 7-28). The key slots controlling sound compression include:

- `sndFrameType`, specifying whether or not to use a codec
- `codecName`, identifying the codec to use
- `bufferCount`, setting how many codec buffers to allocate
- `bufferSize`, setting the size of each codec buffer in bytes
- `compressionType`, specifying the compression type if a codec is not used
- `samplingRate`, specifying the number of samples per second to play or record
- `compressionRatio`, an optional slot specifying the compression ratio for user interface updating

You can compress and decompress sound using one of the built-in codecs, as described in the following subsections.

In addition to using compression, you can control the resulting size of recorded sounds by varying the sampling rate used when recording. The highest quality sound results from using a sampling rate of 21600, however, this also uses the most storage. You can use lower sampling rates to decrease the size of the resulting sound object, with some loss of quality. Reasonable sampling rates are 8000 on the MessagePad 2000 unit and 10000 on the eMate 300 unit.

Note that playing back a sound at a sampling rate different from that at which it was recorded shifts the sound’s pitch.

## Using Codecs to Compress and Decompress Sound

---

You can use one of several different codecs built into the Newton 2.1 OS to compress and decompress sounds. The codec mechanism makes use of intermediate buffers to hold the sound data, so that there are no skips in the sound caused by the system accessing VBO storage. You should use at least two buffers, and typically four buffers of the same size works well. The total

## Sound

size of all the buffers should be enough to hold 2-3 seconds of sound; so the size of the buffers will vary, depending on the sampling rate and the size of each sample.

For example, if the sampling rate is 8000 samples per second, and you are recording 16-bit samples, that yields 16 KB per second. A total buffer size of 40 KB would be adequate to handle the data. You could allocate four 10 KB buffers to satisfy this need. In the following examples, various values for buffer size are shown, depending on the sampling rate.

Note that if the sound sample data is not stored in a VBO, the buffers can be much smaller.

When using a codec, the `samplingRate`, `compressionType`, and `dataType` slots define the format of the data produced by decompression (for playback), or required for compression (for recording). They also define the format of the data in the intermediate buffers.

To compress or decompress sounds using the `muLaw` codec, set up the slots in the sound frame like this:

```
{sndFrameType: 'codec', // use a codec
 codecName: "TMuLawCodec", // select muLaw codec
 bufferSize: 10000, // size of codec buffers
 bufferCount: 4, // # of codec buffers
 compressionType: kSampleLinear, // for playback
 dataType: k16bit, // for playback

 samples: mySamples, // compressed sound object
 samplingRate: 8000, // set to any value you want
 compressionRatio: 1, // for muLaw
}
```

The `muLaw` codec compresses each 16-bit sample to 8 bits.

To use the `IMA` codec, set up the slots in the sound frame like this:

```
{sndFrameType: 'codec', // use a codec
 codecName: "TIMACodec", // select IMA codec
 bufferSize: 12500, // size of codec buffers
 bufferCount: 4, // # of codec buffers
 compressionType: kSampleLinear, // for playback
 dataType: k16bit, // for playback

 samples: mySamples, // compressed sound object
 samplingRate: 10000, // set to any value you want
}
```

## Sound

```
compressionRatio: 64/34, // for IMA
}
```

This example shows the same values as those used for the highest quality recording level on the Newton 2.1 devices (Music on the MessagePad 2000, and High on the eMate 300). A sampling rate of 8000 is used for the next lowest quality level (Voice 4K on the MessagePad 2000, and Low on the eMate 300).

The IMA codec compresses each block of 64 16-bit samples to 34 bytes, saving 73% over no compression.

To use the GSM codec, set up the slots in the sound frame like this:

```
{sndFrameType: 'codec, // use a codec
codecName: "TGSMCodec", // select GSM codec
bufferSize: 10000, // size of codec buffers
bufferCount: 4, // # of codec buffers
compressionType: kSampleLinear, // for playback
dataType: k16bit, // for playback

samples: mySamples, // compressed sound object
samplingRate: 8000, // set to any value you want
compressionRatio: 160/33, // for GSM
}
```

This example shows the same values as those used for the lowest quality recording level on the MessagePad 2000 device. Note that GSM is a mathematically intensive compression technique that works effectively only on fast processors, such as that used in the MessagePad 2000. It is possible to play GSM-encoded sounds on the eMate 300, but not to record them.

The GSM codec compresses each block of 160 16-bit samples to 33 bytes, saving 90% over no compression.

## Synthesizing Sound

---

The Newton 2.1 OS contains a built-in synthesizer codec that can be used to generate tones based on sine wave addition. You can specify up to 12 individual sine wave tones, which are mixed together to create a sound. Alternately, you can use one of several variants of FM synthesis to create up to four modulated tones (from the base 12), which are mixed together to create a sound.

## Sound

Because the system can play up to four sound channels simultaneously, you can generate quite complex synthesized sounds by using multiple sound channels playing at the same time. Note that multiple sound channels use more heap space, so depending on your application, you may be limited if you are low on heap space.

Synthesized sounds are typically much smaller than sampled sounds, so if your application plays custom sounds, you may want to synthesize them rather than using a recorded sample.

To use the synthesizer codec, set up the slots in the sound frame like this:

```
{sndFrameType: 'codec', // use a codec
 codecName: "TDMFCodec", // select synthesizer codec
 bufferSize: 5000, // size of codec buffers
 bufferCount: 4, // # of codec buffers
 compressionType: kSampleLinear, // for playback
 dataType: k16bit, // for playback

 samples: mySynthData, // synthesizer binary data
 samplingRate: 21600,
}
```

There is a shortcut for using the synthesizer codec. You can simply call the `PlaySound` function and pass a binary object of the class `'TDMFCodec'`, like this:

```
PlaySound(mySynthSound); // class of sound object must be 'TDMFCodec'
```

When passed a binary object of this class, `PlaySound` plays it using the synthesizer codec. If the codec is not found, an exception is thrown. This shortcut works for the whole family of `PlaySoundxxx` functions.

The data in the `samples` slot of the sound frame is a binary object of the class `'samples'`. You can create binary synthesizer data by using the compile-time function `MakeBinaryFromHex`.

The data consists of several unsigned short (16-bit) values, structured as follows:

```
type 'DTMF' {
  unsigned short; /* Parameter block type, set to 1 */
  unsigned short; /* Synthesis type (0-4) */
  unsigned short; /* Reserved, set to 0 */
  unsigned short; /* loop count */
  unsigned short = $$CountOf(DTMFTones); /* Number of tone blocks */
  wide array DTMFTones { /* specify 1 to 12 tone blocks */
```

## Sound

```
    unsigned short; /* frequency integer part */
    unsigned short; /* frequency fractional part */
    unsigned short; /* sustain amplitude */
    unsigned short; /* leading silence in ms */
    unsigned short; /* attack in ms */
    unsigned short; /* decay in ms */
    unsigned short; /* sustain in ms */
    unsigned short; /* release in ms */
    unsigned short; /* peak amplitude */
    unsigned short; /* trailing silence in ms */
};
```

Sound

You can specify up to 12 `DTMFTones` blocks describing individual tones that are mixed (added) or modulated together to generate a sound. The different sound synthesis types are explained in Table 7-2.

**Table 7-2**      Sound synthesis types

Synth type	Description
0	This multiple-synthesis type mixes from 1 to 12 pure sine wave tones. Each tone can have its own envelope, amplitude, and frequency. The minimum number of tones required to use this type is 1.
1	This is the most basic FM (frequency modulation) type, and a minimum of 2 tones must be specified to use it. Each tone can have its own amplitude and modulation. The first tone in each pair is the carrier, and the second is the modulation frequency. You can have a maximum of 6 pairs of tones. For example, tones 1 and 2 are the first FM pair, tones 3 and 4 are the second FM pair, and so on. Each pair of tones is modulated together to create one resulting tone, and these resulting tones are then mixed together to generate the sound. So using this type, you can mix up to 6 tones.

Sound

**Table 7-2**      Sound synthesis types

Synth type	Description
2	This is a more complex type of FM, where each resulting tone consists of 3 base tones. Tone 1 is the carrier, tone 2 is the first modulator, and tone 3 is the second modulator. For this type of FM, the two modulators are mixed and then used to modulate the carrier. In this way, each group of 3 tones are mixed and modulated together to create one resulting tone, and these resulting tones are then mixed together to generate the sound. So using this type, you can mix up to 4 resulting tones.
3	This type is a variation on type 3. Again 3 base tones are used to generate one tone. Using this modulation type, the modulator, tone 2, is first modulated by tone 3, then the resulting modulated waveform is used to modulate tone 1, the base carrier. In this way, each group of 3 tones are modulated together to create one resulting tone, and these resulting tones are then mixed together to generate the sound. So using this type, you can mix up to 4 resulting tones.
4	This is the most complex FM mode. This mode requires 4 base tones to generate one tone. The first tone is the carrier, the second tone is modulated by the third tone, and the third tone is modulated by the fourth tone. The result is then used to modulate the carrier. In this way, each group of 4 tones are modulated together to create one resulting tone, and these resulting tones are then mixed together to generate the sound. So using this type, you can mix up to 3 resulting tones.

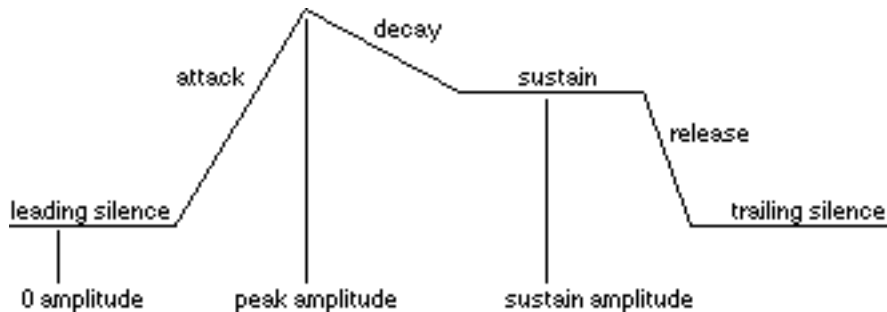
The loop count plays a sound multiple times. This is useful for creating groups of short tone bursts. This value specifies the number of times to repeat the sound after it's played once. So specify 0 to play the sound just once, specify 1 to repeat it once, and so on.

The frequency is specified in Hz, and is a fixed point real number constructed from two fields: the integer part and the fractional part. For example, if you specify 340 for the integer part and 2 for the fractional part, the codec generates a tone of 340.2 Hz.

## Sound

The time values used to play a tone are collectively referred to as the **tone envelope**, illustrated in Figure 7-4.

**Figure 7-4** Tone envelope



All time values specified in the synthesizer data are in milliseconds (1000 ms. = 1 second). There are four time values that make up a tone envelope: attack, decay, sustain, and release. In Figure 7-4, two additional values, leading silence and trailing silence, are also shown.

Attack is the time in which the tone ramps in amplitude from 0 to the peak amplitude. Decay is the time in which the tone ramps from the peak amplitude to the sustain amplitude. Sustain is the time which the tone remains at the sustain amplitude. Finally, release is the time in which the tone amplitude drops from the sustain amplitude to 0.

All of the times except sustain can be 0. This means, for example, that a tone would leap from 0 amplitude to the sustain amplitude and then from the sustain amplitude to 0 at the end, with no ramp. All ramps are linear.

The leading silence and trailing silence values simply specify a period of silence before or after the sound plays. This is useful when mixing multiple tones and you want a delay in the start of one of the tones, or a delay in the finish when looping is used.

The peak amplitude and sustain amplitude are used at two different points in the tone envelope, as shown in Figure 7-4. The maximum amplitude is 32768; anything above that is clipped. The complexity is that if you are



## Sound

mixing (adding) two or more sine waves, then you must limit the total amplitude to 32768 to avoid clipping. Note that you can specify higher amplitude values, resulting in clipping distortion, which is sometimes a desirable part of the sound.

## Using Global Sound Preferences

---

User preferences for sound are maintained in the user configuration data, which is stored in the system soup. A sound channel can override specific preferences, however.

This section describes how to use the `GetUserConfig` and `SetUserConfig` global functions to access sound-related user configuration variables.

### Getting and Setting Input Gain Preference

---

To retrieve the user preference 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 the `inputGain` slot in sound channels used for input. 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 and the input gain in any active channels, but not the value of the `inputGain` slots in any active channels. That is, the input gain for an active channel is changed, but the value of the `inputGain` slot in the sound channel frame is not updated to reflect this change.

## Sound

## Getting and Setting Default Input or Output Devices

---

To retrieve the user preference for the default sound input device, pass the `'inputDevice'` symbol to the `GetUserConfig` function, as in the following example:

```
GetUserConfig('inputDevice');
```

This code returns an integer representing the current default input device. This default can be overridden by putting an `inputDevice` slot in your sound channel. Device definitions are given in “Devices and Channels” (page 7-6).

To set the default input device, pass the `'inputDevice'` symbol and the new value to the `SetUserConfig` function, as in the following example:

```
SetUserConfig('inputDevice, value');
```

This code sets a value that indicates the default input device used during recording. It does not change the device for a currently recording input channel.

To retrieve the user preference for the default sound output device, pass the `'outputDevice'` symbol to the `GetUserConfig` function, as in the following example:

```
GetUserConfig('outputDevice');
```

This code returns an integer representing the current default output device. This default can be overridden by putting an `outputDevice` slot in your sound channel.

To set the default output device, pass the `'outputDevice'` symbol and the new value to the `SetUserConfig` function, as in the following example:

```
SetUserConfig('outputDevice, value');
```

This code sets a value that indicates the default output device used during playback. It does not change the device for a currently playing output channel.

## PlaySound Errata

---

The `PlaySound` function was inadvertently omitted from the Sound chapter in *Newton Programmer's Guide for Newton 2.0* and *Newton Programmer's Reference*.

## Sound

This function was documented for the 1.x system, still works the same under the 2.1 OS, and is documented again in this chapter for completeness.

Note however, it is not the recommended way to play sounds due to interactions with the user preference settings related to sound. `PlaySound` honors user preference 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 the alert sound is the same as the alarm sound and alarm sound effects are disabled.

Rather than `PlaySound`, it is recommended that you use a sound channel or the functions `PlaySoundEffect`, `PlaySoundIrregardless`, or `PlaySoundIrregardlessAtVolume`. The function `PlaySoundEffect` is preferred to all other versions of `PlaySound` for playing pen, alarm, and action sound effects.

The function `PlaySoundAtVolume` is not recommended, since it calls `PlaySound` and thus is subject to the same limitations.

All of these functions (besides `PlaySound`) are documented in *Newton Programmer's Reference*.

## Using the Sound Registry

---

There is a sound registry that you can use to register new sounds that the user can choose as the system alert sound (in the Sound preferences slip). Use the global function `RegSound` to register a new sound with the system under a unique symbol. Remember to append your developer signature to the symbols identifying any sounds that you register. To unregister a sound, use `UnRegSound`.

You can get a list of all registered sounds by calling `SoundList`. This function returns an alphabetized array of sounds suitable for use directly in a picker list. The array contains a series of frames; each frame contains an `item` slot with the name of the sound, and a `soundSymbol` slot containing the unique symbol identifying the sound. Note that there are several built-in sounds listed in the array, in addition to any custom sounds that have been registered.

Sound

The function `GetRegisteredSound` is used to return the sound frame for a registered sound. You pass this function the symbol under which the sound was registered and it returns the sound frame. If the symbol you pass is not found in the registry, the system returns the `simpleBeep` sound frame.

**Note**

These global functions exist in both the Newton 2.0 and 2.1 operating systems. They were not previously documented for 2.0. ♦

# Sound Reference

---

## Constants

---

This section describes constants that your application uses to interact with the sound interface.

## Device Constants

---

The constants described in Table 7-3 are used in the Newton 2.1 OS to identify sound output and input hardware devices.

**Table 7-3**      Sound device constants

Constant	Value	Description
<code>kDefaultDevice</code>	<code>0x00</code>	Default input or output device
<code>kInternalSpeaker</code>	<code>0x01</code>	Internal speaker
<code>kLineOut</code>	<code>0x08</code>	Line out on the interconnect bus
<code>kInternalMic</code>	<code>0x04</code>	Internal microphone
<code>kLineIn</code>	<code>0x10</code>	Line in on the interconnect bus

Sound

Codec Constants

The string literal values described in Table 7-4 are used in the `codecName` slot of sound frames to specify a codec to be used to compress or decompress the sound.

**Table 7-4**      Codec constants

String	Description
"TMuLawCodec"	MuLaw codec.
"TIMACodec"	IMA codec.
"TGSMCodec"	GSM codec.
"TDTMFCodec"	Synthesizer codec (for playback only).

Compression Constants

The constants described in Table 7-5 are used in the `compressionType` slot of sound frames to specify the encoding format of the sound samples.

**Table 7-5**      Compression constants

Constant	Value	Description
<code>kSampleStandard</code>	0	Uncompressed 8-bit samples. This is the only format supported on Newton 2.0 and 1.x devices and corresponds to the value of this slot ( <code>kNone</code> ) used in those versions of the Newton OS. This is the default.
<code>kSampleMuLaw</code>	1	8-bit samples encoded by the muLaw compressor (reduced from 16 to 8 bits).
<code>kSampleLinear</code>	6	Uncompressed 16-bit samples. This is the standard format produced by the recording hardware on Newton 2.1 OS devices.

Sound

Data Type Constants

The constants described in Table 7-6 are used in the `dataType` slot of sound frames to specify the size of each sound sample.

**Table 7-6** Data type constants

Constant	Value	Description
<code>k8Bit</code>	8	Samples are 8 bits each. This is the default.
<code>k16Bit</code>	16	Samples are 16 bits each.

Data Structures

This section describes data structures used in the sound interface.

Sound Frame

A sound frame has the following slots.

**Note**

You can use the new proto, `protoSoundFrame` (page 7-44), as the basis for sound frames in the 2.1 OS. Using this proto provides enhanced features through methods that it includes. ♦

**Slot descriptions**

<code>sndFrameType</code>	Required. A symbol specifying the type of the sound frame. The <code>'simpleSound'</code> symbol indicates a standard sound, while the <code>'codec'</code> symbol indicates a sound compressed with a codec. For information on codecs, see “Compressing Sound” (page 7-15). If you specify the symbol <code>'codec'</code> , you must also supply the <code>codecName</code> , <code>bufferCount</code> , and <code>bufferSize</code> slots.
<code>codecName</code>	A string identifying the codec to be used to compress or decompress the sound. This slot is required if you

## Sound

	specify 'codec in the <code>sndFrameType</code> slot. For more details on the possible string values, see Table 7-4 (page 7-27).
<code>bufferCount</code>	An integer specifying the number of codec buffers to allocate. Each buffer has the size specified in the <code>bufferSize</code> slot. This slot is required if you specify 'codec for the <code>sndFrameType</code> slot. For guidelines on this value, see “Using Codecs to Compress and Decompress Sound” (page 7-15).
<code>bufferSize</code>	An integer specifying the size of each codec buffer in bytes. This slot is required if you specify 'codec for the <code>sndFrameType</code> slot. For guidelines on this value, see “Using Codecs to Compress and Decompress Sound” (page 7-15).
<code>samples</code>	Required. A binary object of the class 'samples, that contains the sound samples. If the synthesizer codec is being used, this slot contains the synthesizer data, which is a binary object of the class 'TDTMFCodec.
<code>samplingRate</code>	Optional. Real or integer value describing the sampling rate of the data in the <code>samples</code> slot. (8000.0, 11013.21586, and 22026.43172 are common values). If missing, the sound channel assumes 22026.43172. When using a codec, this value also describes the sampling rate of data in the codec buffers.
<code>compressionType</code>	Optional. An integer identifying the encoding format of the samples. If present, it must be <code>kSampleStandard</code> (0), <code>kSampleLinear</code> (6), or <code>kSampleMuLaw</code> (1). If missing, <code>kSampleStandard</code> is assumed. If you specify <code>kSampleMuLaw</code> (or <code>kSampleStandard</code> ), you must also set the <code>dataType</code> slot to <code>k8Bit</code> , since this compressor reduces 16-bit samples to 8-bit samples. If you specify <code>kSampleLinear</code> , you must also set the <code>dataType</code> slot to <code>k16bit</code> . For more details on the formats, see Table 7-5 (page 7-27).
<code>compressionRatio</code>	Optional. A ratio of the number of uncompressed sound samples (not bytes) per a number of compressed bytes. For example, for IMA compression, you would specify 64/34, because 64 samples are compressed into 34 bytes. This value is used to update the user interface progress

## Sound

	indicator while a compressed sound is playing or recording.
<code>dataType</code>	Optional. An integer specifying the size of each sample in bits. If present, it must be <code>k8Bit</code> (8) or <code>k16Bit</code> (16). If missing, <code>k8Bit</code> is assumed.
	<p><b>Note</b></p> <p>Older versions of NTK generate sound frames having the value 0 (zero) in the <code>dataType</code> slot. The system assumes 0 (zero) is the same as 8 (<code>k8Bit</code>). ♦</p>
<code>volume</code>	<p>Optional. An integer or real value specifying the volume level at which to play this sound. If missing, the channel's volume setting (see <code>SetVolume</code> method) is used. Note that if <code>volume</code> is an integer it must have the value 0, 1, 2, 3, or 4 corresponding to decibel levels silent, -18 dB, -6 dB, -3 dB, or 0 dB (unity gain) respectively. If <code>volume</code> is a real number, it is treated as the actual dB level, and must be negative.</p> <p>This value overrides the system volume and the channel volume—including values set by functions such as <code>PlaySoundAtVolume</code> and sound channel methods such as <code>SetVolume</code>.</p>
<code>start</code>	Optional. An integer value that is the index of the first sample to play. When this value is missing, 0 is assumed. Omit this slot in sound frames used for recording.
<code>count</code>	Optional. An integer specifying the number of samples to play. When this value is missing, <code>Length(samples) / (dataType/8)</code> is assumed. Omit this slot in sound frames used for recording.
<code>loops</code>	Optional. An integer that is the number of times to repeat the sound. For example, setting <code>loops</code> 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



## Sound

continuous play. Omit this slot in sound frames used for recording.

The following `Callback` method is also part of the sound frame.

**Callback**

---

*soundFrame*:`Callback(state, result)`

Invoked when an operation on the sound frame completes.

<i>state</i>	The state of the sound channel when the callback was executed. Values are:  0 = <code>kSoundCompleted</code> 1 = <code>kSoundAborted</code> 2 = <code>kSoundPaused</code>
<i>result</i>	An integer error code, if present. For a listing of possible values, see “Sound Error Codes” (page 7-52).
return value	Unused; you can return anything.

**DISCUSSION**

This method is invoked when a sound frame is finished playing or recording. During recording, this method is called when each sound frame is filled.

**IMPORTANT**

The `Callback` method may be called slightly before the sound operation completes; if so, the operation will complete within 0.333 seconds. The method is called after the last buffer has been scheduled. ♦

**Sound Result Frame**

---

A sound result frame returns information when the sound channel stops or pauses. This frame is returned by the sound channel methods `Stop` and `Pause`.

## Sound

**Slot descriptions**

<code>sound</code>	The sound frame that was paused or stopped.
<code>index</code>	An integer value that is a zero-based index into the <code>samples</code> slot of the <code>sound</code> frame. This value describes where in the sample data the sound channel was paused or stopped. This value is always between the value of the sound frame <code>start</code> slot and the value <code>(start + count)</code> .

**User Configuration Variables**

---

The following user configuration variables are used in the sound interface. Use the `GetUserConfig` and `SetUserConfig` functions to get and set these values.

**Variable descriptions**

<code>inputGain</code>	The default input gain used during recording. Can be overridden by the channel.
<code>inputDevice</code>	The default input device used for recording. Can be overridden by the channel.
<code>outputDevice</code>	The default output device used for playback. Can be overridden by the channel.
<code>soundVolumeDb</code>	The current system sound volume, in decibels.
<code>alarmVolumeDb</code>	The current system alarm volume, in decibels.

**Synthesized Sound Data Format**

---

The synthesizer codec accepts sound frames whose `samples` slot contains a binary object of the class `'DTMFCodec'`. The data format of the binary object is as follows (note this is a C structure). All values are 16 bits in length.

```
type 'DTMF' {
    unsigned short; /* Parameter block type, set to 1 */
    unsigned short; /* Synthesis type (0-4) */
    unsigned short; /* Reserved, set to 0 */
    unsigned short; /* loop count */
    unsigned short = $$CountOf(DTMFTones); /* Number of tone blocks */
    wide array DTMFTones { /* specify 1 to 12 tone blocks */
        unsigned short; /* frequency integer part */
    }
}
```

## Sound

```

        unsigned short; /* frequency fractional part */
        unsigned short; /* sustain amplitude */
        unsigned short; /* leading silence in ms */
        unsigned short; /* attack in ms */
        unsigned short; /* decay in ms */
        unsigned short; /* sustain in ms */
        unsigned short; /* release in ms */
        unsigned short; /* peak amplitude */
        unsigned short; /* trailing silence in ms */
    };
};

```

## soundRecorder Object

---

The `soundRecorder` object is a child view of the root view. It is a built-in user interface object that lets the user control recording and playback.

After opening the recorder slip with either `OpenPlay` or `OpenRecord`, you can change several slots in the sound recorder base view to override values in the associated sound channel and sound frames created by it. The slots you can change are as follows:

### Slot descriptions

<code>fSoundFrameSlots</code>	A frame that contains any slots that you want to be copied to the sound frames for recording. This frame overrides the default values of the slots in any sound frames created by <code>soundRecorder</code> .
<code>fInputGain</code>	Sets the <code>inputGain</code> slot in the sound channel.
<code>fInputDevice</code>	Sets the <code>inputDevice</code> slot in the sound channel.
<code>fOutputDevice</code>	Sets the <code>outputDevice</code> slot in the sound channel.
<code>fCallback</code>	A function that is called when the user taps the close box to close the recorder. It is passed a single argument, the array of sound frames that were allocated as a result of recording. Note that this slot is set by the <code>OpenRecord</code> method (it's the argument you passed to <code>OpenRecord</code> ), so if you open the recorder by calling <code>OpenRecord</code> , then you don't need to also set this slot.

You can also send the following messages to the `soundRecorder` object.

## Sound

**OpenRecord**

---

```
soundRecorder:OpenRecord(callback) ;
```

Opens the sound recorder slip ready for recording.

*callback*                      A function that is called when the user closes the slip. You must define this function to take one parameter. It is passed the array of sound frames that were allocated as a result of recording. Its return value is not used.

return value                Undefined; do not rely on it.

**OpenPlay**

---

```
soundRecorder:OpenPlay(soundFrame) ;
```

Opens the sound recorder slip ready for playing.

*soundFrame*                A sound frame, or an array of sound frames, that is to be played. For details on the sound frame data structure, see “Sound Frame” (page 7-28).

return value                Undefined; do not rely on it.

**Protos**

---

This section describes protos used in the sound interface.

**protoRecorderView**

---

The `protoRecorderView` system prototype implements a simple set of user interface controls for recording and playback of sounds. This proto is shown in Figure 7-3.

**Slot descriptions**

<code>elapsedTime</code>	An integer giving the number of seconds used in recording.
<code>statusText</code>	Set to <code>nil</code> by default. If you wish to catch recorder view state changes, you can implement a <code>SetState</code> function in a frame in this slot. See the <code>SetState</code> method later in this section for more information.

## Sound

The following methods are used with `protoRecorderView`.

### GetSounds

---

`recorderView: GetSounds()`

Returns the array of sound frames recorded by this object.

return value            An array of sound frames.

### SetState

---

`recorderView.statusText: SetState(oldState, newState, hasSound)`

Sent by the recorder view to indicate a state change when the user taps one of the buttons.

*oldState*            The recorder view's previous state. The possible values are shown in Table 7-7.

*newState*            The recorder view's new state. The possible values are shown in Table 7-7.

*hasSound*            A Boolean that is `true` if the recorder currently has some sound frames defined. This parameter allows you to determine if the recorder view has any data to play (Boolean is `true`), or if it has none (Boolean is `nil`).

return value            Ignored; you can return anything.

### DISCUSSION

Each time there's a state change, the recorder view sends this `SetState` message to the `statusText` slot in the view. By default the `statusText` slot is `nil`. You can catch state changes by implementing a `statusText` slot that contains a frame that contains a `SetState` method.

Sound

The constants described in Table 7-7 are passed for the values of the first and second parameters in the `SetState` message.

**Table 7-7**      `protoRecorderView` state constants

Constant	Value	Description
<code>kInactive</code>	1	Sound channel is stopped (default state).
<code>kRecording</code>	2	Indicates recording is about to be started.
<code>kPlaying</code>	4	Indicates playing is about to be started.
<code>kPlayPaused</code>	8	Indicates playing is paused.
<code>kRecordPaused</code>	16	Indicates recording is paused.
<code>kStopping</code>	32	Indicates the sound channel is about to be stopped.
<code>kSetupStore</code>	64	Indicates recording is about to be started.

**SEE ALSO**

For more details on how to use this method, see “Using the `protoRecorderView`” (page 7-8).

**protoSoundChannel**

The `protoSoundChannel` system prototype represents a virtual connection to a specific piece of sound input or output hardware.

**Note**

`protoSoundChannel` has been modified to support input in the 2.1 OS. All of the slots listed in this section are new in the 2.1 OS, along with the following methods: `IsOpen`, `NewRecording`, `NewInputBlock`, `SetInputGain`, and `GetInputGain`. Both old and new methods are documented here for completeness.

## Sound

**Slot descriptions**

<code>direction</code>	A symbol indicating whether the sound channel is to be used for input or output. This slot can have two valid values: <code>'record</code> or <code>'play</code> . It is set to <code>'record</code> by the <code>NewRecording</code> method. If this slot is invalid or <code>nil</code> , <code>'play</code> is assumed.
<code>outputDevice</code>	Specifies the device that is to be used for sound playback. If <code>nil</code> , then playback is done on the device specified by the user configuration variable of the same name. If that is <code>nil</code> , then the internal speaker is used. The default value of <code>outputDevice</code> is <code>nil</code> . Device constants are listed in Table 7-3 (page 7-26).
<code>inputDevice</code>	Specifies the device that is to be used for sound recording. If <code>nil</code> , then recording is done on the device specified by the user configuration variable of the same name. If that is <code>nil</code> , then the internal microphone is used. The default value of <code>inputDevice</code> is <code>nil</code> . Device constants are listed in Table 7-3 (page 7-26).
<code>inputGain</code>	An integer value between 0 and 255 that specifies how much the input signal is to be amplified before it is recorded. If <code>nil</code> , then the input gain specified by the user configuration variable of the same name is used. If that is <code>nil</code> , then a suitable default value is used to determine an initial value for input gain. The <code>SetInputGain</code> method can change the input gain for the channel, but it does not change the value of the <code>inputGain</code> slot, which remains at its initial setting. For more details on the input gain setting, see “Setting the Input Gain” (page 7-14).
<code>inputBlockSize</code>	The size used by <code>NewInputBlock</code> for the VBO to record into. The default value is 65536 bytes. You can change the size, but the timing of the sound manager may not work for sizes that are too small.

The following methods are supplied by `protoSoundChannel`. They are listed in alphabetical order.

## Sound

**Close**

---

*soundChannel*:Close()

Closes an open sound channel.

return value            Undefined; do not rely on it.

**DISCUSSION**

This method frees the system resources allocated for the sound channel. It throws the exception `|evt.ex.fr|` if an error occurs.

**IMPORTANT**

The operating system memory allocated for a sound channel by the `Open` method is not released when the *soundChannel* frame is garbage-collected. You **must** call the `Close` method explicitly to free this memory and avoid memory leaks. ♦

**GetInputGain**

---

*soundChannel*:GetInputGain()

Returns the input gain value currently used by the sound channel.

**DISCUSSION**

The channel must be open. If it is not, an exception with the error code -30010 is thrown. For information on the meaning of values this method returns, see “Setting the Input Gain” (page 7-14).

**GetVolume**

---

*soundChannel*:GetVolume()

Returns the playback volume level for the channel.

return value	The channel’s playback volume, expressed in decibels. The integer value 0 corresponds to silent, and values 1, 2, 3, and 4 correspond to decibel levels -18 dB, -6 dB, -3 dB, and 0 dB (unity gain) respectively. This method returns <code>nil</code> when the channel does not specify its own volume but instead inherits it from user preference
--------------	--



## Sound

settings (the `soundVolumeDb` variable), which is the default behavior.

**DISCUSSION**

The channel must be open. If it is not, an exception with the error code -30010 is thrown.

If the unit has a hardware volume control, and the sound channel is playing, this method returns the actual playing volume; otherwise, it returns the channel volume as set by the `SetVolume` method. If no volume has been set using `SetVolume`, and the channel is not playing, this method returns `nil`.

**IsActive**


---

*soundChannel*:IsActive()

Returns `true` if the sound channel is active.

return value      True if the sound channel is active (playing, recording, or paused), and `nil` if it is not (`Start` has not yet been called).

**IsOpen**


---

*soundChannel*:IsOpen()

Returns `true` if the sound channel is open.

return value      True if the sound channel is open or `nil` if it is not.

**IsPaused**


---

*soundChannel*:IsPaused()

Returns `true` if the sound channel is paused.

return value      True if the sound channel is paused, and `nil` if it is not.

## Sound

**NewInputBlock**

---

*soundChannel*:NewInputBlock(*Callback*)

Creates and returns a new sound frame that has its slots set up for recording speech at the default combination of fidelity and compression.

*Callback*                      A callback function that you provide, and that is called when the sound frame fills or recording stops. It must accept two parameters, *state* and *result*, as described on page 7-31.

return value                  A sound frame ready to accept input, that looks like this:

```
{
  _proto: protoSoundFrame,
  samples: vbo,           // size determined by inputBlockSize
  callback: callback,     // function specified by caller
  dataType: k8Bit,         // eight bits per sample
  compressionType: kSampleMuLaw, // MuLaw encoded
  samplingRate: 10800,     // 10K samples per second
}
```

**DISCUSSION**

This method allocates a VBO and stores it in the `'samples'` slot of the returned sound frame. This is where the recorded data is stored. The size of the VBO is determined by the value of the `inputBlockSize` slot in the sound channel, which defaults to 65536 bytes.

The sound frame returned has reasonable settings for voice recording. However, you need not rely on these specific values. If you require specific settings, override them yourself.

**NewRecording**

---

*soundChannel*:NewRecording()

Creates and returns a new `protoSoundChannel` frame that is properly initialized for sound input.

return value                  A frame whose `_proto` slot is set to `protoSoundChannel` and whose `direction` slot is set to `'record'`.

## Sound

## DISCUSSION

Note that you send this message directly to the proto itself, not your instantiation of the proto. Sending this message creates a new sound channel based on the proto. Here is an example of how to call this method:

```
myChannel := protoSoundChannel:NewRecording();
```

**Open**

---

```
soundChannel:Open()
```

Instantiates and opens the sound channel.

return value            Undefined; do not rely on it.

## DISCUSSION

This method allocates system resources for the sound channel. It throws the exception `|evt.ex.fr|` if an error occurs.

When you are done using the sound channel, you must close it using the `Close` method; otherwise, memory is not freed.

**Pause**

---

```
soundChannel:Pause()
```

Temporarily suspends play or recording in progress or resumes play or recording already paused.

return value            A sound result frame (page 7-31) indicating which sound frame was stopped, or `nil` if no sound was currently playing or recording.

## DISCUSSION

Scheduled sounds are unaffected by this method.

This method throws the exception `|evt.ex.fr|` if an error occurs.

## Sound

**Schedule**

---

*soundChannel*:Schedule(*soundFrame*)

Queues a sound frame for playback or recording.

*soundFrame*            A sound frame to be scheduled for playback, or an empty sound frame (obtained by calling `NewInputBlock`) to be used for recording. For details on sound frames, see “Sound Frame” (page 7-28).

return value            Undefined; do not rely on it.

**DISCUSSION**

If *soundFrame* defines a callback function, the sound channel sends the callback message to *soundFrame* when play or recording completes.

Note that sounds can be scheduled during a playback or recording operation.

This method can throw the exception `|evt.ex.fr|` if an error occurs.

**SetInputGain**

---

*soundChannel*:SetInputGain(*gain*)

Sets the amplification applied to the signal coming from the input device.

*gain*                    An integer from 0 (no amplification) to 255 (maximum amplification) specifying the amplification to be applied to the input signal.

return value            Undefined; do not rely on it.

**DISCUSSION**

This method changes the input gain, in real time, of the current input channel (and because current 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 sound channel frame.

`SetInputGain` can be called only on open channels, otherwise an exception is thrown.

## Sound

## SEE ALSO

For more information on gain values, see “Setting the Input Gain” (page 7-14).

**SetVolume**

---

*soundChannel*: `SetVolume(volume)`

Sets the playback volume level for the channel.

*volume*                      An integer or `nil`. Value 0 corresponds to silent, and values 1, 2, 3, and 4 correspond to decibel levels -18 dB, -6 dB, -3 dB, and 0 dB (unity gain) respectively. If *volume* is `nil`, then the driver's master (preferred) volume is used.

return value                Undefined; do not rely on it.

**Start**

---

*soundChannel*: `Start(async)`

Starts the sound channel playing or recording.

*async*                        Pass a non-`nil` value to play sounds asynchronously. If this value is `nil`, control does not return until the entire play or record queue is empty (all scheduled sound frames are played or recorded).

return value                Undefined; do not rely on it.

**DISCUSSION**

The sound channel begins playing or recording sound frames in the order they were scheduled.

This method throws the exception `|evt.ex.fr|` if an error occurs.

## Sound

**Stop***soundChannel*:Stop()

Stops playing or recording into the current sound frame, if any.

return value            A sound result frame (page 7-31) indicating which sound frame was stopped, or *nil* if no sound was currently playing or recording.

**DISCUSSION**

This method also sends the *Callback* message, if defined, to each sound frame in the specified channel's queue. The *Callback state* parameter is set to *kSoundAborted*.

When this method returns, all scheduled sounds will have received a *Callback* message, and the queue will be empty.

This method throws the exception `|evt.ex.fr|` if an error occurs.

**protoSoundFrame**

This proto for a sound frame is new for Newton OS 2.1. Not only does it contain all the slots described as valid for sound frame objects (page 7-28), but it also includes some methods that make it easier to work with. Note that sound frames that are not derived from *protoSoundFrame* do not support these methods. To convert an older sound frame to use this proto, simply add a *\_proto* slot containing the magic pointer for *protoSoundFrame*.

Besides all the slots in older sound frame objects (page 7-28), *protoSoundFrame* includes the additional *length* slot and the methods described in this section.

**Slot description**

*length*                    The number of sound samples stored in the *samples* slot. This slot is set only by the *SetRecordingLength* method of *protoSoundFrame*.

## Sound

**GetPlayingTime**

---

*soundFrame*:GetPlayingTime()

Returns the playing time of the sound, in seconds.

return value            Real number describing the playing time of the sound, in seconds.

**DISCUSSION**

This is useful for creating progress indicators for the sound.

**GetSampleCount**

---

*soundFrame*:GetSampleCount()

Returns the number of samples in the sound frame.

return value            An integer indicating the number of samples in the samples slot of the sound frame.

**DISCUSSION**

First, this method checks for the `length` slot in the sound frame. If that slot exists, its value is returned. If it does not exist or is `nil`, this method calculates the number of samples based on the size of the samples binary object divided by the number of bytes in each sample.

Note that the `length` slot in the sound frame is set only by the `SetRecordingLength` method.

**GetSampleSize**

---

*soundFrame*:GetSampleSize()

Returns the size of each sample, in bytes.

return value            An integer indicating the size of each sample, in bytes. This is either 1 or 2, in the 2.1 OS.

## Sound

**GetSamplingRate**

---

*soundFrame*:GetSamplingRate()

Returns the rate at which the data in the sound frame was (or will be) recorded.

**return value**            Real or integer value describing the sampling rate of data in the `samples` slot. This is taken from the `samplingRate` slot of the sound frame. If that slot is `nil`, this method returns the value 22026.43172.

**SetRecordingLength**

---

*soundFrame*:SetRecordingLength(*numSamples*, *Callback*)

Sets the length of the VBO holding the recorded sound samples.

***numSamples***            An integer expressing the number of samples to include in the VBO stored in the `samples` slot of the sound frame.

***Callback***              A callback function that you provide, and that is executed when the `SetLength` operation on the VBO is completed. This function 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 that takes no parameters. Its return value is not used.

**return value**            Undefined; do not rely on it.

**DISCUSSION**

You should call `SetRecordingLength` after the user stops recording sound input. This method sets the `length` slot in the sound frame to the *numSamples* value, then it uses an `AddDeferredSend` call to truncate the length of the VBO to that same value (multiplied by the size of each sample in bytes).

Since the methods `GetPlayingTime` and `GetSampleCount` rely on the `length` slot, both return correct values immediately after `SetRecordingLength` has been called.



## Sound

## Functions

---

This section describes the global functions used in the sound interface.

### GetRegisteredSound

---

GetRegisteredSound(*symbol*)

Returns the sound frame for a registered sound so it can be played.

<i>symbol</i>	A symbol identifying the sound to return. This is the symbol passed to <code>RegSound</code> when the sound was registered.
return value	The sound frame for the sound identified by <i>symbol</i> . For details on sound frames, see “Sound Frame” (page 7-28). If the sound is not found in the registry, the sound frame for the <code>simpleBeep</code> sound is returned.

#### DISCUSSION

##### Note

This global function exists in both the Newton 2.0 and 2.1 operating systems. It was not previously documented for 2.0. ♦

### PlaySound

---

PlaySound(*soundFrame*)

Plays a sound asynchronously.

<i>soundFrame</i>	The sound frame to be played. For details, see “Sound Frame” (page 7-28). Or you can specify a binary object of the class <code>'TDTMFCodec</code> , which contains synthesizer data. In this case, the synthesized sound is played by the synthesizer codec.
return value	Undefined; do not rely on it.

## Sound

## DISCUSSION

The sound is played asynchronously; that is, this function returns immediately and the sound is played as a background process. When the sound is finished playing, the system calls the *callback* function in *soundFrame*, if it's defined.

When passed a binary object of the class 'TDTMFCode, and containing synthesizer data, `PlaySound` is a shortcut for invoking the synthesizer codec to play the sound. This shortcut works for the whole family of `PlaySoundxxx` functions.

Note that `PlaySound` is not a new function in the Newton 2.1 OS. It was inadvertently omitted from the Sound chapter in *Newton Programmer's Guide for Newton 2.0* and *Newton Programmer's Reference*. This function was documented for the 1.x system, still works the same under the 2.1 OS, and is documented again in this chapter for completeness.

## SEE ALSO

For more information about the limitations of this function, see "PlaySound Errata" (page 7-24).

For more information about using the synthesizer codec, see "Synthesizing Sound" (page 7-17). For more information about the format of synthesized sound data, see "Synthesized Sound Data Format" (page 7-32).

**PlaySoundEffect**


---

`PlaySoundEffect(soundFrame, volume, type)`

Plays a sound effect asynchronously, if the user preferences allow the type of effect.

<i>soundFrame</i>	The sound frame to be played. For details, see "Sound Frame" (page 7-28). Or you can specify a binary object of the class 'TDTMFCodec, which contains synthesizer
-------------------	---

## Sound

	data. In this case, the synthesized sound is played by the synthesizer codec.
<i>volume</i>	The volume at which to play the sound. If you specify <i>nil</i> , the system volume is used.
<i>type</i>	Can be one of the symbols 'pen, 'alarm, or 'action, identifying what type of sound effect this is.
return value	Undefined; do not rely on it.

## DISCUSSION

The sound is played only if the user preferences allow sounds of the specified type. For example, if the user preference setting turns off pen sound effects, and *type* is 'pen, then the sound is not played.

This function is preferred to all other versions of `PlaySound` for playing sound effects. If the sound you want to play is not one of the three types of sound effects, then it's best to use a different function for playing sounds.

If *type* is not 'pen, 'alarm, or 'action, then the sound is played at the specified volume (the same as `PlaySoundIrregardlessAtVolume`). After this function executes, the volume is reset back to the level it was at before this function executed; that is, the system volume is not permanently changed.

When passed a binary object of the class 'TDTMFCode, and containing synthesizer data, `PlaySoundEffect` is a shortcut for invoking the synthesizer codec to play the sound. This shortcut works for the whole family of `PlaySoundxxx` functions.

Note that `PlaySoundEffect` is not a new function in the Newton 2.1 OS. It is documented in *Newton Programmer's Reference*, but is duplicated here with some clarification.

## SEE ALSO

For more information about using the synthesizer codec, see "Synthesizing Sound" (page 7-17). For more information about the format of synthesized sound data, see "Synthesized Sound Data Format" (page 7-32).

## Sound

**RegSound**

---

`RegSound(symbol, soundFrame)`

Registers a sound with the system so it is listed as a choice for the system alert sound.

<i>symbol</i>	A symbol identifying the sound. Be sure to append your developer signature to this symbol so the sound is uniquely identified.
<i>soundFrame</i>	A sound frame, as described in “Sound Frame” (page 7-28). This frame should additionally contain a <code>userName</code> slot that holds a string. This string is used as the value of the <code>item</code> slot in the frames constructed by the <code>SoundList</code> function.
return value	Undefined; do not rely on it.

**DISCUSSION**

To unregister a sound, use `UnRegSound`.

You can get a list of all registered sounds by calling `SoundList`.

**Note**

This global function exists in both the Newton 2.0 and 2.1 operating systems. It was not previously documented for 2.0. ♦

**SoundList**

---

`SoundList()`

Returns an array of sounds registered by `RegSound`.

return value	An array of frames. Each frame contains the following slots:
	<code>item</code> A string naming the sound.
	<code>soundSymbol</code> A unique symbol identifying the sound.

## Sound

## DISCUSSION

This function returns an alphabetized array of sounds suitable for use directly in a picker list. For example, you could use this array as the value of the `labelCommands` slot of a `protoLabelPicker`.

Use the function `GetRegisteredSound` to return the sound frame for a registered sound, given its symbol.

**Note**

This global function exists in both the Newton 2.0 and 2.1 operating systems. It was not previously documented for 2.0. ♦

**UnRegSound**

---

`UnRegSound(symbol)`

Unregisters a sound previously registered by `RegisterSound`.

*symbol*                      A symbol identifying the sound to unregister.

return value              Undefined; do not rely on it.

## DISCUSSION

**Note**

This global function exists in both the Newton 2.0 and 2.1 operating systems. It was not previously documented for 2.0. ♦

Sound

# Sound Error Codes

Table 7-8 lists error codes returned by the sound interface and describes possible causes for each.

**Table 7-8** Sound interface error codes

Error code	Possible causes
-30000	Error scheduling node; or codec channel aborted; or no driver found; or channel open; or no sound port; or attempt to do synchronous input.
-30002	Internal sound manager error
-30003	All sound stopped (power off or other problem)
-30006	Internal resource conflict
-30007	Internal resource conflict
-30008	<code>Start</code> called with nothing scheduled; or <code>cancel</code> called for nonexistent node; or codec called with zero-length samples.
-30009	Unable to create codec; or bad value for <code>sndFrameType</code> slot; or bad value for <code>compressionType</code> slot; or bad value for <code>dataType</code> slot; or invalid <code>samplingRate</code> slot; or input not implemented on target hardware; or driver doesn't support output or input.
-30010	Channel aborted ( <code>kChannelAborted</code> returned to sound frame <code>Callback</code> method); or channel stopped (returned to sound frame <code>Callback</code> method); or attempt to make call to closed channel; or unable to start channel; or unable to pause channel; or unable to schedule block; or unable to cancel channel; or unable to stop channel.
-30011	Channel cancelled (sound frame <code>Callback</code> method called); or channel aborted (sound frame <code>Callback</code> method called).

## Sound

## Summary of Sound

---

### Constants

---

```

kDefaultDevice    // 0x00. Default input or output device
kInternalSpeaker  // 0x01. Internal speaker
kInternalMic       // 0x04. Internal microphone
kLineOut          // 0x08. Line out on the interconnect bus
kLineIn           // 0x10. Line in on the interconnect bus

kSampleStandard   // 0. Uncompressed 8-bit samples
kSampleMuLaw      // 1. 8-bit samples encoded by the muLaw compressor
kSampleLinear      // 6. Uncompressed 16-bit samples

k8Bit // 8. Samples are 8 bits each
k16Bit // 16. Samples are 16 bits each

"TMuLawCodec" // MuLaw codec
"TIMACodec"   // IMA codec
"TGSMCodec"   // GSM codec
"TDTMFCodec"  // Synthesizer codec

```

### Data Structures

---

#### Sound Frame

---

```

soundFrame := {
    sndFrameType: symbol, // 'simpleSound or 'codec
    codecName: string, // codec identifier
    bufferCount: integer, // number of codec buffers
    bufferSize: integer, // bytes in each codec buffer
    samples: binary, // sound samples or synthesizer data
    samplingRate: realOrInteger, // sampling rate per second
    compressionType: integer, // encoding format of data
    compressionRatio: real, // ratio of samples to bytes
    dataType: integer, // sample size (8 or 16 bits)

```

## Sound

```

volume: realOrInteger, // volume level
start: integer, // index of first sample to play
count: integer, // number of samples to play
loops: integer, // number of times to repeat sound
Callback: func(state, result)..., // called when play/record finishes
}

```

## Sound Result Frame

---

```

soundResultFrame := {
sound: frame, // sound frame that was paused or stopped
index: integer, // index into sample data where stopped
}

```

## User Configuration Variables

---

```

inputGain // default input gain used during recording
inputDevice // default input device used for recording
outputDevice // default output device used for playback
soundVolumeDb // current system sound volume, in decibels
alarmVolumeDb // current system alarm volume, in decibels

```

## Synthesized Sound Data Format

---

```

type 'DTMF' {
unsigned integer; /* Parameter block type. Set to 1 */
unsigned integer; /* Synthesis type (1-5) */
unsigned integer; /* Reserved set to 0 */
unsigned integer; /* Synth loop count */
unsigned integer = $$CountOf(DTMFTones); /* Number of tone blocks */
    wide array DTMFTones {
        unsigned integer; /* frequency integer part */
        unsigned integer; /* frequency fractional part */
        unsigned integer; /* sustain amplitude */
        unsigned integer; /* leading silence in ms */
        unsigned integer; /* attack in ms */
        unsigned integer; /* decay in ms */
        unsigned integer; /* sustain in ms */
        unsigned integer; /* release in ms */
        unsigned integer; /* peak amplitude */
        unsigned integer; /* trailing silence in ms */
    }
}

```



## Sound

```
    };
};
```

## soundRecorder Object

---

```
sr := GetRoot().soundRecorder;
sr := {
  fSoundFrameSlots: frame, // sound frame overrides for recording
  fInputGain: integer, // sets inputGain slot in the sound channel
  fInputDevice: integer, // sets inputDevice slot in the sound channel
  fOutputDevice: integer, // sets outputDevice slot in the sound channel
  fCallback: func(array)..., // returns sounds when recorder is closed
  OpenRecord: func(callback)..., // opens sound recorder for recording
  OpenPlay: func(soundFrame)..., // opens sound recorder for playing
}
```

## Protos

---

### protoRecorderView

---

```
aProtoRecorderView := {
  _proto: protoRecorderView,
  elapsedTime: integer, // seconds of recorded sound
  statusText: {
    SetState: func(oldState, newState, hasSound)..., // state changed
  }
  GetSounds: func()..., // gets array of sound frames
}
```

### protoSoundChannel

---

```
aProtoSoundChannel := {
  _proto: protoSoundChannel,
  direction: symbol, // 'record or 'play
  outputDevice: integer, // output device identifier
  inputDevice: integer, // input device identifier
  inputGain: integer, // 0-255 or nil (default is used)
  inputBlockSize: integer, // VBO size in bytes, for recording
  Close: func()..., // closes open sound channel
  GetInputGain: func()..., // gets input gain setting
}
```

## Sound

```

GetVolume: func()..., // gets volume setting
IsActive: func()..., // is channel active?
IsOpen: func()..., // is channel open?
IsPaused: func()..., // is channel paused?
NewInputBlock: func(Callback)..., // creates new sound frame/VBO
NewRecording: func()..., // creates new sound channel for recording
Open: func()..., // opens channel
Pause: func()..., // pauses channel
Schedule: func(soundFrame)..., // schedules sound frame
SetInputGain: func(gain)..., // sets input gain
SetVolume: func(volume)..., // sets volume
Start: func(async)..., // starts channel playing/recording
Stop: func()..., // stops channel playing/recording
}

```

protoSoundFrame

---

```

aProtoSoundFrame := {
  _proto: protoSoundFrame,
  length: integer, // number of samples in samples slot
  GetPlayingTime: func()..., // returns playing time in seconds
  GetSampleCount: func()..., // returns number of samples
  GetSampleSize: func()..., // returns size of each sample in bytes
  GetSamplingRate: func()..., // returns sampling rate
  SetRecordingLength: func(numSamples, Callback)..., // sets VBO size
}

```

Functions

---

```

GetRegisteredSound(symbol)
PlaySound(soundFrame)
PlaySoundEffect(soundFrame, volume, type)
RegSound(symbol, soundFrame)
SoundList()
UnRegSound(symbol)

```

# Dial-In Networks

---

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, you 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).

## Dial-in Networks Reference

---

### Data Structures

---

Two data structures are described in the following sections: access frames and network frames.

#### Access Frame

---

An access frame contains the following slots:

##### Slot descriptions

<code>mailNetwork</code>	A symbol for the network.
<code>mailPhone</code>	A string for the phone number.
<code>baud</code>	An integer indicating the baud rate.

#### Network Frame

---

A network frame contains the following slots:

##### Slot descriptions

<code>title</code>	A string describing the network, such as "SprintNet" or "ConcertNet".
<code>id</code>	A symbol uniquely identifying the network.
<code>GetAccessNumbers</code>	A function called to get access numbers for a worksite or city.

##### 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 16-22) in Chapter 16,

## Dial-In Networks

	“Built-in Applications and System Data Reference,” in <i>Newton Programmer’s Reference</i> .
<i>cityFrame</i>	A frame with the same format as the frames returned by the <code>GetCityEntry</code> function; see “GetCityEntry” (page 16-79) in <i>Newton Programmer’s Reference</i> .
return value	Return either an array of access frames, or <code>nil</code> if no numbers are available; access frames are described in “Access Frame” (page 8-2). You should never, however, return the empty array ( <code>[]</code> ).

## DISCUSSION

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.

## Functions

---

The following functions are provided.

### RegDialinNetwork

---

`RegDialinNetwork(networkSym, networkFrame)`

Registers a new dial-in network with the system.

<i>networkSym</i>	A symbol uniquely identifying the network
<i>networkFrame</i>	A network frame, as described in “Network Frame” (page 8-2).
return value	Undefined; do not rely on it.

## DISCUSSION

This function should usually be called from your part’s `InstallScript`, as in the following code sample:

```
DefineGlobalConstant ( 'dudeNetFrame,
{
```

## Dial-In Networks

```

        title: "DudeNet",
        id: 'dudeNet,
        GetAccessNumbers: func(worksites,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*)

Unregisters a dial-in network from the system which had been registered with a call to RegDialinNetwork.

*networkSym*            The symbol used in the call to RegDialinNetwork.

return value            Undefined; do not rely on it.

## Dial-In Networks

## DISCUSSION

This function should usually be called from your part's `RemoveScript`.

**GetLocAccessNums**

`GetLocAccessNums(entry, which)`

Retrieves an array of access frames given a location frame and an array of dial-in network symbols to look for.

*entry*                      A location frame. Can be a worksite or a city location. If `nil`, `GetLocAccessNums` 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 16-22), worksite entries are a type of Names soup entry.

cities                    "GetCityEntry" (page 16-79), the `GetCityEntry` function returns a city location frame.

the current emporium    "User Configuration Variables" (page 16-101), 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.

return value            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. ♦

## Dial-In Networks

**GetAllDialinNetworks**

---

`GetAllDialinNetworks()`

Returns an array of all the dial-in network frames registered in the system.

return value            An array of network frames, see “Network Frame” (page 8-2).

**GetDialinNetwork**

---

`GetDialinNetwork(networkSym)`

Returns the dial-in network frame that corresponds to *networkSym*.

*networkSym*            The symbol of the network whose frame to return.

return value            A network frame; see “Network Frame” (page 8-2).



## Dial-in Networks Summary

---

### Data Structures

---

#### Access Frame

---

```
accessFrame :=
{
    mailNewtork: symbol, // the network's symbol
    mailPhone: string // the phone number
    baud: integer // the baud rate
}
```

#### Network Frame

---

```
networkFrame :=
{
    title: string, //name of network
    id: symbol, //identifies the network
    GetAccessNumbers: function, //get local access numbers
}
```

### Functions

---

```
RegDialinNetwork(networkSym, networkFrame) //regs dial-in network
UnRegDialinNetwork(networkSym) // unregs dial-in network
GetLocAccessNums(entry, which) // gets local access number
GetAllDialinNetworks() // gets all dial-in networks
GetDialinNetwork(networkSym) // // gets a dial-in network
```

Dial-In Networks

# IrDA Communication Tool

---

This chapter describes the IrDA (Infrared Data Association) communication tool built into the Newton 2.1 OS. This communication tool is designed to be accessed and used through the Endpoint interface. For more information about using the Endpoint interface for communications, see Chapter 23, “Endpoint Interface” in *Newton Programmer’s Guide*.

## About the IrDA Communication Tool

---

This section includes general information about the IrDA communication tool.

### Overview

---

The Newton IrDA tool is a communication 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), a transport protocol specification (IrTinyTP), and other protocol layers.

## IrDA Communication Tool

The Newton IrDA tool minimally implements the IrLMP and IrLAP protocols and communicates with a serial driver that implements the SIR protocol.

## Terminology

---

Here is a quick list of the IrDA protocol levels mentioned in this chapter. If you need more definitions and protocol details, you can get them from the IrDA world-wide web site (<http://www.irda.org>).

SIR	(Serial IR), hardware protocol for 9600 to 115.2 Kbps data transmission.
FIR	(Fast serial IR), hardware protocol for 115.2 Kbps to 4 Mbps data transmission.
IrLAP	(Link Access Protocol). Built on top of SIR and/or FIR.
IrLMP	(Link Management Protocol)—Multiplexor, Name Server, Endpoints. Built on IrLMP.
TinyTP	(Transport), built on IrLMP.
IrComm	3-wire/9-wire serial/parallel-like interface. Built on TinyTP.

The only pieces that are supported in this implementation of the Newton IrDA tool are: SIR, IrLAP and IrLMP. This is the minimum required set to be IrDA compliant.

## Using the IrDA Tool

---

Clients of the IrDA tool access it using the Endpoint interface. The IrDA tool service identifier is `kCMSIrDA ("irda")`. Here is an example of how to create an endpoint that uses the IrDA communication tool:

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

## IrDA Communication Tool

All IrDA options are evaluated at endpoint `Connect/Listen` time *only*. For your convenience, the options may be specified earlier at `Instantiate` or `Bind` time, at `connect/listen` time, or even after a connection 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 options.

## Making a Connection

---

Establishing a connection with an IrDA device is a multi-staged process. The first stage of the connection process is the discovery phase. In this stage, the Newton unit probes and accumulates a list of other IrDA devices within beaming range. Each device is loosely identified by a service hints field (PDA/Computer/Printer/etc) and a device nickname. The `kCMOIrDADiscoveryInformation` option is used to specify which device to use and also how the Newton unit should appear to other devices that probe it (in a `Listen` operation).

The second stage of the connection process involves a name lookup of the service to use, or registry of the service provided (in a `Listen` operation). The `kCMOIrDAConnectInformation` option is used to specify this information.

The final stage of the connection process is to make the connection. At this point, a negotiation phase takes place between the Newton unit 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 following options are used to define these negotiation parameters: `kCMOSerialBitRate`, `kCMOIrDAReceiveBuffers`, and `kCMOIrDALinkDisconnectTimeout`.

Note that the `kCMOSerialBitRate` option is not documented in this chapter because it's already covered in the "Built-in Communications Tools" chapter of *Newton Programmer's Reference*. It works slightly different when used with the IrDA tool; it specifies the maximum speed at which you want the Newton device to communicate. The value you specify can be negotiated downwards by the device at the other end of the connection. The default value for the IrDA tool is `k115200bps`, or 115200 bits per second.

For convenience and future compatibility, the known higher FIR speeds defined for IrDA of 576000, 1152000, and 4000000 bps are accepted but

## IrDA Communication Tool

treated as k115200bps (the highest possible speed currently supported by the hardware).

Below is an example NewtonScript option array that specifies all the options that can be 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`.

Typically, the only option that needs to be supplied is the `kCMOIrDAConnectInformation` option, to either identify the 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: kCMOIrDADiscoveryInformation, // IrDA discovery information
   type: 'option',
   opCode: opSetRequired,
   data: {
       arglist: [
           8,
           kIrDASvcHintPDAPalmtop,
           kIrDASvcHintPrinter,
           0,
           1,
       ],
       typelist: [
           'struct',
           'uLong, // num probe slots, default is 8
           'uLong, // my service hint, default is PDA
           'uLong, // service hint mask
           'uLong, // returned devAddr of peer device
           'uLong, // use standard media busy check?
       ],
   },
},
{  label: kCMOIrDAConnectInformation, // IrDA connect info
   type: 'option',
   opCode: opSetRequired,
   data: {
       arglist: [
           0,
           0,
           4, // e.g. strlen("Test")
           5, // e.g. strlen("IrLPT")
       ],
   },
},
]
```

## IrDA Communication Tool

```

        "Test",
        "IrLPT",
    ],
    typelist: [
        'struct',
        'uLong, // my lsap id, default is 0
        'uLong, // peer lsap id, default is 0
        'uLong, // my name length, default is 1
        'uLong, // peer name length, default is 1
        ['array, 'char, 0], // my name, default is "X"
        ['array, 'char, 0], // peer name, default is "X"
    ],
    ),
},
{
    label: kCMOSerialBitRate, // serial bit rate
    type: 'option',
    opCode: opSetRequired,
    data: {
        arglist: [
            115200,
        ],
        typelist: [
            'struct',
            'uLong, // max negotiate speed, default is 115.2k
        ],
    },
},
{
    label: kCMOIrDAReceiveBuffers, // IrDA recv buffers info
    type: 'option',
    opCode: opSetRequired,
    data: {
        arglist: [
            2048,
            1,
        ],
        typelist: [
            'struct',
            'uLong, // size of each receive buffer, default is 512
            'uLong, // number of receive buffers used, default is 1
        ],
    },
},
{
    label: kCMOIrDALinkDisconnectTimeout, // link disconnect threshold
    type: 'option',
    opCode: opSetRequired,
    data: {

```

## IrDA Communication Tool

```

        arglist: [
            8,
        ],
        typelist: [
            'struct',
            uLong, // time before disconnect, default is 40 seconds
        ],
    },
},
];

```

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) 115.2 Kbps, 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.

If the above example option is used to listen, the Newton advertises itself as a palmtop device with the class name "Test". The Newton will communicate (up to) 115.2 Kbps, 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.

## Getting IrDA Tool Information

---

After a connection has been made, you may want to know various results of the connection such as connection speed, buffer size, and so on. The following example option can be used with the endpoint `Option` method to get this information.

```

local connectedOptions :=
[
    {
        label: kCM0IrDADiscoveryInformation, // IrDA discovery info
        type: 'option',
        opCode: opGetCurrent,
        data: {
            arglist: [
                0,
                0,
                0, // service hints of discovered dev
                0, // dev addr of discovered device
                0,
            ],
        },
    },
]

```



## IrDA Communication Tool

```

        typelist: [
            'struct,
            'uLong, // num probe slots, def:8
            'uLong, // my svc hint, default: kIrDASvcHintPDAPalmtop
            'uLong, // service hint mask
            'uLong, // returned devAddr of peer device
            'uLong, // use standard media busy check?
        ],
    },
),
{
    label: kCMOSerialBitRate, // serial bit rate
    type: 'option,
    opCode: opGetCurrent,
    data: {
        arglist: [
            0, // negotiated speed
        ],
        typelist: [
            'struct,
            'uLong, // max negotiate speed, def: 115.2k
        ],
    },
),
{
    label: kCM0IrDAReceiveBuffers, // IrDA recv buffers info
    type: 'option,
    opCode: opGetCurrent,
    data: {
        arglist: [
            0, // negotiated buffer size
            0, // negotiated max buffers in use
        ],
        typelist: [
            'struct,
            'uLong, // size of ea recv buf, def: 512
            'uLong, // num recv bufs used, def: 1
        ],
    },
),
{
    label: kCM0IrDALinkDisconnectTimeout, // link disconn. threshld
    type: 'option,
    opCode: opGetCurrent,
    data: {
        arglist: [
            0, // negotiated disconnect timeout
        ],
        typelist: [

```

## IrDA Communication Tool

```

        'struct,
        'uLong, // Time before disc, def: 40 secs
    ],
},
{
    label: kCMOSlowIRConnect, // "SlowIR" connect info
    type: 'option,
    opCode: opGetCurrent,
    data: {
        arglist: [
            0, // connectOptions results
        ],
        typelist: [
            'struct,
            'uLong, // How did we connect?
        ],
    },
},
];

```

You may also want to know the status of the connection regarding error and retry rates. You can use the `kCMOSlowIRStats` option to return this information. Note that the `kCMOSlowIRStats` option is not documented in this chapter because it's already covered in the “Built-in Communications Tools” chapter of *Newton Programmer's Reference*, under the infrared communications tool.

## Slow IR Connect Option

---

A note about the last option in the previous example. Typically, one communications 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. After the connection is established, one of the two devices has (invisibly to the communications tool client) taken the role of the listener. The return value from the last option in the previous example lets you know if you are the connector or the listener.

If the flag `irActiveConnection` is set in the returned `connect` field, then the communications tool has the role of active connector. If this flag is not set, then the communications tool has the role of passive listener.

IrDA Communication Tool

Note that the `kCMOSlowIRConnect` option is not documented in this chapter because it's already covered in the "Built-in Communications Tools" chapter of *Newton Programmer's Reference*, under the infrared communications tool.

**Note**

The pre-existing infrared communications tool required passing the `kCMOSlowIRConnect` option with `irSymmetricConnect` set in the `connect` field to request "symmetric connecting." But, the IrDA tool does not—the concept is part of IrDA. More importantly, don't confuse the pre-existing `kCMOSlowIRConnect` option and the new IrDA `kCMOIrDAConnectInformation` option. They are two completely different things. ♦

# IrDA Tool Option Reference

This section describes the IrDA communication tool options in detail. Table summarizes the options used with the IrDA tool.

**Table 9-1** Summary of IrDA tool options

Label	Value	Description
<code>kCMOIrDADiscoveryInformation</code>	"irdi"	Discovers other IrDA devices within beaming range.
<code>kCMOIrDAConnectInformation</code>	"irci"	Finds name of service to use, or registers service provided by the unit.
<code>kCMOIrDAReceiveBuffers</code>	"irrb"	Sets size and number of receive buffers.
<code>kCMOIrDALinkDisconnectTimeout</code>	"irdl"	Sets the timeout period.
<code>kCMOIrDAConnectUserData</code>	"ircd"	Advanced option to send or receive out of band data.
<code>kCMOIrDAConnectAttrName</code>	"irca"	Advanced option to register under a different attribute name.

## IrDA Communication Tool

**Table 9-1** Summary of IrDA tool options

Label	Value	Description
kCMOSerialBitRate	"sbps"	Changes the bps rate.
kCMOSlowIRConnect	"irco"	Controls how the connection is made.
kCMOSlowIRStats	"irst"	Read-only option returns statistics about the data received and sent.

Note that the last three options, `kCMOSerialBitRate`, `kCMOSlowIRConnect`, and `kCMOSlowIRStats`, are listed here because they can be used with the IrDA tool, but these options exist in the Newton 2.0 OS—they are not new in 2.1. They are documented in the *Newton Programmer's Reference* under the asynchronous serial tool (`kCMOSerialBitRate`) and the infrared tool (`kCMOSlowIRConnect` and `kCMOSlowIRStats`).

## Discovery Option

In the discovery phase of a connection, the Newton unit probes and accumulates a list of other IrDA devices within beaming range. Each device is loosely identified by a service hints field (PDA/Computer/Printer/etc.) and a device nickname. The `kCM0IrDADiscoveryInformation` option is used to specify which device to use and also how the Newton unit should appear to other devices that probe it.

You can also use this option after a connection has been made, to return information about the connection (the service hints of the discovered device, and its device address).

The following example shows the use of this option:

```
local option := {
  label: kCM0IrDADiscoveryInformation, // "irdi"
  type: 'option',
  opCode: opSetRequired,
  data: {
    arglist: [
      8, // 8 probe slots
      kIrDASvcHintPDAPalmtop, // service hint
      kIrDASvcHintPrinter, // service hint mask
    ]
  }
}
```

IrDA Communication Tool

```
        0,                // devAddr of peer device
        1,                // use standard media busy check
    ],
    typelist: [
        'struct',
        'uLong, // fProbeSlots
        'uLong, // fMyServiceHints
        'uLong, // fPeerServiceHints
        'uLong, // fPeerDevAddr
        'uLong, // fMediaBusyCheck
    ],
    },
};
```

The fields in the IrDA discovery option frame are described in Table 9-2.

**Table 9-2** IrDA discovery option fields

Option field	Description
fProbeSlots	The number of “slots” used during probing. The valid choices are listed in Table 9-3. The default is kIrDA8ProbeSlot. This value relates to an IrDA collision-avoidance mechanism. It is recommended that you use the default setting unless you have a specific reason for changing it.
fMyServiceHints	The category of device that you wish to identify yourself as. The service hints are listed in Table 9-4. The default value is kIrDASvcHintPDAPalmtop.

IrDA Communication Tool

**Table 9-2** IrDA discovery option fields

Option field	Description
fPeerServiceHints	A mask used to identify the categories of devices that you wish to connect to. You can OR together any combination of the values in Table 9-4 to construct this value. The default value is 0xFFFFFFFF (accept any device during discovery phase).
fPeerDevAddr	Read-only. Returns the address of the discovered device.
fMediaBusyCheck	To enable a 600ms. delay before discovery begins, specify true (default). To disable the delay, specify nil. It is highly recommended that you use the default setting. 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 compliancy with other IrDA devices.

Note that the service hints can be bit-OR'd together.

You can specify whatever you wish for the fMyServiceHints field, but the value kIrDASvcHintPDAPalmtop is always OR'd in by the IrDA tool.

**Table 9-3** IrDA discovery option probe slots constants

Constant	Value
kIrDA1ProbeSlot	1
kIrDA6ProbeSlot	6
kIrDA8ProbeSlot	8
kIrDA12ProbeSlot	12

IrDA Communication Tool

**Table 9-4** IrDA discovery option service hint constants

Constant	Value
kIrDASvcHintPnPCompatible	0x00000001
kIrDASvcHintPDAPalmtop	0x00000002
kIrDASvcHintComputer	0x00000004
kIrDASvcHintPrinter	0x00000008
kIrDASvcHintModem	0x00000010
kIrDASvcHintFAX	0x00000020
kIrDASvcHintLanAccess	0x00000040
kIrDASvcHintTelephony	0x00000100
kIrDASvcHintFileServer	0x00000200

When used with the opcode `opGetCurrent`, the field `fPeerServiceHints` returns the service hints reported by the discovered device.

**Note**

Fields used during an endpoint `Connect` operation include `fProbeSlots`, `fPeerServiceHints` and `fMediaBusyCheck`. Fields used during an endpoint `Listen` operation include `fMyServiceHints`. ♦

## Connection Information Option

The second stage of the connection process involves a name lookup of the service to use, or registry of the service provided. The `kCM0IrDAConnectInformation` option is used to specify this information.

The following example shows the use of this option:

```
local option := {
```

## IrDA Communication Tool

```

label: kCM0IrDAConnectInformation, // "irci"
type: 'option',
opCode: opSetRequired,
data: {
    arglist: [
        0,           // LSAP id
        0,           // peer LSAP id
        4,           // strlen("Test")
        5,           // strlen("IrLPT")
        "Test",      // my name
        "IrLPT",     // peer name
    ],
    typelist: [
        'struct',
        'uLong,           // fMyLSAPId
        'uLong,           // fPeerLSAPId
        'uLong,           // fMyNameLength
        'uLong,           // fPeerNameLength
        ['array', 'char', 0], // fMyName
        ['array', 'char', 0], // fPeerName
    ],
},
};

```



IrDA Communication Tool

The fields in the IrDA connection information option frame are described in Table 9-5.

**Table 9-5** IrDA connection information option fields

Option field	Description
fMyLSAPId	Set to 0 and the IrDA comm tool chooses a random LSAP id (between 1 and 31), or you may set a specific value from 1 to 31. The default is 0.
fPeerLSAPId	Set to 0 to do the service lookup by name, or specify any other value from 1 to 111. If the value is non-zero, then the name lookup phase is skipped and connection is made using an LSAP id with that value. The default is 0.
fMyNameLength	The length, in characters, of the fMyName string. The default is 1.
fPeerNameLength	The length, in characters, of the fPeerName string. The default is 1.
fMyName	A string identifying the service provided by the Newton unit. The default is "X".
fPeerName	A string identifying the service provided by the peer device. The default is "X".

Note that the only reason to use a specific value for the fMyLSAPId field would be to advertise your service by number instead of by name—and that is not recommended per IrDA standards. The option is left available in case your application needs to communicate with an older desktop IrDA application.

Likewise for the fPeerLSAPId field—per IrDA standards, services should be identified and looked up by name, not number. But if you need to communicate with an older desktop IrDA application that supports access only by LSAP id (no name look up) then you will be able to by specifying the id number directly.

## IrDA Communication Tool

**Note**

Fields used during an endpoint `Connect` operation include `fPeerLSAPId`, `fPeerNameLength` and `fClassNames`. Fields used during an endpoint `Listen` operation include `fMyLSAPId`, `fMyNameLength` and `fClassNames`. ♦

## Receive Buffers Option

---

The receive buffers option, `kCMOIrDAReceiveBuffers`, sets the number and size of buffers used for receiving data.

The following example shows the use of this option:

```
local option := {
    label: kCMOIrDAReceiveBuffers, // IrDA rcv buffers info
    type: 'option',
    opCode: opSetRequired,
    data: {
        arglist: [
            2048, // buffer is 2048 bytes
            1,    // allocate one buffer
        ],
        typelist: [
            'struct',
            'uLong, // size of each receive buffer, default is 512
            'uLong, // number of receive buffers used, default is 1
        ],
    },
};
```

The first field specifies the size, in bytes, of the receive buffers used by the IrDA tool. The default is 512 bytes. Valid values are 64, 128, 256, 512, 1024, or 2048.

The second field specifies the number of receive buffers used by the IrDA tool (window size in IrDA terminology). The default is 1. Valid values range from 1 to 7.

Note that you may request a large buffer (and/or a large number of buffers), but the actual buffer size and number of buffers may be less if the negotiated speed is less than the maximum that you request.

This option applies to both `Connect` and `Listen` operations.

## IrDA Communication Tool

## Link Disconnect Option

---

The link disconnect option, `kCM0IrDALinkDisconnectTimeout`, sets the time (in seconds) before communication is terminated, if no activity is received from the peer device.

The following example shows the use of this option:

```
local option := {
    label:  kCM0IrDALinkDisconnectTimeout, // link disconnect threshold
    type:   'option',
    opCode: opSetRequired,
    data: {
        arglist: [
            8,      // disconnect after 8 seconds
        ],
        typelist: [
            'struct',
            uLong, // time before disconnect, default is 40 seconds
        ],
    },
};
```

Proper communication protocol between IrDA devices is to send data or a “ready-to-receive” packet every 500ms (minimum). If no such activity is detected for the number of seconds specified by this option, then the IrDA tool is disconnected. The default value is 40 seconds. Valid values are 3, 8, 12, 16, 20, 25, 30, and 40.

This option applies to both `Connect` and `Listen` operations.

Note that after 3 seconds of non-activity, a disconnect warning event is sent from the IrDA tool via the endpoint `EventHandler` method. The values used

IrDA Communication Tool

for the `eventCode` and `data` slots of the event frame passed to `EventHandler` are shown in Table 9-6.

**Table 9-6** Disconnect warning event values

Constant	Value
<b>Event codes</b>	
<code>kEventToolSpecific</code>	1
<code>kEventDisconnect</code>	2
<code>kEventRelease</code>	3
<b>IrDA tool event data</b>	
<code>kDisconnectWarningEvent</code>	1

## Connect User Data Option

This rarely used option that may be needed for comm tools that wish to build upon the base IrDA comm tool. This is the `kCM0IrDAConnectUserData` option, which is used during connect time to send/receive “out of band” data. See the IrDA documentation to get an idea of how this would be used (specifically IrTinyTP and IrComm).

```
local option := {
    label: kCM0IrDAConnectUserData, // IrDA connect user data
    type: 'option',
    opCode: opSetRequired,
    data: {
        arglist: [
            3, // length of data; strlen("foo")
            "foo", // data string
        ],
        typelist: [
            'struct',
            'uLong, // fDataLength; default is 0
            ['array', 'char', 0], // fData
        ],
    },
};
```

## IrDA Communication Tool

The fields `fDataLength` and `fData` are used together to describe data to be sent/received during connect/listen.

## Attribute Name Option

---

The option `kCMOIrDAConnectAttrName` is intended for communications tools (built upon the IrDA 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 communications tool. The default value for the IrDA tool (which is implemented at the IrLMP level) is “IrDA:IrLMP:LsapSel”. For example, a communications tool that implements TinyTP would supply a value of “IrDA:TinyTP:LsapSel”, per the TinyTP spec.

```
local option := {
    label: kCMOIrDAConnectAttrName, // IrDA attribute name
    type: 'option',
    opCode: opSetRequired,
    data: {
        arglist: [
            19, // length of data; strlen("IrDA:TinyTP:LsapSel")
            "IrDA:TinyTP:LsapSel", // data string
        ],
        typelist: [
            'struct',
            'uLong, // fNameLength; default is 0
            ['array, 'char, 0], // fName
        ],
    },
};
```

The fields `fNameLength` and `fName` are used together to describe the LSAP id attribute name registered for Listen/Connect or looked up for Connect operations.

# IrDA Tool Error Codes

Table 9-7 lists error codes returned by the IrDA communications tool.

**Table 9-7** IrDA tool error codes

Error code	Description
-38502	Wrong class specified in the <code>kCM0IrDAConnectInformation</code> option.
-38504	Disconnected on/by the remote side.
-38505	Lost connection with the remote side.

# Summary of IrDA Tool

---

## IrDA Tool Service Option Label

---

kCMSIrDA	"irda"
----------	--------

## IrDA Tool Options

---

kCMOIrDADiscoveryInformation	"irdi"
kCMOIrDAConnectInformation	"irci"
kCMOIrDAReceiveBuffers	"irrb"
kCMOIrDALinkDisconnectTimeout	"irlld"
kCMOIrDAConnectUserData	"ircd"
kCMOIrDAConnectAttrName	"irca"
kCMOSerialBitRate	"sbps"
kCMOSlowIRConnect	"irco"
kCMOSlowIRStats	"irst"

## Constants

---

### IrDA Discovery Information Option Constants

---

kIrDA1ProbeSlot	1
kIrDA6ProbeSlot	6
kIrDA8ProbeSlot	8
kIrDA12ProbeSlot	12
kIrDASvcHintPnPCompatible	0x00000001
kIrDASvcHintPDAPalmtop	0x00000002
kIrDASvcHintComputer	0x00000004
kIrDASvcHintPrinter	0x00000008
kIrDASvcHintModem	0x00000010
kIrDASvcHintFAX	0x00000020
kIrDASvcHintLanAccess	0x00000040
kIrDASvcHintTelephony	0x00000100
kIrDASvcHintFileServer	0x00000200

IrDA Communication Tool

IrDA Link Disconnect Warning Event Constants

---

kEventToolSpecific	1
kEventDisconnect	2
kEventRelease	3
kDisconnectWarningEvent	1



# eMate Multi-User Mode

---

In some classrooms, eMate units are shared among several students who keep their work on the eMate for several days or longer. A particular eMate unit may be used by several different people in one day. This requires the ability to protect each student's data from other students. The teacher can do this by setting up the unit to operate in Classroom mode with multiple users.

The password-protected Teacher Setup application allows the teacher to choose the unit's operating mode (Classroom mode or Full Newton System mode); to select which built-in applications will be available to users in Classroom mode; and to set up user accounts on the unit. When the unit is in Classroom mode and it wakes up, the user is required to log in before they can begin using the unit. Depending on how the teacher has configured the unit, the user may be required to enter a password in addition to their user name.

Note that there is a distinction between Classroom mode and multi-user mode. The unit must be in Classroom mode in order to be in multi-user mode, but it can be in Classroom mode without being in multi-user mode. Classroom mode limits the applications available in the Extras Drawer, while multi-user mode allows multiple users of the unit to keep their data separate.

Applications designed to be used with eMate, including Newton Works, create a separate soup for each user, as well as a generic soup that is used when the unit is in Full Newton System mode. Applications not modified to

## eMate Multi-User Mode

work with eMate in multi-user mode will work just as they do on other Newton units—all users can see all data.

After logging in, the user can begin work on the eMate. For all applications that have been written to work in multi-user mode, the user sees only the data that has been created by himself or herself. For other applications (including all of the built-in applications besides Newton Works) the user sees all of the data.

Other users' data is hidden in separate soups for each user. The name of the user associated with a soup is saved in the soup information frame in the `userName` slot. Each multi-user-aware application displays data only from the soup corresponding to the current user. Each application must register to be notified of changes to the current user variable (`KCurrentUser`) in the system user configuration data so that it can change users after the eMate wakes up.

In a multi-user aware application, if the user switches to Full Newton System mode from Classroom mode, the generic (accessible to all users) soup should be shown.

## Using Multi-user Mode

---

Applications must do a few things to support Classroom multi-user mode.

- When the application opens, check the current user and open the soup corresponding to that user. Note that you can name the application-created soups anything you want, but it is recommended that you use names created by the `GenSoupName` method.
- The application should not show the filing folder tab when it opens and the unit is in Classroom mode. This user interface guideline is used to simplify the interface by removing the filing folder feature and prevents a user from deleting or changing the names of folders that might have been used by others.
- The application must provide a `GetBackupSoupNames` method that returns an array of the names of the soups for the current user, so the Classroom Dock application can back up the data. If this method is not supplied, Classroom Dock cannot back up user's data.

## eMate Multi-User Mode

- The application must be aware that the current user can change while the application is open. To handle this, the application can register (use `RegUserConfigChange`) to be notified of changes to the user configuration variable `KCurrentUser`. Here's an example of a method that responds to a user change:

```
func(changeSym) begin
  if changeSym = 'KCurrentUser then
    GetRoot().HomePage:MultiUserSwitch('newtWorks');
  end
```

The `MultiUserSwitch` method closes and reopens the application when the user changes. This allows the application to begin displaying data for a different user.

- An application that works in Classroom mode can set a `KClassroomAware` slot in its base view to `true`, causing the application to be listed as one of the recommended applications for Classroom mode availability in the Edit Extras slip (which is accessed via the Teacher Setup application). Do not set the `KClassroomAware` slot unless you implement the other requirements in this section.

## Reference

---

### User Configuration Variables

---

This section describes those user configuration variables associated with Classroom and multi-user mode. Note that you should always use the function `GetUserConfig` to access these variables. You can use the function `RegUserConfigChange` to register to be notified of changes to the user configuration data. These functions are documented in *Newton Programmer's Reference*.

#### Variable descriptions

<code>KSimpleMode</code>	True if unit is in Classroom mode, nil otherwise
<code>KCurrentUser</code>	String naming current user, or nil for none

## Functions and Methods

---

The functions and methods in this section are used by applications to implement behavior that supports multi-user mode.

### GetBackupSoupNames

---

*app*:GetBackupSoupNames()

Returns an array of strings corresponding to the names of this application's soups for the current user.

return value	An array of one or more soup name strings corresponding to the current user.
--------------	--

#### DISCUSSION

You must supply this method in your application if you want to support multi-user mode.

This method is called by the Classroom Dock application when the user wants to back up his or her data to the classroom server.

If the unit is in Full Newton System mode, this method should return the name of the generic (accessible to all users) application soup.

The following line of code shows one way to construct an appropriate return value for the `GetBackupSoupNames` method.

```
[
  GetRoot().HomePage:GenSoupName(allSoups.myApp.soupBaseName,
    if GetUserConfig('KSimpleMode)
      then GetUserConfig('KCurrentUser) )
];
```

This example uses the `GenSoupName` method of the `HomePage` application to construct an appropriate soup name string. This method returns a string by concatenating a base soup name with the current user name, or with `nil` if the unit is not in Classroom mode. Note that the application identified as `HomePage` in the root frame contains the Teacher Setup slip, controls display of the simplified Extras Drawer when the unit is in Classroom mode, and performs other functions related to Classroom mode.

## eMate Multi-User Mode

**MultiUserSwitch**

---

`HomePage:MultiUserSwitch(appSymbol)`

Closes and opens an application when the current user changes.

*appSymbol*                      The application symbol of the application you want to close and reopen.

return value                  Undefined; do not rely on it.

**DISCUSSION**

You can call this method to close and reopen your application when the unit is in multi-user mode and the current user changes. You need to call this method only if your application is currently displaying data for a user when the user is changed. For example, if it is displaying a clock or some other data that is not associated with a particular user, then you don't need to call this method.

Call this method like this:

```
GetRoot().HomePage:MultiUserSwitch(myAppSymbol);
```

**GenSoupName**

---

`HomePage:GenSoupName(soupName, currentUser)`

Returns a string that can be used as the soup name for a given user.

*soupName*                      A string identifying a soup. This string must not contain more than 20 characters.

*currentUser*                   A string identifying the current user. This string must not contain more than 19 characters.

return value                  A string formed by concatenating *soupName* with *currentUser*, with a semicolon between them. If *currentUser* is `nil` (indicating that the unit is not in multi-user mode), *soupName* is returned without being modified.

## eMate Multi-User Mode

## DISCUSSION

The result of this method can be used as the soup name for a soup holding data for a particular user, when the unit is in multi-user mode.

If the *soupName* string contains a colon followed by a signature, *GenSoupName* moves the colon and signature to the end of the string it returns, so it follows the *currentUser* portion of the returned string. Here is an example of this situation:

```
soupName := "WidgetPlanner:mySig"; currentUser := "Anna";
GetRoot().HomePage:GenSoupName(soupName,currentUser);

// return value
"WidgetPlanner;Anna:mySig"
```

If the *soupName* string does not contain a colon, then the two strings are simply concatenated, separated by a semicolon, as shown in this example:

```
soupName := "WidgetPlanner"; currentUser := "Anna";
GetRoot().HomePage:GenSoupName(soupName,currentUser);

// return value
"WidgetPlanner;Anna"
```

If the *soupName* string is more than 20 characters long (signature included) and the *currentUser* string is exactly 19 characters, then *GenSoupName* returns a string longer than 39 characters. If you pass this string to a function that creates a soup, the system truncates the soup name to 39 characters, which truncates the developer signature. It is imperative that the *soupName* string be 20 characters or fewer. *GenSoupName* does not check for length.

Note that the system automatically limits the length of user names to a maximum of 19 characters.

## Summary of Multi-User Mode

---

### User Configuration Variables

---

```
kSimpleMode // true if unit is in classroom mode, nil otherwise  
kCurrentUser // string naming current user
```

### Functions and Methods

---

```
app:GetBackupSoupNames()  
HomePage:MultiUserSwitch(appSymbol)  
HomePage:GenSoupName(soupName, currentUser)
```

eMate Multi-User Mode



# Miscellaneous

---

This chapter lists miscellaneous changes to the Newton 2.1 OS that are not lengthy enough to warrant their own chapters. It is organized first by type of information (data structures, protos, constants, and functions and methods), and then by *Newton Programmer's Guide* chapter.

## Reference

---

### Data Structures

---

#### Views

---

A new view slot `hilitedData`, and the format of clipboard data frames are described in this section.

#### View Slot

---

The following view slot is new to Newton 2.1 OS:

## Miscellaneous

**Slot description**`highlightedData`

This slot states whether this view currently has data that can be cut or copied with the global command keys. If a view has this slot with a value of `true`, it will be sent a `ViewAddDragInfoScript` message, when the global command keys are used. If the command was a cut (as opposed to a copy) the view will also be sent a `ViewDropRemoveScript` message.

**Clipboard Data Frame**

---

A clipboard data frame is the frame returned by `GetClipboard`, and passed to `SetClipboard`. It has the following slots:

**Slot descriptions**`label`

A string; the text displayed by the clipboard item.

`types`

Array of types arrays, one types array per item in the clipboard item. The number and order of these types arrays must match that of the data arrays in the `data` slot. Each types array contains symbols representing the types of data in the corresponding data array. Each

## Miscellaneous

symbol specifies the type of data in the corresponding element within the data array.

For example, the following 1-element `types` array describes a clipboard with one item, that can be seen as either a picture or a text:

```
'[ [text,picture] ]
```

Note that the nested array is ordered with preferred type first. If the destination view accepts both text and pictures, the text is passed to the destination view.

This next 2-element `types` array on the other hand, describes a clipboard with two items, a string and a picture:

```
'[ [text],[picture] ]
```

The system can display types of 'text, 'polygon, 'ink, and 'picture. The type of data the system requires for these types is listed in Table 11-1.

data

Array of data arrays, one data array per item on the clipboard. The number and order of data arrays, must match the number and order of types arrays in the `types` slot. Each data array should contain the data corresponding to that type in the array in the `types` slot. For example the data in `clipboardFrame.data[i][j]` should be of the type specified by `clipboardFrame.types[i][j]`.

Each element within the nested arrays can be any NewtonScript object. If you specified a 'text, 'polygon, 'ink, and 'picture data types; these array elements should be frames with the slots listed in Table 11-1.

bounds

A bounds frame; where the data came from in global coordinates.

Miscellaneous

**Table 11-1** Clipboard data types accepted by the system

types	Required Slots	Optional slots
'text	text	any other clParagraphView slots
'polygon	points viewBounds	any other clPolygonView slots
'ink	ink viewBounds	any other clPolygonView slots
'picture	icon viewBounds	any other clPictureView slots

Built-In Applications

A folder symbol has been added for the button bar, the `cityAlias` slot of Names worksite soup entries has changed, and a number of user configuration variables have been added. The soup format of Works word processor entries is described.

Extras Drawer Folder Symbols

The following symbols are used for the `labels` slot of part entries by the Extras Drawer:

<code>nil</code>	Unfiled.
<code>'_extensions</code>	Extensions.
<code>'_help</code>	Help.
<code>'_setup</code>	Set up.
<code>'_soups</code>	Storage.
<code>'_ButtonBar</code>	The button bar.

Names Worksite Soup Entry

Worksite entries in the Names soup contain a `cityAlias` slot. Previous version of the Newton OS stored a entry alias to an undocumented soup in this slot. In Newton 2.1 OS this slot contains an array with information about the city, or `nil` if there is no city information. Note that `ResolveEntryAlias` returns `nil` if passed in an array (or anything other than a valid entry alias).

## Miscellaneous

**Newton Works Word Processor Soup Format**

---

Newton Works word processor soup entries have the following slots.

**Slot descriptions**

<code>class</code>	The symbol 'paper'.
<code>version</code>	Integer, the current version of the entry.
<code>title</code>	String which is the document title.
<code>timeStamp</code>	Creation date of the entry.
<code>realModTime</code>	Date the entry was most recently modified.
<code>saveData</code>	Frame returned from <code>protoTXView Externalize</code> call (page 3-36).
<code>hiliteRange</code>	Frame with the document's highlight range (see "The Range Frame" (page 3-22)).
<code>margins</code>	Frame with slots <code>top</code> , <code>left</code> , <code>bottom</code> , <code>right</code> , which are the document's margins in pixels.

**User Configuration Variables**

---

The following user configuration variables are new to Newton 2.1 OS:

**Slot descriptions**

<code>LCDContrast</code>	On units that support software control of the LCD contrast setting, this slot contains the current contrast setting. It can also be used to modify the current contrast. Use the <code>kGestaltArg_HasSoftContrast Gestalt</code> selector to check if a Newton device has software LCD control, and the maximum and minimum values.
<code>alarmVolumeDb</code>	Sets the system wide alarm volume in decibels. Use the <code>kGestaltArg_VolumeInfo Gestalt</code> selector to find the range of allowable values for the volume.
<code>soundVolumeDb</code>	Sets the system wide volume in decibels. Use the <code>kGestaltArg_VolumeInfo Gestalt</code> selector to find the range of allowable values for the volume.
<code>buttonBarPositions</code>	A 4-element array, specifying the position of the button bar in each of the four possible screen orientations. Each

## Miscellaneous

element in the array can be either `nil`, specifying that the default setting should be used, or one of the following symbols: `'top`, `'left`, `'right`, or `'bottom`.

The array elements are ordered using the screen orientation constants as indices to this array, see “Screen Orientation Constants” (page 11-12). That is, `buttonBarPositions[kPortrait]` should hold information for the portrait screen orientation.

`buttonBarControlsPositions`

A 4-element array, specifying the position of the controls—overview button and scroll arrows—in each of the four screen orientations. Each array element can be `nil`, specifying that the default value be used, or the symbols `'top` and `'bottom` for when the button bar is on the left or right sides of the screen, or `'left` and `'right` for when the button bar is on the top or bottom of the screen.

The array elements are ordered using the screen orientation constants as indices to this array, see “Screen Orientation Constants” (page 11-12). That is, `buttonBarControlsPositions[kPortrait]` should hold information for the portrait screen orientation.

`bellyButtonPositions`

A 4-element array, specifying the position of the overview button relative to the scroll arrows in each of the four screen orientations. Each array element can be `nil`, specifying that the default value be used, or the symbols `'outside`, `'inside`, `'left`, and `'right`.

The array elements are ordered using the screen orientation constants as indices to this array, see “Screen Orientation Constants” (page 11-12). That is, `bellyButtonPositions[kPortrait]` should hold information for the portrait screen orientation.

`buttonBarIconSpacingH`

An integer specifying the number of pixels spacing icons in the button bar when the button bar is laid out horizontally — across the top or bottom of the screen.

## Miscellaneous

The default is 40 on the MessagePad 2000. To restore this settings to its default value, set it to `nil`.

`buttonBarIconSpacingV`

An integer specifying the number of pixels spacing icons in the button bar when the button bar is laid out vertically — across the left or right sides of the screen. The default is 40 on the MessagePad 2000. To restore this settings to its default value, set it to `nil`.

`extrasIconSpacingH`

An integer specifying the horizontal spacing of icons in the Extras Drawer in pixels. The default is 64 in the MessagePad 2000. This value has no effect when the Extras Drawer is in overview mode. To restore this settings to its default value, set it to `nil`. This value is not used in systems prior to Newton 2.1 OS.

`extrasIconSpacingV`

An integer specifying the vertical spacing of icons in the Extras Drawer in pixels. The default is 52 in the MessagePad 2000. This value has no effect when the Extras Drawer is in overview mode. To restore this settings to its default value, set it to `nil`. This value is not used in systems prior to Newton 2.1 OS.

`extraFont`

The font used for the icon labels in both the Extras Drawer and the button bar. While you can use both an integer font spec or a font spec frame, it is strongly recommended that you use only integer font specs, such as `userFont9 + tsPlain` or `simpleFont9 + tsBold`. Using the integer representation in this instance accomplishes two things: it reduces NewtonScript Heap usage and it restricts you to the set of built-in fonts. Using a font that's not in ROM is extremely dangerous, because the font could be removed. This information is stored in a soup. A user may be forced to do a hard reset in order to remove a bad font specification.

This value is not used in systems prior to Newton 2.1 OS.

## Miscellaneous

## Protos

---

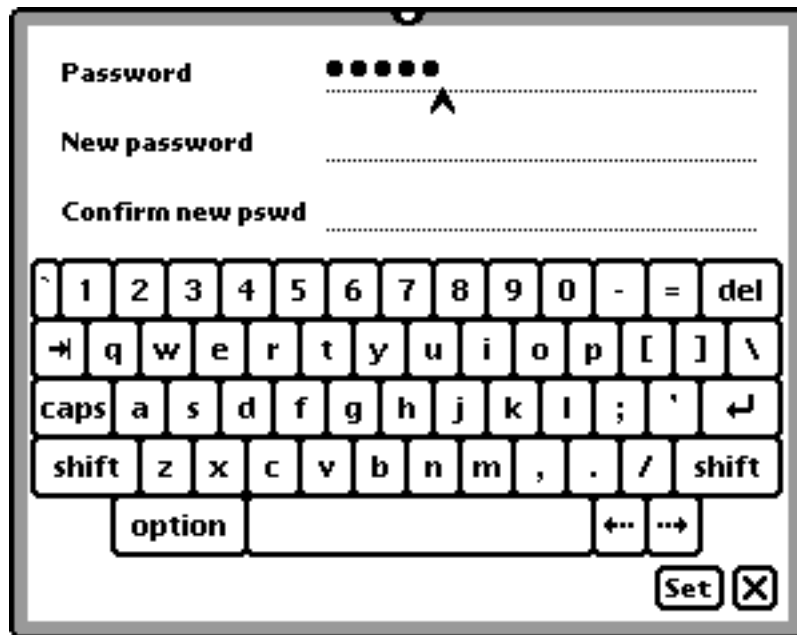
This section describes `protoPasswordSlip` and `protoBlindEntryLine`.

### protoPasswordSlip

---

This proto allows the user to create a new password or enter an existing password without echoing the password in plain text. The typed keys appear as bullets in the input line. A view created from `protoPasswordSlip` is shown in Figure 11-1. Note that the slip does not include an embedded keyboard when created on a Newton device with a hardware keyboard attached.

**Figure 11-1** A view created from `protoPasswordSlip`





## Miscellaneous

**Note**

This proto exists in Newton 2.0 OS, but was not previously documented. ♦

This proto has one slot of interest:

**Slot description**

`verifyPassword`

The symbol `'verifyOnly`, `true`, or `nil`. This slot determines if the password slip is used to just ask for a password, or if it is used to change a password.

A value of `'verifyOnly` specifies a password slip that queries a user for a password, but does not allow the user to change the password. In this case, the slip includes only a “Password” entry line.

A value of `true` means the user is queried for the old password, and may also change the password. This is the default. In this case the slip has all three entry lines: “Password,” “New Password,” and “Confirm Password.”

A value of `nil` means the user can change the password without entering the old one. In this case the slip includes only the “New Password” and “Confirm Password” entry lines.

This proto has the following methods of interest:

**CurrentPassword**


---

`passwordSlip:CurrentPassword()`

Called to retrieve the current password.

return value

A string for the current password, or `nil` if there is no current password.

## Miscellaneous

**SetPassword**

---

*passwordSlip*: SetPassword(*newPassword*)

Called to set a new password.

*newPassword*            A string, the new password to store.

return value            You can return anything; it is ignored.

**DISCUSSION**

Note that the password is a string in plain text, so for maximum security it should be encoded before being stored.

**MatchPassword**

---

*passwordSlip*: MatchPassword(*newPassword*, *currentPassword*)

Called to verify that the correct password has been entered

*newPassword*            A string for the password entered by user*currentPassword*        A string for the current password as returned by  
CurrentPassword.return value            Return `true` if the two match, `nil` if not.**MatchedPassword**

---

*passwordSlip*: MatchedPassword()

Called if a valid password was entered.

return value            You can return anything; it is ignored.

**DISCUSSION**

You must call the inherited method to correctly close the password slip.

**protoBlindEntryLine**

---

This proto allows text to be entered, without echoing the text back to the user. This proto is used in the `protoPasswordSlip`. It is shown in Figure 11-2

## Miscellaneous

**Figure 11-2** A view based on `protoBlindEntryLine`

This proto has three slots of interest:

**Slot descriptions**

<code>dummyChar</code>	Optional. A character containing the text to display instead of the real text. By default, the bullet character is used.
<code>realText</code>	The string the user has typed. You should use this slot for looking up the value of the text (instead of looking in the <code>text</code> slot). Do not modify this slot directly. Use the <code>UpdateText</code> method.
<code>label</code>	Optional. The string used as the label of the entry lines.

**UpdateText**


---

*blindEntryLine*:`UpdateText(newText)`

Sets the value of the `realText` slot to the value in *newText*, and correctly updates the string displayed to the user.

<i>newText</i>	A string, the new value for the blind entry line.
return value	Undefined; do not rely on it.

**Constants****Views**


---











The following constants represent the four possible screen orientations.

Miscellaneous

Screen Orientation Constants

The four screen orientation constants are shown in Figure 11-3 (page 11-12).

Figure 11-3 Screen orientation constants

kPortrait 0	kLandscape 1	kPortraitFlip 2	kLandscapeFlip 3
			
			
	NA	NA	

Built-In Communication Tools

A new serial communication tool option is described.

Serial Communication Tool Sound Option

There is a new serial communication tool option for enabling sound pass-through using a PCMCIA card. Here is an example of what the option looks like:

```
local option := {
type: 'option,
label: kCMOPCMCIAModemSound, //"msnd"
opCode: opSetRequired,
form: 'template,
result: nil,
data: {
    arglist: [nil],
```

## Miscellaneous

```

        typelist: ['struct', 'boolean'],
    },
}

```

The `arglist` value is either `true` or `nil`. If `true`, sound pass-through is enabled. If `nil`, sound pass-through is disabled.

You would normally use this option with a PCMCIA modem using the serial tool. The modem tool automatically enables sound pass-through, so you should not need to use this option with the modem tool.

This option should be used only after the serial endpoint has connected.

**Note**

Sound pass-through should be disabled before the endpoint is disconnected. If it is not, power consumption increases and the speaker emits an annoying sound.

Sound pass-through only works for PCMCIA cards which support it through the PCMCIA specification.

This option is only for use in Newton OS 2.1 and higher. ♦

## Functions and Methods

---

The following methods and functions are either new to Newton 2.1 OS, have changed since previous OS releases, or have existed but were not previously documented. Unless otherwise noted in the COMPATIBILITY section of a function's description, all functions described here are new to Newton 2.1 OS.

## Miscellaneous

Views

---

**DragAndDrop**

---

*view*: DragAndDrop(*unit*, *dragBounds*, *pinBounds*, *copy*, *dragInfo*)

Starts the drag and drop process, returning when the dragged item(s) is dropped into a view or into the clipboard; it is usually called from a ViewClickScript.

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

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

*pinBounds*            A bounds frame or *nil*. The bounds to use when constraining the object within the app area. If you pass *nil*, the drag object's bounds, *dragBounds*, are used. If the object being dragged is almost the size of the app area, you may want to specify a smaller bounds frame than *dragBounds*, otherwise the object may not appear to move far enough. If you specify a bounds frame larger than *dragBounds*, the object cannot be dragged near the edge of the app area.

*copy*                    *Nil* or non-*nil*, indicating whether to drag a copy or the original items. Specify non-*nil* to drag a copy, *nil* to move the original items.

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

*types*                  An array of symbols of the types to which an item can be converted.

*dragRef*                Any valid NewtonScript object. This value is passed to your other methods, such as your ViewGetDropDataScript.

*label*                  An optional string used when the drop is to the clipboard; it is used as the clipboard label. If this slot is missing and the item

## Miscellaneous

has a 'text' type, the text data is used as the label; otherwise a default label is used.

`minDragDistance`

An integer, the minimum distance in pixels that the user must drag the object before it moves. The default is 4.

return value

This method returns one of the following integers:

`kDragNot = 0` The item was not dragged at all.

`kDragged = 1` The item was dragged, but was rejected by the destination.

`kDragNDropped = 2`

The item was dropped into another view.

## DISCUSSION

The `DragAndDrop` method sends several messages to both the source view (the view from which `DragAndDrop` was sent) and the destination view (the view that will receive the items). If you want other views to be able to accept data, these views must implement all of the destination methods. If you have more than one view that can receive a drop, it is easier if you make one drop-aware proto and use it for your other views.

## SEE ALSO

For further information see “Dragging and Dropping with Views” (page 3-40) in *Newton Programmer’s Guide*.

## COMPATIBILITY

The `dragInfo` argument’s `minDragDistance` slot is ignored in Newton operating systems prior to Newton 2.1.

**DragAndDropLtd**


---

```
view:DragAndDropLtd(unit, dragBounds, limitBounds, copy, dragInfo)
    //platform file function
```

## Miscellaneous

Starts the drag and drop process, returning when the dragged item(s) is dropped into a view or into the clipboard; it is usually called from a `ViewClickScript`.

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

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

*limitBounds* A bounds frame, or a frame with two optional slots: `limitBounds` and `pinBounds`. If you specify a bounds frame, it is the bounds in global coordinates in which the object can be dragged.

Otherwise, you may pass in a frame with the following slots:

`limitBounds` A bounds frame, the symbol `'none`, or `nil`. The bounds frame is a rectangle in global coordinates in which the object can be dragged. The symbol `'none` specifies that there is no limiting rectangle, and the object can be dragged anywhere on the screen. If you pass `nil` (or do not include a `limitBounds` slot) the app area is used as the limiting rectangle.

`pinBounds` A bounds frame, the symbol `'none`, or `nil`. The bounds to use when constraining the object within the limiting rectangle defined in the `limitBounds` slot. If you pass `nil`, the drag object's bounds, *dragBounds*, are used. If you pass `'none`, an empty rectangle (with 0 width and height) is specified at the point where the pen went down to drag the object; that is, the object moves until the tip of the pen reaches the limit bounds.

If the object being dragged is small, compared to the size of the `limitBounds`, you may want to specify a `pinBounds`



## Miscellaneous

smaller than *dragBounds*, otherwise the object may not appear to move far enough. If you specify a bounds frame larger than *dragBounds*, the object cannot be dragged near the edge of the `limitBounds`.

*copy*

`Nil` or `non-nil`, indicating whether to drag a copy or the original items. Specify `non-nil` to drag a copy, or `nil` to move the original items.

*dragInfo*

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

`types` An array of symbols of the types to which an item can be converted.

`dragRef` Any valid `NewtonScript` object. This value is passed to your other methods, such as your `ViewGetDropDataScript`.

`label` An optional string used when the drop is to the clipboard; it is used as the clipboard label. If this slot is missing and the item has a `'text` type, the text data is used as the label; otherwise a default label is used.

`minDragDistance`

An integer, the minimum distance in pixels that the user must drag the object before it moves. The default is 4.

## return value

This method returns one of the following integers:

`kDragNot = 0` The item was not dragged at all.

`kDragged = 1` The item was dragged, but was rejected by the destination.

`kDragNDropped = 2`

The item was dropped into another view.

## Miscellaneous

**IMPORTANT**

This function is not defined in all ROM versions and is supplied by the NTK Platform file. This implementation, as a global function, and not as a view method, requires an additional argument *view*, the view calling this function.

Call it using this syntax:

```
call kDragAndDropLtdFunc with (view, unit, dragBounds,
limitBounds, copy, dragInfo));
```

**DISCUSSION**

The `DragAndDropLtd` method sends several messages to both the source view (the view from which `DragAndDropLtd` was sent) and the destination view (the view that will receive the items). If you want other views to be able to accept data, these views must implement all of the destination methods. If you have more than one view that can receive a drop, it is easier if you make one drop-aware proto and use it for your other views.

**ViewAddDragInfoScript**

---

```
view:ViewAddDragInfoScript(dragInfo)
```

Called to retrieve data if the user hits the global command keys, and your view has a `hilitedData` slot set to `true`.

## Miscellaneous

*dragInfo*

An array of frames. You should add a frame to this array if you have something to cut or copy. Your frame should have the following slots:

<i>types</i>	An array of symbols of the types to which an item can be converted.
<i>view</i>	A view object type if the dragged item is a view with a symbol type of 'paragraph, 'polygon, 'picture, and so on.
<i>dragRef</i>	Any value that will be passed to other methods.
<i>label</i>	An optional string used when the drop is to the clipboard; it is used as the clipboard label. If this slot is missing and the item has a 'text type, the text data is used as the label; otherwise a default label is used.
<i>minDragDistance</i>	An integer, the minimum distance in pixels that the user must drag the object before it moves. The default is 4.

**return value** Return `true` if you have added an element to *dragInfo*; that is, something was cut or copied. Return `nil` otherwise.

**GetClipboard**


---

`GetClipboard()`

Returns the contents of the clipboard.

**return value** A clipboard data frame, or `nil` if the clipboard is empty. Clipboard data frames are described in “Clipboard Data Frame” (page 11-2).

## Miscellaneous

**SetClipboard**

---

`SetClipboard(clipboardData)`

Sets the contents of the clipboard.

*clipboardData*      A clipboard data frame, as described in “Clipboard Data Frame” (page 11-2), or `nil` to clear the clipboard. In addition to the slots in a normal clipboard data frame, you may include an `xy` slot in *clipboardData*:

`xy`                    A frame with two slots `x` and `y`. Each slot contains an integer specifying the offset from the origin, in global coordinates, of the label’s position on the screen. By default, the clipboard label is placed on the left side of the screen, a little below the top.

return value          Undefined; do not rely on it.

**DISCUSSION**

You can use this function to perform a paste, use `GetClipboard` to get the contents, then call `SetClipboard` with `nil` to clear the clipboard.

## Miscellaneous

Stationery

---

**RegStationeryChange**

---

`RegStationeryChange(regSymbol, functionBody)`

Registers a function object to be executed when stationery is installed or removed.

<i>regSymbol</i>	A unique symbol that includes your developer signature.								
<i>functionBody</i>	Function object called when stationery changes. This function body takes four arguments: <table> <tr> <td><i>message</i></td><td>A symbol, currently the symbols 'install and 'remove are used.</td></tr> <tr> <td><i>defType</i></td><td>A symbol, currently the symbols 'dataDef and 'viewDef are sent, for the type of stationery that has been installed or removed.</td></tr> <tr> <td><i>symbol1</i></td><td>The dataDef symbol of the installed or removed stationery.</td></tr> <tr> <td><i>symbol2</i></td><td>If <i>defType</i> is 'dataDef, then this is undefined. If <i>defType</i> is 'viewDef, then this is the viewDef symbol of the installed or removed stationery.</td></tr> </table>	<i>message</i>	A symbol, currently the symbols 'install and 'remove are used.	<i>defType</i>	A symbol, currently the symbols 'dataDef and 'viewDef are sent, for the type of stationery that has been installed or removed.	<i>symbol1</i>	The dataDef symbol of the installed or removed stationery.	<i>symbol2</i>	If <i>defType</i> is 'dataDef, then this is undefined. If <i>defType</i> is 'viewDef, then this is the viewDef symbol of the installed or removed stationery.
<i>message</i>	A symbol, currently the symbols 'install and 'remove are used.								
<i>defType</i>	A symbol, currently the symbols 'dataDef and 'viewDef are sent, for the type of stationery that has been installed or removed.								
<i>symbol1</i>	The dataDef symbol of the installed or removed stationery.								
<i>symbol2</i>	If <i>defType</i> is 'dataDef, then this is undefined. If <i>defType</i> is 'viewDef, then this is the viewDef symbol of the installed or removed stationery.								
return value	Undefined; do not rely on it.								

**SPECIAL CONSIDERATIONS**

The function passed in the *functionBody* argument must not itself call `RegStationeryChange` or `UnregStationeryChange`.

## Miscellaneous

**UnRegStationeryChange**

---

`UnRegStationeryChange(regSymbol)`

Unregisters a function body previously registered using `RegStationeryChange`.

*regSymbol*                      The symbol used in the call to `RegStationeryChange`.

return value                      Undefined; do not rely on it.

**Text Input and Display**

---

**GetAllFonts**

---

`GetAllFonts()`

Returns the installed user fonts.

return value                      An array of font frames.

**DISCUSSION**

The system font, Espy, is not included in the returned list.

**MakeFontMenu**

---

`MakeFontMenu(font, families, sizes, styles)`

Creates an array of font menu items.

*font*                                  Nil or a font specification as either a frame or a packed integer that represents the default font. The returned font menu checks the items that correspond to the selected font family, size and style. Passing `nil` results in no items being checked.

*families*                              Nil, the symbols `'all` or `'none`, or an array of font families. This parameter controls which fonts are returned. If this parameter is `nil` the all user fonts in the system are returned (recommended). If you pass the symbol `'all` every font is returned, including system font. If you pass the symbol `'none` family choices are not

## Miscellaneous

	included in the returned menu. An array specifies the list of font families to return for the menu.
<i>sizes</i>	Nil, the symbol 'none, or an array of numbers. This parameter controls which font sizes are returned. If you pass nil, the font size specified in the <i>font</i> parameter is used. If you pass the symbol 'none font size choices are not included in the menu. An array specifies the list of sizes to return for the menu.
<i>styles</i>	Nil, the symbol 'none, or an integer. This parameter controls which style choices are returned. If you pass nil, the default styles in the system are returned. If you pass the symbol 'none style choices are not included in the menu. An integer specifies a list of style choices to return for the menu as a packed integer; the constants specified in "Font Face Constants" (page 7-3) in <i>Newton Programmer's Reference</i> . To specify more than one font face constant, simply add them together, and pass in the sum.
return value	An array of font menu items, suitable for use wherever a pop up menu array is needed, such as in <code>protoPopupButton</code> , <code>protoPopInPlace</code> , and the <code>PopupMenu</code> view method.

## DISCUSSION

Presently, the *styles* argument ignores the constants `kFaceSuperscript` and `kFaceSubscript`.

## Miscellaneous

Recognition

---

**RecognizeTextInStyles**

---

`RecognizeTextInStyles(textFrame, defaultFontSpec)`

Translates the ink words in a frame containing a combination of raw ink and text.

<i>textFrame</i>	A frame with a <code>text</code> and a <code>styles</code> slot.
<i>defaultFontSpec</i>	A font spec, either an integer or a frame. This the font to use for translated ink. For more information on font specs, see Chapter 8, “Text and Ink Input and Display,” in <i>Newton Programmer’s Guide</i> .
return value	If <i>textFrame</i> contains no ink, <i>textFrame</i> is returned. Otherwise a new frame is returned. This frame has a <code>text</code> and a <code>styles</code> slot, containing translated versions of all the ink words.

**DISCUSSION**

The highest confidence match for each ink word is returned.

**RecognizeInkWord**

---

`RecognizeInkWord(inkWord)`

Returns an array of translation options for an ink word.

<i>inkWord</i>	Ink word data from a rich string or from a style array.
return value	An array of frames for each possible match, or <code>nil</code> if no matches were found. The frames in the array contain a <code>word</code> slot which contains a string.

**DISCUSSION**

The array returned is sorted such that higher confidence matches are earlier in the array; that is the first element is the highest confidence match.



## Miscellaneous

System Services

---

**BatteryStatus**

---

`BatteryStatus(which)`

Returns a status frame for the specified battery.

<i>which</i>	An integer identifying the battery for which to return status information. The value 0 specifies the primary battery pack.
return value	A status frame; see DISCUSSION.

**DISCUSSION**

The status frame returned contains the following slots:

<code>batteryType</code>	Contains one of the following symbols, or an integer: 'alkaline     Battery is standard alkaline. 'nicd        Battery is nickel-cadmium. 'nimh        Battery is nickel-metal hydride. 'lithium     Battery is lithium.
<code>batteryVoltage</code>	A real number giving the current battery voltage.
<code>batteryCapacity</code>	An integer, indicating the percentage of a full charge that the battery contains.
<code>batteryLow</code>	An integer, indicating the percentage of a full charge at which the “low battery” warning should be triggered by the system.
<code>batteryDead</code>	An integer, indicating the percentage of a full charge at which the “dead battery” warning should be triggered and the unit shut down by the system.
<code>batteryCurrent</code>	A real number indicating the current drain, in milliamps. This slot is <code>nil</code> if the battery is charging. This slot is new in 2.1.
<code>acPower</code>	Contains a symbol ('yes or 'no) indicating whether or not the unit has AC power applied. Note that this does

## Miscellaneous

	not imply that the battery is charging. See <code>chargeState</code> to determine that.
<code>acVoltage</code>	A real number giving the AC voltage being supplied by an AC adapter, or <code>nil</code> if AC power is not supplied.
<code>chargeState</code>	Contains one of the following symbols, or an integer: <ul style="list-style-type: none"> <li><code>'notCharging</code> The battery is not charging.</li> <li><code>'discharging</code> The battery is discharging.</li> <li><code>'preliminaryCharging</code> The battery is charging under a pulsed duty schedule that raises its voltage to a level at which it can be efficiently fast-charged. This charging mode is used initially for charging a heavily discharged battery.</li> <li><code>'fastCharging</code> The battery is fast-charging.</li> <li><code>'trickleChargeContinuous</code> or <code>'trickleCharging</code> The battery is fully charged and is being maintained in that state by trickle-charging.</li> </ul>
<code>chargeRate</code>	Reserved for future use.
<code>chargeCurrent</code>	A real number indicating the current, in milliamps, being supplied to charge the battery, if it is charging. If the battery is discharging, this is the current supplied from the battery to the system.
<code>ambientTemp</code>	A real number indicating the ambient temperature in degrees Celsius.
<code>batteryTemp</code>	A real number indicating the battery temperature in degrees Celsius.

Miscellaneous

**Note**

A `nil` value for a slot means the underlying hardware cannot supply this information. The slots containing symbol values (`batteryType`, `chargeState`, `acPower`) may contain integers if the battery driver returned something other than the values listed here. ♦

**COMPATIBILITY**

The return value of this function is changed from its Newton 2.0 OS implementation. The `batteryCurrent` slot is new and the possible symbol values for the `chargeState` slot are different.

**Built-In Applications**

---

**GetPartEntryData**

---

*extrasDrawer*:GetPartEntryData(*entry*) //platform file function

Returns a frame containing information about an Extras Drawer part entry.

<i>entry</i>	An entry obtained from a part cursor; by using <code>GetPartCursor</code> .	
return value	The frame returned has the following slots:	
	<code>icon</code>	A bitmap object, containing the bitmap for the part icon displayed in the Extras Drawer on Newton 1.x and 2.0 operating systems.
	<code>iconPro</code>	A frame containing two pix families, for the highlighted icon and the normal icon to display in the Extras Drawer on Newton 2.1 OS. For more information on grey icons and pix families, see Chapter 6, "Drawing and Graphics 2.1."
	<code>text</code>	A string that is the text shown under the part icon.
	<code>labels</code>	A symbol identifying the Extras Drawer folder in which the part is filed. For a list

## Miscellaneous

	of these see “Extras Drawer Folder Symbols” (page 11-4).
<code>appSymbol</code>	A symbol identifying the application, if the part frame has an app slot.
<code>packageName</code>	A string that is the name of the package that contains the part.

**IMPORTANT**

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

```
call kGetPartEntryDataFunc with (entry);
```

**SetEntryAlarm**


---

```
calendar:SetEntryAlarm(mtgText,mtgStartDate,minutesOrDaysBefore)
```

Sets an alarm for the meeting or event with the given text at the given date and time. If the meeting or event is an instance of a repeating meeting or event, the alarm is set for all instances of the repeating meeting or event.

<i>mtgText</i>	A string or rich string that is the meeting text of the meeting or event for which you want to set the alarm time.
<i>mtgStartDate</i>	An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904.
<i>minutesBefore</i>	A non-negative integer, which specifies how far in advance of the meeting or event the alarm should go off. A value of 0 means the alarm goes off at the time of the meeting. This integer should specify the number of minutes before <i>mtgStartDate</i> that you want the alarm to go off for a meeting, and the number of days before <i>mtgStartDate</i> for an event.  You can specify <code>nil</code> to clear an alarm that is currently set.
return value	Undefined; do not rely on it.

## Miscellaneous

## COMPATIBILITY

The version of this function available on Newton 2.0 OS can only be used for meetings. The `kSetEventAlarmFunc` function exists in the 2.0 platform file to set alarms for events.

**SetUserConfigEnMasse**

---

`SetUserConfigEnMasse(changeSym, changeFrame)`

Sets one or more user configuration variables.

<i>changeSym</i>	A symbol passed to functions registered for notification of user configuration changes. This symbol should be one of the slot names in <i>changeFrame</i> . Some functions registered for user configuration variable changes are passed only this symbol, see <code>RegUserConfigChange</code> .
<i>changeFrame</i>	A frame where the names of the slots are the names of the user configuration variables that you wish to set, and the slot values are the values to which the respective user configuration variables should be set.
return value	Undefined; do not rely on it.

**RegUserConfigChange**

---

`RegUserConfigChange(callBackID, callBackFn)`

Registers a function object to be called each time a user configuration variable changes.

<i>callBackID</i>	A unique symbol identifying the function object to be registered; normally, the value of this parameter is the application symbol, which includes your registered signature, or some variation on it.
<i>callBackFn</i>	A function object called when a user configuration variable changes. It is passed either one or two

## Miscellaneous

parameters. This function can be either of either of the following two forms:

```
func(changeSym, changeFrame) begin .... end
func(changeSym) begin .... end
```

On Newton devices where the `SetUserConfigEnMasse` function is not defined, this callback function is always passed one argument. On Newton devices with `SetUserConfigEnMasse` defined, this function will be called with the proper number of arguments; that is, if you define a one argument function, it will be called with only the *changeSym* argument, but if you define it with two arguments, it is called with both the *changeSym* and the *changeFrame* arguments.

For information on the *changeSym* and the *changeFrame* parameters, see `SetUserConfig` (in *Newton Programmer's Reference*) and `SetUserConfigEnMasse`.

The return value of *callBackFn* function is ignored.

return value      Undefined; do not rely on it.

## DISCUSSION

Note that it is up to the application that changed one of these variables to broadcast the change. This is not something that you need to worry about, since the `SetUserConfig` function will always broadcast the change. Also note that the system may change, and broadcast the change of, certain undocumented user configuration variables; you should ignore these symbols.

## SPECIAL CONSIDERATIONS

The function *callBackFn* must not call the `RegUserConfigChange` or `UnRegUserConfigChange` functions.

## Miscellaneous

**KillStdButtonBar**


---

`KillStdButtonBar(buttonBarParams)`

Closes (or restores) the standard button bar, and reserves screen area for a new one.

*buttonBarParams*      A 4-element array or `nil`. Pass the value `nil` to restore the standard button bar. If you pass an array, each element should be a frame specifying where to save screen space for the replacement button bar in the four different screen orientations. The array elements should be ordered as specified by “Screen Orientation Constants” (page 11-12); that is, *buttonBarParams*[`kPortrait`] should hold information for the portrait screen orientation.

These frames should have the following slots:

*buttonBarPosition*

One of the following symbols: `'top`, `'bottom`, `'right`, `'left`, or `'none`. These symbols specify where to reserve space for the replacement button bar. Specify `'none` if you do not wish to reserve this space.

*buttonBarThickness*

An integer specifying how much space to save for the button bar in pixels. You may not omit this slot, unless *buttonBarPosition* is set to `'none`.

return value      Undefined; do not rely on it.

**DISCUSSION**

If the app area becomes less than 320 pixels high as a result of a call to `KillStdButtonBar`, views without a `ReorientToScreen` method cannot open.

## Miscellaneous

**GetPartEntries**

---

*buttonBar*:GetPartEntries()

Returns the part entries of all icons in the button bar.

return value      A frame with the following two slots, *fixed* and *mobile*. Both of these slots contain an array of part entries. The part entries in *fixed* cannot be moved by dragging. Similarly, the part entries in *mobile* can be moved. The ordering of these arrays is important; it determines the order of the icons in the button bar.

**DISCUSSION**

You must not modify the part entries in any way. To obtain information from a part entry, use the Extras Drawer *GetPartEntryData* method.

To send this method use code such as the following:

```
local bb := GetRoot().Buttons;
if (bb.soft) then bb:GetPartEntries()
```

**Reconfigure**

---

*buttonBar*:Reconfigure(*newSetup*)

Reconfigures the button bar.

*newSetup*      A frame with *fixed* and *mobile* slots. Each slot should contain an array of part entries or application symbols. The icons in *fixed* are not draggable, while the ones in *mobile* are. The ordering of these arrays is important; it determines the order of the icons in the button bar.

return value      Undefined; do not rely on it.

**DISCUSSION**

To send this method use code such as the following:

```
local bb := GetRoot().Buttons;
if (bb.soft) then bb:Reconfigure()
```



## Miscellaneous

**IconCapacity**

---

*buttonBar*:IconCapacity()

Returns the number of icons the button bar can currently hold.

return value            An integer, the maximum number of icons.

**DISCUSSION**

To send this method use code such as the following:

```
local bb := GetRoot().Buttons;
if (bb.soft) then bb:IconCapacity()
```

**Transports**

---

**DeleteItem**

---

*transport*:DeleteItem(*item*)

Deletes an item from the In/Out Box.

*item*                    The item to delete. This is an item frame from the In Box.

return value            Undefined; do not rely on it.

**DeleteRemoteItems**

---

*transport*:DeleteRemoteItems()

Causes the transport to delete from the In/Out Box all remote items that have not been fully downloaded.

return value            Undefined; do not rely on it.

**DISCUSSION**

Typically, you use the `DeleteRemoteItems` method after the transport disconnects, to remove from the In/Out Box all remote items that the user chose not to retrieve fully. This method removes all items whose `remote slot` is set to `true`.

## Miscellaneous

## COMPATIBILITY

This 2.1 method replaces the 2.0 method *ownerApp:RemoveTempItems*. If you are writing an application for 2.1 only, then you should use this method instead of *ownerApp:RemoveTempItems*.

**RefreshOwner**

---

*transport:RefreshOwner()*

Causes the transport owner (typically the In/Out Box) to refresh the view of the in box.

return value            Undefined; do not rely on it.

## DISCUSSION

You use *RefreshOwner* to refresh the in box view after remote items are fully retrieved and after remote items that are not fully retrieved are deleted.

## COMPATIBILITY

This 2.1 transport method replaces the 2.0 method *ownerApp:Refresh*. If you are writing an application for 2.1 only, then you should use this method instead of *ownerApp:Refresh*.

**Utility Functions**

---

**ROM\_GetSerialNumber**

---

*ROM\_GetSerialNumber()*

Returns the unique hardware serial number of a Newton device.

return value            An 8 byte binary object containing the Newton device's serial number.

## DISCUSSION

This function is not defined in neither Newton 1.x nor 2.0 OS. You should wrap the call to this function in *try...onException* block, as in the following example:

## Miscellaneous

```

local sn;

try
    sn := call ROM_GetSerialNumber with ()
onException |evt.ex| do
    nil;

if sn then
    // ...

```

The serial number returned in ROM is not the same as the serial number stamped on the Newton device. The ROM serial number is intended for use by programmers.

The `StrHexDump` and `ExtractByte` functions are designed to read binary objects.

**ImportDisabled**


---

```
partFrame: ImportDisabled(unitName, majorVersion, minorVersion)
```

Called after an imported unit has been deactivated to perform housekeeping.

<i>unitName</i>	A symbol, the name of the unit.
<i>majorVersion</i>	An integer, the major version number of the unit.
<i>minorVersion</i>	An integer, the minor version number of the unit.
return value	Either the symbol <code>'ThrillMeChillMeFulfillMe'</code> or anything else.

**DISCUSSION**

The part should deal with the situation as gracefully as possible. For example, you could use alternative data, or put up a message slip with the `Notify` method and/or close your application.

If you return the symbol `'ThrillMeChillMeFulfillMe'`, the system attempts to re-resolve the imports. For example, if version 2 of unit `foo` is disabled and your package's `ImportDisabled` script returns `'ThrillMeChillMeFulfillMe'`, the system looks for other versions of the objects in the unit `foo`.

## Miscellaneous

## COMPATIBILITY

Newton 2.0 OS sends this message, but ignores the return value.

**LegalOrientations**

---

`LegalOrientations()`

Returns the legal values for screen orientations on the Newton device.

return value      An array of integers, possible values are listed in “Screen Orientation Constants” (page 11-12).

## COMPATIBILITY

This function is supported in Newton OS 2.0. On the MessagePad 120 and 130 units, the only possible return values are `kPortrait` (0) and `kLandscapeFlip` (3).

**GetOrientation**

---

`GetOrientation()`

Returns the current orientation of the unit.

return value      An integer, possible values are listed in “Screen Orientation Constants” (page 11-12).

## COMPATIBILITY

This function is supported in Newton OS 2.0. On the MessagePad 120 and 130 units, the only possible return values are 0 and 3.

**SetScreenOrientation**

---

`SetScreenOrientation(orientation)`

Sets the screen orientation.

*orientation*      An integer specifying the new orientation, possible values are listed in “Screen Orientation Constants” (page 11-12).

return value      Nil if the screen orientation was not changed, otherwise a non-nil value is returned.

## Miscellaneous

## DISCUSSION

This function requests the system to rotate the screen to the desired orientation. The user may be prompted if particular applications do not support the new orientation.

**GetAppParams**

---

`GetAppParams()`

Returns a frame containing information about the screen size and other system configuration items.

return value	A frame with the following slots:	
	<code>appAreaTop</code>	The y coordinate of the top-left corner of the application area. Children of the root view are always opened relative to the application area. This value is always 0.
	<code>appAreaLeft</code>	The x coordinate of the top-left corner of the application area. This value is always 0.
	<code>appAreaWidth</code>	The width of the screen in pixels.
	<code>appAreaHeight</code>	The height of the screen in pixels.
	<code>buttonBarPosition</code>	A symbol, either 'top', 'left', 'bottom', 'right, or 'none indicating where the button bar is, if there is one. This is useful

## Miscellaneous

if you want to locate your application flush against the button bar.

`appAreaGlobalTop`

The y coordinate of the top of the application area in global coordinates.

`appAreaGlobalLeft`

The x coordinate of the left of the application area in global coordinates.

`buttonBarBounds`

If there is a soft button bar this slot contains its view bounds.

## COMPATIBILITY

Versions of this function previous to Newton 2.1 OS return a frame without the `appAreaGlobalTop`, `appAreaGlobalLeft`, and `buttonBarBounds` slots.

**Gestalt**


---

`Gestalt(selector)`

Returns information about the Newton system; the type of information returned depends on the value of the *selector* parameter.

*selector*

A constant that specifies the type of information that is returned on the system. The following values are currently allowed: `kGestalt_SystemInfo`, `kGestalt_Backlight`, `kGestaltArg_HasSoftContrast`, and `kGestaltArg_VolumeInfo`.

return value

Depends on *selector*, see DISCUSSION.

## DISCUSSION

The return value of this function depends on the value of *selector*, as follows:

- If *selector* is `kGestalt_SystemInfo`, `Gestalt` returns a frame with the following slots:

## Miscellaneous

**Slot Descriptions**

manufacturer	An integer indicating the manufacturer of the Newton Device.
machineType	An integer indicating the hardware type this ROM was built for.
ROMStage	A decimal integer indicating the language (English, German, French) and the stage of the ROM (alpha, beta, final).
ROMVersion	A packed integer indicating the major and minor ROM version numbers. You can use the following function to convert this number into an array, containing an integer for the ROM major and minor version numbers:

```
func (ROMVersionInteger)
begin
    local minor := BAND(ROMVersionInteger, 0xFFFF);
    local major := BAND(ROMVersionInteger>>16, 0xFFFF);
    [ Floor(StringToNumber(BAND(major>>12, 0xF)
                                & BAND(major>>8, 0xF)
                                & BAND(major>>4, 0xF)
                                & BAND(major, 0xF))),
      Floor(StringToNumber(BAND(minor>>12, 0xF)
                                & BAND(minor>>8, 0xF)
                                & BAND(minor>>4, 0xF)
                                & BAND(minor, 0xF)))]
end
```

Here is another example of code to test if your Newton is running 2.x. The following expression evaluates to a non-nil value if the major version is 2:

```
BAND((Gestalt(kGestalt_SystemInfo).ROMVersion)>>16, 0xFFFF) = 0x0002
```

**IMPORTANT**

Do not assume that if the Newton is running version 2.0 or later that a particular feature exists. You still need to test the Newton to make sure the feature exists. ♦

## Miscellaneous

**Note**

The `machineType`, `ROMStage` and `ROMVersion` slots provide internal configuration information and should not be relied on. ♦

<code>screenWidth</code>	<p>An integer representing the width of the screen in pixels. The width takes into account the current screen orientation.</p> <p>For example, on the MessagePad 120, because the screen width is 240 and the screen height is 320, in portrait orientation <code>Gestalt</code> returns a width of 240. If the screen is rotated, <code>Gestalt</code> returns a width of 320.</p>
<code>screenHeight</code>	<p>An integer representing the height of the screen in pixels.</p>
<code>screenResolutionX</code>	<p>An integer representing the number of horizontal pixels per inch. For screens with square pixels, <code>screenResolutionX</code> equals <code>screenResolutionY</code>. On the MessagePad 120, for example, both <code>screenResolutionX</code> and <code>screenResolutionY</code> equal 85.</p>
<code>screenResolutionY</code>	<p>An integer representing the number of vertical pixels per inch.</p>
<code>screenDepth</code>	<p>The bit depth of the LCD screen. For the MessagePad 120, the LCD supports a monochrome screen depth of 1.</p>



## Miscellaneous

The eMate 300 and MessagePad 200 have 4 bit depth LCD screens.

`patchVersion`

Returns 0 on an unpatched Newton and nonzero on a patched Newton.

`ROMVersionString`

The user-visible string that identifies the version of the installed ROM and the installed patch, if any.

The first part of the string is a “functionality level” indicating the OS version, such as 1.3, 2.0 or 2.1.

The second part of the string is a six-digit number in parentheses that is an encoded representation of ROM and system update information.

`cpuType`

A symbol specifying the type of CPU, possible values are 'strongArm, 'arm710a, and 'arm610a.

`cpuSpeed`

A real indicating the speed of the CPU in megahertz.

- If *selector* is `kGestalt_Backlight`, `Gestalt` returns either `nil`, indicating the unit does not have backlight hardware, or a one element array. If an array is returned the unique element contains either `nil` or a non-`nil` value, indicating whether backlight hardware is present.

The following code correctly tests if a unit has a backlight:

```
local result := Gestalt(kGestalt_Backlight);
if result and result[0] then
    // unit has backlighting
else
    // unit does not have backlighting
```

- If *selector* is `kGestaltArg_HasSoftContrast`, `Gestalt` returns either `nil`, or a 3 element array of the following form:  
`[hasSoftContrast, minContrast, maxContrast]`

## Miscellaneous

**Array Element Descriptions**

<i>hasSoftContrast</i>	True or nil depending on whether there is a soft contrast control.
<i>minContrast</i>	Integer for the minimum contrast.
<i>maxContrast</i>	Integer for the maximum contrast.

You can use the values returned by this selector to set the `LCDCContrast` user configuration variable.

- If *selector* is `kGestaltArg_VolumeInfo`, `Gestalt` returns either nil, or a 7 element array of the following form:  
`[hasInput, hasOutput, hardwareVolControl, headphoneJack, minAudibleDB, numDVLevels, devicesBitfield]`

**Array Element Descriptions**

<i>hasInput</i>	True or nil depending on whether the device can support sound input.
<i>hasOutput</i>	True or nil depending on whether the device can support sound output.
<i>hardwareVolControl</i>	True or nil depending on whether the device has a hardware volume control.
<i>headphoneJack</i>	True or nil depending on whether the device has a built-in headphone jack.
<i>minAudibleDB</i>	An integer, the minimal decibel level for output. The MessagePad 2000 is set to -31.9760.
<i>numDVLevels</i>	An integer, the number of levels between <i>minAudibleDB</i> and 0. The dB increment per level is <i>minAudibleDB</i> / <i>numDVLevels</i> . The MessagePad 2000 is set to 14.
<i>devicesBitfield</i>	A packed integer with information about the built-in sound devices. This integer contains the summation of the applicable device constants. Device constants are described in “Device Constants” (page 7-26) in

## Miscellaneous

Chapter 7, “Sound.” The two important ones are `kInternalSpeaker` and `kInternalMic`.

The following function returns `nil` / `non-nil` if the current device has an internal microphone (use `kInternalSpeaker` to check for an internal speaker):

```
HasMic := func()
begin
    local volInfo := Gestalt(kGestaltArg_VolumeInfo) ;

    return volInfo AND
        (BAND(volInfo[6], kInternalMic) <> 0);
end
```

## COMPATIBILITY

The `kGestalt_Backlight` and `kGestaltArg_VolumeInfo` selectors are not supported on 2.0 devices.

**TimeFrameStr**


---

`TimeFrameStr(timeFrame, timeStrSpec)`

Returns a string representation of the time *timeFrame*, in the specified format.

<i>timeFrame</i>	A date frame as returned by the <code>Date</code> function.
<i>timeStrSpec</i>	A format specification returned by the <code>GetStringSpec</code> function, or one of the format specifications found in <code>ROM_dateTimeStrSpecs</code> .
return value	A string representation <i>timeFrame</i> .

## DISCUSSION

This function is similar to the `TimeStr` function. The `TimeStr` function is passed in the time as an integer, the number of minutes since 1/1/04; this is what the `Time` function returns. Thus, when the a format spec is provided that requires seconds, `TimeStr` returns a string with 00 as the seconds value. `TimeFrameStr`, on the other hand, since it is passed the time as a date frame, can include seconds information.

## Miscellaneous

**LocalTime**

---

`LocalTime(time, where)`

Returns the local time in a distant city.

<i>time</i>	An integer, the time in minutes since 1/1/1904 in the local, Newton device's, time zone. This is the value returned by the <code>Time</code> function.
-------------	--

<i>where</i>	A city entry, as returned by <code>GetCityEntry</code> .
--------------	--

return value	An integer, the time in minutes since 1/1/1904 in the <i>where</i> city, adjusted as necessary for time zone and daylight savings.
--------------	--

**DISCUSSION**

To find out the time in Tokyo:

`Date(LocalTime(Time(), GetCityEntry("Tokyo")[0]))`

This function call returns the following frame

```
{year: 1997, month: 2, Date: 22, dayOfWeek: 6, hour: 8, minute: 1,
second: 0, daysInMonth: 28}
```

**COMPATIBILITY**

This function exists in Newton 2.0 OS, but was not previously documented.

**DSTOffset**

---

`DSTOffset(time, where)`

Returns the Daylight Savings Time component of a given city at a given date.

<i>time</i>	An integer, the time in minutes since 1/1/1904 in the <i>where</i> city.
-------------	--

<i>where</i>	A city entry, as returned by <code>GetCityEntry</code> .
--------------	--

return value	An integer, the number of minutes that daylight savings adjusted that time in that city.
--------------	--

Miscellaneous

**COMPATIBILITY**

This function exists in Newton 2.0 OS, but was not previously documented.

**SymbolName**

---

SymbolName(*symbol*)

Returns a string representation of a symbol.

*symbol*                      A symbol.

return value                A string.

**COMPATIBILITY**

This function exist in Newton 2.0 OS, but was not previously documented.

## Summary

---

### Data Structures

---

### Views

---

### View Slot

---

hilitedData

### Clipboard Data Frame

---

```
aClipboardDataFrame := {
  label : string, //string displayed by clipboard
  types : array, //array of types arrays
  data : array, //array of data arrays
  bounds : frame, //where data came from
  ...}
```

### Built-In Applications

---

### Extras Drawer Folder Symbols

---

```
nil
'_extensions
'_help
'_setup
'_soups
'_ButtonBar
```

### Names Worksite Soup Entry

---

cityAlias

## Miscellaneous

**Newton Works Word Processor Soup Format**

---

```

aWorksWordProcessorSoupEntry :=
{
class: 'paper,
version: integer,
title: string ,
timeStamp: integer,
realModTime: integer,
saveData: frame,
hiliteRange: frame,
margins: frame,
...}

```

**User Configuration Variables**

---

```

LCDContrast
alarmVolumeDb
soundVolumeDb
buttonBarPositions
buttonBarControlsPositions
bellyButtonPositions
buttonBarIconSpacingH
buttonBarIconSpacingV
extrasIconSpacingH
extrasIconSpacingV
extraFont

```

**Protos**

---

**protoPasswordSlip**

---

```

aPassWordSlip := {
 proto : protoPasswordSlip,
CurrentPassword : func() ..., //gets curr password
SetPassword : func(newPassword), //sets curr password
MatchPassword: func(newPassword, currentPassword)..., //do these match
MatchedPassword: func() ..., //called if there was a match
verifyPassword: symbolORtrueORnil, //should password be verified
...}

```

## Miscellaneous

**protoBlindEntryLine**

---

```

aBlindEntryLine := {
  _proto: protoBlindEntryLine,
  dummyChar : character, //char to echo
  UpdateText : func (newText), //updates text
  realText : string, //the real text
  label : string, //entry line label
  ...}

```

**Constants**

---

**Views**

---

**Screen Orientation Constants**

---

kPortrait	0
kLandscape	1
kPortraitFlip	2
kLandscapeFlip	3

**Built-In Communications Tools**

---

**Serial Communication Tool Sound Option**

---

kCMOPCMCIAModemSound	"msnd"
----------------------	--------

**Functions and Methods**

---

**Views**

---

```

view:DragAndDrop(unit, bounds, limitBounds, copy, dragInfo)
    //starts the drag and drop process (2.0 also)
view:DragAndDropLtd(unit, dragBounds, limitBounds, copy, dragInfo)
    //starts the drag and drop process in limited area(platform file)

```



## Miscellaneous

```
view:ViewAddDragInfoScript(dragInfo) //called if hilitedData is true
GetClipboard() //returns the contents of the clipboard
SetClipboard(clipboardData) //sets the contents of the clipboard
```

## Stationery

---

```
RegStationeryChange(regSymbol, functionBody)
                                     //regs callback for stationey change
UnRegStationeryChange(regSymbol) //unregs a stationery change callback
```

## Text Input and Display

---

```
GetAllFonts() // returns the installed user fonts
MakeFontMenu(font, families, sizes, styles) //makes a font menu
```

## Recognition

---

```
RecognizeTextInStyles(textFrame, defaultFontSpec)
                     //recognizes ink in a frame
RecognizeInkWord(inkWord) //recognizes an ink word
```

## System Services

---

```
BatteryStatus(which) //returns info about a battery (2.0 also)
```

## Built-In Applications

---

```
extrasDrawer:GetPartEntryData(entry)
               // gets info about an part entry (platform file - 2.0 also)
calendar:SetEntryAlarm(mtgText, mtgStartDate, minutesOrDaysBefore)
               // sets an alarm for a meeting or event (2.0 also)
SetUserConfigEnMasse(changeSym, changeFrame)
               // sets multiple user configuratation variables
RegUserConfigChange(callBackID, callBackFn)
               //registers a callback for changes in a user configuration vars.
KillStdButtonBar(buttonBarParams)
               // closes (or restores) the button bar
buttonBar:GetPartEntries() //returns part entries for parts in b. bar
buttonBar:ReConfigure(newSetup) //reconfigure the button bar
buttonBar:IconCapacity() // gets number of icons that fit in bb
```

## Miscellaneous

## Transports

---

```
transport:DeleteItem(item) //deletes item from In/Out box
transport:DeleteRemoteItems() //deletes remote items
transport:RefreshOwner() //refreshes the transport owner
```

## Utility Functions

---

```
ROM_GetSerialNumber() //gets a units unique serial number
partFrame:ImportDisabled(unitName, majorVersion, minorVersion)
    //called to clean up when unit is disabled (2.0 also)
LegalOrientations() //gts legal values for screen orientation (2.0 also)
GetOrientation() //gets current screen orientation (2.0 also)
SetScreenOrientation(orientation) //sets screen orientation
GetAppParams() //gets system configuration (2.0 also)
Gestalt(selector) //gets info about the system (2.0 also)
TimeFrameStr(timeFrame, timeStrSpec) //returns string with time
LocalTime(time, where) //gets local time in a distant city (2.0 also)
DSTOffset(time, where) //gets DST component of a city's time(2.0 also)
SymbolName(symbol) //returns a string version of a symbol (2.0 also)
```

# Newton Toolkit Enhancements

---

Newton Toolkit version 1.6.4 provides support for pix families and gray icons. Two editors have been significantly changed for this purpose, the picture editor and the application icon editor. These new build-time functions have also been added to NTK: `GetSoundFrame`, `MakeBinaryFromHex`, `MakeDitheredPattern`, `MakeExtrasIcons`, `MakePixFamily`, and `UnpackRGB`.

## Editors

---

The picture and application icon editors have been changed to support pix families.

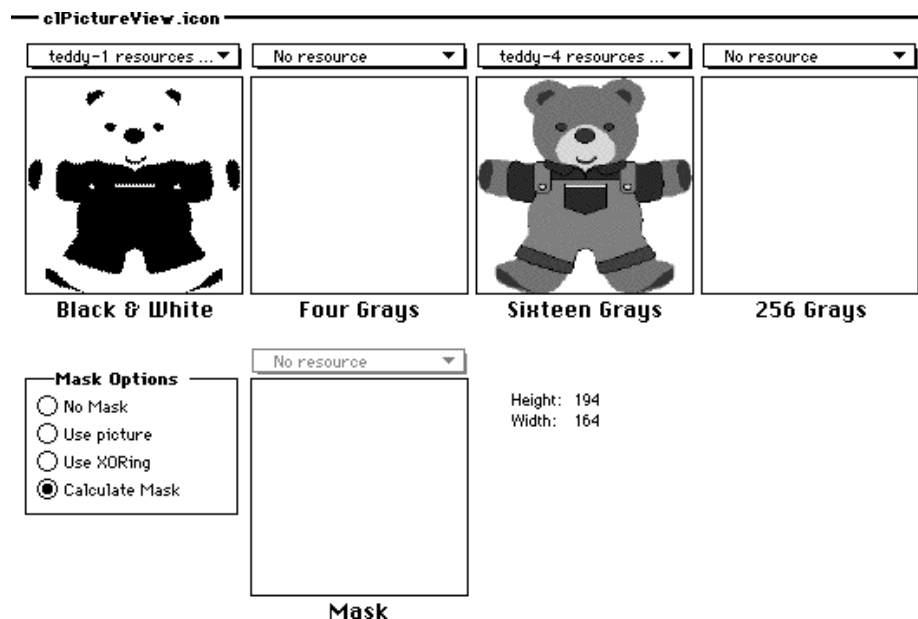
### Picture Slot Editor

---

A picture slot editor, shown in Figure A-1, is used to create a pix family from a number of PICT resources at different bit depths to use for the `icon` slot of a `clPictureView`. The editor allows you to include different PICTs to display on black and white, 4 grays, 16 grays, and 256 grays screens. You may specify any number of these pictures; the system software determines the appropriate image to display at run time.

#### Note

The picture does not have to be the same bit-depth as the picture window it is placed in. For example, an 8-bit picture could be specified for the “Black and White” window. The picture would be properly displayed on a black and white screen. However, this would waste memory, and the picture would be drawn slower. You should reduce the bit depth of each PICT to the appropriate setting in NTK with a graphics utility on the desktop machine. ♦

**Figure A-1** NTK's picture slot editor

NTK displays the width and height in pixels. All included PICTs must be the same size. Each of the images is selected from a popup menu over the image. This popup menu contains all the PICT resources from resource files included in the current project.

A number of options are provided for a picture's mask:

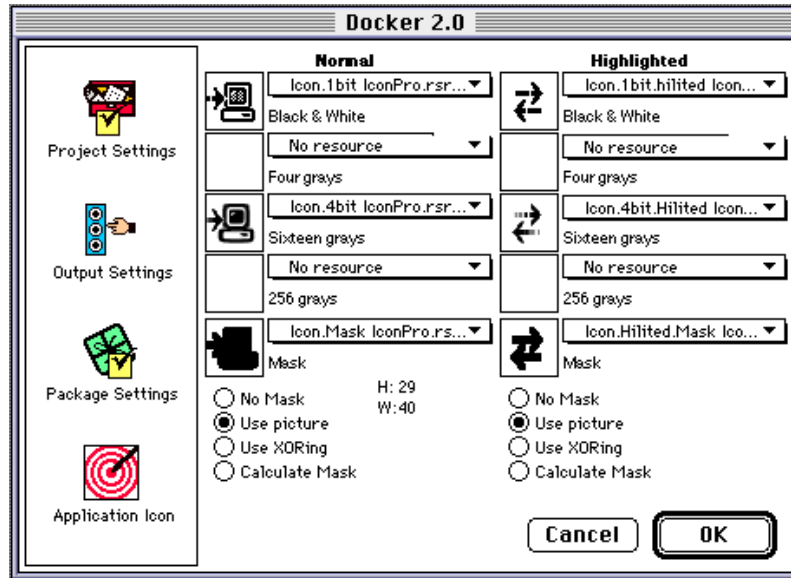
No Mask	The bitmap does not contain a mask.
Use Picture	The mask is a PICT resource that is picked from the popup menu over the Mask window, just like any other image is selected.
Use XORing	A mask is generated such that when this mask is Xor'ed with the 1 bit image, the image selected in the mask field is displayed.
Calculate Mask	A mask is generated automatically from the black and white image. If no black and white image has been selected, one is created by NTK on the fly from one of the available images to generate the mask.

## Application Icon Editor

---

The Application Icon pane of the Project Settings dialog allows you to select gray icons for your form part. There is a field for each of four screen resolutions, 1, 2, 4, and 8 bits, as well as a field for highlighted versions of the icon, and two fields for the normal and highlighted masks.

**Figure A-2** NTK's Application Icon pane of the Project Setting dialog



A PICT is selected for each of these icons with the popup menu to the right of each of these fields. There are four choices available for the icon's mask:

- |                |  |
|----------------|--|
| No Mask        | A mask is not used.  |
| Use Picture    | A black and white PICT is selected.  |
| Use XORing     | A mask is generated such that when this mask is Xor'ed with the 1 bit icon, the image selected in the mask field is displayed. The Extras Drawer in Newton 1.x and 2.0 OS Xor's an icon with its mask when the icon is selected. |
| Calculate Mask | A mask is generated automatically from the black and white image. If no black and white image has been selected, one is created by NTK on the fly from one of the available images, to generate the mask.                        |

## Functions

---

The following build-time functions are new in NTK version 1.6.4. Note that these functions are not available at run time.

### GetSoundFrame

---

GetSoundFrame(*nameString*)

Retrieves a sound from an open Macintosh sound resource.

<i>nameString</i>	A string specifying the name of the sound resource to be retrieved.
return value	A sound frame containing a sound in whatever format is specified in the source sound resource. For details on the sound frame, see “Sound Frame” (page 7-28).

### DISCUSSION

This function is similar to the older `GetSound` and `GetSound11` build-time functions. However, those functions require the sound to conform to a particular sampling rate, while `GetSoundFrame` is capable of loading sounds of any type. The Newton device, of course, will only play certain kinds of sounds.

### MakeBinaryFromHex

---

MakeBinaryFromHex(*hexString*, *classSym*)

Returns a binary object of the specified class from the data in *hexString*.

<i>hexString</i>	A string consisting of an even number of hexdigits. Each set of two hexdigits makes for one byte of the binary object. This string is similar to the string returned by <code>StrHexDump</code> .
<i>classSym</i>	A symbol for the binary object’s class.
return value	A binary object of class <i>classSym</i> .

SEE ALSO

For example calls to this function, see “Black and White Patterns” (page 6-11), “Gray Patterns” (page 6-12), and “Dithered Patterns” (page 6-12).

### MakeDitheredPattern

---

MakeDitheredPattern(*bwPattern*, *foregroundColor*, *backgroundColor*)

Creates a dithered pattern.

<i>bwPattern</i>	A one-bit pattern. A pattern is a binary object containing an 8x8 bitmap of class 'pattern. The constants <code>vfWhite</code> , <code>vfLtGray</code> , <code>vfGray</code> , <code>vfDkGray</code> , and <code>vfBlack</code> specify patterns in the Newton OS ROM.
<i>foregroundColor</i>	A <code>kRGB_GrayXX</code> constant or a packed RGB integer returned by <code>PackRGB</code> .
<i>backgroundColor</i>	A <code>kRGB_GrayXX</code> constant or a packed RGB integer returned by <code>PackRGB</code> .
return value	A dithered pattern frame as defined in “Dithered Pattern” (page 6-28).

DISCUSSION

Using this function, as opposed to creating your own frame ensures that the frame shares a frame map with other dithered pattern frames.

SEE ALSO

For an example use of this function, see “Dithered Patterns” (page 6-12).



## MakeExtrasIcons

`MakeExtrasIcons(iconRsrcSpecs, unhilitedMaskRsrcSpec, hilitedMaskRsrcSpec)`

Creates a frame with an `iconPro`, and optionally, an `icon` slot; these slots can be copied to a part frame.

<i>iconRsrcSpecs</i>	An array of frames with the following format:
<i>unhilitedRsrcSpec</i>	String for the name of a PICT resource to use as the normal icon.
<i>hilitedRsrcSpec</i>	Optional. String for the name of a PICT resource for highlighted icon.
<i>bitDepth</i>	Optional. Integer indicating resource's bit depth. The allowable values are 1, 2, 4, and 8.  If you do not specify a bit depth for a particular PICT, the bit depth is determined automatically from the PICT resource. If you want the icon to be included in your project at a particular bit depth, you should specify it explicitly.

### Note

All the PICTs provided in this array must be of the same size. ♦

<i>unhilitedMaskRsrcSpec</i>	String for the name of a black and white PICT to be the mask for normal icon.
<i>hilitedMaskRsrcSpec</i>	String for the name of the black and white PICT to be a mask for the highlighted icon, or <code>nil</code> if no highlighted icon is provided.
return value	A frame with an <code>iconPro</code> slot, and if 1-bit information is provided in <i>iconRsrcSpecs</i> , an <code>icon</code> slot. These slots can be copied to a part frame.

## DISCUSSION

If the *iconRsrcSpecs* array contains more than one icon, the system determines the appropriate one for the current hardware.

The resource names are for named PICT resources within any resource file included in the current project. If more than one PICT is used, then all the PICTs must have the same size bounds, or this function will throw. This includes all the PICTs referred to in the *iconRsrcSpecs*, *unhilitedMaskRsrcSpec*, and *hilitedMaskRsrcSpec* parameters.

## SEE ALSO

The Project Settings dialog provides an editor to use for an application's part's icon; see "Application Icon Editor" (page A-3). You must use `MakeExtrasIcons` to create icons for other types of parts.

For an example of using this function, see Listing 6-1 (page 6-16).

## MakePixFamily

---

`MakePixFamily (bwRsrcSpec, maskRsrcSpec, colorSpecs)`

Creates pix family from a set of PICTs.

<i>bwRsrcSpec</i>	String for the name of a black and white PICT resource to use in 2.0 and 1.x systems, or <code>nil</code> if backward compatibility is not desired.				
<i>maskRsrcSpec</i>	String for the name of a black and white PICT resource to use as a mask or <code>nil</code> if there is no mask.				
<i>colorSpecs</i>	A color spec or an array of color specs. A color spec is either a string for the PICT resource name or a frame with the following slots: <table> <tr> <td><i>rsrcSpec</i></td><td>Required. A string for the PICT resource name.</td></tr> <tr> <td><i>bitDepth</i></td><td>Optional. An integer for the bit depth of the PICT. The following values are allowed: 1, 2, 4, and 8.  If you do not specify a bit depth for a particular PICT, the bit depth is determined automatically from the PICT</td></tr> </table>	<i>rsrcSpec</i>	Required. A string for the PICT resource name.	<i>bitDepth</i>	Optional. An integer for the bit depth of the PICT. The following values are allowed: 1, 2, 4, and 8.  If you do not specify a bit depth for a particular PICT, the bit depth is determined automatically from the PICT
<i>rsrcSpec</i>	Required. A string for the PICT resource name.				
<i>bitDepth</i>	Optional. An integer for the bit depth of the PICT. The following values are allowed: 1, 2, 4, and 8.  If you do not specify a bit depth for a particular PICT, the bit depth is determined automatically from the PICT				

resource. If you want the image to be included in your project at a particular bit depth, you should specify it explicitly.

**return value** A `pix` family frame, it can be passed to `CopyBits`, used in the icon slot of a `clPictureView`, or passed to `MakeShape` to create a bitmap shape.

## DISCUSSION

If `colorSpecs` contains an array, the system displays the most appropriate image for the current hardware.

The resource names are for named PICT resources within any resource file included in the current project. If more than one PICT is used, then all the PICTs must have the same size bounds, or this function will throw. This includes all the PICTs referred to in the `bwRsrcSpec`, `maskRsrcSpec`, and `colorSpecs` parameters.

## SEE ALSO

NTK's picture slot editor provides a simple way to create a `pix` family. See "Picture Slot Editor" (page A-1).

## UnPackRGB

---

`UnPackRGB(packedRGB)`

Returns a frame with information about the red, green, and blue components of a packed RGB integer.

***packedRGB*** A packed RGB integer, as returned by the function `PackRGB`.

**return value** A frame with `red`, `green`, and `blue` slots. Each slot contains an integer in the range [0, 65535] for that color component's value.

## SPECIAL CONSIDERATIONS

`UnPack(PackRGB(r,g,b))` returns a frame {`red`: `redInt`, `green`: `greenInt`, `blue`: `blueInt`}. Note that `r` might not equal `redInt`, `g` might not equal

`greenInt`, and `b` might not equal `blueInt`. It is only guaranteed that these values are similar, not identical.