EXHIBIT 64 PART 1

Newton Programmer's Guide

For Newton 2.0

Addison-Wesley Publishing Company

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Harlow, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Apple Computer, Inc.
© 1996 Apple Computer, Inc.
All rights reserved.

No part of this publication 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 to make a backup copy of any documentation provided on CD-ROM. Printed in the United States of America.

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.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for printing or clerical errors.

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

Apple, the Apple logo, APDA, AppleLink, AppleTalk, 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.

Apple Press, the Apple Press Signature, eWorld, 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

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.

Table of Contents

Figures and Tables xxxiii

Preface **About This Book** xliii

Who Should Read This Book xliii
Related Books xliii
Newton Programmer's Reference CD-ROM xliv
Sample Code xlv
Conventions Used in This Book xlv
  Special Fonts xlv
  Tap Versus Click xlvi
  Frame Code xlvi
Developer Products and Support xlvii
Undocumented System Software Objects xlviii

Chapter 1 **Overview** 1-1

Operating System 1-1
  Memory 1-3
  Packages 1-4
System Services 1-4
  Object Storage System 1-5
  View System 1-6
  Text Input and Recognition 1-7
  Stationery 1-8
  Intelligent Assistant 1-8
  Imaging and Printing 1-9
  Sound 1-9
  Book Reader 1-10
  Find 1-10
  Filing 1-11

ARENDI-DEFS00003651

Communications Services 1-11
NewtonScript Application Communications 1-13
Routing Through the In/Out Box 1-13
Endpoint Interface 1-14
Low-Level Communications 1-14
Transport Interface 1-14
Communication Tool Interface 1-15
Application Components 1-15
Using System Software 1-17
The NewtonScript Language 1-18
What's New in Newton 2.0 1-18
NewtApp 1-18
Stationery 1-19
Views 1-19
Protos 1-20
Data Storage 1-20
Text Input 1-20
Graphics and Drawing 1-21
System Services 1-21
Recognition 1-22
Sound 1-22
Built-in Applications 1-22
Routing and Transports 1-23
Endpoint Communication 1-23
Utilities 1-24
Books 1-24

Chapter 2 Getting Started 2-1

Choosing an Application Structure 2-1
Minimal Structure 2-1
NewtApp Framework 2-2
Digital Books 2-3
Other Kinds of Software 2-4
Package Loading, Activation, and Deactivation 2-4
Loading 2-5
Activation 2-5
Deactivation 2-6

Effects of System Resets on Application Data 2-7
Flow of Control 2-8
Using Memory 2-8
Localization 2-9
Developer Signature Guidelines 2-9
Signature 2-9
How to Register 2-10
Application Name 2-10
Application Symbol 2-11
Package Name 2-11
Summary 2-12
View Classes and Protos 2-12
Functions 2-12

Chapter 3 Views 3-1

About Views 3-1
Templates 3-2
Views 3-4
Coordinate System 3-6
Defining View Characteristics 3-8
Class 3-9
Behavior 3-9
Location, Size, and Alignment 3-10
Appearance 3-20
Opening and Closing Animation Effects 3-23
Other Characteristics 3-24
Inheritance Links 3-24
Application-Defined Methods 3-26
View Instantiation 3-26
Declaring a View 3-27
Creating a View 3-28
Closing a View 3-29
View Compatibility 3-30
New Drag and Drop API 3-30
New Functions and Methods 3-30
New Messages 3-30
New Alignment Flags 3-31

Changes to Existing Functions and Methods 3-31
New Warning Messages 3-32
Obsolete Functions and Methods 3-32
Using Views 3-32
Getting References to Views 3-32
Displaying, Hiding, and Redrawing Views 3-33
Dynamically Adding Views 3-33
Showing a Hidden View 3-34
Adding to the stepChildren Array 3-34
Using the AddStepView Function 3-35
Using the BuildContext Function 3-36
Creating Templates 3-36
Making a Picker View 3-37
Changing the Values in viewFormat 3-37
Determining Which View Item Is Selected 3-37
Complex View Effects 3-38
Making Modal Views 3-38
Finding the Bounds of Views 3-39
Animating Views 3-40
Dragging a View 3-40
Dragging and Dropping with Views 3-40
Scrolling View Contents 3-41
Redirecting Scrolling Messages 3-42
Working With View Highlighting 3-42
Creating View Dependencies 3-43
View Synchronization 3-43
Laying Out Multiple Child Views 3-43
Optimizing View Performance 3-44
Using Drawing Functions 3-44
View Fill 3-44
Redrawing Views 3-44
Memory Usage 3-45
Scrolling 3-46
Summary of Views 3-47
Constants 3-47
Functions and Methods 3-51

Chapter 4 NewtApp Applications 4-1

About the NewtApp Framework 4-1
The NewtApp Protos 4-2
About newtApplication 4-4
About newtSoup 4-5
The Layout Protos 4-5
The Entry View Protos 4-8
About the Slot View Protos 4-9
Stationery 4-11
NewtApp Compatibility 4-11
Using NewtApp 4-12
Constructing a NewtApp Application 4-12
Using Application Globals 4-13
Using newtApplication 4-14
Using the Layout Protos 4-16
Using Entry Views 4-19
Using the Required NewtApp Install and Remove Scripts 4-21
Using Slot Views in Non-NewtApp Applications 4-22
Modifying the Base View 4-22
Using a False Entry View 4-23
Creating a Custom Labelled Input-Line Slot View 4-24
Summary of the NewtApp Framework 4-25
Required Code 4-25
Protos 4-25

Chapter 5 Stationery 5-1

About Stationery 5-1
The Stationery Buttons 5-2
Stationery Registration 5-4
Getting Information about Stationery 5-5
Compatibility Information 5-5
Using Stationery 5-5
Designing Stationery 5-5
Using FillNewEntry 5-6
Extending the Notes Application 5-7
Determining the SuperSymbol of the Host 5-7

Creating a DataDef 5-8
Defining DataDef Methods 5-9
Creating ViewDefs 5-11
Registering Stationery for an Auto Part 5-13
Using the MinimalBounds ViewDef Method 5-14
Stationery Summary 5-15
Data Structures 5-15
Protos 5-15
Functions 5-17

Chapter 6 Pickers, Pop-up Views, and Overviews 6-1

About Pickers and Pop-up Views 6-1
Pickers and Pop-up View Compatibility 6-2
New Pickers and Pop-up Views 6-2
Obsolete Function 6-4
Picker Categories 6-4
General-Purpose Pickers 6-4
Using protoGeneralPopup 6-7
Map Pickers 6-8
Text Pickers 6-10
Date, Time, and Location Pop-up Views 6-17
Number Pickers 6-21
Picture Picker 6-21
Overview Protos 6-22
Using protoOverview 6-24
Using protoListPicker 6-26
Using the Data Definitions Frame in a List Picker 6-29
Specifying Columns 6-29
Having a Single Selection in a List Picker 6-30
Having Preselected Items in a List Picker 6-30
Validation and Editing in protoListPicker 6-31
Changing the Font of protoListPicker 6-33
Using protoSoupOverview 6-33
Determining Which protoSoupOverview Item Is Hit 6-33
Displaying the protoSoupOverview Vertical Divider 6-34
Roll Protos 6-35
View Classes 6-36

Specifying the List of Items for a Popup 6-37
Summary 6-41
General Picker Protos 6-41
Map Pickers 6-45
Text Picker Protos 6-46
Date, Time, and Location Pop-up Views 6-50
Number Pickers 6-53
Picture Picker 6-53
Overview Protos 6-54
Roll Protos 6-57
View Classes 6-58
Functions 6-59

Chapter 7 Controls and Other Protos 7-1

Controls Compatibility 7-1
Scroller Protos 7-2
Implementing a Minimal Scroller 7-3
Automatic Arrow Feedback 7-3
Scrolling Examples 7-4
Scrolling Lines of Text 7-4
Scrolling in the Dates Application 7-5
Scrolling In a Graphics Application 7-5
Scroll Amounts 7-5
Advanced Usage 7-6
Button and Box Protos 7-6
Implementing a Simple Button 7-10
Selection Tab Protos 7-11
Gauge and Slider Protos 7-12
Implementing a Simple Slider 7-13
Time Protos 7-14
Implementing a Simple Time Setter 7-15
Special View Protos 7-16
View Appearance Protos 7-18
Status Bar Protos 7-19
Summary 7-20
Scroller Protos 7-20
Button and Box Protos 7-22

Selection Tab Protos 7-25
Gauges and Slider Protos 7-25
Time Protos 7-27
Special View Protos 7-28
View Appearance Protos 7-30
Status Bar Protos 7-31

Chapter 8 Text and Ink Input and Display 8-1

About Text 8-1
About Text and Ink 8-1
Written Input Formats 8-2
Caret Insertion Writing Mode 8-3
Fonts for Text and Ink Display 8-3
About Text Views and Protos 8-3
About Keyboard Text Input 8-4
The Keyboard Registry 8-5
The Punctuation Pop-up Menu 8-5
Compatibility 8-6
Using Text 8-6
Using Views and Protos for Text Input and Display 8-6
General Input Views 8-6
Paragraph Views 8-10
Lightweight Paragraph Views 8-11
Using Input Line Protos 8-12
Displaying Text and Ink 8-14
Text and Ink in Views 8-14
Using Fonts for Text and Ink Display 8-17
Rich Strings 8-22
Text and Styles 8-25
Setting the Caret Insertion Point 8-26
Using Keyboards 8-26
Keyboard Views 8-26
Using Keyboard Protos 8-28
Defining Keys in a Keyboard View 8-30
Using the Keyboard Registry 8-36
Defining Tabbing Orders 8-36
The Caret Pop-up Menu 8-38

Handling Input Events 8-38
Testing for a Selection Hit 8-38
Summary of Text 8-39
Text Constants and Data Structures 8-39
Views 8-42
Protos 8-43
Text and Ink Display Functions and Methods 8-47
Keyboard Functions and Methods 8-49
Input Event Functions and Methods 8-50

Chapter 9 Recognition 9-1

About the Recognition System 9-1
Classifying Strokes 9-3
Gestures 9-4
Shapes 9-5
Text 9-6
Unrecognized Strokes 9-7
Enabling Recognizers 9-8
View Flags 9-9
Recognition Configuration Frames 9-9
View Flags vs. RecConfig Frames 9-10
Where to Go From Here 9-10
Recognition Failure 9-11
System Dictionaries 9-11
Correction and Learning 9-13
User Preferences for Recognition 9-14
Handwriting Recognition Preferences 9-15
RecToggle Views 9-18
Flag-Naming Conventions 9-19
Recognition Compatibility 9-20
Using the Recognition System 9-21
Types of Views 9-21
Configuring the Recognition System 9-22
Obtaining Optimum Recognition Performance 9-23
Accepting Pen Input 9-24
Taps and Overlapping Views 9-24
Recognizing Shapes 9-25

Recognizing Standard Gestures 9-25
Combining View Flags 9-26
Recognizing Text 9-27
Recognizing Punctuation 9-28
Suppressing Spaces Between Words 9-28
Forcing Capitalization 9-29
Justifying to Width of Parent View 9-29
Restricting Input to Single Lines or Single Words 9-29
Validating Clipboard and Keyboard Input 9-29
Using the vAnythingAllowed Mask 9-30
Summary 9-31
Constants 9-31
Data Structures 9-33

Chapter 10 Recognition: Advanced Topics 10-1

About Advanced Topics in Recognition 10-1
How the System Uses Recognition Settings 10-1
ProtoCharEdit Views 10-4
Ambiguous Characters in protoCharEdit Views 10-5
Deferred Recognition 10-5
User Interface to Deferred Recognition 10-5
Programmer’s Overview of Deferred Recognition 10-6
Compatibility Information 10-7
Using Advanced Topics in Recognition 10-7
Using recConfig Frames 10-8
Creating a recConfig Frame 10-9
Using RecConfig Frames to Enable Recognizers 10-10
Returning Text, Ink Text or Sketch Ink 10-10
Fine-Tuning Text Recognition 10-12
Manipulating Dictionaries 10-13
Single-Character Input Views 10-13
Creating Single-Letter Input Views 10-15
Changing Recognition Behavior Dynamically 10-17
Using protoRecToggle Views 10-19
Creating the recToggle View 10-19
Configuring Recognizers and Dictionaries for recToggle Views 10-20
Creating the _recogSettings Slot 10-20

Providing the _recogPopup Slot 10-22
Accessing Correction Information 10-23
Using Custom Dictionaries 10-24
Creating a Custom Enumerated Dictionary 10-24
Creating the Blank Dictionary 10-25
Adding Words to RAM-Based Dictionaries 10-26
Removing Words From RAM-Based Dictionaries 10-27
Saving Dictionary Data to a Soup 10-27
Restoring Dictionary Data From a Soup 10-28
Using Your RAM-Based Custom Dictionary 10-28
Removing Your RAM-Based Custom Dictionary 10-30
Using System Dictionaries Individually 10-30
Working With the Review Dictionary 10-30
Retrieving the Review Dictionary 10-31
Displaying Review Dictionary Browsers 10-31
Adding Words to the User Dictionary 10-32
Removing Words From the User Dictionary 10-32
Adding Words to the Expand Dictionary 10-33
Removing Words From the Expand Dictionary 10-34
Retrieving Word Expansions 10-34
Retrieving the Auto-Add Dictionary 10-34
Disabling the Auto-Add Mechanism 10-35
Adding Words to the Auto-Add Dictionary 10-35
Removing Words From the Auto-Add Dictionary 10-36
Using protoCharEdit Views 10-36
Positioning protoCharEdit Views 10-36
Manipulating Text in protoCharEdit Views 10-37
Restricting Characters Returned by protoCharEdit Views 10-38
Customized Processing of Input Strokes 10-40
Customized Processing of Double Taps 10-41
Changing User Preferences for Recognition 10-41
Modifying or Replacing the Correction Picker 10-42
Using Stroke Bundles 10-42
Stroke Bundles Example 10-42
Summary of Advanced Topics in Recognition 10-44
Constants 10-44
Data Structures 10-45
Recognition System Prototypes 10-49
Additional Recognition Functions and Methods 10-54

Chapter 11 Data Storage and Retrieval 11-1

About Data Storage on Newton Devices 11-1
Introduction to Data Storage Objects 11-2
Where to Go From Here 11-6
Stores 11-6
Packages 11-7
Soups 11-7
Indexes 11-8
Saving User Preference Data in the System Soup 11-10
Queries 11-10
Querying for Indexed Values 11-10
Begin Keys and End Keys 11-12
Tag-based Queries 11-14
Customized Tests 11-14
Text Queries 11-15
Cursors 11-16
Entries 11-17
Alternatives to Soup-Based Storage 11-18
Dynamic Data 11-18
Static Data 11-19
Compatibility Information 11-20
Obsolete Store Functions and Methods 11-20
Soup Compatibility Information 11-20
Query Compatibility Information 11-23
Obsolete Entry Functions 11-24
Obsolete Data Backup and Restore Functions 11-24
Using Newton Data Storage Objects 11-25
Programmer's Overview 11-25
Using Stores 11-28
Store Object Size Limits 11-29
Referencing Stores 11-29
Retrieving Packages From Stores 11-29
Testing Stores for Write-Protection 11-30
Getting or Setting the Default Store 11-30
Getting and Setting the Store Name 11-30
Accessing the Store Information Frame 11-31
Using Soups 11-31
Naming Soups 11-31
Registering and Unregistering Soup Definitions 11-32

Retrieving Existing Soups 11-33
Adding Entries to Soups 11-34
Adding an Index to an Existing Soup 11-35
Removing Soups 11-36
Using Built-in Soups 11-36
Making Changes to Other Applications' Soups 11-37
Adding Tags to an Existing Soup 11-37
Using Queries 11-37
Querying Multiple Soups 11-38
Querying on Single-Slot Indexes 11-38
Querying for Tags 11-41
Querying for Text 11-43
Internationalized Sorting Order for Text Queries 11-44
Queries on Descending Indexes 11-45
Querying on Multiple-Slot Indexes 11-47
Limitations of Index Keys 11-51
Using Cursors 11-53
Getting a Cursor 11-53
Testing Validity of the Cursor 11-53
Getting the Entry Currently Referenced by the Cursor 11-54
Moving the Cursor 11-54
Counting the Number of Entries in Cursor Data 11-56
Getting the Current Entry's Index Key 11-56
Copying Cursors 11-56
Using Entries 11-57
Saving Frames as Soup Entries 11-57
Removing Entries From Soups 11-58
Modifying Entries 11-59
Moving Entries 11-60
Copying Entries 11-60
Sharing Entry Data 11-61
Using the Entry Cache Efficiently 11-61
Using Soup Change Notification 11-63
Registering Your Application for Change Notification 11-63
Unregistering Your Application for Change Notification 11-65
Responding to Notifications 11-65
Sending Notifications 11-66
Summary of Data Storage 11-68
Data Structures 11-68
Data Storage Functions and Methods 11-71

Chapter 12 Special-Purpose Objects for Data Storage and Retrieval 12-1

About Special-Purpose Storage Objects 12-1
Entry Aliases 12-1
Virtual Binary Objects 12-2
Parts 12-3
Store Parts 12-4
Mock Entries 12-4
Mock Stores, Mock Soups, and Mock Cursors 12-6
Using Special-Purpose Data Storage Objects 12-7
Using Entry Aliases 12-7
Using Virtual Binary Objects 12-8
Creating Virtual Binary Objects 12-8
Modifying VBO Data 12-10
VBOs and String Data 12-12
Using Store Parts 12-12
Creating a Store Part 12-13
Getting the Store Part 12-14
Accessing Data in Store Parts 12-14
Using Mock Entries 12-14
Implementing the EntryAccess Method 12-15
Creating a New Mock Entry 12-15
Testing the Validity of a Mock Entry 12-16
Getting Mock Entry Data 12-16
Changing the Mock Entry's Handler 12-16
Getting the Mock Entry's Handler 12-16
Implementing Additional Handler Methods 12-16
Summary of Special-Purpose Data Storage Objects 12-17
Data Structures 12-17
Functions and Methods 12-17

Chapter 13 Drawing and Graphics 13-1

About Drawing 13-1
Shape-Based Graphics 13-2
Manipulating Shapes 13-7
The Style Frame 13-7

Drawing Compatibility 13-8
New Functions 13-8
New Style Attribute Slots 13-8
Changes to Bitmaps 13-8
Changes to the HitShape Method 13-8
Changes to View Classes 13-9
Using the Drawing Interface 13-9
How to Draw 13-9
Responding to the ViewDrawScript Message 13-9
Drawing Immediately 13-10
Using Nested Arrays of Shapes 13-10
The Transform Slot in Nested Shape Arrays 13-11
Default Transfer Mode 13-12
Transfer Modes at Print Time 13-12
Controlling Clipping 13-12
Transforming a Shape 13-13
Using Drawing View Classes and Protos 13-14
Displaying Graphics Shapes and Ink 13-14
Displaying Bitmaps, Pictures, and Graphics Shapes 13-15
Displaying Pictures in a clEditView 13-15
Displaying Scaled Images of Other Views 13-15
Translating Data Shapes 13-16
Finding Points Within a Shape 13-16
Using Bitmaps 13-17
Making CopyBits Scale Its Output Bitmap 13-18
Storing Compressed Pictures and Bitmaps 13-18
Capturing a Portion of a View Into a Bitmap 13-18
Rotating or Flipping a Bitmap 13-19
Importing Macintosh PICT Resources 13-20
Drawing Non-Default Fonts 13-20
PICT Swapping During Run-Time Operations 13-21
Optimizing Drawing Performance 13-22
Summary of Drawing 13-23
Data Structure 13-23
View Classes 13-23
Protos 13-24
Functions and Methods 13-26

Chapter 14 Sound 14-1

About Newton Sound 14-1
Event-related Sounds 14-2
Sounds in ROM 14-2
Sounds for Predefined Events 14-2
Sound Data Structures 14-3
Compatibility 14-3
Using Sound 14-4
Creating and Using Custom Sound Frames 14-4
Creating Sound Frames Procedurally 14-5
Cloning Sound Frames 14-5
Playing Sound 14-5
Using a Sound Channel to Play Sound 14-5
Playing Sound Programmatically 14-6
Synchronous and Asynchronous Sound 14-7
Advanced Sound Techniques 14-8
Pitch Shifting 14-9
Manipulating Sample Data 14-10
Summary of Sound 14-11
Data Structures 14-11
Protos 14-11
Functions and Methods 14-12
Sound Resources 14-12

Chapter 15 Filing 15-1

About Filing 15-1
Filing Compatibility Information 15-9
Using the Filing Service 15-10
Overview of Filing Support 15-10
Creating the Labels Slot 15-11
Creating the appName Slot 15-11
Creating the appAll Slot 15-12
Creating the appObjectFileThisIn Slot 15-12
Creating the appObjectFileThisOn Slot 15-12
Creating the appObjectUnfiled Slot 15-12
Specifying the Target 15-13

Creating the labelsFilter slot 15-14
Creating the storesFilter slot 15-14
Adding the Filing Button 15-14
Adding the Folder Tab View 15-14
Customizing Folder Tab Views 15-15
Defining a TitleClickScript Method 15-15
Implementing the FileThis Method 15-15
Implementing the NewFilingFilter Method 15-16
Using the Folder Change Notification Service 15-18
Creating the doCardRouting slot 15-18
Using Local or Global Folders Only 15-19
Adding and Removing Filing Categories Programmatically 15-19
Interface to User-Visible Folder Names 15-19
Summary 15-20
Data Structures for Filing 15-20
Application Base View Slots 15-20
Filing Protos 15-21
Filing Functions and Methods 15-22
Application-Defined Filing Functions and Methods 15-22

Chapter 16 Find 16-1

About the Find Service 16-1
Compatibility Information 16-6
Using the Find Service 16-6
Technical Overview 16-6
Global and Selected Finds 16-9
Checklist for Adding Find Support 16-10
Creating the title Slot 16-11
Creating the appName Slot 16-11
Using the Finder Protos 16-11
Implementing Search Methods 16-14
Using the StandardFind Method 16-15
Using Your Own Text-Searching Method 16-16
Finding Text With a ROM_CompatibleFinder 16-17
Implementing the DateFind Method 16-18
Adding Application Data Sets to Selected Finds 16-19
Returning Search Results 16-21

Implementing Find Overview Support 16-21
The FindSoupExcerpt Method 16-21
The ShowFoundItem Method 16-22
Replacing the Built-in Find Slip 16-24
Reporting Progress to the User 16-24
Registering for Finds 16-25
Summary 16-26
Finder Protos 16-26
Functions and Methods 16-28
Application-Defined Methods 16-28

Chapter 17 Additional System Services 17-1

About Additional System Services 17-1
Undo 17-1
Undo Compatibility 17-2
Idler Objects 17-2
Change Notifications 17-2
Online Help 17-3
Alerts and Alarms 17-3
User Alerts 17-3
User Alarms 17-3
Periodic Alarms 17-4
Alarms Compatibility 17-5
Progress Indicators 17-5
Automatic Busy Cursor 17-5
Notify Icon 17-5
Status Slips With Progress Indicators 17-6
Power Registry 17-7
Power Compatibility Information 17-7
Using Additional System Services 17-7
Using Undo Actions 17-8
The Various Undo Methods 17-8
Avoiding Undo-Related “Bad Package” Errors 17-9
Using Idler Objects 17-9
Using Change Notification 17-10
Using Online Help 17-10

Using Alerts and Alarms 17-11
Using the Notify Method to Display User Alerts 17-11
Creating Alarms 17-11
Obtaining Information about Alarms 17-12
Retrieving Alarm Keys 17-12
Removing Installed Alarms 17-13
Common Problems With Alarms 17-13
Using the Periodic Alarm Editor 17-14
Using Progress Indicators 17-15
Using the Automatic Busy Cursor 17-15
Using the Notify Icon 17-15
Using the DoProgress Function 17-16
Using DoProgress or Creating Your Own protoStatusTemplate 17-18
Using protoStatusTemplate Views 17-18
Using the Power Registry 17-24
Registering Power-On Functions 17-24
Registering Login Screen Functions 17-25
Registering Power-Off Functions 17-25
Using the Battery Information Functions 17-26
Summary of Additional System Services 17-27
Undo 17-27
Idlers 17-27
Notification and Alarms 17-27
Reporting Progress 17-28
Power Registry 17-29

Chapter 18 Intelligent Assistant 18-1

About the Assistant 18-1
Introduction 18-1
Input Strings 18-2
No Verb in Input String 18-2
Ambiguous or Missing Information 18-4
The Task Slip 18-4
Programmer's Overview 18-5
Matching Words With Templates 18-8
The Signature and PreConditions Slots 18-10

The Task Frame 18-11
The Entries Slot 18-11
The Phrases Slot 18-11
The OrigPhrase Slot 18-12
The Value Slot 18-12
Resolving Template-Matching Conflicts 18-13
Compatibility Information 18-14
Using the Assistant 18-15
Making Behavior Available From the Assistant 18-15
Defining Action and Target Templates 18-15
Defining Your Own Frame Types to the Assistant 18-16
Implementing the PostParse Method 18-17
Defining the Task Template 18-18
Registering and Unregistering the Task Template 18-19
Displaying Online Help From the Assistant 18-19
Routing Items From the Assistant 18-20
Summary 18-21
Data Structures 18-21
Templates 18-21
Developer-Supplied Task Template 18-22
Developer-Supplied Action Templates 18-25
Developer-Supplied Target Templates 18-27
Assistant Functions and Methods 18-27
Developer-Supplied Functions and Methods 18-28
Application Base View Slots 18-28

Chapter 19 Built-in Applications and System Data 19-1

Names 19-2
About the Names Application 19-2
Names Compatibility 19-3
Using the Names Application 19-4
Adding a New Type of Card 19-4
Adding a New Data Item 19-4
Adding a New Card Layout Style 19-5
Adding New Layouts to the Names Application 19-6
Using the Names Methods and Functions 19-6
Using the Names Soup 19-7
Using the Names Protos 19-7

Dates 19-8
About the Dates Application 19-8
Dates Compatibility 19-9
Using the Dates Application 19-10
Adding Meetings or Events 19-11
Deleting Meetings and Events 19-12
Finding Meetings or Events 19-13
Moving Meetings and Events 19-14
Getting and Setting Information for Meetings or Events 19-15
Creating a New Meeting Type 19-17
Examples of Creating New Meeting Types 19-19
Miscellaneous Operations 19-20
Controlling the Dates Display 19-21
Using the Dates Soups 19-22
To Do List 19-22
About the To Do List Application 19-22
To Do List Compatibility 19-23
Using the To Do List Application 19-23
Creating and Removing Tasks 19-24
Accessing Tasks 19-24
Checking-Off a Task 19-25
Miscellaneous To Do List Methods 19-26
Using the To Do List Soup 19-26
Time Zones 19-27
About the Time Zones Application 19-27
Time Zone Compatibility 19-27
Using the Time Zone Application 19-28
Obtaining Information About a City or Country 19-28
Adding a City to a Newton Device 19-29
Using Longitude and Latitude Values 19-30
Setting the Home City 19-30
Notes 19-30
About the Notes Application 19-31
Notes Compatibility 19-31
Using the Notes Application 19-32
Creating New Notes 19-32
Adding Stationery to the Notes Application 19-33
Using the Notes Soup 19-33

Fax Soup Entries 19-34
About Fax Soup Entries 19-34
Using Fax Soup Entries 19-34
Prefs and Formulas Rolls 19-35
About the Prefs and Formulas Rolls 19-35
Prefs and Formulas Compatibility 19-36
Using the Prefs and Formulas Interfaces 19-36
Adding a Prefs Roll Item 19-36
Adding a Formulas Roll Item 19-36
Auxiliary Buttons 19-36
About Auxiliary Buttons 19-36
Auxiliary Buttons Compatibility 19-36
Using Auxiliary Buttons 19-37
Icons and the Extras Drawer 19-38
About Icons and the Extras Drawer 19-38
Extras Drawer Compatibility 19-39
Using the Extras Drawer's Interface for Icon Management 19-39
Using Extras Drawer Cursors 19-40
Changing Icon Information 19-40
Adding a Soup Icon 19-40
Removing a Soup Icon 19-41
Creating a Script Icon 19-42
Using the Soupervisor Mechanism 19-43
System Data 19-44
About System Data 19-44
Using System Data 19-44
Functions for Accessing User Configuration Data 19-45
Storing Application Preferences in the System Soup 19-45
Summary 19-46
Constants and Variables 19-46
User Configuration Variables 19-47
Protos 19-48
Soup Formats 19-49
Functions and Methods 19-53

Chapter 20

Localizing Newton Applications 20-1

About Localization 20-1
The Locale Panel and the International Frame 20-1
Locale and ROM Version 20-2
How Locale Affects Recognition 20-2
Using the Localization Features of the Newton 20-3
Defining Language at Compile Time 20-3
Defining a Localization Frame 20-4
Using LocObj to Reference Localized Objects 20-4
Use ParamStr Rather Than "&" and "&&" Concatenation 20-5
Measuring String Widths at Compile Time 20-6
Determining Language at Run Time 20-6
Examining the Active Locale Bundle 20-6
Changing Locale Settings 20-7
Creating a Custom Locale Bundle 20-7
Adding a New Bundle to the System 20-8
Removing a Locale Bundle 20-8
Changing the Active Locale 20-9
Using a Localized Country Name 20-9
Summary: Customizing Locale 20-9
Localized Output 20-10
Date and Time Values 20-10
Currency Values 20-13
Summary of Localization Functions 20-14
Compile-Time Functions 20-14
Locale Functions 20-14
Date and Time Functions 20-14
Utility Functions 20-15

Chapter 21

Routing Interface 21-1

About Routing 21-1
The In/Out Box 21-1
The In Box 21-2
The Out Box 21-3
Action Picker 21-3

Routing Formats 21-5
Current Format 21-8
Routing Compatibility 21-8
Print Formats 21-8
Using Routing 21-8
Providing Transport-Based Routing Actions 21-9
Getting and Verifying the Target Object 21-10
Getting and Setting the Current Format 21-11
Supplying the Target Object 21-12
Storing an Alias to the Target Object 21-13
Storing Multiple Items 21-14
Using the Built-in Overview Data Class 21-14
Displaying an Auxiliary View 21-15
Registering Routing Formats 21-16
Creating a Print Format 21-18
Page Layout 21-18
Printing and Faxing 21-19
Creating a Frame Format 21-21
Creating a New Type of Format 21-22
Providing Application-Specific Routing Actions 21-22
Performing the Routing Action 21-24
Handling Multiple Items 21-24
Handling No Target Item 21-25
Sending Items Programmatically 21-26
Creating a Name Reference 21-27
Specifying a Printer 21-28
Opening a Routing Slip Programmatically 21-29
Supporting the Intelligent Assistant 21-30
Receiving Data 21-31
Automatically Putting Away Items 21-31
Manually Putting Away Items 21-33
Registering to Receive Foreign Data 21-34
Filing Items That Are Put Away 21-34
Viewing Items in the In/Out Box 21-34
View Definition Slots 21-35
Advanced Alias Handling 21-36
Summary of the Routing Interface 21-37
Constants 21-37
Data Structures 21-37

Protos 21-38
Functions and Methods 21-39
Application-Defined Methods 21-40

Chapter 22 Transport Interface 22-1

About Transports 22-1
Transport Parts 22-2
Item Frame 22-2
Using the Transport Interface 22-5
Providing a Transport Object 22-5
Installing the Transport 22-5
Setting the Address Class 22-6
Grouping Transports 22-7
Sending Data 22-8
Sending All Items 22-9
Converting an E-Mail Address to an Internet Address 22-9
Receiving Data 22-9
Handling Requests When the Transport Is Active 22-12
Canceling an Operation 22-13
Obtaining an Item Frame 22-13
Completion and Logging 22-16
Storing Transport Preferences and Configuration Information 22-17
Extending the In/Out Box Interface 22-17
Application Messages 22-19
Error Handling 22-20
Power-Off Handling 22-20
Providing a Status Template 22-21
Controlling the Status View 22-23
Providing a Routing Information Template 22-25
Providing a Routing Slip Template 22-26
Using protoFullRouteSlip 22-27
Using protoAddressPicker 22-31
Providing a Preferences Template 22-33
Summary of the Transport Interface 22-36
Constants 22-36
Protos 22-36
Functions and Methods 22-39

Chapter 23 **Endpoint Interface** 23-1

About the Endpoint Interface 23-1
Asynchronous Operation 23-2
Synchronous Operation 23-3
Input 23-3
Data Forms 23-4
Template Data Form 23-5
Endpoint Options 23-7
Compatibility 23-7
Using the Endpoint Interface 23-8
Setting Endpoint Options 23-8
Initialization and Termination 23-10
Establishing a Connection 23-11
Sending Data 23-11
Receiving Data Using Input Specs 23-12
Specifying the Data Form and Target 23-13
Specifying Data Termination Conditions 23-14
Specifying Flags for Receiving 23-15
Specifying an Input Time-Out 23-16
Specifying Data Filter Options 23-16
Specifying Receive Options 23-17
Handling Normal Termination of Input 23-17
Periodically Sampling Incoming Data 23-18
Handling Unexpected Completion 23-18
Special Considerations 23-18
Receiving Data Using Alternative Methods 23-19
Streaming Data In and Out 23-20
Working With Binary Data 23-20
Canceling Operations 23-21
Asynchronous Cancellation 23-21
Synchronous Cancellation 23-22
Other Operations 23-22
Error Handling 23-23
Power-Off Handling 23-23
Linking the Endpoint With an Application 23-24
Summary of the Endpoint Interface 23-25
Constants and Symbols 23-25
Data Structures 23-26
Protos 23-28
Functions and Methods 23-30

Chapter 24 Built-in Communications Tools 24-1

Serial Tool 24-1
Standard Asynchronous Serial Tool 24-1
Serial Tool with MNP Compression 24-4
Framed Asynchronous Serial Tool 24-4
Modem Tool 24-6
Infrared Tool 24-8
AppleTalk Tool 24-9
Resource Arbitration Options 24-10
AppleTalk Functions 24-12
The Net Chooser 24-13
Summary 24-16
Built-in Communications Tool Service Option Labels 24-16
Options 24-16
Constants 24-18
Functions and Methods 24-21

Chapter 25 Modem Setup Service 25-1

About the Modem Setup Service 25-1
The Modem Setup User Interface 25-2
The Modem Setup Process 25-3
Modem Communication Tool Requirements 25-4
Defining a Modem Setup 25-5
Setting Up General Information 25-5
Setting the Modem Preferences Option 25-5
Setting the Modem Profile Option 25-6
Setting the Fax Profile Option 25-7
Summary of the Modem Setup Service 25-9
Constants 25-9

ARENDI-DEFS00003677

Chapter 26 **Utility Functions** 26-1

Compatibility 26-2
New Functions 26-2
New Object System Functions 26-2
New String Functions 26-3
New Array Functions 26-3
New Sorted Array Functions 26-3
New Integer Math Functions 26-4
New Financial Functions 26-4
New Exception Handling Functions 26-4
New Message Sending Functions 26-4
New Deferred Message Sending Functions 26-4
New Data Stuffing Functions 26-5
New Functions to Get and Set Globals 26-5
New Debugging Functions 26-5
New Miscellaneous Functions 26-5
Enhanced Functions 26-6
Obsolete Functions 26-6
Summary of Functions and Methods 26-7
Object System Functions 26-7
String Functions 26-8
Bitwise Functions 26-9
Array Functions 26-9
Sorted Array Functions 26-9
Integer Math Functions 26-10
Floating Point Math Functions 26-10
Financial Functions 26-12
Exception Functions 26-12
Message Sending Functions 26-12
Deferred Message Sending Functions 26-12
Data Extraction Functions 26-13
Data Stuffing Functions 26-13
Getting and Setting Global Variables and Functions 26-13
Debugging Functions 26-13
Miscellaneous Functions 26-14

Appendix The Inside Story on Declare A-1

Compile-Time Results A-1
Run-Time Results A-2

Glossary GL-1

Index IN-1

Figures and Tables

Chapter 1 **Overview** 1-1

Figure 1-1 System software overview 1-2
Figure 1-2 Communications architecture 1-12
Figure 1-3 Using components 1-16

Chapter 3 **Views** 3-1

Figure 3-1 Template hierarchy 3-3
Figure 3-2 View hierarchy 3-5
Figure 3-3 Screen representation of view hierarchy 3-6
Figure 3-4 View system coordinate plane 3-7
Figure 3-5 Points and pixels 3-7
Figure 3-6 Bounds parameters 3-11
Figure 3-7 View alignment effects 3-18
Figure 3-8 Transfer modes 3-22

Table 3-1 `viewJustify` constants 3-14

Chapter 4 **NewtApp Applications** 4-1

Figure 4-1 The main protos in a NewtApp-based application 4-3
Figure 4-2 A roll-based application (left) versus a card-based application 4-6
Figure 4-3 Calls is an example of a page-based application 4-7
Figure 4-4 Multiple entries visible simultaneously 4-8
Figure 4-5 An Information slip 4-9
Figure 4-6 The smart name view and system-provided people picker 4-11
Figure 4-7 The message resulting from a `nil` value for `forceNewEntry` 4-17
Figure 4-8 The overview slots 4-17
Figure 4-9 The information button and picker. 4-20

Chapter 5 Stationery 5-1

Figure 5-1 The IOU extension in the New picker 5-3
Figure 5-2 The IOU extension to the Notes application 5-3
Figure 5-3 The Show menu presents different views of application data 5-4
Figure 5-4 The default viewDef view template 5-12

Chapter 6 Pickers, Pop-up Views, and Overviews 6-1

Figure 6-1 A `protoPopupButton` example 6-5
Figure 6-2 A `protoPopInPlace` example 6-5
Figure 6-3 A `protoLabelPicker` example 6-5
Figure 6-4 A `protoPicker` example 6-6
Figure 6-5 A `protoGeneralPopup` example 6-6
Figure 6-6 A `protoTextList` example 6-7
Figure 6-7 A `protoTable` example 6-7
Figure 6-8 A `protoCountryPicker` example 6-9
Figure 6-9 A `protoProvincePicker` example 6-9
Figure 6-10 A `protoStatePicker` example 6-9
Figure 6-11 A `protoWorldPicker` example 6-10
Figure 6-12 A `protoTextPicker` example 6-10
Figure 6-13 A `protoDateTextPicker` example 6-11
Figure 6-14 A `protoDateDurationTextPicker` example 6-12
Figure 6-15 A `protoDateNTimeTextPicker` example 6-13
Figure 6-16 A `protoTimeTextPicker` example 6-13
Figure 6-17 A `protoDurationTextPicker` example 6-14
Figure 6-18 A `protoTimeDeltaTextPicker` example 6-14
Figure 6-19 A `protoMapTextPicker` example 6-15
Figure 6-20 A `protoUSstatesTextPicker` example 6-15
Figure 6-21 A `protoCitiesTextPicker` example 6-16
Figure 6-22 A `protoLongLatTextPicker` example 6-16
Figure 6-23 A `protoDatePopup` example 6-17
Figure 6-24 A `protoDatePicker` example 6-17
Figure 6-25 A `protoDateNTimePopup` example 6-18
Figure 6-26 A `protoDateIntervalPopup` example 6-18
Figure 6-27 A `protoMultiDatePopup` example 6-19
Figure 6-28 A `protoYearPopup` example 6-19
Figure 6-29 A `protoTimePopup` example 6-19
Figure 6-30 A `protoAnalogTimePopup` example 6-20
Figure 6-31 A `protoTimeDeltaPopup` example 6-20
Figure 6-32 A `protoTimeIntervalPopup` example 6-20
Figure 6-33 A `protoNumberPicker` example 6-21
Figure 6-34 A `protoPictIndexer` example 6-21

Figure 6-35 A protoOverview example 6-22
Figure 6-36 A protoSoupOverview example 6-23
Figure 6-37 A protoListPicker example 6-24
Figure 6-38 A ProtoListPicker example 6-26
Figure 6-39 Creating a new name entry 6-27
Figure 6-40 Highlighted row 6-27
Figure 6-41 Selected row 6-27
Figure 6-42 Pop-up view displayed over list 6-28
Figure 6-43 Slip displayed for gathering input 6-28
Figure 6-44 A protoRoll example 6-35
Figure 6-45 A protoRollBrowser example 6-36
Figure 6-46 Example of an expandable text outline 6-36
Figure 6-47 Example of a month view 6-37
Figure 6-48 Cell highlighting example for protoPicker 6-40

Table 6-1 Item frame for strings and bitmaps 6-38
Table 6-2 Item frame for string with icon 6-38
Table 6-3 Item frame for two-dimensional grid 6-39

Chapter 7 Controls and Other Protos 7-1

Figure 7-1 A protoHorizontal2DScroller view 7-2
Figure 7-2 A protoLeftRightScroller view 7-2
Figure 7-3 A protoUpDownScroller view 7-3
Figure 7-4 A protoHorizontalUpDownScroller view 7-3
Figure 7-5 A protoTextButton view 7-6
Figure 7-6 A protoPictureButton view 7-7
Figure 7-7 A protoInfoButton view 7-7
Figure 7-8 A protoOrientation view 7-7
Figure 7-9 A cluster of protoRadioButtons 7-8
Figure 7-10 A cluster of protoPictRadioButtons 7-8
Figure 7-11 A protoCloseBox view 7-8
Figure 7-12 A protoLargeCloseBox view 7-9
Figure 7-13 A protoCheckBox view 7-9
Figure 7-14 A protoRCheckBox view 7-9
Figure 7-15 A protoAZTabs view 7-11
Figure 7-16 A protoAZVertTabs view 7-11
Figure 7-17 A protoSlider view 7-12
Figure 7-18 A protoGauge view 7-12
Figure 7-19 A protoLabeledBatteryGauge view 7-12
Figure 7-20 A clGaugeView view 7-13
Figure 7-21 A protoDigitalClock view 7-14
Figure 7-22 A protoNewSetClock view 7-15

Figure 7-23 A protoAMPMCluster view 7-15
Figure 7-24 A protoDragger view 7-16
Figure 7-25 A protoDragNGo view 7-16
Figure 7-26 A protoGlance view 7-17
Figure 7-27 A protoStaticText view 7-17
Figure 7-28 A protoBorder view 7-18
Figure 7-29 A protoDivider view 7-18
Figure 7-30 A protoTitle view 7-18
Figure 7-31 A protoStatus view 7-19
Figure 7-32 A protoStatusBar view 7-19

Table 7-1 Scroller bounds frame slots 7-4

Chapter 8 Text and Ink Input and Display 8-1

Figure 8-1 The Punctuation pop-up menu 8-5
Figure 8-2 An example of a protoLabelInputLine 8-13
Figure 8-3 The Recognition menu 8-15
Figure 8-4 Resized and recognized ink 8-16
Figure 8-5 A paragraph view containing an ink word and text 8-25
Figure 8-6 The built-in alphanumeric keyboard 8-26
Figure 8-7 The built-in numeric keyboard 8-27
Figure 8-8 The built-in phone keyboard 8-27
Figure 8-9 The built-in time and date keyboard 8-27
Figure 8-10 An example of a protoKeyboard 8-29
Figure 8-11 The keyboard button 8-29
Figure 8-12 The small keyboard button 8-30
Figure 8-13 A generic keyboard view 8-31
Figure 8-14 Keyboard codes 8-34
Figure 8-15 Independent tabbing orders within a parent view 8-37

Table 8-1 Views and protos for text input and display 8-4
Table 8-2 viewStationery slot value for clEditView children 8-9
Table 8-3 Font family symbols 8-18
Table 8-4 Font style (face) values 8-18
Table 8-5 Built-in font constants 8-19
Table 8-6 Font packing constants 8-21
Table 8-7 Rich string functions 8-24
Table 8-8 Key descriptor constants 8-34

Chapter 9

Recognition 9-1

Figure 9-1 Recognizers create units from input strokes 9-5
Figure 9-2 Recognition-related view flags 9-9
Figure 9-3 Text-corrector picker 9-14
Figure 9-4 Handwriting Recognition preferences 9-16
Figure 9-5 Text Editing Settings slip 9-17
Figure 9-6 Fine Tuning handwriting preferences slips 9-17
Figure 9-7 Handwriting Settings slip 9-18
Figure 9-8 Use of `protoRecToggle` view in the Notes application 9-19

Chapter 10

Recognition: Advanced Topics 10-1

Figure 10-1 Example of `protoCharEdit` view 10-4
Figure 10-2 User interface to deferred recognition, with inverted ink 10-6
Figure 10-3 Single-character editing box specified by `rcBaseInfo` frame 10-13
Figure 10-4 Two-dimensional array of input boxes specified by `rcGridInfo` frame 10-14
Figure 10-5 One `recToggle` controls all views 10-21
Figure 10-6 Each `recToggle` view controls a single input view 10-21
Figure 10-7 Example of a `protoCharEdit` view 10-36

Table 10-1 Recognition failure in paragraph or edit view controlled by `recToggle` 10-12
Table 10-2 Symbols appearing in the `_recogPopup` slot 10-22

Chapter 11

Data Storage and Retrieval 11-1

Figure 11-1 Stores, soups and union soups 11-4
Figure 11-2 An index provides random access and imposes order 11-11
Figure 11-3 Using `beginKey` and `endKey` values to specify an index subrange 11-12
Figure 11-4 Using `beginExclKey` and `endExclKey` values to specify a subrange 11-13
Figure 11-5 Cursor presents discontiguous index key values contiguously 11-16
Figure 11-6 Cursor operations on descending index 11-46
Figure 11-7 Specifying ends of a descending index 11-47

Table 11-1 Effect of functions and methods on entry cache 11-63

Chapter 12 Special-Purpose Objects for Data Storage and Retrieval 12-1

Table 12-1 Parts and type identifiers 12-4

Chapter 13 Drawing and Graphics 13-1

Figure 13-1 A line drawn with different bit patterns and pen sizes 13-3
Figure 13-2 A rectangle 13-3
Figure 13-3 An oval 13-4
Figure 13-4 An arc and a wedge 13-4
Figure 13-5 A rounded rectangle 13-5
Figure 13-6 A polygon 13-6
Figure 13-7 A region 13-6
Figure 13-8 A simple picture 13-7
Figure 13-9 Example of nested shape arrays 13-11
Figure 13-10 Example of `ViewIntoBitmap` method 13-19
Figure 13-11 Example of `MungeBitmap` method 13-19

Table 13-1 Summary of drawing results 13-11

Chapter 15 Filing 15-1

Figure 15-1 Two examples of filing button views 15-2
Figure 15-2 Filing slip 15-3
Figure 15-3 Creating a local folder 15-4
Figure 15-4 Filing slip without external store 15-5
Figure 15-5 Filing slip for `'onlyCardRouting` 15-5
Figure 15-6 A `protoNewFolderTab` view 15-6
Figure 15-7 A `protoClockFolderTab` view 15-7
Figure 15-8 Choosing a filing filter 15-8

Chapter 16 Find 16-1

Figure 16-1 The system-supplied Find slip 16-2
Figure 16-2 Specifying text or date searches in the Find slip 16-2
Figure 16-3 A local Find operation 16-3
Figure 16-4 Searching selected applications 16-3
Figure 16-5 Progress slip 16-4
Figure 16-6 The Find overview 16-5
Figure 16-7 Find status message 16-5
Figure 16-8 Strings used in a Find overview 16-8

Figure 16-9 The `ShowFoundItem` method displays the view of an overview item 16-9
Figure 16-10 Typical status message 16-24

Table 16-1 Overview of `ROM_SoupFinder` methods 16-13

Chapter 17 Additional System Services 17-1

Figure 17-1 User alert 17-3
Figure 17-2 Alarm slip with Snooze button 17-4
Figure 17-3 A view based on protoPeriodicAlarmEditor 17-4
Figure 17-4 Busy cursor 17-5
Figure 17-5 Notify icon 17-5
Figure 17-6 Progress slip with barber pole gauge 17-6
Figure 17-7 A user alert 17-11
Figure 17-8 Built-in status view configurations 17-20

Chapter 18 Intelligent Assistant 18-1

Figure 18-1 Assist slip 18-3
Figure 18-2 The Please picker 18-3
Figure 18-3 Calling task slip 18-4
Figure 18-4 Simplified overview of the Assistant's matching process 18-7

Chapter 19 Built-in Applications and System Data 19-1

Figure 19-1 Names application Card and All Info views 19-3
Figure 19-2 Dates application Day and Day's Agenda views 19-9
Figure 19-3 The To Do List application 19-23
Figure 19-4 The Time Zones application 19-27
Figure 19-5 Time Zones application's All Info view 19-28
Figure 19-6 Notes note and Checklist views 19-31
Figure 19-7 Note added using `NewNote` method 19-33
Figure 19-8 Custom Prefs and Formulas Panels 19-35
Figure 19-9 The Notes application with and without an auxiliary button 19-37
Figure 19-10 The information slips for an application's soup that do and do not support the soupervisor mechanism (note extra filing button) 19-39

Chapter 20 Localizing Newton Applications 20-1

Figure 20-1 The Locale settings in Preferences 20-2

Table 20-1 Using the `kIncludeAllElements` constant 20-13

Chapter 21 Routing Interface 21-1

Figure 21-1 In Box and Out Box overviews 21-2
Figure 21-2 Action picker 21-3
Figure 21-3 Transport selection mechanism for action picker 21-6
Figure 21-4 Format picker in routing slip 21-7
Figure 21-5 Auxiliary view example 21-15

Table 21-1 Routing data types 21-7

Chapter 22 Transport Interface 22-1

Figure 22-1 Status view subtypes 22-22
Figure 22-2 Routing information view 22-26
Figure 22-3 `protoFullRouteSlip` view 22-27
Figure 22-4 Complete routing slip 22-29
Figure 22-5 `protoPeoplePicker` view 22-31
Figure 22-6 Address picker with remembered names 22-32
Figure 22-7 Address picker set up by Intelligent Assistant 22-32
Figure 22-8 Information picker and preferences view 22-33
Figure 22-9 `protoTransportPrefs` view 22-34
Figure 22-10 Print preferences 22-35

Table 22-1 Status view subtypes 22-21

Chapter 23 Endpoint Interface 23-1

Table 23-1 Data form applicability 23-5
Table 23-2 Input spec slot applicability 23-13

ARENDI-DEFS00003688

Chapter 24 Built-in Communications Tools 24-1

Figure 24-1 Default serial framing 24-5
Figure 24-2 NetChooser view while searching 24-14
Figure 24-3 NetChooser view displaying printers 24-14

Table 24-1 Summary of serial options 24-2
Table 24-2 Summary of serial tool with MNP options 24-4
Table 24-3 Summary of framed serial options 24-5
Table 24-4 Summary of modem options 24-7
Table 24-5 Summary of Infrared Options 24-8
Table 24-6 Summary of AppleTalk options 24-10
Table 24-7 Resource arbitration options 24-11
Table 24-8 AppleTalk functions 24-13

Chapter 25 Modem Setup Service 25-1

Figure 25-1 Modem preferences view 25-3

Table 25-1 Summary of configuration string usage 25-7

Chapter 26 Utility Functions 26-1

Table 26-1 Summary of copying functions 26-2

Appendix The Inside Story on Declare A-1

Figure A-1 Declare example A-3

P R E F A C E

About This Book

This book, *Newton Programmer's Guide*, is the definitive guide to Newton programming, providing conceptual information and instructions for using the Newton application programming interfaces.

This book is a companion to *Newton Programmer's Reference*, which provides comprehensive reference documentation for the routines, system prototypes, data structures, constants, and error codes defined by the Newton system. *Newton Programmer's Reference* is included on the CD-ROM that accompanies this book.

Who Should Read This Book

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

Before using this guide, you should read *Newton Toolkit User's Guide* to learn how to install and use Newton Toolkit, which is the development environment for writing NewtonScript programs for Newton. You may also want to read *The NewtonScript Programming Language* either before or concurrently with this book. That book describes the NewtonScript language, which is used throughout the *Newton Programmer's Guide*.

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

Related Books

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

- *Newton Toolkit User's Guide*. This book comes with the Newton Toolkit development environment. It introduces the Newton development environment and shows how to develop applications using Newton Toolkit. You should read this book first if you are a new Newton application developer.

ARENDI-DEFS00003691

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 45 of 200 PageID #: 28689

- *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.

Newton Programmer's Reference CD-ROM

This book is accompanied by a CD-ROM disc that contains the complete text of *Newton Programmer's Reference*. *Newton Programmer's Reference* is the comprehensive reference to the Newton programming interface. It documents all routines, prototypes, data structures, constants, and error codes defined by the Newton system for use by NewtonScript developers.

The companion CD-ROM includes three electronic versions of *Newton Programmer's Reference*. The CD-ROM contains these items, among others:

- The complete *Newton Programmer's Reference* in QuickView format for the Mac OS — the same format used by the *Macintosh Programmer's Toolbox Assistant*. In this format, you can use the extremely fast full-text searching capabilities and ubiquitous hypertext jumps to find reference information quickly.
- The complete *Newton Programmer's Reference* in Windows Help format. This format provides quick and convenient access to the reference information for developers working on Windows platforms.
- The complete *Newton Programmer's Reference* in Adobe Acrobat format. This format provides a fully formatted book with page-numbered table of contents, index, and cross-references. You can print all or portions of the book, and you can also view it online. When viewing online, you can use the indexed search facilities of Adobe Acrobat Reader 2.1 for fast lookup of any information in the book.

The companion CD-ROM also includes an Adobe Acrobat version of this book, *Newton Programmer's Guide*, and a demo version of the Newton Toolkit development environment for the Mac OS.

ARENDI-DEFS00003692

Sample Code

The Newton Toolkit development environment, from Apple Computer, includes many sample code projects. You can examine these samples, learn from them, and experiment with them. These sample code projects illustrate most of the topics covered in this book. They are an invaluable resource for understanding the topics discussed in this book and for making your journey into the world of Newton programming an easier one.

The Newton Developer Technical Support team continually revises the existing samples and creates new sample code. The latest sample code is included each quarter on the Newton Developer CD, which is distributed to all Newton Developer Program members and to subscribers of the Newton monthly mailing. Sample code is updated on the Newton Development side on the World Wide Web (`http://dev.info.apple.com/newton`) shortly after it is released on the Newton Developer CD. For information about how to contact Apple Computer regarding the Newton Developer Program, see the section "Developer Products and Support," on page xlvii.

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

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

Conventions Used in This Book

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

Special Fonts

This book uses the following special fonts:

- **Boldface**. Key terms and concepts appear in boldface on first use. These terms are also defined in the Glossary.
- `Courier typeface`. Code listings, code snippets, and special identifiers in the text such as predefined system frame names, slot names, function names, method names, symbols, and constants are shown in the Courier typeface to distinguish them from regular body text. If you are programming, items that appear in Courier should be typed exactly as shown.

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 47 of 200 PageID #: 28691

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

Tap Versus Click

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

Frame Code

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

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

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

    ViewSetupDoneScript: func()
        :UpdateDisplay(),

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

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

1. On the NTK template palette, find the view class or proto shown in the book. Draw out a view using that template. If the frame shown in the book contains a `_proto` slot, use the corresponding proto from the NTK template palette. If the frame shown in the book contains a `viewClass` slot instead of a `_proto` slot, use the corresponding view class from the NTK template palette.

ARENDI-DEFS00003694

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 48 of 200 PageID #: 28692

2. Edit the `viewBounds` slot to match the values shown in the book.
3. Add each of the other slots you see listed in the frame, setting their values to the values shown in the book. Slots that have values are attribute slots, and those that contain functions are method slots.

Developer Products and Support

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

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

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

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	ORDER.ADC
Internet	order.adc@applelink.apple.com
World Wide Web	http://www.devcatalog.apple.com

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

For Newton-specific information, see the Newton developer World Wide Web page at: `http://dev.info.apple.com/newton`

ARENDI-DEFS00003695

Undocumented System Software Objects

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

C H A P T E R 1

Overview

This chapter describes the general architecture of the Newton system software, which is divided into three levels, as shown in Figure 1-1 (page 1-2).

The lowest level includes the operating system and the low-level communications system. These parts of the system interact directly with the hardware and perform basic operations such as memory management, input and output, and task switching. NewtonScript applications have no direct access to system services at this level.

The middle level consists of system services that NewtonScript applications can directly access and interact with to accomplish tasks. The system provides hundreds of routines that applications can use to take advantage of these services.

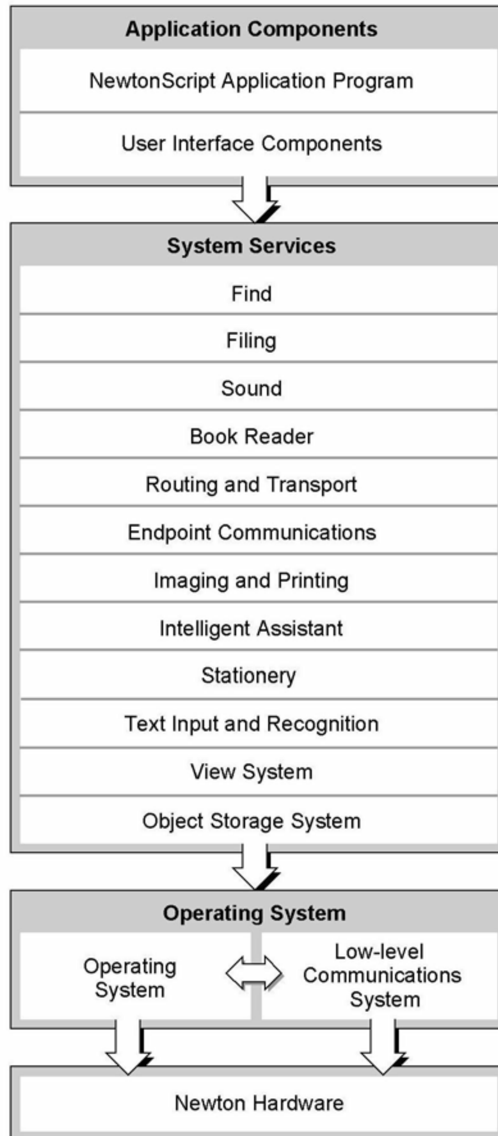
At the highest level are components that applications can use to construct their user interfaces. These reusable components neatly package commonly needed user interface objects such as buttons, lists, tables, input fields, and so on. These components incorporate NewtonScript code that makes use of the system services in the middle level, and that an application can override to customize an object.

Operating System

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.

A significant part of the operating system is concerned with low-level communication functions. The communication subsystem runs as a separate task. It manages the hardware communication resources available in the system. These include serial, fax modem, AppleTalk networking, and infrared. The communication architecture is extensible, and new communication protocols can be installed and removed at run time, to support additional services and external devices that may be added.

Figure 1-1 System software overview



Another operating system task of interest is the Inker. The Inker task is responsible for gathering and displaying input from the electronic tablet overlaying the screen when the user writes on the Newton. The Inker exists as a separate task so that the Newton can gather input and display electronic ink at the same time as other operations are occurring.

All Newton applications, including the recognition system, built-in applications, and applications you develop, run in a single operating system task, called the Application task.

NewtonScript applications have no direct access to the operating system level of software. Access to certain low-level resources, such as communications, is provided by higher-level interfaces.

Memory

It is helpful to understand the use of random access memory (RAM) in the system, since this resource is shared by the operating system and all applications. Newton RAM is divided into separate domains, or sections, that have controlled access. Each domain has its own heap and stack. It is important to know about three of these domains:

- The operating system domain. This portion of memory is reserved for use by the operating system. Only operating system tasks have access to this domain.
- The storage domain. This portion of memory is reserved for permanent, protected storage of user data. All soups, which store the data, reside here, as well as any packages that have been downloaded into the Newton. To protect the data in the storage domain from inadvertent damage, it can only be accessed through the object storage system interface, described in Chapter 11, “Data Storage and Retrieval.” If the user adds a PCMCIA card containing RAM, Flash RAM, or read-only memory (ROM) devices, the memory on the card is used to extend the size of the storage domain.

  The storage domain occupies special persistent memory; that is, this memory is maintained even during a system reset. This protects user data, system software updates, and downloaded packages from being lost during system resets. The used and free space in the storage domain is reported to the user in the Memory Info slip in the Extras Drawer.
- The application domain. This portion of memory is used for dynamic memory allocation by the handwriting recognizers and all Newton applications. A fixed part of this domain is allocated to the NewtonScript heap. The NewtonScript heap is important because most objects allocated as a result of your NewtonScript application code are allocated from the NewtonScript heap. These are the only memory objects to which you have direct access. The NewtonScript heap is shared by all applications.

ARENDI-DEFS00003699

The system performs automatic memory management of the NewtonScript heap. You don't need to worry about memory allocation or disposal in an application. The system automatically allocates memory when you create a new object in NewtonScript. When references to an object no longer exist, it is freed during the next garbage collection cycle. The system performs garbage collection automatically when it needs additional memory.

The Newton operating system optimizes use of memory by using compression. Various parts of memory are compressed and decompressed dynamically and transparently, as needed. This occurs at a low level, and applications don't need to be concerned with these operations.

Packages

A **package** is the unit in which software is installed on and removed from the Newton. Packages can combine multiple pieces of software into a single unit. The operating system manages packages, which can be installed from PCMCIA cards, from a serial connection to a desktop computer, a network connection, or via modem. When a package comes into the Newton system, the system automatically opens it and dispatches its parts to appropriate handlers in the system.

A package consists of a header, which contains the package name and other information, and one or more **parts,** which contain the software. Parts can include applications, communication drivers, fonts, and system updates (system software code loaded into RAM that overrides or extends the built-in ROM code). A package can also export objects for use by other packages in the system, and can import (use) objects that are exported by other packages.

Packages are optionally stored compressed on the Newton. Compressed packages occupy much less space (roughly half of their uncompressed size), but applications in compressed packages may execute somewhat slower and use slightly more battery power, because of the extra work required to decompress them when they are executed.

For more information about packages, refer to Chapter 11, "Data Storage and Retrieval."

System Services

The Newton system software contains hundreds of routines organized into functional groups of services. Your application can use these routines to accomplish specific tasks such as opening and closing views, storing and retrieving data, playing sounds, drawing shapes, and so on. This section includes brief descriptions of the more important system services with which your application will need to interact. Note that communications services are described in a separate section following this one.

Object Storage System

This system is key to the Newton information architecture. The object storage system provides persistent storage for data.

Newton uses a unified data model. This means that all data stored by all applications uses a common format. Data can easily be shared among different applications, with no translation necessary. This allows seamless integration of applications with each other and with system services.

Data is stored using a database-like model. Objects are stored as **frames,** which are like database records. A frame contains named **slots,** which hold individual pieces of data, like database fields. For example, an address card in the Names application is stored as a frame that contains a slot for each item on the card: name, address, city, state, zip code, phone number, and so on.

Frames are flexible and can represent a wide variety of structures. Slots in a single frame can contain any kind of NewtonScript object, including other frames, and slots can be added or removed from frames dynamically. For a description of NewtonScript objects, refer to *The NewtonScript Programming Language*.

Groups of related frames are stored in **soups,** which are like databases. For example, all the address cards used by the Names application are stored in the Names soup, and all the notes on the Notepad are stored in the Notes soup. All the frames stored in a soup need not contain identical slots. For example, some frames representing address cards may contain a phone number slot and others may not.

Soups are automatically indexed, and applications can create additional indexes on slots that will be used as keys to find data items. You retrieve items from a soup by performing a query on the soup. Queries can be based on an index value or can search for a string, and can include additional constraints. A query results in a **cursor**—an object representing a position in the set of soup entries that satisfy the query. The cursor can be moved back and forth, and can return the current entry.

Soups are stored in physical repositories, called **stores.** Stores are akin to disk volumes on personal computers. The Newton always has at least one store—the internal store. Additional stores reside on PCMCIA cards.

The object storage system interface seamlessly merges soups that have the same name on internal and external stores in a **union** soup. This is a virtual soup that provides an interface similar to a real soup. For example, some of the address cards on a Newton may be stored in the internal Names soup and some may be stored in another Names soup on a PCMCIA card. When the card is installed, those names in the card soup are automatically merged with the existing internal names so the user, or an application, need not do any extra work to access those additional names. When the card is removed, the names simply disappear from the card file union soup.

The object storage system is optimized for small chunks of data and is designed to operate in tight memory constraints. Soups are compressed, and retrieved entries are not allocated on the NewtonScript heap until a slot in the entry is accessed.

You can find information about the object storage system interface in Chapter 11, "Data Storage and Retrieval."

View System

Views are the basic building blocks of most applications. A view is simply a rectangular area mapped onto the screen. Nearly every individual visual item you see on the screen is a view. Views display information to the user in the form of text and graphics, and the user interacts with views by tapping them, writing in them, dragging them, and so on. A view is defined by a frame that contains slots specifying view attributes such as its bounds, fill color, alignment relative to other views, and so on.

The view system is what you work with to manipulate views. There are routines to open, close, animate, scroll, highlight, and lay out views, to name just a few operations you can do. For basic information about views and descriptions of all the routines you can use to interact with the view system, refer to Chapter 3, "Views."

An application consists of a collection of views all working together. Each application has an **application base view** from which all other views in the application typically descend hierarchically. In turn, the base view of each application installed in the Newton descends from the system **root view.** (Think of the hierarchy as a tree structure turned upside down, with the root at the top.) Thus, each application base view is a **child** of the root view. We call a view in which child views exist the **parent** view of those child views. Note that occasionally, an application may also include views that don't descend from the base view but are themselves children of the root view.

The system includes several different primitive **view classes** from which all views are ultimately constructed. Each of these view classes has inherently different behavior and attributes. For example, there are view classes for views that contain text, shapes, pictures, keyboards, analog gauges, and so on.

As an application executes, its view frames receive messages from the system and exchange messages with each other. System messages provide an opportunity for a view to respond appropriately to particular events that are occurring. For example, the view system performs default initialization operations when a view is opened. It also sends the view a `ViewSetupFormScript` message. If the view includes a method to handle this message, it can perform its own initialization operations in that method. Handling system messages in your application is optional since the system performs default behaviors for most events.

Text Input and Recognition

The Newton recognition system uses a sophisticated multiple-recognizer architecture. There are recognizers for text, shapes, and gestures, which can be simultaneously active (this is application-dependent). An arbitrator examines the results from simultaneously active recognizers and returns the recognition match that has the highest confidence.

Recognition is modeless. That is, the user does not need to put the system in a special mode or use a special dialog box in order to write, but can write in any input field at any time.

The text recognizers can handle printed, cursive, or mixed handwriting. They can work together with built-in dictionaries to choose words that accurately match what the user has written. The user can also add new words to a personal dictionary.

Depending on whether or not a text handwriting recognizer is enabled, users can enter handwritten text that is recognized or not. Unrecognized text is known as ink text. Ink text can still be manipulated like recognized text—words can be inserted, deleted, moved around, and reformatted—and ink words can be intermixed with recognized words in a single paragraph. Ink words can be recognized later using the deferred recognition capability of the system.

The shape recognizer recognizes both simple and complex geometric objects, cleaning up rough drawings into shapes with straight lines and smooth curves. The shape recognizer also recognizes symmetry, using that property, if present, to help it recognize and display objects.

For each view in an application, you can specify which recognizers are enabled and how they are configured. For example, the text recognizer can be set to recognize only names, or names and phone numbers, or only words in a custom dictionary that you supply, among other choices.

Most recognition events are handled automatically by the system view classes, so you don't need to do anything in your application to handle recognition events, unless you want to do something special. For example, when a user writes a word in a text view, that view automatically passes the strokes to the recognizer, accepts the recognized word back, and displays the word. In addition, the view automatically handles corrections for you. The user can double-tap a word to pop up a list of other possible matches for it, or to use the keyboard to correct it.

For information on methods for accepting and working with text input, refer to Chapter 8, "Text and Ink Input and Display." For information on controlling recognition in views and working with dictionaries, refer to Chapter 9, "Recognition."

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 57 of 200 PageID #: 28701

Stationery

Stationery is a capability of the system that allows applications to be extended by other developers. The word “stationery” refers to the capability of having different kinds of data within a single application (such as plain notes and outlines in the Notepad) and/or to the capability of having different ways of viewing the same data (such as the Card and All Info views in the Names file). An application that supports stationery can be extended either by adding a new type of data to it (for example, adding recipe cards to the Notepad), or by adding a new type of viewer for existing data (a new way of viewing Names file entries or a new print format, for example).

To support stationery, an application must register with the system a frame, called a data definition, that describes the data with which it works. The different data definitions available to an application are listed on the pop-up menu attached to the New button. In addition, an application must register one or more view definitions, which describe how the data is to be viewed or printed. View definitions can include simple read-only views, editor-type views, or print formats. The different view definitions available in an application (not including print formats) are listed on the pop-up menu attached to the Show button.

Stationery is well integrated into the NewtApp framework, so if you use that framework for your application, using stationery is easy. The printing architecture also uses stationery, so all application print formats are registered as a kind of stationery.

For more information about using stationery, see Chapter 5, “Stationery.”

Intelligent Assistant

A key part of the Newton information architecture is the Intelligent Assistant. The Intelligent Assistant is a system service that attempts to complete actions for the user according to deductions it makes about the task that the user is currently performing. The Assistant is always instantly available to the user through the Assist button, yet remains nonintrusive.

The Assistant knows how to complete several built-in tasks; they are Scheduling (adding meetings), Finding, Reminding (adding To Do items), Mailing, Faxing, Printing, Calling, and getting time information from the Time Zones map. Each of these tasks has several synonyms; for example, the user can write “call,” “phone,” “ring,” or “dial” to make a phone call.

Applications can add new tasks so that the Assistant supports their special capabilities and services. The Newton unified data model makes it possible for the Assistant to access data stored by any application, thus allowing the Assistant to be well integrated in the system.

For details on using the Intelligent Assistant and integrating support for it into your application, see Chapter 18, “Intelligent Assistant.”

ARENDI-DEFS00003704

Imaging and Printing

At the operating system level, the Newton imaging and printing software is based on an object-oriented, device-independent imaging model. The imaging model is monochrome since the current Newton screen is a black-and-white screen.

NewtonScript application programs don't call low-level imaging routines directly to do drawing or image manipulation. In fact, most drawing is handled for applications by the user interface components they incorporate, or when they call other routines that display information. However, there is a versatile set of high-level drawing routines that you can call directly to create and draw shapes, pictures, bitmaps, and text. When drawing, you can vary the pen thickness, pen pattern, fill pattern, and other attributes. For details on drawing, refer to Chapter 13, "Drawing and Graphics."

The Newton text imaging facility supports Unicode directly, so the system can be easily localized to display languages using different script systems. The system is extensible, so it's possible to add additional fonts, font engines, and printer drivers.

The high-level interface to printing on the Newton uses a model identical to that used for views. Essentially, you design a special kind of view called a print format to specify how printed information is to be laid out on the page. Print formats use a unique view template that automatically adjusts its size to the page size of the printer chosen by the user. When the user prints, the system handles all the details of rendering the views on the printer according to the layout you specified.

The Newton offers the feature of deferred printing. The user can print even though he or she is not connected to a printer at the moment. An object describing the print job is stored in the Newton Out Box application, and when a printer is connected later, the user can then select that print job for printing. Again, this feature is handled automatically by the system and requires no additional application programming work.

For information on how to add printing capabilities to an application, refer to Chapter 21, "Routing Interface."

Sound

The Newton includes a monophonic speaker and can play sounds sampled at rates up to 22 kHz. You can attach sounds to particular events associated with a view, such as showing it, hiding it, and scrolling it. You can also use sound routines to play sounds synchronously or asynchronously at any other time.

Newton can serve as a phone dialer by dialing phone numbers through the speaker. The dialing tones are built into the system ROM, along with several other sounds that can be used in applications.

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 59 of 200 PageID #: 28703

Besides the sounds that are built into the system ROM, you can import external sound resources into an application through the Newton Toolkit development environment.

For information about using sound in an application, see Chapter 14, "Sound."

Book Reader

Book Reader is a system service that displays interactive digital books on the Newton screen. Digital books can include multiple-font text, bitmap and vector graphics, and on-screen controls for content navigation. Newton digital books allow the user to scroll pages, mark pages with bookmarks, access data directly by page number or subject, mark up pages using digital ink, and perform text searches. Of course, the user can copy and paste text from digital books, as well as print text and graphics from them.

Newton Press and Newton Book Maker are two different development tools that you use to create digital books for the Newton. Nonprogrammers can easily create books using Newton Press. Newton Book Maker is a more sophisticated tool that uses a text-based command language allowing you to provide additional services to the user or exercise greater control over page layout. Also, using Book Maker, you can attach data, methods, and view templates to book content to provide customized behavior or work with the Intelligent Assistant.

The Book Maker application can also be used to create on-line help for an application. The installation of on-line help in an application package requires some rudimentary NewtonScript programming ability; however, nonprogrammers can create on-line help content, again using only a word processor and some basic Book Maker commands.

Refer to the book *Newton Book Maker User's Guide* for information on Book Reader, the Book Maker command language, and the use of Newton Toolkit to create digital book packages and on-line help. Refer to the *Newton Press User's Guide* for information on using Newton Press.

Find

Find is a system service that allows users to search one or all applications in the system for occurrences of a particular string. Alternatively, the user can search for data time-stamped before or after a specified date. When the search is completed, the Find service displays an overview list of items found that match the search criteria. The user can tap an item in the list and the system opens the corresponding application and displays the data containing the selected string. Users access the Find service by tapping the Find button.

ARENDI-DEFS00003706

If you want to allow the user to search for data stored by your application, you need to implement certain methods that respond to find messages sent by the system. You'll need to supply one method that searches your application's soup(s) for data and returns the results in a particular format, and another method that locates and displays the found data in your application if the user taps on it in the find overview. The system software includes routines and templates that help you support find in your application. For details on supporting the Find service, refer to Chapter 16, "Find."

Filing

The Filing service allows users to tag soup-based data in your application with labels used to store, retrieve, and display the data by category. The labels used to tag entries are represented as folders in the user interface; however, no true hierarchical filing exists—the tagged entries still reside in the soup. Users access the filing service through a standard user interface element called the file folder button, which looks like a small file folder.

When the user chooses a category for an item, the system notifies your application that filing has changed. Your application must perform the appropriate application-specific tasks and redraw the current view, providing to the user the illusion that the item has been placed in a folder. When the user chooses to display data from a category other than the currently displayed one, the system also notifies your application, which must retrieve and display data in the selected category.

The system software includes templates that help your application implement the filing button and the selector that allows the user to choose which category of data to display. Your application must provide methods that respond to filing messages sent by the system in response to user actions such as filing an item, changing the category of items to display, and changing the list of filing categories. For details on supporting the Filing service, refer to Chapter 15, "Filing."

Communications Services

This section provides an overview of the communications services in Newton system software 2.0.

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

ARENDI-DEFS00003707

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 61 of 200 PageID #: 28705

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

The Newton communications architecture is illustrated in Figure 1-2.

Figure 1-2 Communications architecture

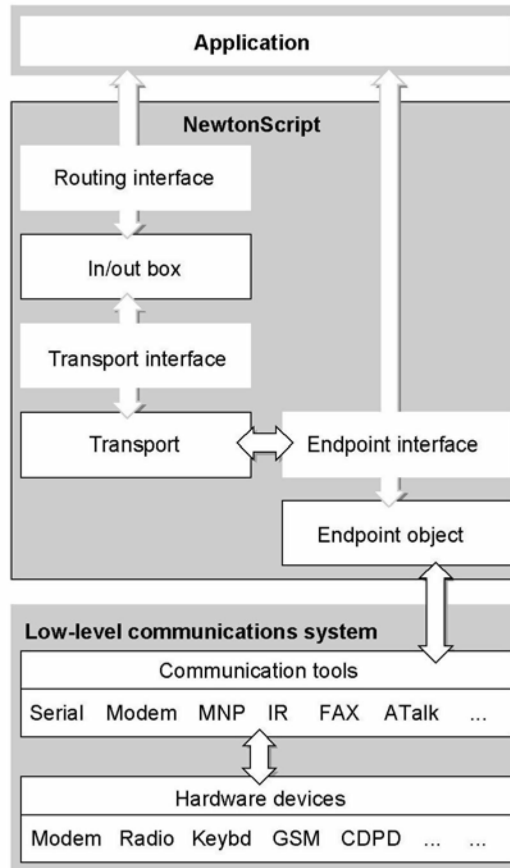


Figure 1-2 shows four unique communications interfaces available for you to use:

- routing interface
- endpoint interface

ARENDI-DEFS00003708

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 62 of 200 PageID #: 28706

- transport interface
- communication tool interface

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

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

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

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

NewtonScript Application Communications

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

Typically, an application uses only one of these types of communication, but sometimes both are needed. These two types of communication are described in the following sections.

Routing Through the In/Out Box

The routing interface is the highest-level NewtonScript communication interface. The routing interface allows an application to communicate with the In/Out Box and lets users send data and receive data from outside the system. In applications, users access routing services through a standard user interface element called the Action button, which looks like a small envelope. Users access the In/Out Box application through icons in the Newton Extras Drawer. The In/Out Box provides a common user interface for all incoming and outgoing data in the system.

The routing interface is best suited for user-controlled messaging and transaction-based communications. For example, the Newton built-in applications use this interface for e-mail, beaming, printing, and faxing. Outgoing items can be stored in the Out Box until a physical connection is available, when the user can choose to transmit the items, or they can be sent immediately. Incoming items are received in the In Box, where the user can get new mail and beamed items, for example.

For information on the routing interface, refer to Chapter 21, "Routing Interface."

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

If you are writing an application that takes advantage of only the transports currently installed in the Newton system, you need to use only the routing

ARENDI-DEFS00003709

interface. You need to use the transport or endpoint interfaces only when writing custom communication tools.

Endpoint Interface

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

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

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

The basic endpoint interface is described in Chapter 23, "Endpoint Interface."

Low-Level Communications

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

These two interfaces are described in the following sections.

Transport Interface

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

When the user taps the Action button in an application, the Action picker appears. Built-in transports available on the Action picker include printing, faxing, and beaming. Any new transports that you provide are added to this list.

For more information, refer to Chapter 22, "Transport Interface."

Communication Tool Interface

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

The built-in communication tools include:

- Synchronous and asynchronous serial
- Fax/data modem (data is V.34 with MNP/V.42 and fax is V.17 with Class 1, 2, and 2.0 support)
- Point-to-point infrared—called beaming (Sharp 9600 and Apple IR-enhanced protocols)
- AppleTalk ADSP protocol

For information about configuring the built-in communication tools through the endpoint interface, refer to Chapter 24, “Built-in Communications Tools.”

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

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

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

Application Components

At the highest level of system software are dozens of components that applications can use to construct their user interfaces and other nonvisible objects. These reusable components neatly package commonly needed user interface objects such as buttons, lists, tables, input fields, and so on. These components incorporate NewtonScript code that makes use of other system services, and which an application can override to customize an object.

These components are built into the Newton ROM. When you reference one of these components in your application, the code isn’t copied into your application—your application simply makes a reference to the component in the ROM. This conserves memory at run time and still allows your application to easily override any attributes of the built-in component. Because you can build much of your

ARENDI-DEFS00003711

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 65 of 200 PageID #: 28709

application using these components, Newton applications tend to be much smaller in size than similar applications on desktop computers.

A simple example of how you can construct much of an application using components is illustrated in Figure 1-3. This simple application accepts names and phone numbers and saves them into a soup. It was constructed in just a few minutes using three different components.

The application base view is implemented by a single component that includes the title bar at the top, the status bar at the bottom, the clock and the close box, and the outer frame of the application. The Name and Phone input lines are each created from the same component that implements a simple text input line; the two buttons are created from the same button component. The only code you must write to make this application fully functional is to make the buttons perform their actions. That is, make the Clear button clear the input lines and make the Save button get the text from the input lines and save it to a soup.

Figure 1-3 Using components

The components available for use by applications are shown on the layout palette in Newton Toolkit. These components are known as **protos,** which is short for "prototypes." In addition to the built-in components, Newton Toolkit lets you create your own reusable components, called user protos. The various built-in components are documented throughout the book in the chapter containing information related to each proto. For example, text input protos are described in Chapter 8, "Text and Ink Input and Display;" protos that implement pickers and lists are described in Chapter 6, "Pickers, Pop-up Views, and Overviews;" and protos that implement controls and other miscellaneous protos are described in Chapter 7, "Controls and Other Protos."

ARENDI-DEFS00003712

The NewtApp framework consists of a special collection of protos that are designed to be used together in a layered hierarchy to build a complete application. For more information about the NewtApp protos, refer to Chapter 4, “NewtApp Applications.”

Using System Software

Most of the routines and application components that comprise the Newton system software reside in ROM, provided in special chips contained in every Newton device. When your application calls a system routine, the operating system executes the appropriate code contained in ROM.

This is different from traditional programming environments where system software routines are accessed by linking a subroutine library with the application code. That approach results in much larger applications and makes it harder to provide new features and fix bugs in the system software.

The ROM-based model used in the Newton provides a simple way for the operating system to substitute the code that is executed in response to a particular system software routine, or to substitute an application component. Instead of executing the ROM-based code for some routine, the operating system might choose to load some substitute code into RAM; when your application calls the routine, the operating system intercepts the call and executes the RAM-based code.

RAM-based code that substitutes for ROM-based code is called a system update. Newton system updates are stored in the storage memory domain, which is persistent storage.

Besides application components, the Newton ROM contains many other objects such as fonts, sounds, pictures, and strings that might be useful to applications. Applications can access these objects by using special references called **magic pointers.** Magic pointers provide a mechanism for code written in a development system separate from the Newton to reference objects in the Newton ROM or in other packages. Magic pointer references are resolved at run time by the operating system, which substitutes the actual address of the ROM or package object for the magic pointer reference.

Magic pointers are constants defined in Newton Toolkit. For example, the names of all the application components, or protos, are actually magic pointer constants. You can find a list of all the ROM magic pointer constants in the Newton 2.0 Defs file, included with Newton Toolkit.

The NewtonScript Language

You write Newton applications in NewtonScript, a dynamic object-oriented language developed especially for the Newton platform, though the language is highly portable. NewtonScript is designed to operate within tight memory constraints, so is well suited to small hand-held devices like Newton.

NewtonScript is used to define, access, and manipulate objects in the Newton system. NewtonScript frame objects provide the basis for object-oriented features such as inheritance and message sending.

Newton Toolkit normally compiles NewtonScript into byte codes. The Newton system software contains a byte code interpreter that interprets the byte codes at run time. This has two advantages: byte codes are much smaller than native code, and Newton applications are easily portable to other processors, since the interpreter is portable. Newton Toolkit can also compile NewtonScript into native code. Native code occupies much more space than interpreted code, but in certain circumstances it can execute much faster.

For a complete reference to NewtonScript, refer to *The NewtonScript Programming Language*.

What's New in Newton 2.0

Version 2.0 of the Newton System Software brings many changes to all areas. Some programming interfaces have been extended; others have been completely replaced with new interfaces; and still other interfaces are brand new. For those readers familiar with previous versions of system software, this section gives a brief overview of what is new and what has changed in Newton 2.0, focusing on those programming interfaces that you will be most interested in as a developer.

NewtApp

NewtApp is a new application framework designed to help you build a complete, full-featured Newton application more quickly. The NewtApp framework consists of a collection of protos that are designed to be used together in a layered hierarchy. The NewtApp framework links together soup-based data with the display and editing of that data in an application. For many types of applications, using the NewtApp framework can significantly reduce development time because the protos automatically manage many routine programming tasks. For example, some of the tasks the protos support include filing, finding, routing, scrolling, displaying an overview, and soup management.

The NewtApp framework is not suited for all Newton applications. If your application stores data as individual entries in a soup, displays that data to the user in views, and allows the user to edit some or all of the data, then it is a potential candidate for using the NewtApp framework. NewtApp is well suited to "classic" form-based applications. Some of the built-in applications constructed using the NewtApp framework include the Notepad and the Names file.

Stationery

Stationery is a new capability of Newton 2.0 that allows applications to be extended by other developers. If your application supports stationery, then it can be extended by others. Similarly, you can extend another developer's application that supports stationery. You should also note that the printing architecture now uses stationery, so all application print formats are registered as a kind of stationery.

Stationery is a powerful capability that makes applications much more extensible than in the past. Stationery is also well integrated into the NewtApp framework, so if you use that framework for your application, using stationery is easy. For more information about stationery, see the section "Stationery" (page 1-8).

Views

New features for the view system include a drag-and-drop interface that allows you to provide users with a drag-and-drop capability between views. There are hooks to provide for custom feedback to the user during the drag process and to handle copying or moving the item.

The system now includes the capability for the user to view the display in portrait or landscape orientation, so the screen orientation can be changed (rotated) at any time. Applications can support this new capability by supporting the new `ReorientToScreen` message, which the system uses to alert all applications to re-layout their views.

Several new view methods provide features such as bringing a view to the front or sending it to the back, automatically sizing buttons, finding the view bounds including the view frame, and displaying modal dialogs to the user.

There is a new message, `ViewPostQuitScript`, that is sent to a view (only on request) when it is closing, after all of the view's child views have been destroyed. This allows you to do additional clean-up, if necessary. And, you'll be pleased to know that the order in which child views receive the `ViewQuitScript` message is now well defined: it is top-down.

Additionally, there are some new `viewJustify` constants that allow you to specify that a view is sized proportionally to its sibling or parent view, horizontally and/or vertically.

Protos

There are many new protos supplied in the new system ROM. There are new pop-up button pickers, map-type pickers, and several new time, date, and duration pickers. There are new protos that support the display of overviews and lists based on soup entries. There are new protos that support the input of rich strings (strings that contain either recognized characters or ink text). There are a variety of new scroller protos. There is an integrated set of protos designed to make it easy for you to display status messages to the user during lengthy or complex operations.

Generic list pickers, available in system 1.0, have been extended to support bitmap items that can be hit-tested as two-dimensional grids. For example, a phone keypad can be included as a single item in a picker. Additionally, list pickers can now scroll if all the items can't fit on the screen.

Data Storage

There are many enhancements to the data storage system for system software 2.0. General soup performance is significantly improved. A tagging mechanism for soup entries makes changing folders much faster for the user. You can use the tagging mechanism to greatly speed access to subsets of entries in a soup. Queries support more features, including the use of multiple slot indexes, and the query interface is cleaner. Entry aliases make it easy to save unique references to soup entries for fast access later without holding onto the actual entry.

A new construct, the virtual binary object, supports the creation and manipulation of very large objects that could not be accommodated in the NewtonScript heap. There is a new, improved soup change-notification mechanism that gives applications more control over notification and how they respond to soup changes. More precise information about exactly what changed is communicated to applications. Soup data can now be built directly into packages in the form of a store part. Additionally, packages can contain protos and other objects that can be exported through magic pointer references, and applications can import such objects from available packages.

Text Input

The main change to text input involves the use of ink text. The user can choose to leave written text unrecognized and still manipulate the text by inserting, deleting, reformatting, and moving the words around, just as with recognized text. Ink words and recognized words can be intermixed within a single paragraph. A new string format, called a rich string, handles both ink and recognized text in the same string.

There are new protos, `protoRichInputLine` and `protoRichLabelInputLine`, that you can use in your application to allow users to enter ink text in fields. In addition, the view classes `clEditView` and

clParagraphView now support ink text. There are several new functions that allow you to manipulate and convert between regular strings and rich strings. Other functions provide access to ink and stroke data, allow conversion between strokes, points, and ink, and allow certain kinds of ink and stroke manipulations.

There are several new functions that allow you to access and manipulate the attributes of font specifications, making changing the font attributes of text much easier. A new font called the handwriting font is built in. This font looks similar to handwritten characters and is used throughout the system for all entered text. You should use it for displaying all text the user enters.

The use of on-screen keyboards for text input is also improved. There are new proto buttons that your application can use to give users access to the available keyboards. It's easier to include custom keyboards for your application. Several new methods allow you to track and manage the insertion caret, which the system displays when a keyboard is open. Note also that a real hardware keyboard is available for the Newton system, and users may use it anywhere to enter text. The system automatically supports its use in all text fields.

Graphics and Drawing

Style frames for drawing shapes can now include a custom clipping region other than the whole destination view, and can specify a scaling or offset transformation to apply to the shape being drawn.

Several new functions allow you to create, flip, rotate, and draw into bitmap shapes. Also, you can capture all or part of a view into a bitmap. There are new protos that support the display, manipulation, and annotation of large bitmaps such as received faxes. A new function, InvertRect, inverts a rectangle in a view.

Views of the class clPictureView can now contain graphic shapes in addition to bitmap or picture objects.

System Services

System-supplied Filing services have been extended; applications can now filter the display of items according to the store on which they reside, route items directly to a specified store from the filing slip, and provide their own unique folders. In addition, registration for notification of changes to folder names has been simplified.

Two new global functions can be used to register or unregister an application with the Find service. In addition, Find now maintains its state between uses, performs "date equal" finds, and returns to the user more quickly.

Applications can now register callback functions to be executed when the Newton powers on or off. Applications can register a view to be added to the user preferences roll. Similarly, applications can register a view to be added to the formulas roll.

The implementation of undo has changed to an undo/redo model instead of two levels of undo, so applications must support this new model.

Recognition

Recognition enhancements include the addition of an alternate high-quality recognizer for printed text and significant improvements in the cursive recognizer. While this doesn't directly affect applications, it does significantly improve recognition performance in the system, leading to a better user experience. Other enhancements that make the recognition system much easier to use include a new correction picker, a new punctuation picker, and the caret insertion writing mode (new writing anywhere is inserted at the caret position).

Specific enhancements of interest to developers include the addition of a `recConfig` frame, which allows more flexible and precise control over recognition in individual input views. A new proto, `protoCharEdit`, provides a comb-style entry view in which you can precisely control recognition and restrict entries to match a predefined character template.

Additionally, there are new functions that allow you to pass ink text, strokes, and shapes to the recognizer to implement your own deferred recognition. Detailed recognition corrector information (alternate words and scores) is now available to applications.

Sound

The interface for playing sounds is enhanced in Newton 2.0. In addition to the existing sound functions, there is a new function to play a sound at a particular volume and there is a new `protoSoundChannel` object. The `protoSoundChannel` object encapsulates sounds and methods that operate on them. Using a sound channel object, sound playback is much more flexible—the interface supports starting, stopping, pausing, and playing sounds simultaneously through multiple sound channels.

Built-in Applications

Unlike in previous versions, the built-in applications are all more extensible in version 2.0. The Notepad supports stationery, so you can easily extend it by adding new "paper" types to the New pop-up menu. The Names file also supports stationery, so it's easy to add new card types, new card layout styles, and new data items to existing cards by registering new data definitions and view definitions for the Names application. There's also a method that adds a new card to the Names soup.

ARENDI-DEFS00003718

The Dates application includes a comprehensive interface that gives you the ability to add, find, move, and delete meetings and events. You can get and set various kinds of information related to meetings, and you can create new meeting types for the Dates application. You can programmatically control what day is displayed as the first day of the week, and you can control the display of a week number in the Calendar view.

The To Do List application also includes a new interface that supports creating new to do items, retrieving items for a particular date or range, removing old items, and other operations.

Routing and Transports

The Routing interface is significantly changed in Newton 2.0. The system builds the list of routing actions dynamically, when the user taps the Action button. This allows all applications to take advantage of new transports that are added to the system at any time. Many hooks are provided for your application to perform custom operations at every point during the routing operation. You register routing formats with the system as view definitions. A new function allows you to send items programmatically.

Your application has much more flexibility with incoming items. You can choose to automatically put away items and to receive foreign data (items from different applications or from a non-Newton source).

The Transport interface is entirely new. This interface provides several new protos and functions that allow you to build a custom communication service and make it available to all applications through the Action button and the In/Out Box. Features include a logging capability, a system for displaying progress and status information to the user, support for custom routing slips, and support for transport preferences.

Endpoint Communication

The Endpoint communication interface is new but very similar to the 1.0 interface. There is a new proto, `protoBasicEndpoint`, that encapsulates the connection and provides methods to manage the connection and send and receive data. Additionally, a derivative endpoint, `protoStreamingEndpoint`, provides the capability to send and receive very large frame objects.

Specific enhancements introduced by the new endpoint protos include the ability to handle and identify many more types of data by tagging the data using data forms specified in the `form` slot of an endpoint option. Most endpoint methods can now be called asynchronously, and asynchronous operation is the recommended way to do endpoint-based communication. Support is also included for time-outs and multiple termination sequences. Error handling is improved.

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 73 of 200 PageID #: 28717

There have been significant changes in the handling of binary (raw) data. For input, you can now target a direct data input object, resulting in significantly faster performance. For output, you can specify offsets and lengths, allowing you to send the data in chunks.

Additionally, there is now support for multiple simultaneous communication sessions.

Utilities

Many new utility functions are available in Newton 2.0. There are several new deferred, delayed, and conditional message-sending functions. New array functions provide ways to insert elements, search for elements, and sort arrays. Additionally, there's a new set of functions that operate on sorted arrays using binary search algorithms. New and enhanced string functions support rich strings, perform conditional substring substitution, tokenize strings, and perform case-sensitive string compares. A new group of functions gets, sets, and checks for the existence of global variables and functions.

Books

New Book Reader features include better browser behavior (configurable auto-closing), expanded off-line bookkeeping abilities, persistent bookmarks, the ability to remove bookmarks, and more efficient use of memory.

New interfaces provide additional ways to navigate in books, customize Find behavior, customize bookmarks, and add help books. Book Reader also supports interaction with new system messages related to scrolling, turning pages, installing books, and removing books. Additional interfaces are provided for adding items to the status bar and the Action menu.

ARENDI-DEFS00003720

C H A P T E R 2

Getting Started

This chapter describes where to begin when you're thinking about developing a Newton application. It describes the different kinds of software you can develop and install on the Newton and the advantages and disadvantages of using different application structures.

Additionally, this chapter describes how to create and register your developer signature.

Before you read this chapter, you should be familiar with the information described in Chapter 1, "Overview."

Choosing an Application Structure

When you create an application program for the Newton platform, you can use one of the following basic types of application structures:

- minimal predefined structure, by basing the application on a view class of `clView` or the `protoApp` proto
- highly structured, by basing the application on the NewtApp framework of protos
- highly structured and specialized for text, by building a digital book

Alternatively, you might want to develop software that is not accessed through an icon in the Extras Drawer. For example, you might want to install stationery, a transport, or some other kind of specialized software that does something like creating a soup and then removing itself.

These various approaches to software development are discussed in the following sections.

Minimal Structure

The minimalist approach for designing a Newton application starts with an empty or nearly empty container that provides little or no built-in functionality—thus the "minimalist" name. This approach is best suited for specialized applications that

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 75 of 200 PageID #: 28719

don't follow the "classic" form-based model. For example, some types of applications that might use this approach include games, utilities, calculators, and graphics applications.

The advantage of using the minimalist approach is that it's simple and small. Usually you'd choose this approach because you don't need or want a lot of built-in support from a comprehensive application framework, along with the extra size and overhead that such support brings.

The disadvantage of the minimalist approach is that it doesn't provide any support from built-in features, like the NewtApp framework does. You get just a simple container in which to construct your application.

To construct an application using the minimalist approach, you can use the view class `clView` or the proto `protoApp` as your application base view. The view class `clView` is the bare minimum you can start with. This is the most basic of the primitive view classes. It provides nothing except an empty container. The `protoApp` provides a little bit more, it includes a framed border, a title at the top, and a close box so the user can close it. For details on these objects, see `clView` (page 1-1) and `protoApp` (page 1-2) in *Newton Programmer's Reference*.

Neither of these basic containers provide much built-in functionality. You must add functionality yourself by adding other application components to your application. There are dozens of built-in protos that you can use, or you can create your own protos using NTK. Most of the built-in protos are documented in these two chapters: Chapter 6, "Pickers, Pop-up Views, and Overviews,"and Chapter 7, "Controls and Other Protos." Note also that certain protos in the NewtApp framework can be used outside of a NewtApp application. For information on NewtApp protos, see Chapter 4, "NewtApp Applications."

NewtApp Framework

NewtApp is an application framework that is well suited to "classic" form-based applications. Such applications typically gather and store data in soups, display individual soup entries to users in views, and allow the user to edit some or all of the data. For example, some types of applications that might use NewtApp include surveys and other data gathering applications, personal information managers, and record-keeping applications. Some of the built-in applications constructed using NewtApp include the Notepad, Names file, In/Out Box, Calls, and Time Zones.

The advantage of NewtApp is that it provides a framework of protos designed to help you build a complete, full-featured Newton application more quickly than if you started from scratch. The NewtApp protos are designed to be used together in a layered hierarchy that links together soup-based data with the display and editing of that data in an application. For many types of applications, using the NewtApp framework can significantly reduce development time because the protos

ARENDI-DEFS00003722

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 76 of 200 PageID #: 28720

automatically manage many routine programming tasks. For example, some of the tasks the protos support include filing, finding, routing, scrolling, displaying an overview, and soup management.

The disadvantage of NewtApp is that it is structured to support a particular kind of application—one that allows the creation, editing, and display of soup data. And particularly, it supports applications structured so that there is one data element (card, note, and so on) per soup entry. If your application doesn't lend itself to that structure or doesn't need much of the support that NewtApp provides, then it would be better to use a different approach to application design.

For details on using the NewtApp framework to construct an application, see Chapter 4, "NewtApp Applications."

Digital Books

If you want to develop an application that displays a large amount of text, handles multiple pages, or needs to precisely layout text, you may want to consider making a digital book instead of a traditional application. In fact, if you are dealing with a really large amount of text, like more than a few dozen screens full, then you could make your job much easier by using the digital book development tools.

Digital books are designed to display and manipulate large amounts of text and graphics. Digital books can include all the functionality of an application—they can include views, protos, and methods that are executed as a result of user actions. In fact, you can do almost everything in a digital book that you can do in a more traditional application, except a traditional application doesn't include the text layout abilities.

The advantage of using a digital book structure is that you gain the automatic text layout and display abilities of Book Reader, the built-in digital book reading application. Additionally, the book-making tools are easy to use and allow you to quickly turn large amounts of text and graphics into Newton books with minimal effort.

The disadvantage of using a digital book is that it is designed to support a particular kind of application—one that is like a book. If your application doesn't lend itself to that structure or doesn't need much of the text-handling support that Book Reader provides, then it would be better to use a different approach to application design.

For information on creating digital books using the Book Maker command language and/or incorporating NewtonScript code and objects into digital books, see *Newton Book Maker User's Guide*. For information on creating simpler digital books see *Newton Press User's Guide*.

ARENDI-DEFS00003723

Other Kinds of Software

There are other kinds of software you can develop for the Newton platform that are not accessed by the user through an icon in the Extras drawer. These might include new types of stationery that extend existing applications, new panels for the Preferences or Formulas applications, new routing or print formats, communication transports, and other kinds of invisible applications. Such software is installed in a kind of part called an auto part (because its part code is `auto`).

You can also install a special kind of auto part that is automatically removed after it is installed. The `InstallScript` function in the auto part is executed, and then it is removed. (For more information about the `InstallScript` function, see the section "Package Loading, Activation, and Deactivation" beginning on page 2-4.) This kind of auto part is useful to execute some code on the Newton, for example, to create a soup, and then to remove the code. This could be used to write an installer application that installs just a portion of the data supplied with an application. For example, you might have a game or some other application that uses various data sets, and the installer could let the user choose which data sets to install (as soups) to save storage space.

Any changes made by an automatically removed auto part are lost when the Newton is reset, except for changes made to soups, which are persistent.

For additional information about creating auto parts and other kinds of parts such as font, dictionary, and store parts, refer to *Newton Toolkit User's Guide*.

Package Loading, Activation, and Deactivation

When a package is first loaded onto the Newton store from some external source, the system executes the `DoNotInstallScript` function in each frame part in the package. This function gives the parts in the package a chance to prevent installation of the package. If the package is not prevented from being installed, next it is activated.

When a package containing an application or auto part is activated on the Newton, the system executes a special function in those parts: the `InstallScript` function. A package is normally activated as a result of installing it—by inserting a storage card containing it, by moving it from one store to another, by downloading it from a desktop computer, by downloading it via modem or some other communication device, or by soft resetting the Newton device. Packages can also exist in an inactive state on a Newton store, and such a package can be activated by the user at a later time.

When a package is deactivated, the system executes another special function in each of the application and auto parts in the package: the `RemoveScript` function. A package is normally deactivated when the card it resides on is removed,

ARENDI-DEFS00003724

when it is moved to another store (it is deactivated then reactivated), or when the user deletes the application icon in the Extras Drawer. Packages can also be deactivated without removing them from the store.

When a package is removed as a result of the user deleting it from the Extras Drawer, the system also executes the `DeletionScript` function in each of the package frame parts. This occurs before the `RemoveScript` function is executed.

The following sections describe how to use these functions.

Loading

The `DoNotInstallScript` function in a package part is executed when a package is first loaded onto a Newton store from some external source (this does not include inserting a storage card containing the package or moving it between stores). This function applies to all types of frame parts (for example, not store parts).

This method gives the parts in the package a chance to prevent installation of the entire package. If any of the package parts returns a non-`nil` value from this function, the package is not installed and is discarded.

You should provide the user with some kind of feedback if package installation is prevented, rather than silently failing. For example, to ensure that a package is installed only on the internal store you could write a `DoNotInstallScript` function like the following:

```
func()
begin
   if GetStores()[0] <> GetVBOStore(ObjectPkgRef('foo')) then
   begin
   GetRoot():Notify(kNotifyAlert, kAppName,
               "This package was not installed.
               It can be installed only onto the internal store.");
   true;
   end;
end
```

Activation

The `InstallScript` function in a package part is executed when an application or auto part is activated on the Newton or whenever the Newton is reset.

This function lets you perform any special installation operations that you need to do, any initialization, and any registration for system services.

ARENDI-DEFS00003725

IMPORTANT
Any changes that you make to the system in the `InstallScript` function must be reversed in the `RemoveScript` function. For example, if you register your application for certain system services or install print formats, stationery, or other objects in the system, you must reverse these changes and remove or unregister these objects in the `RemoveScript` function. If you fail to do this, such changes cannot be removed by the user, and if your application is on a card, they won't be able to remove the card without getting a warning message to put the card back. ▲

Only applications and auto parts use the `InstallScript` function. Note that the `InstallScript` function takes one extra argument when used for an auto part. Applications built using the NewtApp framework require special `InstallScript` and `RemoveScript` functions. For details, see Chapter 4, "NewtApp Applications."

Deactivation

The `RemoveScript` function in a package part is executed when an application or auto part is deactivated.

This function lets you perform any special deinstallation operations that you need to do, any clean-up, and any unregistration for system services that you registered for in the `InstallScript` function.

Note that automatically removed auto parts do not use the `RemoveScript` function since such auto parts are removed immediately after the `InstallScript` is executed—the `RemoveScript` is not executed.

In addition to the `RemoveScript` function, another function, `DeletionScript`, is executed when the user removes a package by deleting it from the Extras Drawer. This function applies to all types of frame parts, and is actually executed before the `RemoveScript` function.

The `DeletionScript` function is optional. It lets you do different clean-up based on the assumption that the user is permanently deleting a package, rather than simply ejecting the card on which it happens to reside. For example, in the `DeletionScript` function, you might want to delete all the soups created by the application—checking with the user, of course, before performing such an irreversible operation.

ARENDI-DEFS00003726

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 80 of 200 PageID #: 28724

Effects of System Resets on Application Data

Two kinds of reset operations—hard resets and soft resets—can occur on Newton devices. All data in working RAM (the NewtonScript heap and the operating system domain) is erased when a hard or soft reset occurs.

Unless a hard reset occurs, soups remain in RAM until they are removed explicitly, even if the Newton device is powered down. Soups are not affected by soft resets, as they are stored in the protected storage domain. The remainder of this section describes reset operations in more detail and suggests ways to ensure that your application can deal with resets appropriately.

A hard reset occurs at least once in the life of any Newton device—when it is initially powered on. The **hard reset** returns all internal RAM to a known state: all soups are erased, all caches are purged, all application packages are erased from the internal store, application RAM is reinitialized, the NewtonScript heap is reinitialized, and the operating system restarts itself. It's the end (or beginning) of the world as your application knows it.

Note
Data on external stores is not affected by a hard reset. ◆

A hard reset is initiated only in hardware by the user. Extreme precautions have been taken to ensure that this action is deliberate. On the MessagePad, the user must simultaneously manipulate the power and reset switches to initiate the hardware reset. After this is accomplished, the hardware reset displays two dialog boxes warning the user that all data is about to be erased; the user must confirm this action in both dialog boxes before the hard reset takes place.

It is extremely unlikely that misbehaving application software would cause a hard reset. However, a state similar to hardware reset may be achieved if the battery that backs up internal RAM is removed or fails completely.

It's advisable to test your application's ability to install itself and run on a system that has been initialized with a hard reset. The exact sequence of steps required to hard reset a Newton device is documented in its user guide.

Newton devices may also perform a soft reset operation. A **soft reset** erases all data stored by applications in the NewtonScript heap, for example all data stored in slots in views or other frames in memory. A soft reset also reinitializes the data storage system frames cache, while leaving soup data intact. Any frames in the cache are lost, such as new or modified entries that have not been written back to the soup. A soft reset can be initiated in software by the operating system or from hardware by the user.

ARENDI-DEFS00003727

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 81 of 200 PageID #: 28725

When the operating system cannot obtain enough memory to complete a requested operation, it may display a dialog box advising the user to reset the Newton device. The user can tap the Reset button displayed in the dialog box to reset the system, or can tap the Cancel button and continue working.

The user may also initiate a soft reset by pressing a hardware button provided for this purpose. This button is designed to prevent its accidental use. On the MessagePad, for example, it is recessed inside the battery compartment and must be pressed with the Newton pen or similarly-shaped instrument.

A soft reset may also be caused by misbehaving application software. One way to minimize the occurrence of unexpected resets is to utilize exception-handling code where appropriate.

The only way applications can minimize the consequences of a soft reset is to be prepared for one to happen at any time. Applications need to store all permanent data in a soup and write changed entries back to the soup as soon as is feasible.

It's advisable to test your application's ability to recover from a soft reset. The exact sequence of steps required to soft-reset a particular Newton device is documented in its user guide.

Flow of Control

The Newton system is an event-driven, object-oriented system. Code is executed in response to messages sent to objects (for example, views). Messages are sent as a result of user events, such as a tap on the screen, or internal system events, such as an idle loop triggering. The flow of control in a typical application begins when the user taps on the application icon in the Extras Drawer. As a result of this event, the system performs several actions such as reading the values of certain slots in your application base view and sending a particular sequence of messages to it.

For a detailed discussion of the flow of control and the order of execution when an application "starts up," see the section "View Instantiation" beginning on page 3-26.

Using Memory

The tightly-constrained Newton environment requires that applications avoid wasting memory space on unused references. As soon as possible, applications should set to `nil` any object reference that is no longer needed, thereby allowing the system to reclaim the memory used by that object. For example, when an application closes, it needs to clean up after itself as much as possible, removing its references to soups, entries, cursors, and any other objects. This means you should set to `nil` any application base view slots that refer to objects in RAM.

ARENDI-DEFS00003728

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 82 of 200 PageID #: 28726

IMPORTANT
If you don't remove references to unused soups, entries, cursors, and other objects, the objects will not be garbage collected, reducing the amount of RAM available to the system and other applications. ▲

Localization

If your application displays strings, and you want your application to run on localized Newton products, you should consider localizing your application. This involves translating strings to other languages and using other formats for dates, times, and monetary values.

There are some features of NTK that make string localization simple, allowing you to define the language at compile time to build versions in different languages without changing the source files. Refer to *Newton Toolkit User's Guide* for more information.

For details on localizing an application, see Chapter 20, "Localizing Newton Applications."

Developer Signature Guidelines

To avoid name conflicts with other Newton application, you need to register a single developer signature with Newton DTS. You can then use this signature as the basis for creating unique application symbols, soup names and other global symbols and strings according to the guidelines described in this section.

Signature

A **signature** is an arbitrary sequence of approximately 4 to 10 characters. Any characters except colons (:) and vertical bars(|) can be used in a signature. Case is not significant.

Like a handwritten signature, the developer signature uniquely identifies a Newton application developer. The most important characteristic of a signature is that it is unique to a single developer, which is why Newton DTS maintains a registry of developer signatures. Once you have registered a signature with Newton DTS it is yours, and will not be assigned to any other developer.

ARENDI-DEFS00003729

Examples of valid signatures include

```
NEWTONDTS
Joe's Cool Apps
1NEWTON2DTS
What the #$*? SW
```

How to Register

To register your signature, you need to provide the following information to the Newton Development Information Group at Apple.

```
Company Name:
Contact Person:
Mailing Address:
Phone:
Email Address:
Desired Signature 1st choice:
Desired Signature 2nd choice:
```

Send this information to the e-mail address

`NEWTONDEV@applelink.apple.com`

or send it via US Mail to:

NewtonSysOp
c/o: Apple Computer, Inc.
1 Infinite Loop, M/S: 305-2A
Cupertino, CA 95014
USA

Application Name

The **application name** is the string displayed under your application's icon in the Extras drawer. Because it is a string, any characters are allowed.

This name does not need to be unique, because the system does not use it to identify the application. For example, it is possible for there to be two applications named `Chess` on the market. The application name is used only to identify the application to the user. If there were in fact two applications named `Chess` installed on the same Newton device, hopefully the user could distinguish one from the other by some other means, perhaps by the display of different icons in the Extras drawer.

Examples of valid application names include

```
Llama
Good Form
2 Fun 4 U
Chess
```

Note
It's recommended that you keep your application names short so that they don't crowd the names of other applications in the Extras drawer. ◆

Application Symbol

The **application symbol** is created by concatenating the application name, a colon (:), and your registered developer signature. This symbol is not normally visible to the end user. It is used to uniquely identify an application in the system. Because application symbols contain a colon (:), they must be enclosed by vertical bars (|) where they appear explicitly in NewtonScript code.

Examples of valid application symbols include:

```
'|Llama:NEWTONDTS|
'|2 Fun 4 U:Joe's Cool Apps|
```

You specify the application symbol in the Output Settings dialog of NTK. At the beginning of a project build, NTK 1.5 or newer defines a constant for your project with the name `kAppSymbol` and sets it to the symbol you specify as the application symbol. Use of this constant throughout your code makes it easier to maintain your code.

At the end of the project build, if you've not created a slot with the name `appSymbol` in the application base view of your project, NTK creates such a slot and places in it the application symbol. If the slot exists already, NTK doesn't overwrite it.

Package Name

The **package name** is usually a string version of the application symbol. The package name may be visible to the user if no application name is provided. Package names are limited to 26 characters, so this places a practical limit on the combined length of application names and signatures.

ARENDI-DEFS00003731

Summary

View Classes and Protos

clView

```
aView := {
viewClass: clView, // base view class
viewBounds: boundsFrame, // location and size
viewJustify: integer, // viewJustify flags
viewFlags: integer, // viewFlags flags
viewFormat: integer, // viewFormat flags
...
}
```

protoApp

```
anApp := {
_proto: protoApp, // proto application
title: string, // application name
viewBounds: boundsFrame, // location and size
viewJustify: integer, // viewJustify flags
viewFlags: integer, // viewFlags flags
viewFormat: integer, // viewFormat flags
declareSelf: 'base, // do not change
...
}
```

Functions

Application-Defined Functions

```
InstallScript(partFrame) // for application parts
InstallScript(partFrame, removeFrame) // for auto parts
DeletionScript()
DoNotInstallScript()
RemoveScript(frame)
```


ARENDI-DEFS00003732

C H A P T E R 3

Views

This chapter provides the basic information you need to know about views and how to use them in your application.

You should start with this chapter if you are creating an application for Newton devices, as views are the basic building blocks for most applications. Before reading this chapter, you should be familiar with the information in *Newton Toolkit User's Guide* and *The NewtonScript Programming Language*.

This chapter introduces you to views and related items, describing

- views, templates, the view coordinate system, and the instantiation process for creating a view
- common tasks, such as creating a template, redrawing a view, creating special view effects, and optimizing a view's performance
- view constants, methods, and functions

About Views

Views are the basic building blocks of most applications. Nearly every individual visual item you see on the screen—for example, a radio button, or a checkbox—is a view, and there may even be views that are not visible. Views display information to the user in the form of text and graphics, and the user interacts with views by tapping them, writing in them, dragging them, and so on.

Different types of views have inherently different behavior, and you can include your own methods in views to further enhance their behavior. The primitive view classes provided in the Newton system are described in detail in Table 2-2 (page 2-4) in the *Newton Programmer's Reference*.

You create or lay out a view with the Newton Toolkit's graphic editor. The Newton Toolkit creates a template; that is, a data object that describes how the view will look and act on the Newton. Views are then created from templates when the application runs on the Newton.

ARENDI-DEFS00003733

This section provides detailed conceptual information on views and other items related to views. Specifically, it covers the following:

- templates and views and how they relate to each other
- the coordinate system used in placing views
- components used to define views
- application-defined methods that the system sends to views
- the programmatic process used to create a view
- new functions, methods, and messages added for 2.0 as well as modifications to existing view code

Templates

A **template** is a frame containing a description of an object. (In this chapter the objects referred to are views that can appear on the screen.) Templates contain data descriptions of such items as fields for the user to write into, graphic objects, buttons, and other interactive objects used to collect and display information. Additionally, templates can include **methods,** which are functions that give the view behavior.

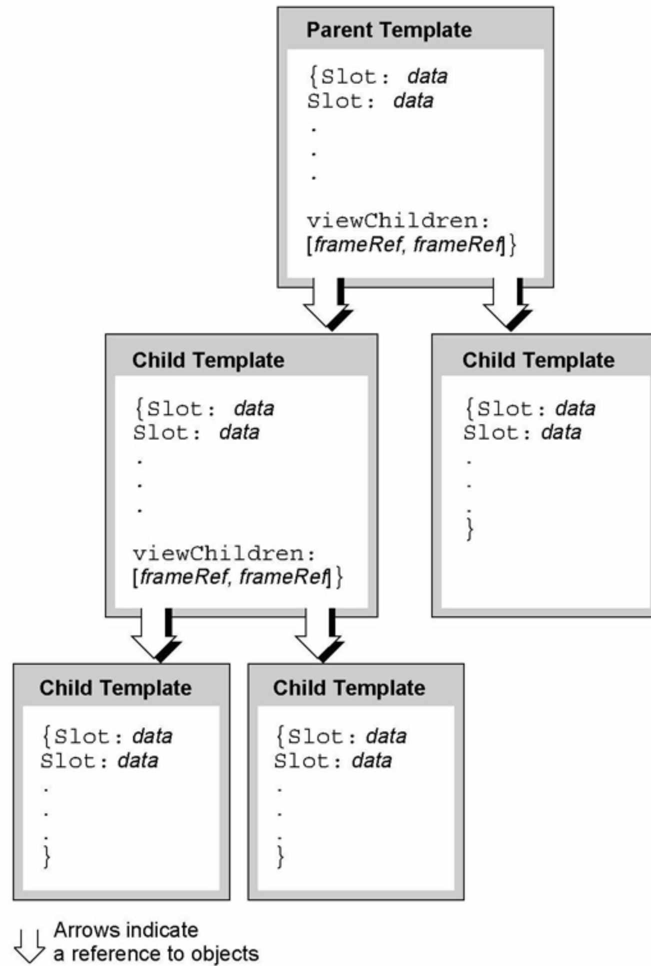
Note
A template can also describe nongraphic objects like communication objects. Such objects have no visual representation and exist only as logical objects. ◆

An application exists as a collection of templates, not just a single template. There is a **parent** template that defines the application window and its most basic features. From this parent template springs a hierarchical collection of **child** templates, each defining a small piece of the larger whole. Each graphic object, button, text field, and so on is defined by a separate template. Each child template exists within the context of its parent template and inherits characteristics from its parent template, though it can override these inherited characteristics.

Within the Newton object system, a template for a view exists as a special kind of **frame;** that is, a frame containing or inheriting a particular group of **slots** (`viewClass`, `viewBounds`, `viewFlags`, and some other optional slots) that define the template's class, dimensions, appearance, and other characteristics. Templates are no different from any other frames, except that they contain or inherit these particular slots (in addition to others). For more information about frames, slots, and the NewtonScript language, see *The NewtonScript Programming Language*.

Figure 3-1 shows a collection of template frames that might make up an application. The frame at the top represents the highest-level parent template. Each template that has children contains a `viewChildren` (or `stepChildren`) slot whose value is an array of references to its child templates.

Figure 3-1 Template hierarchy



ARENDI-DEFS00003735

Views

A template is a data description of an object. A **view** is the visual representation of the object that is created when the template is instantiated. The system reads the stored description in the template and creates a view on the screen—for example, a framed rectangle containing a title.

Besides the graphic representation you see on the screen, a view consists of a memory object (a frame) that contains a reference to its template and also contains transient data used to create the graphic object. Any changes to view data that occur during run time are stored in the view, not in its template. This is an important point—after an application has started up (that is, once the views are instantiated from their templates), all changes to slots occur in the view; the template is never changed.

This distinction between templates and views with respect to changing slot values occurs because of the NewtonScript inheritance mechanism. During run time, templates, containing static data, are prototypes for views, which contain dynamic data. To understand this concept, it is imperative that you have a thorough understanding of the inheritance mechanism as described in *The NewtonScript Programming Language*.

You can think of a template as a computer program stored on a disk. When the program starts up, the disk copy (the template) serves as a template; it is copied into dynamic memory, where it begins execution. Any changes to program variables and data occur in the copy of the program in memory (the view), not in the original disk version.

However, the Newton system diverges from this metaphor in that the view is not actually a copy of the template. To save RAM use, the view contains only a reference to the template. Operations involving the reading of data are directed by reference to the template if the data is not first found in the view. In operations in which data is written or changed, the data is written into the view.

Because views are transient and data is disposed of when the view is closed, any data written into a view that needs to be saved permanently must be saved elsewhere before the view disappears.

A view is linked with its template through a `_proto` slot in the view. The value of this slot is a reference to the template. Through this reference, the view can access slots in its template. Templates may themselves contain `_proto` slots which reference other templates, called **protos,** on which they are built.

Views are also linked to other views in a parent-child relationship. Each view contains a `_parent` slot whose value is a reference to its parent view; that is, the view that encloses it. The top-level parent view of your application is called the

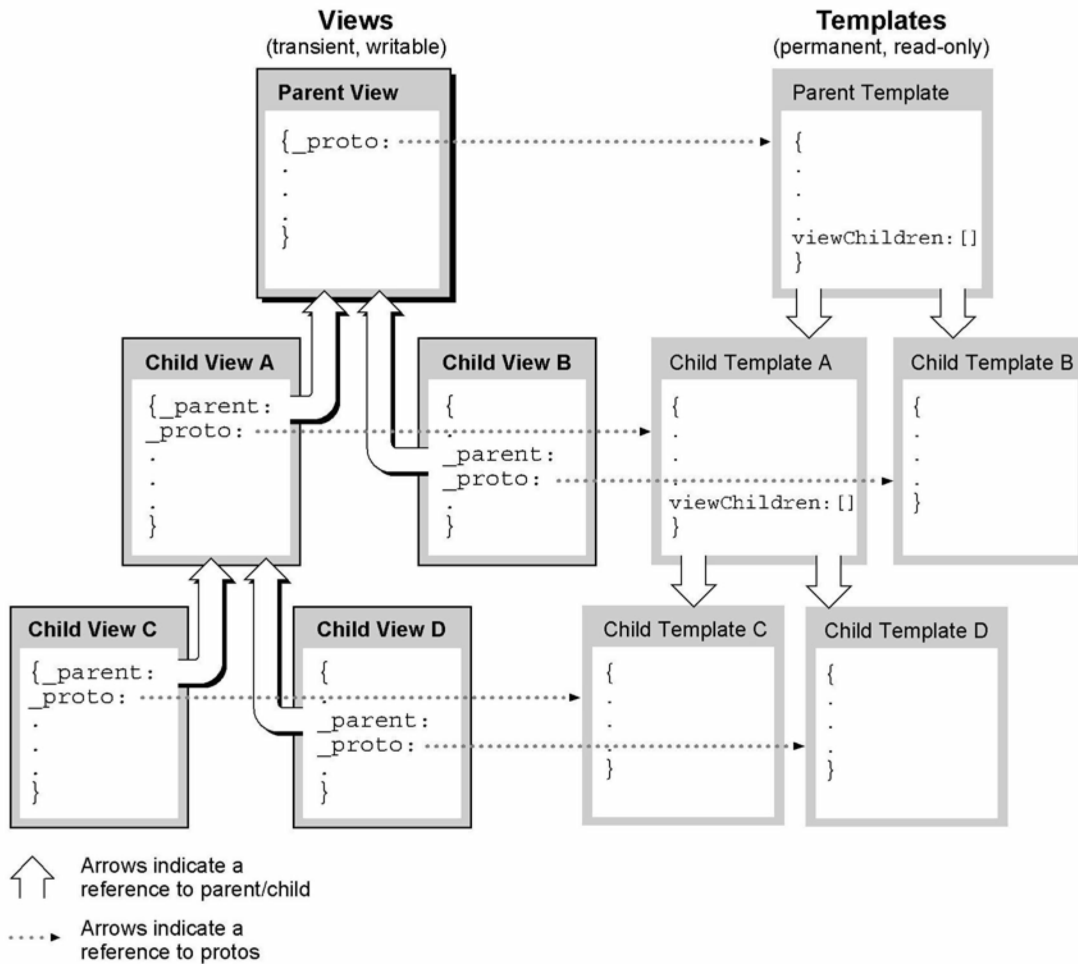
ARENDI-DEFS00003736

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 90 of 200 PageID #: 28734

application base view. (Think of the view hierarchy as a tree structure in which the tree is turned upside down with its root at the top. The top-level parent view is the root view.)

Figure 3-2 shows the set of views instantiated from the templates shown in Figure 3-1. Note that this example is simplified in that it shows a separate template for each view. In practice, multiple views often share a single template. Also, this example doesn't show templates that are built on other protos.

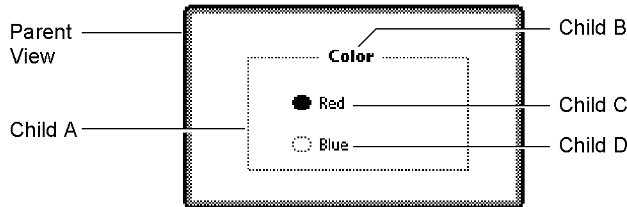
Figure 3-2 View hierarchy



ARENDI-DEFS00003737

Figure 3-3 shows an example of what this view hierarchy might represent on the screen.

Figure 3-3 Screen representation of view hierarchy



The application base view of each application exists as a child of the system **root view.** The root view is essentially the blank screen that exists before any other views are drawn. It is the ancestor of all other views that are instantiated.

Coordinate System

The view coordinate system is a two-dimensional plane. The (0, 0) **origin** point of the plane is assigned to the upper-left corner of the Newton screen, and coordinate values increase to the right and (unlike a Cartesian plane) down. Any pixel on the screen can be specified by a vertical coordinate and a horizontal coordinate. Figure 3-4 (page 3-7) illustrates the view system coordinate plane.

Views are defined by rectangular areas that are usually subsets of the screen. The origin of a view is usually its upper-left corner, though the origin can be changed. The coordinates of a view are relative to the origin of its parent view—they are not screen coordinates.

It is helpful to conceptualize the coordinate plane as a two-dimensional grid. The intersection of a horizontal and vertical grid line marks a point on the coordinate plane.

Note the distinction between points on the coordinate grid and pixels, the dots that make up a visible image on the screen. Figure 3-5 illustrates the relationship between the two: the pixel is down and to the right of the point by which it is addressed.

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 92 of 200 PageID #: 28736

Figure 3-4 View system coordinate plane

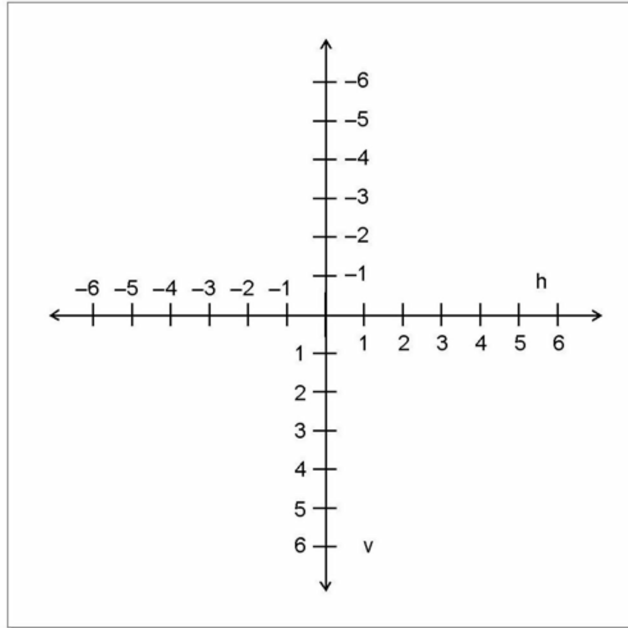
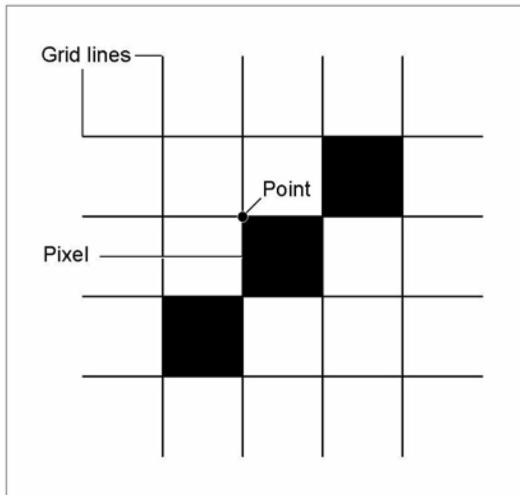


Figure 3-5 Points and pixels



ARENDI-DEFS00003739

As the grid lines are infinitely thin, so a point is infinitely small. Pixels, by contrast, lie *between* the lines of the coordinate grid, not at their intersections.

This relationship gives them a definite physical extent, so that they can be seen on the screen.

Defining View Characteristics

A template that describes a view is stored as a frame that has slots for view characteristics. Here is a NewtonScript example of a template that describes a view:

```
{viewClass: clView,
viewBounds: RelBounds( 20, 50, 94, 142 ),
viewFlags: vNoFlags,
viewFormat:vfFillWhite+vfFrameBlack+vfPen(1),
viewJustify: vjCenterH,
viewFont: simpleFont10,
declareSelf: 'base,
debug: "dialer",
};
```

Briefly, the syntax for defining a frame is:

```
{slotName: slotValue,
 slotName: slotValue,
...};
```

where *slotName* is the name of a slot, and *slotValue* is the value of a slot. For more details on NewtonScript syntax, refer to *The NewtonScript Programming Language*.

Frames serving as view templates have slots that define the following kinds of view characteristics:

Class
: The `viewClass` slot defines the class of graphic object from which the view is constructed.

Behavior
: The `viewFlags` slot defines other primary view behaviors and controls recognition behavior.

Location, size, and alignment
: The `viewBounds` and `viewJustify` slots define the location, size, and alignment of the view and its contents.

Appearance
: The `viewFormat` slot defines the frame and fill characteristics. The `viewFillPattern` and `viewFramePattern` slots control custom patterns. Transfer modes used in drawing the view are controlled by the `viewTransferMode` slot.

Opening and closing animation effects	The `viewEffect` slot defines an animation to be performed when the view is displayed or hidden.
Other attributes	Some other slots define view characteristics such as font, copy protection, and so on.
Inheritance links	The `_proto`, `_parent`, `viewChildren`, and `stepChildren` slots contain links to a view's template, parent view, and child views.

These different categories of view characteristics are described in the following sections.

Class

The `viewClass` slot defines the view class. This information is used by the system when creating a view from its template. The view class describes the type of graphic object to be used to display the data described in the template. The view classes built into the system serve as the primitive building blocks from which all visible objects are constructed. The view classes are listed and described in Table 2-2 (page 2-4) in the *Newton Programmer's Reference*.

Behavior

The `viewFlags` slot defines behavioral attributes of a view other than those that are derived from the view class. Each attribute is represented by a constant defined as a bit flag. Multiple attributes are specified by adding them together, like this:

```
vVisible+vFramed
```

Note that in the NTK viewFlags editor, multiple attributes are specified simply by checking the appropriate boxes.

Some of the `viewFlags` constants are listed and described in Table 2-4 (page 2-11) in the *Newton Programmer's Reference*. There are also several additional constants you can specify in the `viewFlags` slot that control what kinds of pen input (taps, strokes, words, letters, numbers, and so on) are recognized and handled by the view. These other constants are described in "Recognition" (page 9-1).

View behavior is also controlled through methods in the view that handle system messages. As an application executes, its views receive messages from the system, triggered by various events, usually the result of a user action. Views can handle system messages by having methods that are named after the messages. You control the behavior of views by providing such methods and including code that operates on the receiving view or other views.

For a detailed description of the messages that views can receive, and information on how to handle them, see "Application-Defined Methods" (page 3-26)."

Handling Pen Input

The use of the `vClickable viewFlags` constant to control pen input is important to understand, so it is worth covering here, even though it is discussed in more detail in "Recognition" (page 9-1). The `vClickable` flag must be set for a view to receive input. If this flag is not set for a view, that view cannot accept any pen input.

If you have a view whose `vClickable` flag is not set, pen events, such as a tap, will "fall through" that view and be registered in a background view that does accept pen input. This can cause unexpected results if you are not careful. You can prevent pen events from registering in the wrong view by setting the `vClickable` flag for a view and providing a `ViewClickScript` method in the view that returns non-`nil`. This causes the view to capture all pen input within itself, instead of letting it "fall through" to a different view. If you want to capture pen events in a view but still prevent input (and electronic ink), do not specify any other recognition flags besides `vClickable`.

If you want strokes or gestures but want to prevent clicks from falling through up the parent chain, return the symbol `'skip.` This symbol tells the view system not to allow the stroke to be processed by the parent chain, but instead allows the stroke to be processed by the view itself for recognition behavior.

Several other `viewFlags` constants are used to control and constrain the recognition of text, the recognition of shapes, the use of dictionaries, and other input-related features of views. For more information, refer to "Recognition" (page 9-1).

Location, Size, and Alignment

The location and size of a view are specified in the `viewBounds` slot of the view template. The `viewJustify` slot affects the location of a view relative to other views. The `viewJustify` slot also controls how text and pictures within the view are aligned and limits how much text can appear in the view (one line, one word, and so on).

The `viewOriginX` and `viewOriginY` slots control the offset of child views within a view.

View Bounds

The `viewBounds` slot defines the size and location of the view on the screen. The value of the `viewBounds` slot is a frame that contains four slots giving the view coordinates (all distances are in pixels). For example:

```
{left: leftValue,
 top: topValue,
 right: rightValue,
 bottom: bottomValue
}
```

leftValue The distance from the left origin of the parent view to the left edge of the view.

topValue The distance from the top origin of the parent view to the top edge of the view.

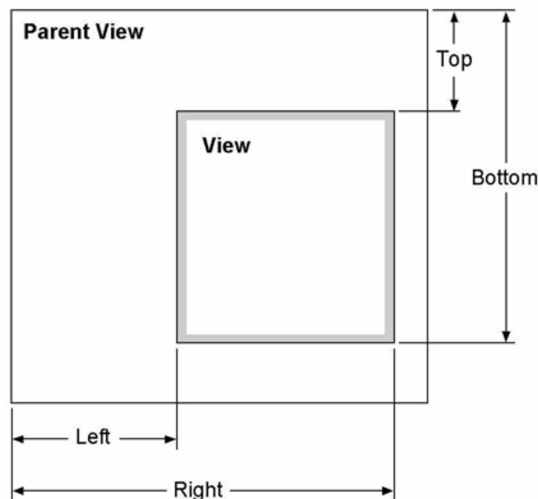
rightValue The distance from the left origin of the parent view to the right edge of the view.

bottomValue The distance from the top origin of the parent view to the bottom edge of the view.

Note
The values in the `viewBounds` frame are interpreted as described here only if the view alignment is set to the default values. Otherwise, the view alignment setting changes the way `viewBounds` values are used. For more information, see "View Alignment" (page 3-13). ◆

As shown in Figure 3-6, all coordinates are relative to a view's parent, they are not actual screen coordinates.

Figure 3-6 Bounds parameters



When you are using the Newton Toolkit (NTK) to lay out views for your application, the `viewBounds` slot is set automatically when you drag out a view in the layout window. If you are writing code in which you need to specify a `viewBounds` slot, you can use one of the global functions such as `SetBounds` or `RelBounds`, which are described in "Finding the Bounds of Views" (page 3-39).

View Size Relative to Parent Size

A view is normally entirely enclosed by its parent view. You shouldn't create a view whose bounds extend outside its parent's bounds. If you do create such a view, for example containing a picture that you want to show just part of, you need to set the `vClipping` flag in the `viewFlags` slot of the parent view.

If you do not set the `vClipping` flag for the parent view, the behavior is unpredictable. The portions of the view outside the parent's bounds may or may not draw properly. All pen input is clipped to the parent's bounds.

Note that the base views of all applications (all root view children, in fact) are automatically clipped, whether or not the `vClipping` flag is set.

If your application base view is very small and you need to create a larger floating child view, for example, a slip, you should use the `BuildContext` function. This function creates a special view that is a child of the root view. To open the view, you send the `Open` message to it.

Using Screen-Relative Bounds

Newton is a family of products with varying screen sizes. If you want your application to be compatible with a variety of individual Newton products, you should design your application so that it sizes itself dynamically (that is, at run time), accounting for the size of the screen on which it is running, which could be smaller or larger than the original Newton MessagePad screen.

You may want to dynamically size the base view of your application so that it changes for different screen sizes, or you may want it to remain a fixed size on all platforms. In the latter case, you should still check the actual screen size at run time to make sure there is enough room for your application.

You can use the global function `GetAppParams` to check the size of the screen at run time. This function returns a frame containing the coordinates of the drawable area of the screen, as well as other information (see "Utility Functions Reference" (page 23-1) in the *Newton Programmer's Reference* for a description). The frame returned looks like this:

```
{appAreaLeft: 0,
 appAreaTop: 0,
 appAreaWidth: 240,
 appAreaHeight: 320,
...}
```

The following example shows how to use the `ViewSetupFormScript` method in your application base view to make the application a fixed size, but no larger than the size of the screen:

```
viewSetupFormScript: func()
   begin
   local b := GetAppParams();
   self.viewbounds := RelBounds(
      b.appAreaLeft,
      b.appAreaTop,
      min(200, b.appAreaWidth),    // 200 pixels wide max
      min(300, b.appAreaHeight)); // 300 pixels high max
   end
```

Don't blindly size your application to the full extents of the screen. This might look odd if your application runs on a system with a much larger screen.

Do include a border around your application base view. That way, if the application runs on a screen that is larger than the size of your application, the user will be able to clearly see its boundaries.

The important point is to correctly size the application base view. Child views are positioned relative to the application base view. If you have a dynamically sizing application base view, make sure that the child views also are sized dynamically, so that they are laid out correctly no matter how the dimensions of the base view change. You can ensure correct layout by using parent-relative and sibling-relative view alignment, as explained in the next section, "View Alignment."

One additional consideration you should note is that on a larger screen, it may be possible for the user to move applications around. You should not rely on the top-left coordinate of your application base view being fixed. To prevent this from happening check your application's current location when you work with global coordinates. To do this, send the `GlobalBox` message to your application base view.

View Alignment

The `viewJustify` slot is used to set the view alignment and is closely linked in its usage and effects with the `viewBounds` slot.

The `viewJustify` slot specifies how text and graphics are aligned within the view and how the bounds of the view are aligned relative to its parent or sibling views. (**Sibling** views are child views that have a common parent view.)

In the `viewJustify` slot, you can specify one or more alignment attributes, which are represented by constants defined as bit flags. You can specify one alignment attribute from each of the following groups:

- horizontal alignment of view contents (applies to views of class `clParagraphView` and `clPictureView` only)
- vertical alignment of view contents (applies to views of class `clParagraphView` and `clPictureView` only)

- horizontal alignment of the view relative to its parent or sibling view
- vertical alignment of the view relative to its parent or sibling view
- text limits

For example, you could specify these alignment attributes for a button view that has its text centered within the view and is placed relative to its parent and sibling views:

```
vjCenterH+vjCenterV+vjSiblingRightH+vjParentBottomV+oneLineOnly
```

If you don't specify an attribute from a group, the default attribute for that group is used.

The view alignment attributes and the defaults are listed and described in Table 3-1. The effects of these attributes are illustrated in Figure 3-7, following the table.

Sibling setting are not used if the view has not previous setting, instead the parent settings are used.

Table 3-1 `viewJustify` constants

Constant	Value	Description
Horizontal alignment of view contents		
`vjLeftH`	0	Left alignment (default).
`vjCenterH`	2	Center alignment (default for `clPictureView` only).
`vjRightH`	1	Right alignment.
`vjFullH`	3	Stretches the view contents to fill the entire view width.
Vertical alignment of view contents[1]		
`vjTopV`	0	Top alignment (default).
`vjCenterV`	4	Center alignment (default for `clPictureView` only).
`vjBottomV`	8	Bottom alignment.
`vjFullV`	12	For views of the `clPictureView` class only; stretches the picture to fill the entire view height.
Horizontal alignment of the view relative to its parent or sibling view[2]		
`vjParentLeftH`	0	The left and right view bounds are relative to the parent's left side (default).

continued

Table 3-1 `viewJustify` constants (continued)

Constant	Value	Description
`vjParentCenterH`	16	The difference between the left and right view bounds is used as the width of the view. If you specify zero for left, the view is centered in the parent view. If you specify any other number for left, the view is offset by that much from a centered position (for example, specifying left = 10 and right = width+10 offsets the view 10 pixels to the right from a centered position).
`vjParentRightH`	32	The left and right view bounds are relative to the parent's right side, and will usually be negative.
`vjParentFullH`	48	The left bounds value is used as an offset from the left edge of the parent and the right bounds value as an offset from the right edge of the parent (for example, specifying left = 10 and right = –10 leaves a 10-pixel margin on each side).
`vjSiblingNoH`	0	(Default) Do not use sibling horizontal alignment.
`vjSiblingLeftH`	2048	The left and right view bounds are relative to the sibling's left side.
`vjSiblingCenterH`	512	The difference between the left and right view bounds is used as the width of the view. If you specify zero for left, the view is centered in relation to the sibling view. If you specify any other number for left, the view is offset by that much from a centered position (for example, specifying left = 10 and right = width+10 offsets the view 10 pixels to the right from a centered position).
`vjSiblingRightH`	1024	The left and right view bounds are relative to the sibling's right side.
`vjSiblingFullH`	1536	The left bounds value is used as an offset from the left edge of the sibling and the right bounds value as an offset from the right edge of the sibling (for example, specifying left = 10 and right = –10 indents the view 10 pixels on each side relative to its sibling).
Vertical alignment of the view relative to its parent or sibling view[3]		
`vjParentTopV`	0	The top and bottom view bounds are relative to the parent's top side (default).

continued

Table 3-1 `viewJustify` constants (continued)

Constant	Value	Description
`vjParentCenterV`	64	The difference between the top and bottom view bounds is used as the height of the view. If you specify zero for top, the view is centered in the parent view. If you specify any other number for top, the view is offset by that much from a centered position (for example, specifying top = –10 and bottom = height–10 offsets the view 10 pixels above a centered position).
`vjParentBottomV`	128	The top and bottom view bounds are relative to the parent's bottom side.
`vjParentFullV`	192	The top bounds value is used as an offset from the top edge of the parent and the bottom bounds value as an offset from the bottom edge of the parent (for example, specifying top = 10 and bottom = –10 leaves a 10-pixel margin on both the top and the bottom).
`vjSiblingNoV`	0	(Default) Do not use sibling vertical alignment.
`vjSiblingTopV`	16384	The top and bottom view bounds are relative to the sibling's top side.
`vjSiblingCenterV`	4096	The difference between the top and bottom view bounds is used as the height of the view. If you specify zero for top, the view is centered in relation to the sibling view. If you specify any other number for top, the view is offset by that much from a centered position (for example, specifying top = –10 and bottom = height–10 offsets the view 10 pixels above a centered position).
`vjSiblingBottomV`	8192	The top and bottom view bounds are relative to the sibling's bottom side.
`vjSiblingFullV`	12288	The top bounds value is used as an offset from the top edge of the sibling and the bottom bounds value as an offset from the bottom edge of the sibling (for example, specifying top = 10 and bottom = –10 indents the view 10 pixels on both the top and the bottom sides relative to its sibling).

continued

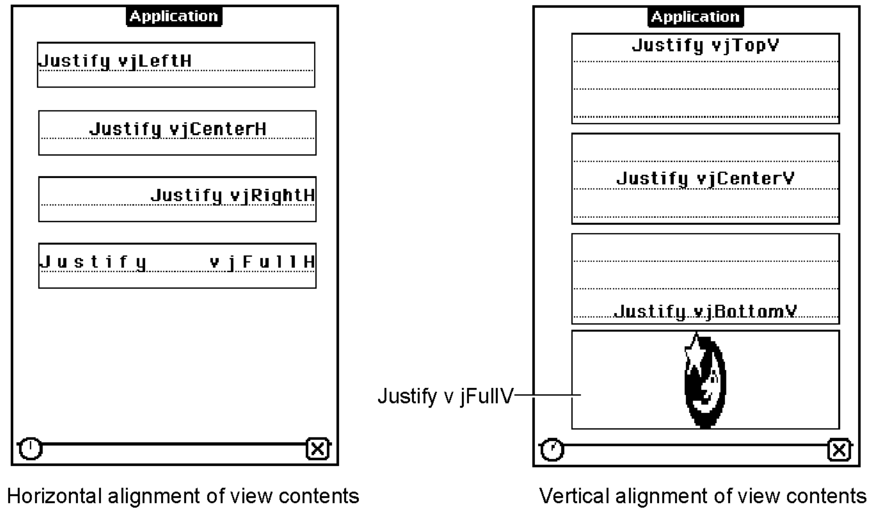
Table 3-1 viewJustify constants (continued)

Constant	Value	Description
Text limits		
noLineLimits	0	(Default) No limits, text wraps to next line.
oneLineOnly	8388608	Allows only a single line of text, with no wrapping.
oneWordOnly	16777216	Allows only a single word. (If the user writes another word, it replaces the first.)
Indicate that a bounds value is a ratio		
vjNoRatio	0	(Default) Do not use proportional alignment.
vjLeftRatio	67108864	The value of the slot viewBounds.left is interpreted as a percentage of the width of the parent or sibling view to which this view is horizontally justified.
vjRightRatio	134217728	The value of the slot viewBounds.right is interpreted as a percentage of the width of the parent or sibling view to which this view is horizontally justified.
vjTopRatio	268435456	The value of the slot viewBounds.top is interpreted as a percentage of the height of the parent or sibling view to which this view is vertically justified.
vjBottomRatio	–536870912	The value of the slot viewBounds.bottom is interpreted as a percentage of the height of the parent or sibling view to which this view is vertically justified.
vjParentAnchored	256	The view is anchored at its location in its parent view, even if the origin of the parent view is changed. Other sibling views will be offset, but not child views with this flag set.

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

ARENDI-DEFS00003749

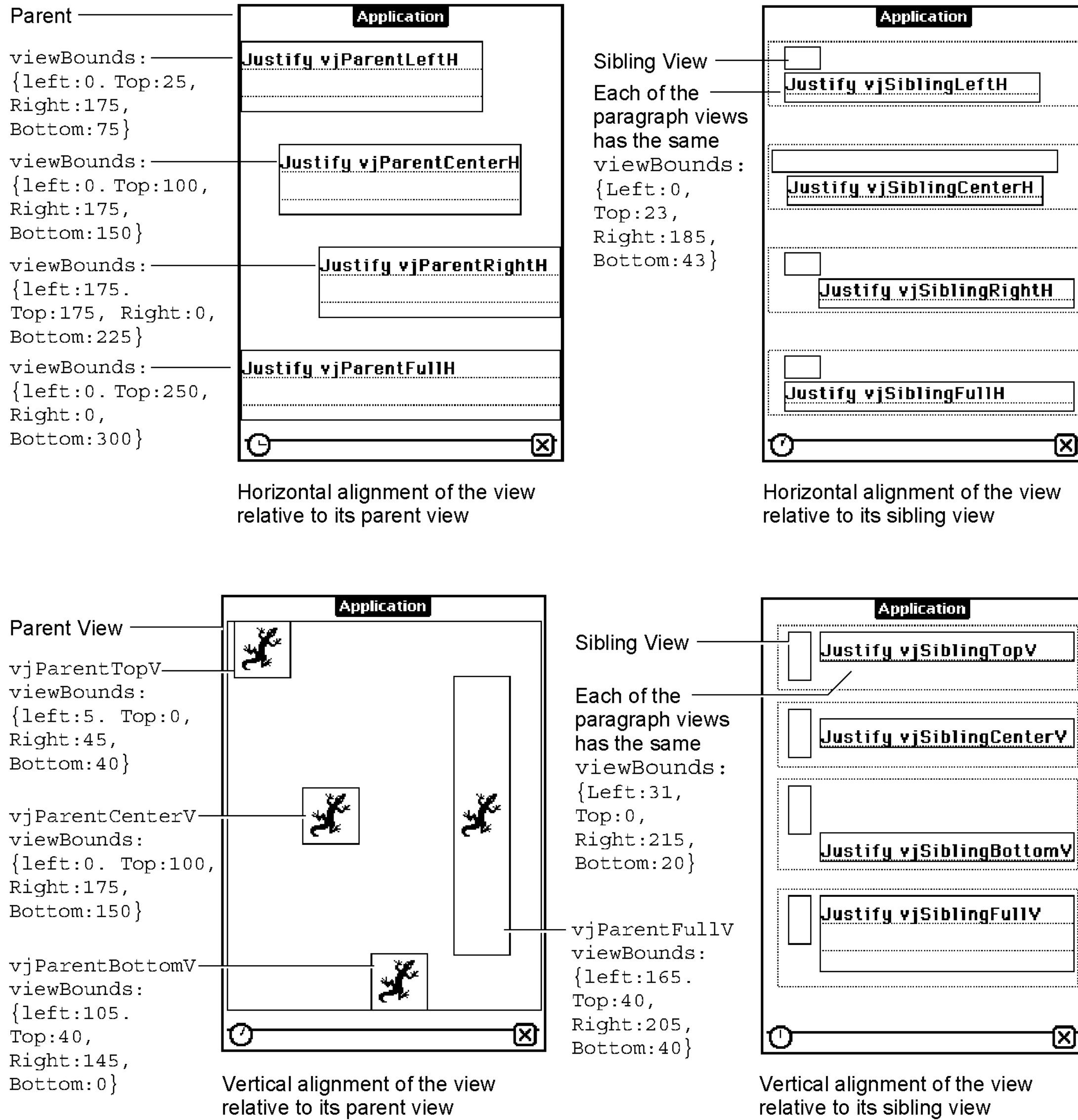
Figure 3-7 View alignment effects



Horizontal alignment of view contents

Vertical alignment of view contents

Figure 3-7 View alignment effects (continued)



Horizontal alignment of the view relative to its parent view

Horizontal alignment of the view relative to its sibling view

Vertical alignment of the view relative to its parent view

Vertical alignment of the view relative to its sibling view

viewOriginX and viewOriginY Slots

These slots can be read but not written or set. Instead, use `Setorigin` to set the origin offset for a view. For more information, see "Scrolling View Contents" (page 3-41).

If you use these slots to specify an offset, the point you specify becomes the new origin. Child views are drawn offset by this amount. This is useful for displaying different portions of a view whose content area is larger than its visible area.

Appearance

The `viewFormat` slot defines view attributes such as its fill pattern, frame pattern, frame type, and so on. Custom fill and frame patterns are defined using the `viewFillPattern` and `viewFramePattern` slots.

The `viewTransferMode` slot controls the appearance of the view when it is drawn on the screen; that is, how the bits being drawn interact with bits on the screen.

View Format

The `viewFormat` slot defines visible attributes of a view such as its fill pattern, frame type, and so on. In the `viewFormat` slot, you can specify one or more format attributes, which are represented by constants defined as bit flags. You can specify one format attribute from each of the following groups:

- view fill pattern
- view frame pattern
- view frame thickness
- view frame roundness
- view frame inset (this is the white space between the view bounds and view frame)
- view shadow style
- view line style (these are solid or dotted lines drawn in the view to make it look like lined paper)

Multiple attributes are specified by adding them together like this:

```
vfFillWhite+vfFrameBlack+vfPen(2)+vfLinesGray
```

Note that the frame of a view is drawn just outside of the view bounding box, not within it.

The fill for a view is drawn before the view contents and the frame is drawn after the contents.

IMPORTANT
Many views need no fill pattern, so you may be inclined to set the fill pattern to "none" when you create such a view. However, it's best to fill the view with white, if the view may be explicitly dirtied (in need of redrawing) and if you don't need a transparent view. This increases the performance of your application because when the system is redrawing the screen, it doesn't have to update views behind those filled with a solid color such as white. However, don't fill all views with white, since there is some small overhead associated with fills; only use this technique if the view is one that is usually dirtied.

Also, note that the application base view always appears opaque, as do all child views of the root view. That is, if no fill is set for the application base view, it automatically appears to be filled with white. ▲

The view format attributes are listed and described in Table 2-5 (page 2-13) in the *Newton Programmer's Reference*.

Custom Fill and Frame Patterns

Custom fill and custom view frame patterns are set for a view by using the `vfCustom` flag, as shown in Table 2-5 (page 2-13) in the *Newton Programmer's Reference*, and by using following two slots:

`viewFillPattern`
: Sets a custom fill pattern that is used to fill the view.

`viewFramePattern`
: Sets a custom pattern that is used to draw the frame lines around the view, if the view has a frame.

You can use custom fill and frame patterns by setting the value of the `viewFillPattern` and `viewFramePattern` slots to a binary data structure containing a custom pattern. A pattern is simply an eight-byte binary data structure with the class `'pattern`.

You can use this NewtonScript trick to create binary pattern data structures "on the fly":

```
DefineGlobalConstant('myPat,SetLength(SetClass(Clone
  ("\uAAAAAAAAAAAAAAAA"),'pattern), 8));
```

This code clones a string, which is already a binary object, and changes its class to `'pattern`. The string is specified with hexadecimal character codes whose binary representation is used to create the pattern. Each two-digit hex code creates one byte of the pattern.

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 107 of 200 PageID #: 28751

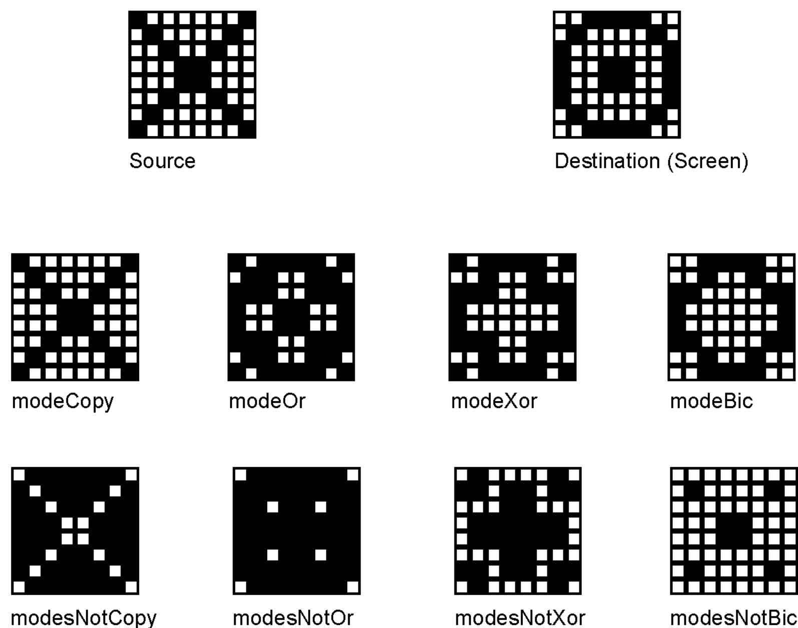
Drawing Transfer Mode for Views

The `viewTransferMode` slot specifies the transfer mode to be used for drawing in the view. The transfer mode controls how bits being drawn are placed over existing bits on the screen. The constants that you can specify for the `viewTransferMode` slot are listed and described in Table 2-6 (page 2-14) in the *Newton Programmer's Reference.*

The transfer mode is used to specify how bits are copied onto the screen when something is drawn in a view. For each bit in the item to be drawn, the system finds the existing bit on the screen, performs a Boolean operation on the pair of bits, and displays the resulting bit.

The first eight transfer modes are illustrated in Figure 3-8. The last transfer mode, in addition to those shown, `modeMask`, is a special one, and its effects are dependent on the particular picture being drawn and its mask.

Figure 3-8 Transfer modes



In Figure 3-8, the Source item represents something being drawn on the screen. The Destination item represents the existing bits on the screen. The eight patterns below these two represent the results for each of the standard transfer modes.

ARENDI-DEFS00003754

Opening and Closing Animation Effects

Another attribute of a view that you can specify is an animation that occurs when the view is opened or closed on the screen. If an effect is defined for a view, it occurs whenever the view is sent an `Open`, `Close`, `Show`, `Hide`, or `Toggle` message.

Use the `viewEffect` slot to give the view an opening or closing animation. Alternately, you can perform one-time effects on a view by sending it one of these view messages: `Effect`, `SlideEffect`, `RevealEffect`, or `Delete`. These methods are described in "Animating Views" (page 3-40).

The `viewEffect` slot specifies an animation that occurs when a view is shown or hidden. If this slot is not present, the view will not animate at these times. There are several predefined animation types. You can also create a custom effect using a combination of `viewEffect` flags from Table 2-7 (page 2-86) in *Newton Programmer's Reference*. To use one of the predefined animation types, specify the number of animation steps, the time per step, and the animation type, with the following values:

`fxSteps(`*x*`)` In *x* specify the number of steps you want, from 1 to 15.

`fxStepTime(`*x*`)` In *x* specify the number of ticks that you want each step to take, from zero to 15 (there are 60 ticks per second).

Specify one of the following values to select the type of animation effect desired:

- `fxCheckerboardEffect`—reveals a view using a checkerboard effect, where adjoining squares move in opposite (up and down) directions.
- `fxBarnDoorOpenEffect`—reveals a view from center towards left and right edges, like a barn door opening where the view is the inside of the barn.
- `fxBarnDoorCloseEffect`—reveals a view from left and right edges towards the center, like a barn door closing where the view is painted on the doors.
- `fxVenetianBlindsEffect`—reveals a view so that it appears behind venetian blinds that open.
- `fxIrisOpenEffect`—changes the size of an invisible "aperture" covering the view, revealing an ever-increasing portion of the full-size view as the aperture opens.
- `fxIrisCloseEffect`—like `fxIrisOpenEffect`, except that it decreases the size of an invisible "aperture" covering the view, as the aperture closes.
- `fxPopDownEffect`—reveals a view as it slides down from its top boundary.
- `fxDrawerEffect`—reveals a view as it slides up from its bottom boundary.
- `fxZoomOpenEffect`—expands the image of the view from a point in the center until it fills the screen; that is, the entire view appears to grow from a point in the center of the screen.

- `fxZoomCloseEffect`—opposite of `fxZoomOpenEffect`. This value shrinks the image of the view from a point in the center until it disappears or closes on the screen.
- `fxZoomVerticalEffect`—the view expands out from a horizontal line in the center of its bounds. The top half moves upward and lower half moves downward.

A complete `viewEffect` specification might look like this:

```
fxVenetianBlindsEffect+fxSteps(6)+fxStepTime(8)
```

You can omit the `fxSteps` and `fxStepTime` constants and appropriate defaults will be used, depending on the type of the effect.

Table 2-7 (page 2-86) in *Newton Programmer's Reference* lists the constants that you can use in the `viewEffect` slot to create custom animation effects. You combine these constants in different ways to create different effects. For example, the predefined animation type `fxCheckerboardEffect` is defined as:

```
fxColumns(8)+fxRows(8)+fxColAltPhase+fxRowAltPhase+fxDown
```

It is difficult to envision what the different effects will look like in combination, so it is best to experiment with various combinations until you achieve the effect you want.

Other Characteristics

Other view characteristics are controlled by the following slots:

`viewFont` Specifies the font used in the view. This slot applies only to views that hold text, that is, views of the class `clParagraphView`. For more information about how to specify the font, see the section "Using Fonts for Text and Ink Display" (page 8-17) in "Text and Ink Input and Display"

`declareSelf` When the template is instantiated, a slot named with the value of this slot is added to the view. Its value is a reference to itself. For example, if you specify `declareSelf:'base`, a slot named `base` is added to the view and its value is set to a reference to itself. Note that this slot is not inherited by the children of a view; it applies only to the view within which it exists.

Inheritance Links

These slots describe the template's location in the inheritance chain, including references to its proto, parent, and children. The following slots are not inherited by children of the template.

`_proto` Contains a reference to a proto template. This slot is created when the view opens.

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 110 of 200 PageID #: 28754

_parent	Contains a reference to the parent template. This slot is created when the view opens. Note that it's best to use the Parent function to access the parent view at run time, rather than directly referencing the _parent slot.
stepChildren	Contains an array that holds references to each of the template's child templates. This slot is created and set automatically when you graphically create child views in NTK. This slot is for children that you add to a template.
viewChildren	Contains an array that holds references to each of a system proto's child templates. Because this slot is used by system protos, you should never modify it or create a new one with this name. If you do so, you may be inadvertently overriding the children of a system proto. An exception to this rule occurs for clEditView; you might want to edit the viewChildren slot of a clEditView. See Table 2-1, "View class constants," (page 2-2) in *Newton Programmer's Guide* for details.

The reason for the dual child view slots is that the viewChildren slot is used by the system protos to store their child templates. If you create a view derived from one of the system protos and change the viewChildren slot (for example, to add your own child templates programmatically), you would actually be creating a new viewChildren slot that would override the one in the proto, and the child templates of the proto would be ignored.

The stepChildren slot has been provided instead as a place for you to put your child templates, if you need to do so from within a method. By adding your templates to this slot, the viewChildren slot of the proto is not overridden. Both groups of child views are created when the parent view is instantiated.

If you are only creating views graphically using the Newton Toolkit palette, you don't need to worry about these internal details. The Newton Toolkit always uses the stepChildren slot for you.

You may see either viewChildren, stepChildren, or both slots when you examine a template at run time in the Newton Toolkit Inspector window. Child templates can be listed in either slot, or both. When a view is instantiated, all the child views from both of these two slots are also created. Note that the templates in the viewChildren slot are instantiated first, followed by the templates in the stepChildren slot.

If you are adding child views in a method that will not be executed until run time, you need to use the stepChildren slot to do this. If there isn't a stepChildren slot, create one and put your views there.

ARENDI-DEFS00003757

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 111 of 200 PageID #: 28755

IMPORTANT
Remember that the `viewChildren` and `stepChildren` arrays contain templates, *not* views. If you try to send a message like `Hide` to one of the objects listed in this array, the system will probably throw an exception because it is not a view.

During run time, if you want to obtain references to the child views of a particular view, you must use the `ChildViewFrames` method. This method returns views from both the `viewChildren` and `stepChildren` slots. This method is described in "Getting References to Views" (page 3-32). ▲

Application-Defined Methods

As your application executes, it receives **messages** from the system that you can choose to handle by providing **methods** that are named after the messages. These messages give you a chance to perform your own processing as particular events are occurring.

For example, with views, the system performs default initialization operations when a view is instantiated. It also sends a view a `ViewSetupFormScript` message. If you provide a method to handle this message, you can perform your own initialization operations in the method. However, handling system messages in your application is optional.

The system usually performs its own actions to handle each event for which it sends your view messages. Your system message-handling methods do not override these system actions. You cannot change, delete, or substitute for the default system event-handling actions. Your system message-handling methods augment the system actions.

For example, when the view system receives a Show command for a view, it displays the view. It also sends the view the `ViewShowScript` message. If you have provided a `ViewShowScript` method, you can perform any special processing that you need to do when the view is displayed.

The system sends messages to your application at specific times during its handling of an event. Some messages are sent before the system does anything to respond to the event, and some are sent after the system has already performed its actions. The timing is explained in each of the message descriptions in "Application–Defined Methods" (page 2-65) in the *Newton Programmer's Reference.*

View Instantiation

View instantiation refers to the act of creating a view from its template. The process of view instantiation includes several steps and it is important to understand when and in what order the steps occur.

ARENDI-DEFS00003758

Declaring a View

Before diving into the discussion of view instantiation, it is important to understand the term **declaring.** Declaring a view is something you do during the application development process using the Newton Toolkit (NTK). Declaring a view allows it to be accessed symbolically from another view.

In NTK, you declare a view using the Template Info command. (Although the phrase "declaring a view" is being used here, at development time, you're really just dealing with the view template.) In the Template Info dialog, you declare a view by checking the box entitled "Declare To," and then choosing another view in which to declare the selected view. The name you give your view must be a valid symbol, and not a reserved word or the name of a system method.

You always declare a view in its parent or in some other view farther up the parent chain. It's best, for efficiency and speed, to declare a view in the lowest level possible in the view hierarchy; that is, in its parent view or as close to it as possible. If you declare a view in a view other than the parent view, it may get the wrong parent view. Because the view's parent is wrong, its coordinates will be wrong as well, so it will show up at the wrong position on screen.

Declaring a view simply puts the declared view in the named slot. See Appendix A, "The Inside Story on Declare," for a complete description. The slot name is the name of the view you are declaring. The slot value, at run time, will hold a reference to the declared view.

The base view of your application is always declared in the system root view. Note that the application base view is declared in a slot named with its application symbol, specified in the Application Symbol field of the Project Settings slip in NTK.

Why would you want to declare a view? When a view is declared in another view, it can be accessed symbolically from that other view. The NewtonScript inheritance rules already allow access from a view to its parent view, but there is no direct access from a parent view to its child views, or between child views of a common parent. Declaring a view provides this access.

For example, if you have two child views of a common parent, and they need to send messages to each other, you need to declare each of them in the common parent view. Or, if a parent view needs to send messages to one of its child views, the child view must be declared in the parent view.

One key situation requiring a declared view is when you want to send the `Open` message to show a nonvisible view. The `Open` message can only be sent to a declared view.

Declaring a view has a small amount of system overhead associated with it. This is why the system doesn't just automatically declare every view you create. You should only declare views that you need to access from other views.

ARENDI-DEFS00003759

For a more detailed technical description of the inner workings of declaring a view, see Appendix A, "The Inside Story on Declare."

Creating a View

A view is created in two stages. First, a view memory object (a frame) is created in RAM. This view memory object contains a reference to its template, along with other transient run-time information. In the following discussion, the phrase, "creating the view" is used to describe just this part of the process. Second, the graphic representation of the view is created and shown on the screen. In the following discussion, the phrase, "showing the view" is used to describe just this part of the process.

A view is created and shown at different times, depending on whether or not it is a declared view.

- If the view is declared in another open (shown) view, it is created when the view in which it is declared is sent the `Open` message. For example, a child view declared in the parent of its parent view is created when that "grandparent" view is opened. Note, however, that the child view is not necessarily shown at the same time it is created.
- If the view is not declared in any view, it is created and also shown when its immediate parent view is sent the `Open` message. (Note that if a nondeclared view's `vVisible` flag is not set, that view can never be created.)

Here is the view creation sequence for a typical application installed in the Newton Extras Drawer and declared in the system root view:

1. When your application is installed on the Newton device, its base view is automatically created, but not shown.
2. When the user taps on the icon representing your application in the Extras Drawer, the system sends the `ButtonToggleScript` message to the application's base view.
3. When the application is launched from the Extras Drawer, a view is created (but not shown yet) for each template declared in the base view. Slots with the names of these views are created in the base view. These slots contain references to their corresponding views.
4. The `ViewSetupFormScript` message is sent to the base view, `viewFlags`, `viewFormat`, `viewBounds`, `viewJustify`, and `declareSelf` slots, and so on, are read from the view template. The global bounds of the view are adjusted to reflect the effects of the `viewJustifyflags`, but the `viewBounds` values are not changed, and the `ViewSetupChildrenScript` message is sent to the base view.

5. The `viewChildren` and `stepChildren` slots are read and the child views are instantiated using this same process. As part of the process, the following messages are sent to each child view, in this order: `ViewSetupFormScript`, `ViewSetupChildrenScript`, and `ViewSetupDoneScript`.
6. The `ViewSetupDoneScript` message is sent to the view.
7. The view is displayed if its `vVisible` `viewFlags` bit is set.
8. The `ViewShowScript` message is sent to the view and then the `ViewDrawScript` message is sent to the view. (Note that the `ViewShowScript` message is not sent to any child views, however.)
9. Each of the child views is drawn, in hierarchical order, and the `ViewDrawScript` message is sent to each of these views, immediately after it is drawn.

As you can see from step 5, when a view is opened, all child views in the hierarchy under it are also shown (as long as they are flagged as visible). A nonvisible child view can be subsequently shown by sending it the `Open` message—as long as it has been declared.

Closing a View

When you send a view the `Close` message, the graphic representation of the view (and of all of its child views) is destroyed, but the view memory object is not necessarily destroyed. There are two possibilities:

- If the view was declared, and the view in which it was declared is still open, the frame is preserved. You can send the view another `Open` or `Toggle` message to reopen it at a later time.

  A view memory object is finally destroyed when the view in which it was declared is closed. That is, when a view is closed, all views declared in it are made available for garbage collection.
- If the view being closed was not declared, both its graphic representation and its view memory object are made available for garbage collection when it is closed.

When a view is closed, the following steps occur:

1. If the view is closing because it was directly sent the `Close` or `Toggle` message, the system sends it the `ViewHideScript` message. (If the view is closing because it is a child of a view being closed directly, the `ViewHideScript` message is not sent to it.)
2. The graphic representation of the view is removed from the screen.
3. The view is sent the `ViewQuitScript` message.

The view itself may or may not be marked for garbage collection, depending on whether or not it was declared.

View Compatibility

The following new functionality has been added for the 2.0 release of Newton System Software. See the *Newton Programmer's Reference* for complete descriptions on each new function and method.

New Drag and Drop API

A drag and drop API has been added. This API now lets users drag a view, or part of a view, and drop it into another view. See "Dragging and Dropping with Views" (page 3-40) for details.

New Functions and Methods

The following functions and methods have been added.

- `AsyncConfirm` creates and displays a slip that the user must dismiss before continuing.
- `ButtonToggleScript` lets the application perform special handling when its icon is tapped in the Extras Drawer.
- `DirtyBox` marks a portion of a view (or views) as needing redrawing.
- `GetDrawBox` returns the bounds of the area on the screen that needs redrawing.
- `GlobalOuterBox` returns the rectangle, in global coordinates, of the specified view, including any frame that is drawn around the view.
- `ModalConfirm` creates and displays a slip.
- `MoveBehind` moves a view behind another view, redrawing the screen as appropriate.
- `StdButtonWidth` returns the size that a button needs to be in order to fit some specified text.

New Messages

The following messages have been added.

- `ReorientToScreen` is sent to each child of the root view when the screen orientation is changed.
- `ViewPostQuitScript` is sent to a view following the `ViewQuitScript` message and after all of the view's child views have been destroyed.

ARENDI-DEFS00003762

New Alignment Flags

The `viewJustify` slot contains new constants that allow you to specify that a view is sized proportionally to its sibling or parent view, both horizontally and/or vertically.

A change to the way existing `viewJustify` constants work is that when you are using sibling-relative alignment, the first sibling uses the parent alignment settings (since it has no sibling to which to justify itself).

Changes to Existing Functions and Methods

The following changes have been made to existing functions and methods for 2.0.

- `RemoveStepView`. This function now removes the view template from the `stepChildren` array of the parent view. You do not need to remove the template yourself.
- `SetValue`. You can now use this global function to change the recognition behavior of a view at run time by setting new recognition flags in the `viewFlags` slot. The new recognition behavior takes effect immediately following the `SetValue` call.
- `GlobalBox`. This method now works properly when called from the `ViewSetupFormScript` method of a view. If called from the `ViewSetupFormScript` method, `GlobalBox` gets the `viewBounds` and `ViewJustify` slots from the view, calculates the effects of the sibling and parent alignment on the view bounds, and then returns the resulting bounds frame in global coordinates.
- `LocalBox`. This method now works properly when called from the `ViewSetupFormScript` method of a view. If called from the `ViewSetupFormScript` method, `LocalBox` gets the `viewBounds` and `ViewJustify` slots from the view, calculates the effects of the sibling and parent alignment on the view bounds, and then returns the resulting bounds frame in local coordinates.
- `ViewQuitScript`. When this message is sent to a view, it propagates down to child views of that view. In system software version 1.0, the order in which child views received this message and were closed was undefined.

  In system software version 2.0, the order in which this message is sent to child views is top-down. Also, each view has the option of having `ViewPostQuitScript` called in child-first order. The return value of the `ViewQuitScript` method determines whether or not the `ViewPostQuitScript` message is sent.

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 117 of 200 PageID #: 28761

New Warning Messages

Warning messages are now printed to the inspector when a NewtonScript application calls a view method in situations where the requested operation is unwise, unnecessary, ambiguous, invalid, or just a bad idea.

Obsolete Functions and Methods

The following functions and methods are obsolete with version 2.0 of the Newton System Software:

- `Confirm`, which created and displayed an OK/Cancel slip. Use `AsyncConfirm` instead.
- `DeferredConfirmedCall` and `DeferredConfirmedSend` have both been replaced by `AsyncConfirm`.

Using Views

This section describes how to use the view functions and methods to perform specific tasks. See “Summary of Views” (page 3-47) for descriptions of the functions and methods discussed in this section.

Getting References to Views

Frequently, when performing view operations, you need access to the child or parent views of a view, or to the root view in the system. You need to use the `ChildViewFrames` and `Parent` methods as well as the `GetRoot` and `GetView` functions to return references to these “related” views.

To test whether an application is open or not, for example, you can use the `GetRoot` function and the application’s signature, together with the global function `kViewIsOpenFunc`:

```
call kViewIsOpenFunc with (GetRoot().appsignature)
```

The `ChildViewFrames` method is an important method you must use if you need access to the child views of a view. It returns the views in the same order in which they appear in the view hierarchy, from back to front. The most recently opened views (which appear on top of the hierarchy) will appear later in the list. Views with the `vFloating` flag (which always appear above nonfloating views) will be at the end of the array.

ARENDI-DEFS00003764

Displaying, Hiding, and Redrawing Views

To display a view (and its visible child views), send it one of the following view messages:

- `Open`—to open the view
- `Toggle`—to open or close the view
- `Show`—to show the view if it had previously been opened, then hidden
- To hide a view (and its child views), send it one of the following view messages:
- `Close`—to hide and possibly delete it from memory
- `Toggle`—to close or open the view
- `Hide`—to hide it temporarily

You can cause a view (and its child views) to be redrawn by using one of the following view messages or global functions:

- `Dirty`—flags the view as "dirty" so it is redrawn during the next system idle loop
- `RefreshViews`—redraws all dirty views immediately
- `SetValue`—sets the value of a slot and possibly dirties the view
- `SyncView`—redraws the view if its bounds have changed

Dynamically Adding Views

Creating a view dynamically (that is, at run time) is a complex issue that has multiple solutions. Depending on what you really need to do, you can use one of the following solutions:

- Don't create the view dynamically because it's easier to accomplish what you want by creating an invisible view and opening it later.
- Create the view by adding a new template to its parent view's `stepChildren` array in the `ViewSetupChildrenScript` method.
- Create the template and the view at run time by using the `AddStepView` function.
- Create the template and the view at run time by using the `BuildContext` function.
- If you want a pop-up list view, called a **picker,** use the `PopupMenu` function to create and manage the view.

These techniques are discussed in the following sections. The first four techniques are listed in order from easiest to most complex (and error prone). You should use the easiest solution that accomplishes what you want. The last technique, for creating a picker view, should be used if you want that kind of view.

Showing a Hidden View

In many cases, you might think that you need to create a view dynamically. However, if the template can be defined at compile time, it's easier to do that and flag the view as not visible. At the appropriate time, send it the `Open` message to show it.

The typical example of this is a slip, which you can usually define at compile time. Using the Newton Toolkit (NTK), simply do not check the `vVisible` flag in the `viewFlags` slot of the view template. This will keep the view hidden when the application is opened.

Also, it is important to declare this view in your application base view. For information on declaring a view, see the section "View Instantiation" (page 3-26).

When you need to display the view, send it the `Open` message using the name under which you have declared it (for example, `myView:Open()`).

This solution even works in cases where some template slots cannot be set until run time. You can dynamically set slot values during view instantiation in any of the following view methods: `ViewSetupFormScript`, `ViewSetupChildrenScript`, and `ViewSetupDoneScript`. You can also set values in a declared view before sending it the `Open` message.

Adding to the stepChildren Array

If it is not possible to define the template for a view at compile time, the next best solution is to create the template (either at compile time or run time) and add it to the `stepChildren` array of the parent view using the `ViewSetupChildrenScript` method. This way, the view system takes care of creating the view at the appropriate time (when the child views are shown).

For example, if you want to dynamically create a child view, you first define the view template as a frame. Then, in the `ViewSetupChildrenScript` method of its parent view, you add this frame to the `stepChildren` array of the parent view. To ensure that the `stepChildren` array is in RAM, use this code:

```
if not HasSlot(self, 'stepChildren) then
   self.stepChildren := Clone(self.stepChildren);
AddArraySlot(self.stepChildren, myDynamicTemplate);
```

The `if` statement checks whether the `stepChildren` slot already exists in the current view (in RAM). If it does not, it is copied out of the template (in ROM) into RAM. Then the new template is appended to the array.

All of this takes place in the `ViewSetupChildrenScript` method of the parent view, which is before the `stepChildren` array is read and the child views are created.

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 120 of 200 PageID #: 28764

If at some point after the child views have been created you want to modify the contents of the `stepChildren` array and build new child views from it, you can use the `RedoChildren` view method. First, make any changes you desire to the `stepChildren` array, then send your view the `RedoChildren` message. All of the view's current children will be closed and removed. A new set of child views will then be recreated from the `stepChildren` array.

Also, note that reordering the `stepChildren` array and then calling `RedoChildren` or `MoveBehind` is the way to reorder the child views of a view dynamically.

For details on an easy way to create a template dynamically, see "Creating Templates" (page 3-36).

Using the AddStepView Function

If you need to create a template and add a view yourself at run time, use the function `AddStepView`. This function takes two parameters: the parent view to which you want to add a view, and the template for the view you want to create. The function returns a reference to the view it creates. Be sure to save this return value so you can access the view later.

The `AddStepView` function also adds the template to the parent's `stepChildren` array. This means that the `stepChildren` array needs to be modifiable, or `AddStepView` will fail. See the code in the previous section for an example of how to ensure that the `stepChildren` array is modifiable.

The `AddStepView` function doesn't force a redraw when the view is created, so you must take one of the following actions yourself:

- Send the new view a `Dirty` message.
- Send the new view's parent view a `Dirty` message. This is useful if you're using `AddStepView` to create several views and you want to show them all at the same time.
- If you created the view template with the `vVisible` bit cleared, the new view will remain hidden and you must send it the `Show` message to make it visible. This technique is useful if you want the view to appear with an animation effect (specified in the `viewEffect` slot in the template).

Do not use the `AddStepView` function in a `ViewSetupFormScript` method or a `ViewSetupChildrenScript` method—it won't work because that's too early in the view creation process of the parent for child views to be created. If you are tempted to do this, you should instead use the second method of dynamic view creation, in which you add your template to the `stepChildren` array and let the view system create the view for you.

ARENDI-DEFS00003767

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 121 of 200 PageID #: 28765

To remove a view created by `AddStepView`, use the `RemoveStepView` function. This function takes two parameters: the parent view from which you want to remove the child view, and the view (not its template) that you want to remove.

For details on an easy way to create a template dynamically, see "Creating Templates" (page 3-36).

Using the BuildContext Function

Another function that is occasionally useful is `BuildContext`. It takes one parameter, a template. It makes a view from the template and returns it. The view's parent is the root view. The template is not added to any `viewChildren` or `stepChildren` array. Basically, you get a free-agent view.

Normally, you won't need to use `BuildContext`. It's useful when you need to create a view from code that isn't part of an application (that is, there's no base view to use as a parent). For instance, if your `InstallScript` or `RemoveScript` needs to prompt the user with a slip, you use `BuildContext` to create the slip.

`BuildContext` is also useful for creating a view, such as a slip, that is larger than your application base view.

For details on an easy way to create a template dynamically, see the next section, "Creating Templates"

Creating Templates

The three immediately preceding techniques require you to create templates. You can do this using NewtonScript to define a frame, but then you have to remember which slots to include and what kinds of values they can have. It's easy to make a mistake.

A simple way of creating a template is to make a user proto in NTK and then use it as a template. That allows you to take advantage of the slot editors in NTK.

If there are slots whose values you can't compute ahead of time, it doesn't matter. Leave them out of the user proto, and then at run time, create a frame with those slots set properly and include a `_proto` slot pointing to the user proto. A typical example might be needing to compute the bounds of a view at run time. If you defined all the static slots in a user proto in the file called `dynoTemplate`, you could create the template you need using code like this:

```
template := {viewBounds: RelBounds(x, y, width, height),
             _proto: GetLayout("DynoTemplate"),
            }
```

This really shows off the advantage of a prototype-based object system. You create a small object "on the fly" and the system uses inheritance to get the rest of the

ARENDI-DEFS00003768

needed values. Your template is only a two-slot object in RAM. The user proto resides in the package with the rest of your application. The conventional, RAM-wasting alternative would have been:

```
template := Clone(PT_dynoTemplate);
template.viewBounds := RelBounds(x, y, width, height);
```

Note that for creating views arranged in a table, there is a function called `LayoutTable` that calculates all the bounds. It returns an array of templates.

Making a Picker View

To create a transient pop-up list view, or picker, you can use the function `PopupMenu`. This kind of view pops up on the screen and is a list from which the user can make a choice by tapping it. As soon as the user chooses an item, the picker view is closed.

You can also create a picker view by defining a template using the `protoPicker` view proto. See "Pickers, Pop-up Views, and Overviews" (page 6-1) for information on `protoPicker` and `PopupMenu`.

Changing the Values in viewFormat

You can change the values in the `viewFormat` slot of a view without closing and reopening a view. Use the `SetValue` function to update the view with new settings. For example:

```
SetValue(myView, 'viewFormat, 337)
// 337 = vfFillWhite + vfFrameBlack+vfPen(1)
```

`SetValue`, among other things, calls `Dirty` if necessary, so you don't need to call it to do a task that the view system already knows about, such as changing `viewBounds` or text slots in a view.

Determining Which View Item Is Selected

To determine which view item is selected in a view call `GetHiliteOffsets`. You must call this function in combination with the `HiliteOwner` function. When you call `GetHiliteOffsets`, it returns an array of arrays. Each item in the outer array represents selected subviews, as in the following example:

```
x:= gethiliteoffsets()
#440CA69                  [[{#4414991}, 0, 2],
                          [{#4417B01}, 0, 5],
                          [{#4418029}, 1, 3]]
```


ARENDI-DEFS00003769

Each of the three return values contains three elements:

- Element 0: the subview that is highlighted. This subview is usually a `clParagraphView`, but you need to check to make sure. A `clPolygonView` is not returned here even if `HiliteOwner` returns a `clEditView` when a `clPolygonView` child is highlighted.
- Element 1: the start position of the text found in the text slot of a `clParagraphView`.
- Element 2: the end position of the text found in the text slot of a `clParagraphView`.

To verify that your view is a `clParagraphView`, check the `viewClass` slot of the view. The value returned (dynamically) sometimes has a high bit set so you need to take it into consideration using a mask constant, `vcClassMask`:

```
theviews.viewClass=clParagraphView OR
theView.viewClass - vcClassMask=clParagraphView
BAnd(thViews.viewClass, BNot(vcClassMask))=clParagraphView
```

If a graphic is highlighted and `HiliteOwner` returns a `clEditView`, check its view children for non-`nil` values of the '`hilites` slot (the '`hilites` slot is for use in any view but its contents are private).

Complex View Effects

If you have an application that uses `ViewQuitScript` in numerous places, your view may close immediately, but to the user the Newton may appear to be hung during the long calculations. A way to avoid this is to have the view appear open until the close completes.

You can accomplish this effect in one of two ways. First, put your code in `ViewHideScript` instead of `ViewCloseScript`. Second, remove the view's `ViewEffect` and manually force the effect at the end of `ViewQuitScript` using the `Effect` method.

Making Modal Views

A modal view is one that primarily restricts the user to interacting with that view. All taps outside the modal view are ignored while the modal view is open.

In the interest of good user interface design, you should avoid using modal views unless they are absolutely necessary. However, there are occasions when you may need one.

Typically, modal views are used for slips. For example, if the user was going to delete some data in your application, you might want to display a slip asking them to confirm or cancel the operation. The slip would prevent them from going to another operation until they provide an answer.

Use `AsyncConfirm` to create and display a slip that the user must dismiss before continuing. The slip is created at a deferred time, so the call to `AsyncConfirm` returns immediately, allowing the currently executing NewtonScript code to finish. You can also use `ModalConfirm` but this method causes a separate OS task to be created and doesn't return until after the slip is closed. It is less efficient and takes more system overhead.

Once you've created a modal view, you can use the `FilterDialog` or `ModalDialog` to open it. Using `FilterDialog` is the preferred method as it returns immediately. As with `ModalConfirm`, `ModalDialog` causes a separate OS task to be created.

Finding the Bounds of Views

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

Run-time functions:

- `RelBounds`— calculates the right and bottom values of a view and returns a bounds frame.
- `SetBounds`—returns a frame when the left, top, right, and bottom coordinates are given.
- `GlobalBox`—returns the rectangle, in coordinates, of a specified view.
- `GlobalOuterBox`—returns the rectangle, in coordinates, of a specified view including any frame that is drawn around a view.
- `LocalBox`—returns a frame containing the view bounds relative to the view itself.
- `MoveBehind`— moves a view behind another view.
- `DirtyBox`— marks a portion of a view as needing redrawing.
- `GetDrawBox`— returns the bounds of an area on the screen that needs redrawing.

Compile-time functions:

- `ButtonBounds`—returns a frame when supplied with the width of a button to be placed in the status bar.
- `PictBounds`— finds the width and height of a picture and returns the proper bounds frame.

ARENDI-DEFS00003771

Animating Views

There are four view methods that perform special animation effects on views. They are summarized here:

- `Effect`—performs any animation that can be specified in the `viewEffect` slot.
- `SlideEffect`—slides a whole view or its contents up or down.
- `RevealEffect`—slides part of a view up or down.
- `Delete`—crumples a view and tosses it into a trash can.

Note that these animation methods only move bits around on the screen. They do not change the actual bounds of a view, or do anything to a view that would change its contents. When you use any of these methods, you are responsible for supplying another method that actually changes the view bounds or contents. Your method is called just before the animation occurs.

Dragging a View

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

Dragging and Dropping with Views

Dragging and dropping a view means allowing a user to drag an item and drop it into another view.

To enable dragging and dropping capability, you must first create a frame that contains slots that specify how the drop will behave. For example, you specify the types of objects that can be dropped into a view, if any. Examples include `'text` or `'picture`. See the *dragInfo* parameter to the `DragAndDrop` method (page 2-46) in the *Newton Programmer's Reference* for a complete description of the slots.

You must set up code to handle a drag and drop in one of two ways: either add code to create a frame and code to call `DragAndDrop`'s view method in each source and destination view that accepts a drag and drop message, or you can create a proto and use it as a template for each view.

Each view must also have the following methods. The system calls these methods in the order listed.

- `ViewGetDropTypesScript`— is sent to the destination view. It is called repeatedly while the pen is down. `ViewGetDropTypesScript` is passed the current location as the dragged item is moved from its source location to its destination location. An array of object types is also returned. In this method, you must return an array of object types that can be accepted by that location.

- GetDropDataScript— is sent to the source view when the destination view is found.
- ViewDropScript— is sent to the destination view. You must add the object to the destination view.
- ViewDropMoveScript— is sent to the source view. It is used when dragging an object within the same view. ViewDropRemoveScript and ViewDropScript are not called in this case.
- ViewDropRemoveScript — is sent to the source view. It is used when dragging an object from one view to another. You must delete the original from the source view when the drag completes.

Additional optional methods can also be added. If you do not include these, the default behavior occurs.

- ViewDrawDragDataScript — is sent to the source view. It draws the image that will be dragged. If you don't specify an image, the area inside the rectangle specified by the DragAndDrop bounds parameter is used.
- ViewDrawDragBackgroundScript— is sent to the source view. It draws the image that will appear behind the dragged image.
- ViewFindTargetScript— is sent to the destination view. It lets the destination view change the drop point to a different view.
- ViewDragFeedbackScript— is sent to the destination view. It provides visual feedback while items are dragged.
- ViewDropDoneScript— is sent to the destination view to tell it that the object has been dropped.

Scrolling View Contents

There are different methods of scrolling a view, supported by view methods you call to do the work. Both methods described here operate on the child views of the view to which you send a scroll message.

One method is used to scroll all the children of a view any incremental amount in any direction, within the parent view. Use the SetOrigin method to perform this kind of scrolling. This method changes the view origin by setting the values of the viewOriginX and viewOriginY slots in the view.

Another kind of scrolling is used for a situation in which there is a parent view containing a number of child views positioned vertically, one below the other. The SyncScroll method provides the ability to scroll the child views up or down the height of one of the views. This is the kind of scrolling you see on the built-in Notepad application.

In the latter kind of scrolling, the child views are moved within the parent view by changing their view bounds. Newly visible views will be opened for the first time, and views which have scrolled completely out-of-view will be closed. The viewOriginX and viewOriginY slots are not used.

For information about techniques you can use to optimize scrolling so that it happens as fast as possible, see "Scrolling" (page 3-46), and "Optimizing View Performance" (page 3-44).

Redirecting Scrolling Messages

You can redirect scrolling messages from the base view to another view. Scrolling and overview messages are sent to the frontmost view; this is the same view that is returned if you call `GetView('viewFrontMost)`.

The `viewFrontMost` view is found by looking recursively at views that have both the `vVisible` and `vApplication` bits set in their `viewFlags`. This means that you can set the `vApplication` bit in a descendant of your base view, and as long as `vApplication` is set in all of the views in the parent chain for that view, the scrolling messages will go directly to that view. The `vApplication` bit is not just for base views, despite what the name might suggest.

If your situation is more complex, where the view that needs to be scrolled cannot have `vApplication` set or is not a descendant of your base view, you can have the base view's scrolling scripts call the appropriate scripts in the view you wish scrolled.

Working With View Highlighting

A highlighted view is identified visually by being inverted. That is, black and white are reversed.

To highlight or unhighlight a view, send the view the `Hilite` message.

To highlight or unhighlight a single view from a group, send the view the `HiliteUnique` message. (The group is defined as all of the child views of one parent view.)

To highlight a view when the current pen position is within it, send the view the `TrackHilite` message. The view is unhighlighted when the pen moves outside the view bounds. If the view is a button, you can send the view the `TrackButton` message to accomplish the same task.

To get the view containing highlighted data, you can call the global function `HiliteOwner`; to get the highlighted text use `GetHiliteOffsets`.

To highlight some or all of the text in a paragraph, you can use the `SetHilite` method.

ARENDI-DEFS00003774

To determine if a given view is highlighted, check the `vSelected` bit in the `viewFlags`. `vSelected` should not be set by your application, but you can test it to see if a view is currently selected (that is, highlighted.) If `BAND(viewflags,vSelected) <> 0` is non-`nil`, the view is selected.

Creating View Dependencies

You can make one view dependent upon another by using the global function `TieViews`. The dependent view is notified whenever the view it is dependent on changes.

This dependency relationship is set up outside the normal inheritance hierarchy. That is, the views don't have to be related to each other in any particular way in the hierarchy. The views must be able to access each other, and so need references to each other. Declaring them to a common parent view is one way to accomplish this.

View Synchronization

View synchronization refers to the process of synchronizing the graphic representation of the view with its internal data description. You need to do this when you add, delete, or modify the children of a view, in order to update the screen.

Typically you would add or remove elements from the `stepChildren` array of a parent view, and then call one of the view synchronization functions to cause the child views to be redrawn, created, or closed, as appropriate. Remember that if you need to modify the `stepChildren` array of a view, the array must be copied into RAM; you can't modify the array in the view template, since that is usually stored in ROM or in a package. To ensure that the `stepChildren` array is in RAM, use this code:

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

To redraw all the child views of a view, you can send two different messages to a view: `RedoChildren` or `SyncChildren`. These work similarly, except that `RedoChildren` closes and reopens all child views, while `SyncChildren` only closes obsolete child views and opens new child views.

Laying Out Multiple Child Views

Two different methods are provided to help lay out a view that is a table or consists of some other group of child views.

To lay out a view containing a table in which each cell is a child view, send the view the message `LayoutTable`.

ARENDI-DEFS00003775

To lay out a view containing a vertical column of child views, send the view the message `LayoutColumn`.

Optimizing View Performance

Drawing, updating, scrolling, and performing other view operations can account for a significant amount of time used during the execution of your application. Here are some techniques that can help speed up the view performance of your application.

Using Drawing Functions

Use the drawing functions to draw lines, rectangles, polygons, and even text in a single view, rather than creating these objects as several separate specialized views. This technique increases drawing performance and reduces the system overhead used for each view you create. The drawing functions are described in "Drawing and Graphics" (page 13-1)

View Fill

Many views need no fill color, so you may be inclined to set the fill color to "none" when you create such a view. However, it's best to fill the view with white, if it may be individually dirtied and you don't need a transparent view. This increases the performance of your application because when the system is redrawing the screen, it doesn't have to update views behind those filled with a solid color such as white. However, don't fill all views with white, since there is some small overhead associated with fills; use this technique only if the view is one that is usually dirtied.

Redrawing Views

A view is flagged as dirty (needing redrawing) if you send it the `Dirty` message, or as a result of some other operation, such as calling the `SetValue` function for a view. All dirty views are redrawn the next time the system event loop executes. Often this redrawing speed is sufficient since the system event loop usually executes several times a second (unless a lengthy or slow method is executing).

However, sometimes you want to be able to redraw a view immediately. The fastest way to update a single view immediately is to send it the `Dirty` message and then call the global function `RefreshViews`. In most cases, only the view you dirtied will be redrawn.

If you call `RefreshViews` and there are multiple dirty views, performance can be significantly slower, depending on where the dirty views are on the screen and how many other views are between them. In this case, what is redrawn is the rectangle that is the union of all the dirty views (which might include many other nondirty

views). Also, if there are multiple dirty views that are in different view hierarchies, their closest common ancestor view is redrawn, potentially causing many other views to be redrawn needlessly.

If you want to dirty and redraw more than one view at a time, it may be faster to send the `Dirty` message to the first view, then call `RefreshViews`, send the `Dirty` message to the second view, then call `RefreshViews`, and so on, rather than just calling `RefreshViews` once after all views are dirtied. The performance is highly dependent on the number of views visible on the screen, the location of the dirty views, and their positions in the view hierarchy, so it's best to experiment to find the solution that gives you the best performance.

Memory Usage

Each view that you create has a certain amount of system overhead associated with it. This overhead exists in the form of frame objects allocated in a reserved area of system memory called the NewtonScript heap. The amount of space that a frame occupies is entirely dependent on the complexity and content of the view to which it corresponds. As more and more views are created, more of the NewtonScript heap is used, and overall system performance may begin to suffer as a result.

This is not usually an issue with relatively simple applications. However, complex applications that have dozens of views open simultaneously may cause the system to slow down. If your application fits this description, try to combine and eliminate views wherever possible. Try to design your application so that it has as few views as possible open at once. This can increase system performance.

You should also be aware of some important information regarding hidden and closed views and the use of memory. This information applies to any view that is hidden, it has been sent the `Hide` message, or to any declared view that is closed but where the view it is declared in is still open. In these cases, the view memory object for the view still exists, even though the view is not visible on the screen. If the hidden or closed view contains large data objects, these objects continue to occupy space in the NewtonScript heap.

You can reduce memory usage in the NewtonScript heap by setting to `nil` those slots that contain large objects and that you don't need when the view is hidden or closed. You can do this in the `ViewHideScript` or `ViewQuitScript` methods, and then reload these slots with data when the view is shown or opened again, using the `ViewShowScript` or `ViewSetupFormScript` methods. Again, the performance impact of these techniques is highly application-dependent and you should experiment to see what works best.

Note that this information applies to the base view of your application, since it is automatically declared in the system root view. As long as it is installed in the Newton, slots that you set in the base view of your application will continue to exist, even after the application is closed. If you store large data objects in the base

view of your application, you should set to `nil` those slots that aren't needed when the application is closed, since they are wasting space in the NewtonScript heap. It is especially important to set to `nil` slots that reference soups and cursors, if they are not needed, since they use relatively much space.

If your application is gathering data from the user that needs to be stored, store the data in a soup, rather than in slots in one of the application views. Data stored in soups is protected, while slots in views are transient and will be lost during a system restart.

For information on declaring views, see "View Instantiation" (page 3-26). For information on storing data in soups, see Chapter 11, "Data Storage and Retrieval."

Scrolling

Scrolling the contents of a view can sometimes seem slow. Here are some techniques you can use to improve the speed:

- Scroll multiple lines at a time, rather than just a single line at a time, when the user taps a scroll arrow.
- In general, reduce the number of child views that need to be redrawn, if possible. For example, make a list that is implemented as several paragraphs (separate views) into a single paragraph.
- Set the view fill to white. For more information, see "View Fill" (page 3-44).

ARENDI-DEFS00003778

Summary of Views

Constants

Class Constants

Constant	Value
`clView`	74
`clPictureView`	76
`clEditView`	77
`clKeyboardView`	79
`clMonthView`	80
`clParagraphView`	81
`clPolygonView`	82
`clRemoteView`	88
`clPickView`	91
`clGaugeView`	92
`clOutline`	105

viewFlags Constants

Constant	Value
`vNoFlags`	0
`vVisible`	1
`vReadOnly`	2
`vApplication`	4
`vCalculateBounds`	8
`vClipping`	32
`vFloating`	64
`vWriteProtected`	128
`vClickable`	512
`vNoScripts`	134217728

viewFormat Constants

Constant	Value
vfNone	0
vfFillWhite	1
vfFillLtGray	2
vfFillGray	3
vfFillDkGray	4
vfFillBlack	5
vfFillCustom	14
vfFrameWhite	16
vfFrameLtGray	32
vfFrameGray	48
vfFrameDkGray	64
vfFrameBlack	80
vfFrameDragger	208
vfFrameCustom	224
vfFrameMatte	240
vfPen (*pixels*)	*pixels* 256
vfLinesWhite	4096
vfLinesLtGray	8192
vfLinesGray	12288
vfLinesDkGray	16384
vfLinesBlack	20480
vfInset (*pixels*)	*pixels* 65536
vfLinesCustom	57344
vfShadow (*pixels*)	*pixels* 262144
vfRound (*pixels*)	*pixels* 16777216

viewTransferMode Constants

Constant	Value
modeCopy	0
modeOr	1
modeXor	2
modeBic	3
modeNotCopy	4
modeNotOr	5
modeNotXor	6
modeNotBic	7
modeMask	8

viewEffect Constants

Constant	Bit Flag	Integer Value
fxColumns(*x*)	((x-1) << fxColumnsShift)	*x*-1
fxRows(*x*)	((x-1) << fxRowsShift)	(*x*-1)*32
fxHStartPhase	(1 << fxHStartPhaseShift)	1024
fxVStartPhase	(1 << fxVStartPhaseShift)	2048
fxColAltHPhase	(1 << fxColAltHPhaseShift)	4096
fxColAltVPhase	(1 << fxColAltVPhaseShift)	8192
fxRowAltHPhase	(1 << fxRowAltHPhaseShift)	16384
fxRowAltVPhase	(1 << fxRowAltVPhaseShift)	32768
fxMoveH	(1 << fxMoveHShift)	65536
fxRight	fxMoveH	65536
fxLeft	fxHStartPhase+fxMoveH	66560
fxUp	fxVStartPhase+fxMoveV	133120
fxDown	fxMoveV	131072
fxMoveV	(1 << fxMoveVShift)	131072
fxVenetianBlindsEffect	fxRows(8)+fxDown	131296
fxDrawerEffect	fxUp	133120

continued

Constant	Bit Flag	Integer Value
fxCheckerboardEffect		
	fxColumns(8)+fxRows(8)+fxColAltVPhase+ fxRowAltHPhase+fxDown	155879
fxZoomVerticalEffect		
	fxColumns(1)+fxRows(2)+fxUp+ fxRowAltVPhase	165920
fxZoomCloseEffect		
	fxColumns(2)+fxRows(2)+fxUp+fxLeft	199713
fxZoomOpenEffect		
	fxColumns(2)+fxRows(2)+fxUp+fxLeft+ fxColAltHPhase+fxRowAltVPhase	236577
fxRevealLine	(1 << fxRevealLineShift)	262144
fxPopDownEffect		
	fxDown+fxRevealLine	393216
fxWipe	1 << fxWipeShift)	524288
fxBarnDoorCloseEffect		
	fxColumns(2)+fxColAltHPhase+ fxRowAltVPhase+fxRight+fxWipe	626689
fxBarnDoorOpenEffect		
	fxColumns(2)+fxColAltHPhase+ fxRowAltVPhase+fxLeft+fxWipe	627713
fxIrisCloseEffect		
	fxColumns(2)+fxRows(2)+fxUp+fxLeft+ fxRevealLine+fxWipe	986145
fxIrisOpenEffect		
	fxColumns(2)+fxRows(2)+fxUp+fxLeft+ fxColAltHPhase+fxRowAltVPhase+ fxRevealLine+fxWipe	1023009
fxFromEdge	(1 << fxFromEdgeShift)	1048576
fxSteps(*x*)	((num-1) << fxStepsShift)	(*x*-1)* 2097152
fxStepTime(*x*)	((num) << fxStepTimeShift)	*x**33554432

Functions and Methods

Getting References to Views

view:ChildViewFrames()
view:Parent()
GetRoot()
GetView(*symbol*)

Displaying, Hiding, and Redrawing Views

view:Open()
view:Close()
view:Toggle()
view:Show()
view:Hide()
view:Dirty()
RefreshViews()
SetValue(*view*, *slotSymbol*, *value*)
view:SyncView()
viewToMove:MoveBehind(*view*)

Dynamically Adding Views

AddStepView(*parentView*, *childTemplate*)
RemoveStepView(*parentView*, *childView*)
AddView(*parentView*, *childTemplate*)
BuildContext(*template*)

Making Modal Views

AsyncConfirm(*confirmMessage*, *buttonList*, *fn*)
ModalConfirm(*confirmMessage*, *buttonList*)
view:ModalDialog()
view:FilterDialog()

Setting the Bounds of Views

RelBounds(*left*, *top*, *width*, *height*)
SetBounds(*left*, *top*, *right*, *bottom*)
view:GlobalBox()
view:GlobalOuterBox()
view:LocalBox()
viewToMove:MoveBehind(*view*)
view:DirtyBox(*boundsFrame*)
view:GetDrawBox()
ButtonBounds(*width*)
PictBounds(*name*, *left*, *top*)

Animating Views

view:Effect(*effect*, *offScreen*, *sound*, *methodName*, *methodParameters*)
view:SlideEffect(*contentOffset*, *viewOffset*, *sound*, *methodName*,
methodParameters)

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

Dragging a View
view:`Drag`(*unit*, *dragBounds*)

Dragging and Dropping an Item
view:`DragAndDrop`(*unit*, *bounds*, *limitBounds*, *copy*, *dragInfo*)

Scrolling View Contents
view:`SetOrigin`(*originX*, *originY*)
view:`SyncScroll`(*What*, *index*, *upDown*)

Working With View Highlighting
view:`Hilite`(*on*)
view:`HiliteUnique`(*on*)
view:`TrackHilite`(*unit*)
view:`TrackButton`(*unit*)
`HiliteOwner()`
`GetHiliteOffsets()`
view:`SetHilite`(*start*, *end*, *unique*)

Creating View Dependencies
`TieViews`(*mainView*, *dependentView*, *methodName*)

Synchronizing Views
view:`RedoChildren()`
view:`SyncChildren()`

Laying Out Multiple Child Views
view:`LayoutTable`(*tableDefinition*, *columnStart*, *rowStart*)
view:`LayoutColumn`(*childViews*, *index*)

Miscellaneous View Operations
view:`SetPopup()`
`GetViewFlags`(*template*)
`Visible`(*view*)
`ViewIsOpen`(*view*) `//platform file function`

Application-Defined Methods
```
ViewSetupFormScript()
ViewSetupChildrenScript()
ViewSetupDoneScript()
ViewQuitScript()
ViewPostQuitScript()
ViewShowScript()
ViewHideScript()
ViewDrawScript()
ViewHiliteScript(on)
ViewScrollDownScript()
```

```
ViewScrollUpScript()
ViewOverviewScript()
ViewAddChildScript(child)
ViewChangedScript(slot, view)
ViewDropChildScript(child)
ViewIdleScript()
sourceView:ViewDrawDragDataScript(bounds)
sourceView:ViewDrawDragBackgroundScript(bounds, copy)
destView:ViewGetDropTypesScript(currentPoint)
src: ViewGetDropDataScript(dragType, dragRef)
destView:ViewDragFeedbackScript(dragInfo, currentPoint, show)
sourceView:ViewDropApproveScript(destView)
sourceView:ViewGetDropDataScript(dragType, dragRef)
destView:ViewDropScript(dropType, dropData, dropPt)
sourceView:ViewDropMoveScript(dragRef, offset, lastDragPt, copy)
destView:ViewFindTargetScript(dragInfo)
sourceView:ViewDropRemoveScript(dragRef)
destView:ViewDropDoneScript()
```


ARENDI-DEFS00003785

C H A P T E R 4

NewtApp Applications

NewtApp is a collection of prototypes that work together in an application framework. Using these protos you can quickly construct a full-featured application that includes functionality like finding and filing.

Whether or not you have written an application for the Newton platform before, you should read this chapter. If you're new at writing Newton applications, you'll find that using NewtApp is the best way to start programming for the Newton platform. If you've created Newton applications before, the process of putting together a NewtApp application will be familiar, though you'll find the time required is significantly less.

Newton applications can be created with the NewtApp framework protos, which are described in this chapter, or by constructing them from protos described in almost every other chapter of this book. Chapter 2, "Getting Started," gives you an overview of the process.

Before reading this chapter you should be familiar with the concepts of views, templates, protos, soups, and stores. However, you don't need to know the details of the interfaces to these objects before proceeding with NewtApp. Simply read the first part of the appropriate chapters to get a good overview of the information. These subjects are covered in Chapter 3, "Views," Chapter 11, "Data Storage and Retrieval," Chapter 16, "Find," Chapter 15, "Filing," and Chapter 21, "Routing Interface."

To work with the examples in this chapter, you should also be familiar with Newton Toolkit (NTK) which is described in the *Newton Toolkit User's Guide.*

About the NewtApp Framework

You can construct an entire application by using the protos in the NewtApp framework, without recreating a lot of support code; that is, the code necessary for providing date and text searching, filing, setting up and registering soups, flushing entries, notifying the system of soup changes, formatting data for display, displaying views, and handling write-protected cards. You set the values of a prescribed set of slots, and the framework does the rest.

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 141 of 200 PageID #: 28785

You can create most kinds of applications with the NewtApp framework. If your application is similar to a data browser or editor, or if it implements an automated form, you can save yourself a significant amount of time by using the NewtApp framework.

If you're creating a specialized application (for example, a calculator) or if you need to display more than one soup at a time, you shouldn't construct it with NewtApp, but should use the protos described in other chapters of this book. These chapters include Chapter 3, "Views," Chapter 6, "Pickers, Pop-up Views, and Overviews," Chapter 7, "Controls and Other Protos," Chapter 8, "Text and Ink Input and Display," Chapter 13, "Drawing and Graphics," Chapter 16, "Find," and Chapter 15, "Filing."

Some NewtApp protos work in nonframework applications. For example, you may want to update an existing application to take advantage of the functionality provided by the NewtApp slot view protos. Updating requires a bit of retrofitting, but it can be done. See "Using Slot Views in Non-NewtApp Applications" (page 4-22) for an example.

When you use the NewtApp framework protos, your user interface is updated as the protos change with new system software releases, thereby staying consistent with the latest system changes. In addition, the built-in code that manages system services for these protos is also automatically updated and maintained as the system software advances.

A NewtApp-based application can present many different views of your data. For example, the Show button displays different views of information; the New button creates new formats for data input.

NewtApp applications use a programming device known as **stationery**—a collective term for data definitions (known as dataDefs) and view definitions (known as viewDefs)—to enable this feature. You should use viewDefs to add different views of your data and dataDefs to create different data formats. Stationery is documented in Chapter 5; its use in a NewtApp application is demonstrated in this chapter.

The NewtApp Protos

When you put the application protos together in a programming environment like Newton Toolkit and set the values of slots, the framework takes care of the rest. Your applications automatically take advantage of extensive system management functionality with little additional work on your part. For example, to include your application in system-wide date searches, just set a slot in the base view of your application called `dateFindSlot`. (See "newtApplication" (page 3-8) in *Newton Programmer's Reference.*)

ARENDI-DEFS00003788

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 142 of 200 PageID #: 28786

The parts of the NewtApp framework are designed to fit together using the two-part NewtonScript inheritance scheme. Generally speaking, the framework is constructed so the user interface components of your application (such as views and buttons) use proto inheritance to make methods and application-state variables, which are provided by NewtApp (and transparent to you), available to your application. Parent inheritance implements slots that keep track of system details.

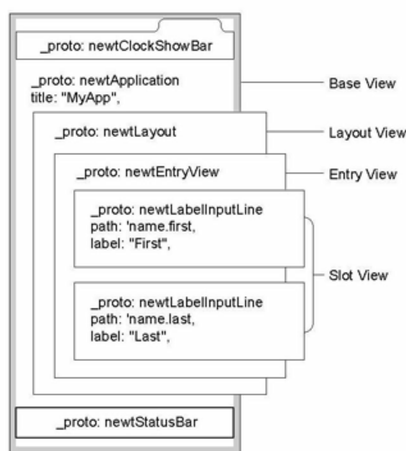
Because the NewtApp framework structure is dependent on both the parent and proto structure of your application, it requires applications to be constructed in a fairly predictable way. Children of certain NewtApp framework protos are required to be particular protos; for example, the application base view must be a `newtApplication` proto.

▲ **WARNING**
When you override system service methods and functions be careful to use the conditional message send operator (`:?`) to avoid inadvertently overriding built-in functionality; otherwise, your code will break.

There may also be alternate ways to construct a NewtApp application, other than those recommended in this chapter and in Chapter 5, "Stationery." Be forewarned that applications using alternate construction methods are not guaranteed to work in the future. ▲

Figure 4-1 shows the four conceptual layers of NewtApp protos that you use to construct an application: the application base view, the layout view, the entry view, and the slot views.

Figure 4-1 The main protos in a NewtApp-based application



ARENDI-DEFS00003789

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 143 of 200 PageID #: 28787

Note
This drawing does not depict the protos as they would appear in a Newton Toolkit layout window. ◆

The basic NewtApp protos are defined here in very general terms. Note that unlike Figure 4-1, this list includes the proto for storing data, which does not have a visual representation in a layout file.

- The `newtApplication` proto is the application's base view. As in nonframework applications, the base view proto either contains or has references to all the other application parts.
- The `newtSoup` proto is used to create and manage the data storage soup for your application; it is not displayed.
- The `newtLayout` protos govern the overall look of your data.
- The `newtEntryView` protos is the view associated with current soup entry and is contained in the default layout view. A `newtEntryView` proto does not display on the screen, but instead manages operations on a soup.
- The slot views are a category of protos used to edit and/or display data from the slots in your application's soup entry frames.

About newtApplication

The `newtApplication` proto serves as the base view for your application; it contains all other application protos. The `allSoups` slot of this proto is where you set up the application soup (based on the `newtSoup` proto).

The functionality defined in this proto layer manages application-wide functions, events, and globals. For example, the functionality for opening and registering soups, dispatching events, and maintaining state information and application globals is implemented in this proto layer.

Also managed by this proto layer are the application-wide user interface elements.

Application-wide Controls

Several control protos affect the entire application. Because of this, the protos are generally placed in the `newtApplication` base view layer. The buttons include the standard Information and Action buttons, as well as the New and Show stationery buttons. Stationery buttons, which you can use to tie viewDefs and dataDefs into your application, are defined in Chapter 5, "Stationery." The NewtApp controls that should be in the `newtApplication` base view include the standard status bar, the folder tab, and the A-Z alphabet tabs.

ARENDI-DEFS00003790

About newtSoup

Application data is stored in persistent structures known as soups in any Newton application. In a NewtApp application, soup definitions, written in the `newtApplication.allSoups` slot, must be based on the `newtSoup` proto.

Within a soup, data is stored in frames known as entries. In turn, entries contain the individual slots in which you store your application's data. The data in these slots is accessed by using a programming construct known as a cursor.

The `newtSoup` proto defines its own version of a set of the data storage objects and methods. If you are not already familiar with these concepts and objects, you should read the introductory parts of Chapter 11, "Data Storage and Retrieval," before trying to use the `newtSoup` proto.

The Layout Protos

Each NewtApp application must have two basic views of the application data, known as layouts, which are:

- an overview—seen when the Overview button is tapped
- a default view—seen when the application is first opened

Three kinds of layouts correspond to three basic application styles:

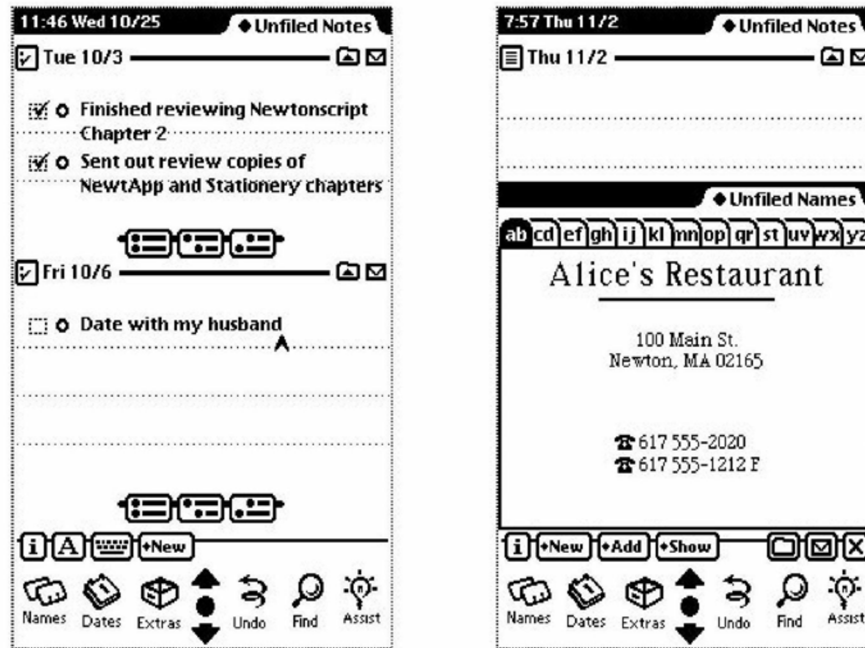
- the card (see `newtLayout`)
- the continuous roll (see `newtRollLayout`)
- the page (see `newtPageLayout`)

Card-based and roll-based applications differ in the number of entries each may have visible at one time. The built-in Names application is a card-based application. For this type of application, only one entry is displayed at a time. In contrast, the built-in Notes application, which is a roll-based application, can have multiple entries visible at once. They must be separated by a header, that incorporates Action and Filing buttons to make it obvious to which entry a button action should apply. Examples of card-based and a roll-based applications are shown in Figure 4-2.

ARENDI-DEFS00003791

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 145 of 200 PageID #: 28789

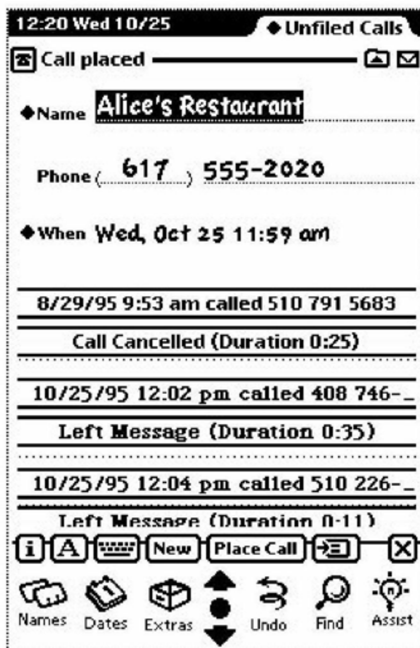
Figure 4-2 A roll-based application (left) versus a card-based application



The page-based application is a hybrid of the card-based and roll-based applications. Like the card-based application, the page-based application shows only one entry at a time. However, unlike the card-based application but like the roll-based application, an entry may be longer than a screen's length. The built-in Calls application, shown in Figure 4-3, is an example of a page-based application.

ARENDI-DEFS00003792

Figure 4-3 Calls is an example of a page-based application



The overview protos are also layouts; they include the `newtOverLayout` and `newtRollOverLayout` protos.

The NewtApp framework code that governs soups, scrolling, and all the standard view functionality, is implemented in the layout protos. A main (default) view layout and an overview layout must be declared in the `allLayouts` slot of the `newtApplication` base view. See "newtApplication" (page 3-8) in *Newton Programmer's Reference* for details.

Your layout can also control which buttons show on the status bar; you can set the `menuLeftButtons` and `menuRightButtons` slots of the layout proto, along with the `statusBarSlot` of the base view (`newtApplication` proto). This control becomes important when more than one entry is shown on the screen, as in a roll-style application. For example, when multiple entries are showing on one screen, the Action and Filing buttons would not be on the status bar. Instead, they would be on the header of each entry, so the entry on which to perform an action is unambiguous.

ARENDI-DEFS00003793

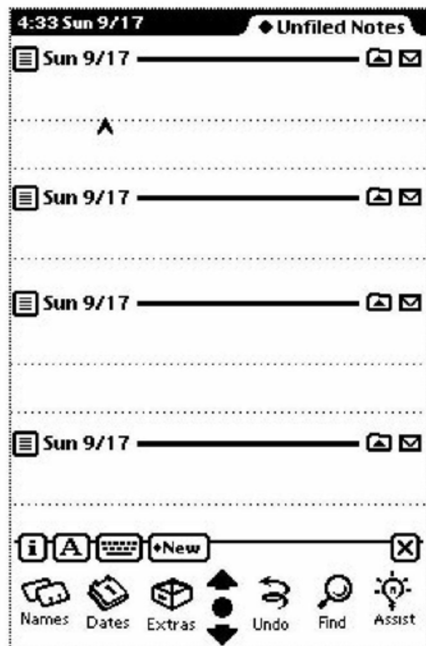
The Entry View Protos

The entry view is the focal point for operations that happen on one soup entry frame at a time. These include functions such as displaying and updating data stored in the entry's slots.

The NewtApp framework has three entry view protos: `newtEntryView`, `newtRollEntryView`, and `newtFalseEntryView`. The `newtEntryView` and `newtRollEntryView` protos are used within a NewtApp application, while the `newtFalseEntryView` and `newtRollEntryView` protos allows you to use the framework's slot views in an application that is not based on the NewtApp framework.

The entry view also contains the user interface components that perform operations on one entry at a time. These components include the header bars, which are used as divider bars to separate multiple entries displayed simultaneously. This behavior happens in the Notes application. An example of the Notes application with multiple entries and header bars is shown in Figure 4-4.

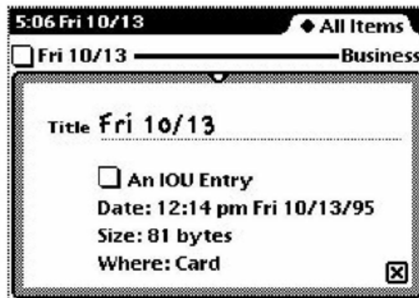
Figure 4-4 Multiple entries visible simultaneously



Note that the header bar contains the Action and Filing buttons on its right side. These buttons appear on the header bar to prevent any ambiguity regarding the entry to be acted upon by those buttons.

In addition, the header bar contains a Title and icon on the left. When the icon is tapped, the Information slip appears, as shown in Figure 4-5. This slip is created from a `newtInfoBox` proto and displays an informational string, which it obtains from the `description` slot of the dataDef. See Chapter 5, "Stationery," for more information about dataDefs.

Figure 4-5 An Information slip



It is at the entry view level of your application that the specific slots for accessing and displaying data in your application soup are set up. The target entry, which is the entry to be acted on, is set in the entry view. The target view is then created by the entry view; the view in which the data in that entry appears. Finally, the data cursor is created by the entry view and is used to access the entries.

The entry view protos also contain important methods that act on individual entries. These methods include functionality for managing and changing existing data in the soup, such as the `FlushData` method.

About the Slot View Protos

The slot view protos retrieve, display, edit, and save changes to any type of data stored in the slots of your application soup's entry frame.

Unless they are contained by either a `newtEntryView` or a `newtFalseEntryView`, the slot views do not work. This is because the entry views are responsible for setting references to a specific entry. These references are used by the slot view to display data.

Slot views exist in two varieties: simple slot views and labelled input-line slot views. Both kinds of slot views are tailored to display and edit a particular kind of

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 149 of 200 PageID #: 28793

data which they format appropriately. For example, the number views (`newtNumberView` and `newtRONumberView`) format number data (according to the value of a `format` slot you set).

The labelled input-line slot view protos provide you with a label, which you may specify, for the input line. Additionally, the label may include a picker (pop-up menu).

These views also format a particular kind of data. To do this they use a special NewtApp object known as a filter to specify a value for the `flavor` slot of the labelled input-line slot views.

The filter object essentially acts as a translator between the target data frame (or more typically, a slot in that frame) and the text field visible to the user. For example, in the `newtDateInputLine` proto, a filter translates the time from a time-in-minutes value to a string, which is displayed. The filter then translates the string back to a time-in-minutes value, which is saved in the soup.

You can create custom filters by basing them on the proto `newtFilter` or on the other filters documented in Table 3-1 (page 3-60) in the *Newton Programmer's Reference*. You can also create custom labelled input-line slot views. See the example in "Creating a Custom Labelled Input-Line Slot View," beginning on page 4-24.

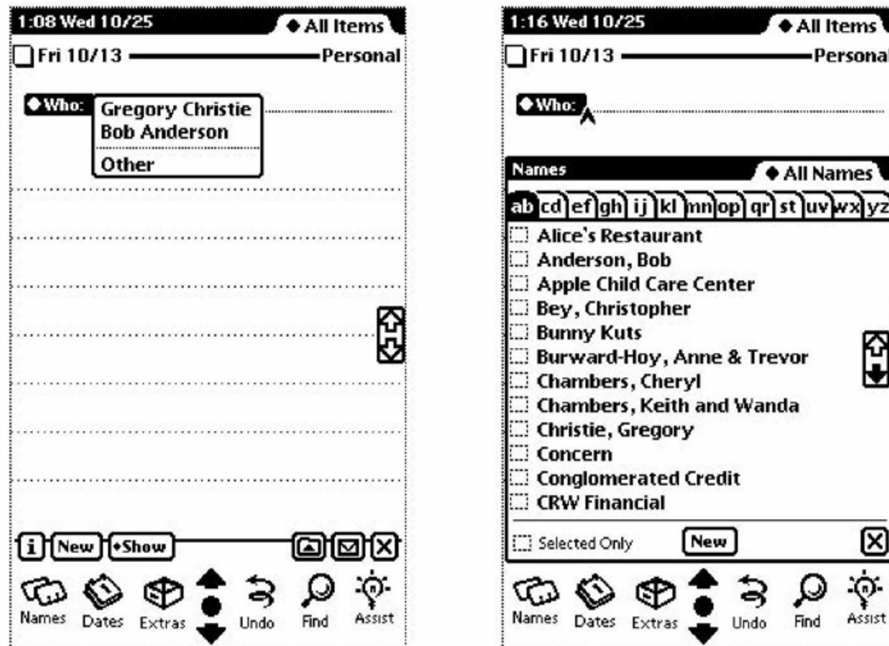
You can have your label input-line protos remember a list of recent items. To do so, all you need do is assign a symbol to the `'memory` slot of your template. This symbol must incorporate your developer signature. The system automatically maintains the list of recent items for your input line. To use the list, you need to use the same symbol with the `AddMemoryItem`, `AddMemoryItemUnique`, `GetMemoryItems`, and `GetMemorySlot` functions, which are described in Chapter 26, "Utility Functions."

In addition, one special slot view, called the `newtSmartNameView` proto, allows the user to choose a name from the soup belonging to the built-in Names application. It adds the pop-up menu item, Other, to the picker; when the user chooses Other from the `newtSmartNameView` proto, it displays the names in the Names application soup in a system-provided people picker.

After you choose a name and close the view displaying the Names soup, that name is displayed on the input line. The name is also put into the Picker menu. A `newtSmartNameView` proto is shown in Figure 4-6.

ARENDI-DEFS00003796

Figure 4-6 The smart name view and system-provided people picker



Stationery

Stationery, an extension you can add to any NewtApp application, is tightly integrated with the NewtApp framework.

Stationery consists of two components that work together: a data definition (dataDef) and a view definition (viewDef). The dataDef provides a definition of the data to be used in the stationery. It is registered in conjunction with its display component, which is a viewDef.

These extensions are available to the user through the New and Show stationery buttons in the NewtApp application. The names of the viewDefs are displayed in the Show menu. The New button is used either to propagate the new entry defined in the dataDef or to display the names of the dataDefs. For more detailed information, see Chapter 5, "Stationery."

NewtApp Compatibility

The NewtApp framework did not exist prior to version 2.0 of Newton system software. Applications created with NewtApp protos will not run on previous versions of the Newton system.

Some NewtApp protos are usable in your non-NewtApp applications. For example, there is a `newtStatusBarNoClose` proto, see page 3-29 in the *Newton Programmer's Reference*, that is unique to NewtApp, which may be used, without special provision, in a non-NewtApp application.

Other NewtApp protos—specifically the slot views—can function only within a simulated NewtApp environment. The mechanism for creating this setup is the `newtFalseEntryView` proto, described on page 3-44 in the *Newton Programmer's Reference*.

The slot views, documented in "Slot View Protos" (page 3-49) in *Newton Programmer's Reference*, provide convenient display and data manipulation functionality that you can use to your advantage in an existing application.

Using NewtApp

The protos in the NewtApp application framework can be used to

- create an application that has one data soup and can be built as a data viewer or editor
- add functionality to non-NewtApp applications
- create and incorporate stationery extensions

When you use the set of protos that make up the NewtApp application framework, you can quickly create an application that takes full advantage of the Newton system services.

In addition, many of the protos may be used in applications built without the framework. In particular, the slot views, used to display data, have built-in functionality you may wish to use.

The framework works best when used with stationery to present different views of and formats for the application's data. The sample application, described in the following sections uses a single piece of stationery, which consists of a dataDef with two viewDefs. Stationery is documented fully in Chapter 5, "Stationery."

The sample application is built using the Newton Toolkit (NTK) development environment. See *Newton Toolkit User's Guide* for more information about using NTK.

Constructing a NewtApp Application

The sample "starter" application presented here shows how to get a NewtApp application underway quickly. You may incorporate this sample code into your applications without restriction. Although every reasonable effort has been made to make sure the application is operable, the code is provided "as is." The

responsibility for its operation is 100% yours. If you are going to redistribute it, you must make it clear in your source files that the code descended from Apple-provided sample code and you have made changes.

The sample is an application for gathering data that supports the system services routing, filing, and finding. It presents two views of the data to be collected: a required default view; "IOU Info" (and an alternate "IOU Notes" view); and a required overview. IOU Info and IOU Notes are stationery and appear as items in the Show button's picker. In addition, it shows how to implement the application in the three styles of NewtApp applications: card, page, and roll. See the DTS sample code for details.

The application starts with three basic NTK layout files:

- The application base view—a `newtApplication` proto.
- A default layout—one of the layout protos.
- An overview layout—either the `newtOverLayout` or `newtRollOverLayout` proto.

The application also contains the NTK layout files for the stationery, a dataDef, and its two corresponding viewDefs:

- `iouDataDef`
- `iouDefaultViewDef`
- `iouNotesViewDef`

The creation of these files is shown in Chapter 5, "Stationery."

A NewtApp application must include standard `InstallScript` and `RemoveScript` functions. Any icons must be included with a resource file; the example uses `CardStarter.rsrc`. In the example, there is also a text file, `Definitions.f`, in which application globals are defined. Neither the resource file nor the text file is required.

The basic view slots, `viewBounds`, `viewFlags`, and `viewJustify`, are discussed in Chapter 3, "Views," and are called out in the samples only when there is something unusual about them.

Using Application Globals

These samples use several application globals. When you use NTK as your development system, they are defined in a definitions file, which we named `Definitions.f`.

The values of the constants `kSuperSymbol` and `kDataSymbol` are set to the application symbol. They are used to set slots that must have unique identifying symbols. You are not required to use the application symbol for this purpose, but it is a good idea, because the application symbol is known to be unique.

ARENDI-DEFS00003799

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 153 of 200 PageID #: 28797

One other global, unique to this application, is set. It is the constant `kAppTitle`, set to the string `"Card Starter"`.

Using newtApplication

This proto serves as the template for the application base view. This section shows you how to use it to set up the

- application base view
- application soup
- status bar; for layout-level control of the appearance and disappearance of its buttons.
- layout slots
- stationery slots

Setting Up the Application Base View

The application base view template, `newtApplication`, should contain the basic application element protos. When you use NTK to create the layout for the `newtApplication` proto, you add to it a `newtStatusBar` proto (the status bar at the bottom of the application) and a `newtClockShowBar` (the folder tab across the top of the application).

Follow these steps to create the application base view:

1. Create a new layout and draw a `newtApplication` proto in it.
2. Place a `newtStatusBar` across the bottom of the layout.
3. Name the `newtStatusBar` proto `status`.
4. Place a `newtClockShowBar` proto across the top of the layout.
5. Save the layout file as `baseView.t`.
6. Name the layout frame `baseView`.

There are more than a dozen slots that need to be set in a `newtApplication` proto. Several of the `newtApplication` slots can be set quickly. Set these slots as follows:

- Set the `title` slot to `kAppTitle`. Note that you must define this constant.
- Set the `appSymbol` slot to `kAppSymbol`. This constant is automatically defined by NTK.
- Set the `appObject` slot to `["Item", "Items"]`.
- Set the `appAll` slot to `"All Items"`. Note that you'll see this displayed on a folder tab.

ARENDI-DEFS00003800

- Optional. Set the `statusBarSlot` to contain the declared name of the status bar so layouts can use it to control the buttons displayed on it. Use the symbol `'status` to set it.

If you wish to override a system message like `ViewSetupFormScript`, which is called before a view is displayed on the screen, make sure to call the inherited method at the end of your own `ViewSetupFormScript` method. Also, you may wish to add a `ReOrientToScreen` method to the `newtApplication` base view so your application can rotate to a landscape display. This message is sent to each child of the root view when the screen orientation is changed. See `ReOrientToScreen` (page 2-73) in *Newton Programmer's Reference* for details.

Finally, be sure to add the layout file `baseView.t` to your project and mark it as the application base view.

Tying Layouts Into the Main Application

The `allLayouts` slot in the `newtApplication` proto is a frame that contains symbols for the application's layout files. It must contain two slots, named `default` and `overview`, that refer to the two layout files used for those respective views.

The section "Using the Layout Protos," beginning on page 4-16, shows how to use the NewtApp layout protos to construct these files. Assume they are named Default Layout and Overview Layout for the purpose of setting the references to them in the `allLayouts` slot. The following code segment sets the `allLayouts` slot appropriately:

```
allLayouts:= {
            default: GetLayout("Default Layout"),
            overview: GetLayout("Overview Layout"),
            }
```

Setting Up the Application Soup

The `newtApplication` proto uses the values in its `allSoups` slot to set up and register your soup with the system.

The framework also looks in the `allSoups` slot to get the appropriate soup information for each layout. It does this by matching the value of the layout's `masterSoupSlot` to the name of a frame contained in the `newtApplication.allSoups` slot. See the section "Using the Layout Protos," following this one.

This application contains only one soup, though a NewtApp application can contain more than one. Each soup defined for a NewtApp application must be based on the `newtSoup` proto. The slots `soupName`, `soupIndices`, and `soupQuery` must be defined within the `allSoups` soup definition frame.

ARENDI-DEFS00003801

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 155 of 200 PageID #: 28799

Use code similar to the following example to set the `allSoups` slot:

```
allSoups:=
{   IOUSoup: {_proto: newtSoup,
                  soupName: "IOU:PIEDTS",
                  soupIndices: [
                              {structure: 'slot,
                              path: 'title,
                              type: 'string},

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

                              { structure: 'slot,
                              path: 'labels,
                              type: 'tags }
                              ],

                  soupQuery: {type: 'index, indexPath:
                              'timeStamp},
                  soupDescr: "The IOU soup.",
                  defaultDataType: '|BasicCard:sig|,}
}
```

Using the Layout Protos

Each NewtApp Application requires exactly two layouts: a default layout, displayed when the application is opened, and an overview layout, displayed when the Overview button is tapped.

The NewtApp framework layout proto you choose for your default view, sets up your application as either a card-, roll-, or page-style application.

Unique slots in the layout protos include:

- `masterSoupSlot`
- `forceNewEntry`

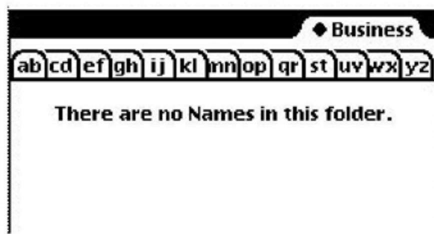
The `masterSoupSlot` is the most important. It contains a reference to the application soup in the `newtApplication.allSoups` slot, from which the layout gets its data.

- The `forceNewEntry` slot allows your application to deal gracefully with the situation created when someone opens a folder that is empty. If the `forceNewEntry` slot is set to `true` in that situation, an entry is automatically created. Otherwise, an alert slip announces that there are no *items* in this list,

ARENDI-DEFS00003802

where *items* is replaced by the value of the `appObject` slot set in the `newtApplication` base view. An example of this message from the Names application is shown in Figure 4-7.

Figure 4-7 The message resulting from a `nil` value for `forceNewEntry`



Using newtOverLayout

The slots you must set for an overview are shown in the Overview Layout browser in Figure 4-8.

Figure 4-8 The overview slots

Follow these steps to create the required overview layout:

1. Open a new layout window and drag out a `newtOverLayout` proto.
2. Name it `Overview Layout`.

ARENDI-DEFS00003803

3. Set the `masterSoupSlot` to the symbol `'IOUSoup`. This correlates to the name of the soup as it is set up in the `newtApplication.allSoups` slot. See "Setting Up the Application Soup," beginning on page 4-15.
4. Add the `forceNewEntry` slot. Leave it with the default value `true`. This causes a new entry to be created if a user tries to open an empty folder.
5. Add a `viewFormat` slot and set the `Fill` value to `White`. This makes the data it displays look better and keeps anything from inadvertently showing through. In addition, the white fill improves the speed of the display and enhances view performance.
6. Set the name slot to a string like `"Overview"`.
7. Add a `centerTarget` slot and set it to `true`. This assures that the entries are centered for display in the Overview.

Controlling Menu Buttons From Layouts

Once the name of the status bar is declared to the application base view (in the `newtApplication.statusBarSlot`), you may control the appearance and disappearance of buttons on the status bar, from the layout view, as needed.

To do this, you must specify which buttons should appear on the status bar by using the slots `menuLeftButtons` and `menuRightButtons`. Each of these is an array that must contain the name of the button proto(s) that you wish to appear on the menu bar's left and right sides. When you use these arrays, the button protos listed in them are automatically placed correctly on the status bar, according to the current human interface guidelines.

To appropriately set up the appearance of the status bar for display in the Overview, first add the optional slots `menuLeftButtons` and `menuRightButtons`. The buttons you name in these slots replace the menu bar buttons from the main layout, since the `statusBarSlot` is set there.

Set the `menuLeftButtons` slot to an array that includes the protos for the Information and New buttons. These buttons are automatically laid out on the status bar, going from left to right.

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

Set the `menuRightButtons` slot to an array that includes the protos for the Action and Filing buttons. These buttons are automatically laid out on the status bar from right to left.

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

Be sure to add the Overview Layout template file to your NTK Project window.

Creating the Default Layout

This is the view you see upon opening the application. Since it will eventually contain views that display the data, it needs to know about the application soup.

The `masterSoupSlot` identifies the application soup to the layout proto. The symbol in this slot must match the name of a soup declared in the `allSoups` slot of the `newtApplication` base view, which was `IOUSoup`. In the layout it is used as a symbol to set the value of the `masterSoupSlot`.

Follow these steps to create the required default layout:

1. Open a new layout window in NTK and drag out a `newtLayout` proto.
2. Name it `default`.
3. Set the `masterSoupSlot` to the symbol `'IOUSoup`. This correlates to the name of the soup as it is set up in the `newtApplication.allSoups` slot. See “Setting Up the Application Soup,” beginning on page 4-15.
4. Add a `forceNewEntry` slot`.` Leave the default value `true`. This causes a new entry to be created when a user tries to open an empty folder.
5. Set the `viewFormat` slot’s Fill value to White. This makes the data it displays look better and keeps anything from inadvertently showing through. In addition, the white fill improves the speed of the display and enhances view performance.

Be sure to add the default template file to your NTK Project window.

Using Entry Views

Entry views are used as containers for the slot views that display data from the slots in the target entry of the application soup. They are also the containers for the different header bars. Note that entry views are not necessary in the overview layout, since the overview layout displays items as shapes.

The entry view sets values needed to locate the data to be displayed in the slot views it will contain. These values include references to the data cursor (the `dataCursor` slot), the soup entry that contains the stored data (the `target` slot), and the view to display data (the `targetView` slot).

ARENDI-DEFS00003805

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 159 of 200 PageID #: 28803

Follow these steps to ready your application for the slot views:

1. Drag out a `newtEntryView` proto on top of the `newtLayout` proto.
2. Optional. Name it `theEntry`.

There are no unusual slots to set in an entry view. Therefore, you are ready to add the header and slot view protos.

3. Drag out a `newtEntryPageHeader` across the top of the `newtEntryView`.
4. Under the header, drag out a `newtStationeryView` proto that covers the rest of the entry view. This special view is not be visible; its function is to provide a bounding box for the viewDef that will eventually be displayed.

The layout should look like the screen shot shown in Figure 4-9.

Figure 4-9 The information button and picker.

Registering DataDefs and ViewDefs

Several slots in the `newtApplication` base view enable you to identify the stationery in your application. These slots include the `allViewDefs`, `allDataDefs`, and `superSymbol` slots.

Note
To see how to create the stationery used as part of this application, consult Chapter 5, "Stationery." The `allDataDefs` and `allViewDefs` slots, which are discussed here, contain references to those dataDefs and viewDefs. ◆

The `allDataDefs` and `allViewDefs` slots are assigned references to the NTK layout files containing your dataDefs and viewDefs. Once this is done, the NewtApp framework automatically registers your stationery with the Newton system registry when your application is installed on a Newton device.

Each `allDataDefs` and `allViewDefs` slot contains frames that are required to contain slots with identical names, to indicate the dataDefs and viewDefs that work together. (A dataDef must be registered with its set of viewDefs because dataDefs use viewDefs to display their data.)

In the `allDataDefs` slot is a frame containing a reference to the NTK layout template for a single dataDef. In the frame within the `allViewDefs` slot is the

ARENDI-DEFS00003806

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 160 of 200 PageID #: 28804

frame containing slots with references to all the viewDef layout templates that work with that dataDef.

The recommended way to name the corresponding `allDataDefs` and `allViewDefs` slots is to set the slot names to the data symbol constant, as shown in the following code examples.

Set the `allDataDefs` slot to return a frame with references to all the application's dataDefs, as follows:

```
result := {};

result.(kDataSymbol) := GetLayout("IOUDataDef");
// result.(kData2Symbol) := ... to add a 2nd DataDef
result;
```

Set the `allViewDefs` slot to return a frame with references to all the application's viewDefs, in a parallel manner, as shown in the following code:

```
result := {};

result.(kDataSymbol) := {
    default: GetLayout("IOUDefaultViewDef"),
    notes:   GetLayout("IOUNotesViewDef"),
    iouPrintFormat: GetLayout("IOUPrintFormat"),
    // Use for routing (beaming, mailing, transports):
    frameFormat: {_proto: protoFrameFormat},
};
// Use to add a 2nd DataDef:
// result.(kData2Symbol) := {...}

result;
```

A NewtApp application only accepts stationery when a dataDef has a `superSymbol` with a value matching the value of the `newtApplication` base view's `superSymbol` slot. For this reason you want the value of the `superSymbol` slot to be a unique symbol. This sample application uses the constant `kSuperSymbol`, which is set to the application symbol `'|IOU:PIEDTS|`, to set the `superSymbol` slot.

Using the Required NewtApp Install and Remove Scripts

An `InstallScript` function and `RemoveScript` function are required to register your NewtApp application with the system for the various system services. These scripts are boilerplate functions you should copy unaltered.

ARENDI-DEFS00003807

You should create a text file, which you save as `Install&Remove.f`, into which to copy the functions:

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

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

This file should be the last one processed when your application is built. (In NTK this means that it should appear at the bottom of the Project file list.)

If you have included the stationery files built in Chapter 5, "Stationery," you may now build, download, and run your NewtApp application.

Using Slot Views in Non-NewtApp Applications

The NewtApp slot view protos have a lot of functionality built into them which you may want to use in a non-NewtApp application. You can do this by keeping your existing application base view, removing the existing entry view layer and its contents, replacing it with a `newtFalseEntryView` proto, and placing the slot views in the `newtFalseEntryView`.

The following requirements must be satisfied for slot views to work outside a NewtApp application:

- The parent of the `newtFalseEntryView` must have the following slots:
  - `target`
  - `targetView`
- The slot views must be contained in a `newtFalseEntryView` proto.
- The `newtFalseEntryView` must receive a `Retarget` message whenever entries are changed.

Modifying the Base View

This discussion assumes that you already have a base view set up as part of your NTK project and that a `newtFalseEntryView` will be added to it later. If that is the case, you already have slots set with specifications for a soup name, soup indices, a soup query, and a soup cursor (among numerous others.)

ARENDI-DEFS00003808

Certain slots must be added to these base view slots for your application to be able to utilize the false entry view and the slot views. First, you must be sure to add a `target` slot and `targetView` slot, so that the false entry view can set them when an entry is changed. Second, you should include a method that sends the `Retarget` message to the false entry view when an entry is changed. As an example, you may wish to implement the following method, or one like it:

```
baseView.DoReTargeting := func()
                            theFalseEntryView:Retarget()
```

There are several places in your code where this message could be sent. For instance, if your application scrolls through entries, you should send the `DoReTargeting` message, defined above, to `ViewScrollUpScript` and `ViewScrollDownScript`. Following is an example of a `ViewScrollUpScript` method that scrolls through soup entries:

```
func()
   begin
      EntryChange(target);
      cardSoupCursor:Prev();
      :ResetTarget();
      :DoRetargeting();
   end
```

Other places where you may want to send the `Retarget` message include a delete action method, a `ViewSetupDoneScript` method (which executes immediately before a view is displayed or redisplayed), or even the `ButtonClickScript` method of a button that generates new entries and thus changes the soup and its display.

Using a False Entry View

The example used here, in which the `newtFalseEntryView` is implemented, is a non-NewtApp application that supports the use of slot views. If you want to adopt slot views into an existing application, you must use `newtFalseEntryView`.

Once you have an application base view set up, you may add the following slots to your `newtFalseEntryView`:

- Add a `dataCursorSlot` and set it to the symbol `'cardSoupCursor`. This symbol should match a slot defined in your application base view. The slot may be omitted if your base application view's cursor slot is set to the default name `dataCursor`.
- Add a `dataSoupSlot` and set it to the symbol `'cardSoup`. This symbol should match a slot defined in your application base view. The slot may be

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 163 of 200 PageID #: 28807

omitted if your base application view's soup slot is set to the default name `dataSoup`.

- Add a `soupQuerySlot` and set it to the symbol `'cardSoupQuerySpec`. This symbol should match a slot defined in your application base view. The slot may be omitted if your base application view's soup query slot is set to the default name `soupQuery`.

Finally, you should make sure to declare the `newtFalseEntryView` to the application base view so the base view can send `Retarget` messages to the false entry view when data is changed.

For more information about the `newtFalseEntryView` see the *Newton Programmer's Reference*.

Creating a Custom Labelled Input-Line Slot View

You may find situations in which you need to create a custom slot view to get one that does exactly what your application requires. For example, the NewtApp framework does not yet contain a slot view that can display a picture. This is possible after you know more about how the slot views work.

In general, a slot view performs the following functions:

- Target data; that is, updates a soup entry from its contents and vice versa.
- Format data by using a filter.
- Allow you to place ("jam") the data from another soup entry in this slot view. Of the built-in slot views, the `newtSmartName` proto does this.

All slot views assume a soup entry has been set by the parent view as the value of the `target` slot. The `target` slot contains a reference to the soup entry. The soup entry contains the slot with the data to be displayed in a given slot view and stores the new data.

Slot views also require a `path` slot which refers to the specific slot within the `target` entry. The path expression must lead to a slot that holds the correct kind of data for a given slot view. For instance, the `path` slot of a `newtROTextDateView` proto must refer to a slot in an entry that contain a integer date.

In the label input-line slot view protos, formatting is accomplished by selecting the correct NewtApp data filter as the value of the slot view's `flavor` slot. Note that some of the NewtApp data filters also specify a particular system picker which will be available when you use the `popup` option for your slot view. See the DTS sample code to see how to create a new newt proto.

ARENDI-DEFS00003810

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 164 of 200 PageID #: 28808

Summary of the NewtApp Framework

Required Code

Required InstallScript

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

Required RemoveScript

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

Protos

newtSoup

```
myNewtSoup := {
_proto: newtSoup, // NewtApp soup proto
soupName: "MyApp:SIG", // a string unique to your app.

soupIndices: [    //soup particulars, may vary
                  {structure: 'slot, //describing a slot
                  path: 'title, // named "title" which
                  type: 'string}, //contains a string
               ...], // more descriptions may follow

soupQuery: {                  // a soup query
            type: 'index,
            indexPath:'timeStamp}, // slot to use as index

soupDescr:"The Widget soup."//string describing the soup
defaultDataType:'soupType ,   //type for your soup entry
```


ARENDI-DEFS00003811

```
AddEntry: //Adds the entry to the specified store
    func(entry, store) ...

AdoptEntry: // Adds entry to the application soup while
    func(entry, type)... // preserving dataDef entry slots

CreateBlankEntry: // Returns a blank entry
    func() ...

DeleteEntry: // Removes an entry from its soup
    func(entry) ...

DuplicateEntry: // Clones and returns entry
    func(entry) ...

DoneWithSoup: // Unregisters soup changes and soup
    func(appSymbol) ...

FillNewSoup:   // Called by MakeSoup to add soup
    func()   ...// values to a new soup

MakeSoup:         // Used by the newtApplication proto
    func(appSymbol)... // to return and register a new soup

GetCursor: // Returns the cursor
    func()   ...

SetupCursor:      // Sets the cursor to an entry in the
func(querySpec) ... // master soup

Query: // Performs a query on a newtSoup
    func(querySpec) ...

GetAlias: // Returns an entry alias.
    func(entry)...

GetCursorPosition: // Returns an entry alias.
    func() ...

GoToAlias: // Returns entry referenced by the alias.
    func(alias)...

}
```


ARENDI-DEFS00003812

newtApplication

```
myNewtAppBaseView := {
_proto: newtapplication, // Application base view proto
   appSymbol: '|IOU:DTS| //Unique application symbol
   title: "Roll Starter" // A string naming the app
   appObject: ["Ox", "Oxen"]// Array with singular and
         // plural strings describing application's data
   appAll: "All Notes" // Displayed in folder tab picker

   allSoups: { //Frame defining all known soups for app
               mySoup: {
               _proto: newtSoup,
               ...       }
            }

   allLayouts: {
               // Frame with references to layout files;
               // both default and overview required.
                  default:GetLayout("DefaultLayoutFile"),
                  overview:GetLayout("OverviewLayoutFile"),
                }

   scrollingEndBehavior:'beepAndWrap // How scrolling is
   // handled at end of view; can also be 'wrap, 'stop, or
   // 'beepAndStop.

   scrollingUpBehavior: 'bottom //Either 'top or 'bottom

   statusBarSlot: 'myStatusBar //Declare name to base so
                               //layouts may send messages

   allDataDefs: {'|appName:SIG|:GetLayout("yourDataDef")}
      //Frame with dataDef symbols as slot names. Slot
      // values are references to dataDef layout files.

   allViewDefs:
      {'|appName:SIG|: {default:GetLayout("yourViewDef")}}
   // Frame with dataDef symbols as slot names. Slot
   // values are references to frames of viewDef
   // layout files.

   superSymbol: '|appName:SIG| //Unique symbol identifying
                         //superSet of application's soups

   doCardRouting:true or 'onlyCardRouting //Enables
                                   // filing and routing
```

```
   dateFindSlot: pathExpression // Enables dateFind for your
   // app. Path must lead to a slot containing a date.
   routeScripts:  //Contains default Delete and Duplicate
                  //route scripts.
   labelsFilter: //Set dynamically for filing settings
   layout:        // Set to the current layout
   newtAppBase: //Set dynamically to identify, for
      //instance, view to be closed when close box tapped
   retargetChain: // Dynamically set array of views
                  // to update.
   targetView: // Dynamically set to the view where
               // target soup entry is displayed
   target: // Set to the soup entry to be displayed

AddEntryFromStationery: //Returns blank entry with class
   func(stationerySymbol) ....// slot set to stationerySymbol

AdoptEntryFromStationery: // Returns entry with all slots
   func(adoptee, stationerySymbol, store) ...// from adopted frame
                     //and class slot set to stationerySymbol

AdoptSoupEntryFromStationery: //Same as above plus
   func(adoptee, stationerySymbol, store, soup) ... // you specify
                                                    //soup & store

FolderChanged: //Changes folder tab to new value
   func(soupName, oldFolder, newFolder) ....

FilterChanged: //Updates folder labels for each soup
   func()         .... //in the allSoups frame.

ChainIn: //Adds views needing to be notified for
   func(chainSymbol) .... //retargeting to chainSymbol array.

ChainOut:                  //Removes views from
   func(chainSymbol) .... //chainSymbol array.

GetTarget:                 //Returns current soup entry.
   func()         ....

GetTargetView:             //Returns view in which the
   func()         ....     // target entry is displayed.

DateFind: // Default DateFind method defined in framework.
          // Set dateFindSlot in base view to enable it.
   func(date, findType, results, scope, findContext) ....
```

```
Find:     // Default Find method as defined in framework.
   func(text, results, scope, findContext) ...

ShowLayout:// Switches display to specified layout.
   func(layout) ...

NewtDeleteScript:// Deletes entry.
   func(entry, view) ... // Referenced in routeScripts array

NewtDuplicateScript:// Duplicates entry.
   func(entry, view) ... // Referenced in routeScripts array

GetAppState:// Gets app preferences, sets app, & returns
   func() ... // prefs. Override to add own app prefs.

GetDefaultState:// Sets default app preferences.
   func() ... // Override to add own app prefs.

SaveAppState:// Sets default app preferences.
   func() ... // Override to add own app prefs.
```

newtInfoButton

```
infoButton := {          // The standard "i" info button
_proto: newtInfoButton,// Place proto in menuLeftButtons
DoInfoHelp:              //Opens online help book
   func()...,
DoInfoAbout:             //Either opens or closes an
   func()...,            // About view
DoInfoPrefs:             //Either opens or closes a
   func()...,}           // Preferences view
```

newtActionButton

```
actionButton := {        // the standard action button
_proto: newtActionButton } // place in menuRightButtons
```

newtFilingButton

```
filingButton := {        // the standard filing button
_proto: newtFilingButton } // place in menuRightButtons
```

newtAZTabs

```
myAZTab:= {            // the standard A-Z tabs
_proto: newtAZTabs,
PickActionScript:      //Default definition keys to
   func(letter)...}    // 'indexPath of allSoups soup query
```

newtFolderTab

```
myFolderTab:= {        // the plain folder tab
_proto: newtFolderTab }
```

newtClockFolderTab

```
myClockFolderTab:= {   // digital clock and folder tabs
_proto: newtClockFolderTab }
```

newtStatusBarNoClose

```
aStatusBarNoClose:= {  // status bar with no close box
_proto: newtStatusBarNoClose,
menuLeftButtons: [], //array of button protos
                     // laid out left to right
menuRightButtons: [], // array of button protos laid out
                     // right to left
```

newtStatusBar

```
aStatusBar:= {         // status bar with close box
_proto: newtStatusBar
menuLeftButtons: [], //array of button protos
                     // laid out left to right
menuRightButtons: [], // array of button protos laid out
                     // right to left }
```

newtFloatingBar

```
aFloatingBar:= {       // status bar with close box
_proto: newtFloatingBar,
menuButtons: [], // array of button protos }
```

newtAboutView

```
anAboutView:= {        // The about view
_proto: newtAboutView }
```

newtPrefsView

```
aPrefsView:= {           // The preferences view
_proto: newtPrefsView  }
```

newtLayout

```
aBasicLayout:= {         // The basic layout view
_proto: newtLayout,
   name: "",                      // Optional.
   masterSoupSlot: 'mainSoup, // Required.
         // Symbol referring to soup from allSoups slot
   forceNewEntry: true, //Forces new entry when empty
                      //folder opened.
   menuRightButtons:[], //Replaces slot in status bar
   menuLeftButtons:[], //Replaces slot in status bar
   dataSoup: 'soupSymbol,//Set to soup for this layout
   dataCursor: ,// Set to top visible entry; main cursor

FlushData:     //Flushes all children's entries
   func(),

NewTarget:     //Utility resets origin and
   func(),     // resets screen

ReTarget:      //Sets the dataCursor slot and resets
   func(setViews),// screen if setViews is true

ScrollCursor:  //Moves cursor delta entries and resets it.
   func(delta),

SetUpCursor:   //Sets cursor and returns entry.
   func(),

Scroller:      //Moves numAndDirection entries. Scrolls
   func(numAndDirection)...,//up if numAndDirection <0.

ViewScrollDownScript: // Calls scroller with the
   func()...,            //value of 1.

ViewScrollUpScript:  // Calls scroller with the
   func()...,            //value of -1.

DoRetarget():          // Calls the "right" retarget
   func()...,
         }
```


ARENDI-DEFS00003817

newtRollLayout

```
myRollLayout:= {  // Dynamically lays out child views
_proto: newtRollLayout, // using protoChild as default
   protoChild: GetLayout("DefaultEntryView"), // Default view
   name: "",                     // Optional.
   masterSoupSlot: 'mainSoup, // Required.
         // Symbol referring to soup from allSoups slot
   forceNewEntry: true, //Forces new entry when empty
                        //folder opened.
   menuRightButtons:[], //Replaces slot in status bar
   menuLeftButtons:[], //Replaces slot in status bar
   dataSoup: 'soupSymbol,//Set to soup for this layout
   dataCursor: ,// Set to top visible entry; main cursor

               // All newtLayout methods are inherited.
            }
```

newtPageLayout

```
myPageLayout:= { // Dynamically lays out child views
_proto: newtPageLayout, // using protoChild as default
   protoChild: GetLayout("DefaultEntryView"), // Default view
   name: "",                     // Optional.
   masterSoupSlot: 'mainSoup, // Required.
         // Symbol referring to soup from allSoups slot
   forceNewEntry: true, //Forces new entry when empty
                        //folder opened.
   menuRightButtons:[], //Replaces slot in status bar
   menuLeftButtons:[], //Replaces slot in status bar
   dataSoup: 'soupSymbol,//Set to soup for this layout
   dataCursor: ,// Set to top visible entry; main cursor

               // All newtLayout methods are inherited.
            }
```

newtOverLayout

```
myOverLayout:= { // Overview for page and card type layout
_proto: newtOverLayout
   centerTarget: nil, // True centers entry in overview
   masterSoupSlot: 'mainSoup, // Required.
         // Symbol referring to soup from allSoups slot
   name: "",                  // Required but not used.
```

```
   forceNewEntry: true, //Creates blank entry for layout
   menuRightButtons:[], //Replaces slot in status bar
   menuLeftButtons:[], //Replaces slot in status bar
   nothingCheckable: nil, //True suppresses checkboxes
Abstract: //Returns shapes for items in overviews
   func(targetEntry, bbox)..., //Override to extract text
GetTargetInfo: //Returns frame with target information
   func(targetType)...,
HitItem: //Called when overview item is tapped.
   func(index, x, y)...,

         // All newtLayout methods are inherited.
            }
```

newtRollOverLayout

```
myOverLayout:= { // Overview for roll-type application
_proto: newtRollOverLayout //Same as newtOverLayout
   centerTarget: nil, // True centers entry in overview
   masterSoupSlot: 'mainSoup, // Required.
         // Symbol referring to soup from allSoups slot
   name: "",                  // Required but not used.
   menuRightButtons:[], //Replaces slot in status bar
   menuLeftButtons:[], //Replaces slot in status bar
   forceNewEntry: true, //Creates blank entry for layout
   nothingCheckable: nil, //True suppresses checkboxes
Abstract: //Returns shapes for items in overviews
   func(targetEntry, bbox)..., //Override to extract text
GetTargetInfo: //Returns frame with target information
   func(targetType)...,
HitItem: //Called when overview item is tapped.
   func(index, x, y)...,

               // All newtLayout methods are inherited.
            }
```

newtEntryView

```
anEntryView:= {   // Invisible container for slot views
_proto: newtEntryView
   entryChanged: //Set to true for flushing
   entryDirtied: //Set to true if flush occurred
   target: //Set to entry for display
   currentDataDef: //Set to current dataDef
```


ARENDI-DEFS00003819

```
   currentViewDef: //Set to current viewDef
   currentStatView: //Set to current context of viewDef
StartFlush: // Starts timer that flushes entry
   func()...,
EndFlush: // Called when flush timer fires
   func()...,
EntryCool: // Is target read-only? True report
   func(report)..., //displays write-protected message
JamFromEntry: // Finds children's jamFromEntry and sends
   func(otherEntry)..., // message if found, then retargets
Retarget: // Changes stationery's display then sends
   func()...,//message on to child views
DoRetarget: // Calls the "right" retarget
   func()...,//
         }
```

newtFalseEntryView

```
aFalseEntryView:= {// Use as container for slot views in
_proto: newtFalseEntryView, // non-NewtApp applications.
   targetSlot: 'target, //Parent needs to have slots
   dataCursorSlot: 'dataCursor, //with names
   targetSlot: 'dataSoup, //that match each of
   dataSoup: 'soupQuery // these symbols.
// newtFalseEntryView inherits all newtEntryView methods.
         }
```

newtRollEntryView

```
aRollEntryView:= { // Entry view for paper roll-style apps
_proto: newtRollEntryView, //stationery required.
   bottomlessHeight: kEntryViewHeight, //Optional
// Inherits slots and methods from newtEntryView.
         }
```

newtEntryPageHeader

```
aPageHeader:= {  // Header bar for card or page-style apps
_proto: newtEntryPageHeader,
   // contains no additional slots or methods
         }
```

newtEntryRollHeader

```
aRollHeader:= {   // Header/divider bar for page or
                  // roll-style apps
_proto: newtEntryRollHeader,
   hasFiling: true // Nil is no filing or action buttons
   isResizable: true // Nil is no drag resizing
            }
```

newtEntryViewActionButton

```
anEntryActionButton:= {// Action button to use on headers
                      // and within entry views
_proto: newtEntryViewActionButton
            }
```

newtEntryViewFilingButton

```
anEntryFilingButton:= {// Filing button to use on headers
                      // and within entry views
_proto: newtEntryViewFilingButton
            }
```

newtInfoBox

```
anInfoBox:= {     // Floating view displayed when header
_proto: newtInfoBox, //icon tapped
   icon: ,// Optional, default provided.
   description: "",// Displayed in view next to icon.
            }
```

newtROTextView

```
readOnlyTextView:= {// All simple slot views based on this
_proto: newtROTextView,
   path: 'pathExpr,// Text stored and retrieved from here
   styles: nil,// Plain text.
   tabs: nil,// Tabs not enabled.
   jamSlot: 'jamPathExpr,// New path for JamFromEntry.
TextScript: // Returns a text representation of data
   func()..., //
JamFromEntry: // Retargets to jamPathExpr if not nil
   func(jamPathExpr)..., //
            }
```

newtTextView

```
editableTextView:= {// This is the editable text view
_proto: newtTextView,
   path: 'pathExpr,// Text stored/retrieved from here
   styles: nil,// Plain text.
   tabs: nil,// Tabs not enabled.
   jamSlot: 'jamPathExpr,// New path for JamFromEntry.
TextScript: // Returns a text representation of data
func()..., //
JamFromEntry: // Retargets to jamPathExpr if not nil
   func(jamPathExpr)..., //
            }
```

newtRONumView

```
readOnlyNumberView:= {// Read-only number view
_proto: newtRONumView,
   path: 'pathExpr,// Numbers stored/retrieved from here
   format: %.10g,// For 10-place decimal; you may change
   integerOnly: true,// Text to num conversion is int
TextScript: // Returns a text representation of data
func()..., //
JamFromEntry: // Retargets to jamPathExpr if not nil
   func(jamPathExpr)..., //
            }
```

newtNumView

```
editableNumberView:= {// Editable number view
_proto: newtNumView,
   path: 'pathExpr,// Numbers stored/retrieved from here
   format: %.10g,// For 10-place decimal; you may change
   integerOnly: true,// Text to num conversion is int
TextScript: // Returns a text representation of data
func()..., //
JamFromEntry: // Retargets to jamPathExpr if not nil
   func(jamPathExpr)..., //
            }
```

newtROTextDateView

```
readOnlyTextDateView:= {// Read-only text and date view. One
_proto: newtROTextDateView, //format slot must be non-nil
   path: 'pathExpr,// Data stored/retrieved from here
```

```
   longFormat: yearMonthDayStrSpec,// for LongDateStr
   shortFormat: nil,       // for ShortDateStr function
TextScript: // Returns a text representation of data
func()..., //
JamFromEntry: // Retargets to jamPathExpr if not nil
   func(jamPathExpr)..., //
            }
```

newtTextDateView

```
editableTextDateView:= {// Editable text and date view. One
_proto: newtTextDateView, //format slot must be non-nil
   path: 'pathExpr,// Data stored/retrieved from here
   longFormat: yearMonthDayStrSpec,// for LongDateStr
   shortFormat: nil,       // for ShortDateStr function
TextScript: // Returns a text representation of data
func()..., //
JamFromEntry: // Retargets to jamPathExpr if not nil
   func(jamPathExpr)..., //
            }
```

newtROTextTimeView

```
readOnlyTextTimeView:= {// Displays and formats time text
_proto: newtROTextTimeView,
   path: 'pathExpr,// Data stored/retrieved from here
   format: ShortTimeStrSpec,// for TimeStr function
TextScript: // Returns a text representation of data
func()..., //
JamFromEntry: // Retargets to jamPathExpr if not nil
   func(jamPathExpr)..., //
            }
```

newtTextTimeView

```
editableTextTimeView:= {// Editable time text
_proto: newtTextTimeView,
   path: 'pathExpr,// Data stored/retrieved from here
   format: ShortTimeStrSpec,// for TimeStr function
TextScript: // Returns a text representation of data
func()..., //
JamFromEntry: // Retargets to jamPathExpr if not nil
   func(jamPathExpr)..., //
            }
```


ARENDI-DEFS00003823

newtROTextPhoneView

```
readOnlyTextPhoneView:= {// Displays phone numbers
_proto: newtROTextPhoneView,
   path: 'pathExpr,// Data stored/retrieved from here
TextScript: // Returns a text representation of data
func()..., //
JamFromEntry: // Retargets to jamPathExpr if not nil
   func(jamPathExpr)..., //
            }
```

newtTextPhoneView

```
EditableTextPhoneView:= {// Displays editable phone numbers
_proto: newtTextPhoneView,
   path: 'pathExpr,// Data stored/retrieved from here
   TextScript: // Returns a text representation of data
   func()..., //
JamFromEntry: // Retargets to jamPathExpr if not nil
   func(jamPathExpr)..., //
            }
```

newtAreaCodeLine

```
protonewtAreaCodeLine : = {
_proto: protonewtAreaCodeLine,
flavor: newtPhoneFilter
access: 'query
label: string  //text to display in the highlight window
path: 'pathExpr,// Data stored/retrieved from here
}
```

newtAreaCodePhoneLine

```
protonewtAreaCodeLine : = {
_proto: protonewtAreaCodeLine,
flavor: newtPhoneFilter
access: 'query
label: string  //text to display in the highlight window
path: 'pathExpr,// Data stored/retrieved from here
}
```

newtROEditView

```
readOnlyEditView:= {  // A text display view, which
                      // may have scrollers
_proto: newtROEditView,
   optionFlags: kNoOptions,          // disables scroller
                      //kHasScrollersOption enables scroller
   doCaret: true, //caret is autoset
   viewLineSpacing: 28,
   path: 'pathExpr,// Data stored/retrieved from here
ScrolltoWord: // Finds words, scrolls to it, and high-
   func(words, hilite)..., // lights it (if hilite is true)
            }
```

newteditView

```
editView:= {          // A text edit view, which
                      // may have scrollers
_proto: newtEditView,
   optionFlags: kNoOptions,          // disables scroller
                      //kHasScrollersOption enables scroller
   doCaret: true, //caret is autoset
   viewLineSpacing: 28,
   path: 'pathExpr,// Data stored/retrieved from here
ScrolltoWord: // Finds words, scrolls to it, and high-
   func(words, hilite)..., // lights it (if hilite is true)
            }
```

newtCheckBox

```
checkBoxView:= {      // A checkbox
_proto: newtCheckBox
   assert: true,// Data stored/retrieved from here
   negate: nil,// Data stored/retrieved from here
   path: 'pathExpr,// Data stored/retrieved from here
ViewSetupForm: // Is target.(path)= assert?
   func()..., //
ValueChanged: // Changes target.(path) value to its
   func()..., // opposite either true or false
            }
```

newtStationeryView

```
stationeryView:= {     // Used as bounding box and container
                       // view for viewDef
_proto: newtStationeryView
            }
```

newtEntryLockedIcon

```
entryLockedIcon:= {  //Shows lock if slot is on locked media
_proto: newtEntryLockedIcon
   icon: nil,// Can also be: lockedIcon
Retarget : // displays either lock or unlocked icon
   func()...,
SetIcon: // Changes target.(path) value to its
   func()..., // opposite either true or false
            }
```

newtProtoLine

```
basicInputLine:= {           // Base for input line protos
_proto: newtProtoLine,
   label: "",// Text for input line label
   labelCommands: ["", "",],// Picker options
   curLabelCommand: 1,// Integer for current command
   usePopup: true,// When true with labelCommands array
                  // picker is enabled
   path: 'pathExpr,// Data stored/retrieved from here
   access: 'readWrite,// Could be 'readOnly or 'pickOnly
   flavor: newtFilter,// Don't change
   memory: nil,            // most recent picker choices
ChangePopup: // change picker items before they display
   func(item, entry)..., //
UpdateText: // Used with Undo to update text to new text
   func(newText)..., //
            }
```

newtLabelInputLine

```
aLabelInputLine:= {          // Labelled input line for text
_proto: newtLabelInputLine,
   label: "",// Text for input line label
   labelCommands: ["", "",],// Picker options
   curLabelCommand: integer,// Integer for current command
```

```
   usePopup: true,// When true with labelCommands array
                  // picker is enabled
   access: 'readWrite,// Could be 'readOnly or 'pickOnly
   flavor: newtTextFilter,//
   memory: nil,            // most recent picker choices
   path: 'pathExpr,// Data stored/retrieved from here
ChangePopup: // change picker items before they display
   func(item, entry)..., //
UpdateText: // Used with Undo to update text to new text
   func(newText)..., //
            }
```

newtROLabelInputLine

```
aLabelInputLine:= {         // Labelled display line for text
_proto: newtROLabelInputLine,
   label: "",// Text for input line label
      flavor: newtTextFilter,//
   memory: nil,            // most recent picker choices
   path: 'pathExpr,// Data stored/retrieved from here
ChangePopup: // change picker items before they display
   func(item, entry)..., //
UpdateText: // Used with Undo to update text to new text
   func(newText)..., //
            }
```

newtLabelNumInputLine

```
aLabelNumberInputLine:= {    // Labelled number input line
_proto: newtLabelNumInputLine,
   label: "",// Text for input line label
   labelCommands: ["", "",],// Picker options
   curLabelCommand: integer,// Integer for current command
   usePopup: true,// When true with labelCommands array
                  // picker is enabled
   access: 'readWrite,// Could be 'readOnly or 'pickOnly
   flavor: newtNumberFilter,//
   memory: nil,            // most recent picker choices
   path: 'pathExpr,// Data stored/retrieved from here
ChangePopup: // change picker items before they display
   func(item, entry)..., //
UpdateText: // Used with Undo to update text to new text
   func(newText)..., //
            }
```

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 181 of 200 PageID #: 28825

newtROLabelNumInputLine

```
aDisplayLabelNumberInputLine:= {// Labelled number display line
_proto: newtROLabelNumInputLine,
   label: "",// Text for input line label
   flavor: newtNumberFilter,//
   path: 'pathExpr,// Data stored/retrieved from here
UpdateText: // Used with Undo to update text to new text
   func(newText)..., //
            }
```

newtLabelDateInputLine

```
editableLabelNumberInputLine:= {// Labelled date input line
_proto: newtLabelDateInputLine,
   label: "",// Text for input line label
   labelCommands: ["", "",],// Picker options
   curLabelCommand: integer,// Integer for current command
   memory: nil,            // most recent picker choices
   usePopup: true,// When true with labelCommands array
               // picker is enabled
   access: 'readWrite,// Could be 'readOnly or 'pickOnly
   flavor: newtDateFilter,//
   path: 'pathExpr,// Data stored/retrieved from here
ChangePopup: // change picker items before they display
   func(item, entry)..., //
UpdateText: // Used with Undo to update text to new text
   func(newText)..., //
            }
```

newtROLabelDateInputLine

```
displayLabelDateLine:= {     // Labelled number display line
_proto: newtROLabelDateInputLine,
   label: "",// Text for input line label
   flavor: newtDateFilter,// Don't change
   path: 'pathExpr,// Data stored/retrieved from here
UpdateText: // Used with Undo to update text to new text
   func(newText)..., //
            }
```

newtLabelSimpleDateInputLine

```
editableLabelSimpleDateLine:= {// Labelled date display line
               // accepts dates like 9/15 or 9/15/95
```


ARENDI-DEFS00003828

```
_proto: newtLabelSimpleDateInputLine,
   label: "",// Text for input line label
   access: 'readWrite,// Could be 'readOnly or 'pickOnly
   flavor: newtSimpleDateFilter,//
   path: 'pathExpr,// Data stored/retrieved from here
UpdateText: // Used with Undo to update text to new text
   func(newText)..., //
            }
```

newtNRLabelDateInputLine

```
pickerLabelDateInputLine:= {   // Input through DatePopup picker
_proto: newtNRLabelDateInputLine,
   label: "",// Text for input line label
   access: 'pickOnly,// Could be 'readOnly
   flavor: newtDateFilter,//
   path: 'pathExpr,// Data stored/retrieved from here
UpdateText: // Used with Undo to update text to new text
   func(newText)..., //
            }
```

newtROLabelTimeInputLine

```
displayLabelTimeLine:= {      // Labelled time display line
_proto: newtROLabelTimeInputLine,
   label: "",// Text for input line label
   flavor: newtTimeFilter,// Don't change
   path: 'pathExpr,// Data stored/retrieved from here
            }
```

newtLabelTimeInputLine

```
aLabelTimeInputLine:= {       // Labelled time input line
_proto: newtLabelTimeInputLine,
   label: "",// Text for input line label
   labelCommands: ["", "",],// Picker options
   curLabelCommand: integer,// Integer for current command
   usePopup: true,// When true with labelCommands array
                  // picker is enabled
   access: 'readWrite,// Could be 'readOnly or 'pickOnly
   flavor: newtTimeFilter,// Don't change
   memory: nil,            // most recent picker choices
   path: 'pathExpr,// Data stored/retrieved from here
ChangePopup: // change picker items before they display
   func(item, entry)..., //
```

```
UpdateText: // Used with Undo to update text to new text
   func(newText)..., //
            }
```

newtNRLabelTimeInputLine

```
pickerLabelTimeInputLine:= {  // Input through TimePopup picker
_proto: newtNRLabelTimeInputLine,
   label: "",// Text for input line label
   access: 'pickOnly,// Could be 'readOnly
   flavor: newtTimeFilter,// Don't change
   path: 'pathExpr,// Data stored/retrieved from here
UpdateText: // Used with Undo to update text to new text
   func(newText)..., //
            }
```

newtLabelPhoneInputLine

```
aLabelPhoneInputLine:= {     // Labelled phone input line
_proto: newtLabelPhoneInputLine,
   label: "",// Text for input line label
   usePopup: true,// When true with labelCommands array
                  // picker is enabled
   access: 'readWrite,// Could be 'readOnly or 'pickOnly
   flavor: newtPhoneFilter,// Don't change
   memory: nil,            // most recent picker choices
   path: 'pathExpr,// Data stored/retrieved from here
ChangePopup: // change picker items before they display
   func(item, entry)..., //
UpdateText: // Used with Undo to update text to new text
   func(newText)..., //
            }
```

newtSmartNameView

```
smartNameLine:= {          // protoPeoplePicker Input
_proto: newtSmartNameView, // from Names soup
   label: "",// Text for input line label
   access: 'readWrite,// Could be 'readOnly or 'pickOnly
   flavor: newtSmartNameFilter,// Don't change
   path: 'pathExpr,// Data stored/retrieved from here
UpdateText: // Used with Undo to update text to new text
   func(newText)...,
            }
```

CHAPTER 5

Stationery

Stationery, which consists of new data formats and different views of your data, may be built into an application or added as an extension. Once incorporated, these data formats and views are available through the pickers (pop-up menus) of the New and Show buttons.

Stationery works best when incorporated into a NewtApp application. It is part of the NewtApp framework and is tightly integrated into its structures. If you are building applications using the NewtApp framework, you'll probably want to read this chapter.

Before you begin you should already be familiar with the concepts documented in Chapter 4, "NewtApp Applications," as well as the concepts of views and templates, soups and stores, and system services like finding, filing, and routing. These subjects are covered in Chapter 3, "Views," Chapter 11, "Data Storage and Retrieval," Chapter 16, "Find," Chapter 15, "Filing," and Chapter 21, "Routing Interface."

The examples in this chapter use the Newton Toolkit (NTK) development environment. Therefore, you should also be familiar with NTK before you try the examples. Consult *Newton Toolkit User's Guide* for information about NTK.

This chapter describes:

- how to create stationery and tie it into an application
- how to create, register, and install an extension
- the stationery protos, methods, and global functions

About Stationery

Stationery application extensions provide different ways of structuring data and various ways to view that data. To add stationery to your application, you must create a **data definition**, also called a dataDef, and an adjunct **view definition**, also called a viewDef. Both of the stationery components are created as view templates, though only the viewDef displays as a view at run time. Stationery always consists of at least one dataDef which has one or more viewDefs associated with it.

ARENDI-DEFS00003831

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 185 of 200 PageID #: 28829

A **dataDef** is based on the `newtStationery` proto and is used to create alternative data structures. The dataDef contains slots that define, describe, and identify its data structures. It also contains a slot, called `superSymbol`, that identifies the application into which its data entries are to be subsumed. It also contains a `name` slot where the string that names the dataDef is placed. This is the name that appears in the New picker. Note that each of the items shown in the New menu of the Notes application in Figure 5-1 is a dataDef name.

The **viewDef** is based on any general view proto, depending upon the characteristics you wish to impart, but must have a specified set of slots added to it. (For more information about the slots required in viewDefs and dataDefs, see the "Stationery Reference" chapter in *Newton Programmer's Reference*.) The viewDef is the view template you design as the input and display device for your data. It is the component of stationery that imparts the "look and feel" for that part of the application. Each dataDef must have at least one viewDef defined to display it, though it can have several.

You may include or add stationery to any NewtApp application or any application that already uses stationery. The stationery components you create appear as items in the pickers (pop-up menus) of the New and Show buttons.

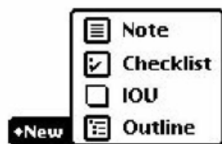
The Stationery Buttons

The stationery buttons are necessary to integrate stationery definitions with an application. They must be in the application which is to display your stationery components. They are defined as part of the NewtApp framework and work only when included in a NewtApp application. (You can use the `newtStationeryPopupButton` proto to create your own non-NewtApp buttons.)

The New button offers new data formats generated from dataDefs. For example, the New button in the built-in Calls application creates one new data entry form by default; if it contained more dataDefs there would be a New picker available. The New button of the built-in Notes application offers a picker whose choices create a new Note, Checklist, or Outline format for entering notes. The example used in this chapter extends the built-in Notes application by adding the dataDef item IOU to the New menu, as shown in Figure 5-1.

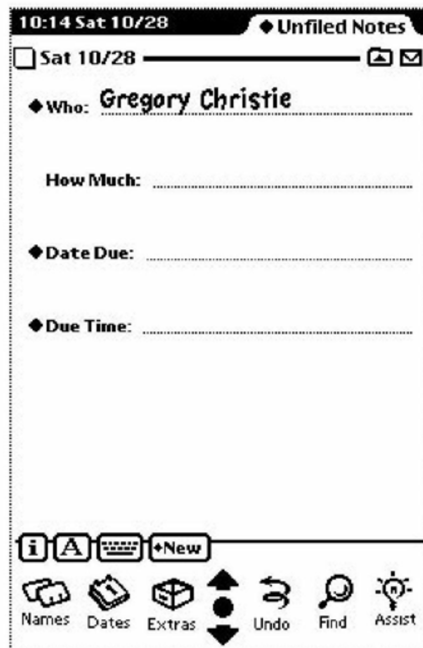
ARENDI-DEFS00003832

Figure 5-1 The IOU extension in the New picker



When you choose IOU from the New picker, an IOU entry is displayed, as shown in Figure 5-2.

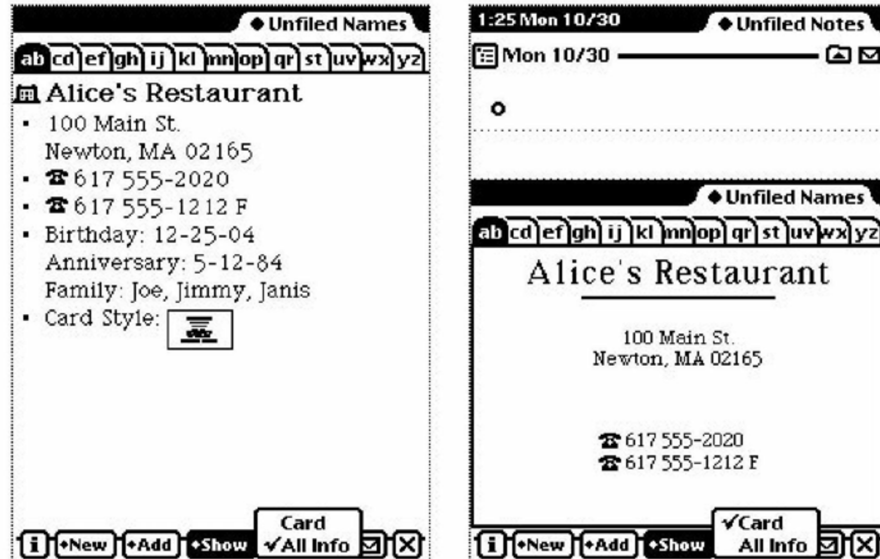
Figure 5-2 The IOU extension to the Notes application



The Show button offers different views for the display of application data. These are generated by the viewDefs defined for an application. For example, the choices in the Show button of the built-in Names application include a Card and All Info view of the data. These views appear as shown in Figure 5-3.

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 187 of 200 PageID #: 28831

Figure 5-3 The Show menu presents different views of application data



Stationery Registration

Your stationery, which may be built as part of an application or outside of an application (as an NTK auto part), must be registered with the system when an application is installed and unregistered when an application is uninstalled. DataDef and viewDef registry functions coordinate those stationery parts by registering the viewDef with its dataDef symbol, as well as its view template. The dataDef registry function adds its view templates to the system registry.

When it is part of a NewtApp application, stationery registration is done automatically–after you set slots with the necessary symbols. If you create your stationery outside of a NewtApp application, you must register (and unregister) your stationery manually by using the global functions provided for that purpose (`RegDataDef`, `UnRegDataDef`, `RegisterViewDef`, and `UnRegisterViewDef`) in the `InstallScript` and `RemoveScript` functions in your application part.

Once stationery is registered, applications can make use of those dataDefs whose `superSymbol` slot matches the application's `superSymbol` slot.

ARENDI-DEFS00003834

Getting Information about Stationery

By using the appropriate global function, you can get information about all the dataDefs and viewDefs that have been registered and thus are part of the system registry. These functions include `GetDefs`, `GetDataDefs`, `GetAppDataDefs`, `GetViewDefs`, and so on. For details on these functions, see *Newton Programmer's Reference*.

You can also obtain application-specific stationery information. This enables applications that are registered for stationery to be extended by other developers.

Compatibility Information

The stationery feature and programming interface is new in Newton OS version 2.0. It is not supported on earlier system versions.

Using Stationery

Stationery allows you to:

- Create discrete data definitions and view definitions.
- Extend your own and other applications.
- Create print formats.

Designing Stationery

Whether you use stationery in an application or an auto part, it is important to keep the data and view definitions as discrete as possible. Encapsulating them, by keeping all references confined to the code in the data or view definition, will make them maximally reusable.

You should keep in mind that these extensions may be used in any number of future programming situations that you cannot foresee. If your stationery was created for an application (which you may have written at the same time), resist any and all urges to make references to structures contained in that application, thereby "hard-wiring" it to depend on that application. In addition, you should provide public interfaces to any values you want to share outside the dataDef.

If your stationery is designed for a NewtApp, the stationery soup entries, which are defined in the dataDef component of stationery, are adopted into the soup of a NewtApp application (via the `AdoptEntry` method) so that your stationery's slots are added to those already defined in the main application. This allows the stationery and the host application to have discrete soup structures. See the description of `AdoptEntry` (page 3-5) in *Newton Programmer's Reference*.

The dataDef component of your stationery should use a `FillNewEntry` method to define its own discrete soup entry structure. Note that it is your responsibility to set a `class` slot within each entry. The value of the `class` slot must match the dataDef symbol and is used by the system when routing the entry (via faxing, mailing, beaming, printing, and so on). An example of how to use `FillNewEntry` follows.

Using FillNewEntry

You use the `FillNewEntry` method in your dataDef to create an entry structure that is tailored to your data. This approach is recommended when your stationery is implemented as part of a NewtApp application.

The `FillNewEntry` method works in conjunction with the NewtApp framework's `newtSoup.CreateBlankEntry` method. The `FillNewEntry` method takes a new entry, as returned by the `CreateBlankEntry` method, as a parameter. This is done with a `CreateBlankEntry` implementation put in the `newtApplication.allSoups` slot of your NewtApp application, as shown in the following example:

```
CreateBlankEntry: func()
   begin
   local newEntry := Clone({class:nil,
                           viewStationery: nil,
                           title: nil,
                           timeStamp: nil,
                           height: 176});
   newEntry.title := ShortDate(time());
   newEntry.timeStamp := time();
   newEntry;
   end;
```

This new entry contains an entry template. In the following code example, that new entry is passed as a parameter to the `FillNewEntry` method, which is implemented in the stationery's dataDef. `FillNewEntry` adds a slot named `kDataSymbol`, which contains an entry template for the stationery's data definition. It then adds a `class` slot to the new entry, which is set to the same constant (`kDataSymbol`). A `viewStationery` slot is then added and set to the same constant (only needed for vestigial compatibility with the Notes application). Finally, it adds a value to the `dueDate` slot of the `kDataSymbol` entry.

```
FillNewEntry: func(newEntry)
   begin
   newEntry.(kDataSymbol) :=
            Clone({who: "A Name",
                  howMuch: 42,
                  dueDate: nil});
```

```
newEntry.class := kDataSymbol;
newEntry.viewStationery := kDataSymbol;
newEntry.(kDataSymbol).dueDate:=time();
newEntry;
end;
```

Extending the Notes Application

You may extend an existing application, such as the built-in Notes application, by adding your own stationery. This is done by building and downloading an NTK auto part that defines your stationery extensions.

The sample project used to illustrate many of the following sections consists of these files, in the processing order shown:

- `ExtendNotes.rsrc`
- `ExtendNotes Definitions.f`
- `iouDataDef`
- `iouDefaultViewDef`
- `iouPrintFormat`
- `ExtendNotes Install & Remove.f`

Of these, the `iouDataDef`, `iouDefaultViewDef`, and `ExtendNotes Install & Remove.f` files are used in the examples in this chapter. The resource file (`ExtendNotes.rsrc`) contains the icon that is displayed next to the dataDef name in the New menu (as shown in Figure 5-1). The definitions file (`ExtendNotes Definitions.f`) is the file in which the constants, some of which are used in examples, are defined. Finally, the `iouPrintFormat` file defines a print format for the stationery.

Determining the SuperSymbol of the Host

Using stationery requires the presence of a matching `superSymbol` slot in both the host application and the dataDef component of your stationery. The value in the `superSymbol` slot is used to link a dataDef to an application.

If you do not know the value of the `superSymbol` slot for an application that is installed on your Newton device, you may use the global function `GetDefs` to see all the dataDefs that are registered by the system.

A call to the global function `GetDefs` in the NTK Inspector window returns a series of frames describing dataDefs that have been registered with the system. An excerpt of the output from a call made in the Inspector window follows.

```
GetDefs('dataDef,nil,nil)

#44150A9  [{_proto: {@451},
            symbol: paperroll,
            name: "Note",
            superSymbol: notes,
            description: "Note",
            icon: {@717},
            version: 1,
            metadata: NIL,
            MakeNewEntry: <function, 0 arg(s) #46938D>,
            StringExtract: <function, 2 arg(s) #4693AD>,
            textScript: <function, 2 arg(s) #4693CD>},
           {_proto: {@451},
            symbol: calllog,
            name: "Calls",
            superSymbol: callapp,
            description: "Phone Message",
            icon: {@718},
            version: 1,
            metadata: NIL,
            taskSlip: |PhoneHome:Newton|,
            MakeNewEntry: <function, 0 arg(s) #47F9A9>,
            StringExtract: <function, 2 arg(s) #47F969>,
            textScript: <function, 2 arg(s) #47F989>},
           ...]
```

`GetDefs` and other stationery functions are documented in *Newton Programmer's Reference*.

Creating a DataDef

You create a dataDef by basing it on a `newtStationery` proto. In NTK it is created as a layout file, even though it is never displayed. The following steps lead you through the creation of the dataDef that is used to extend the built-in Notes application.

Note again that the data definition is adopted into an application's soup only when the application and dataDef have matching values in their `superSymbol` slots. For instance, when you are building a dataDef as an extension to the Notes application, as we are in this example, your dataDef must have `'notes` as the value of its `superSymbol` slot.

ARENDI-DEFS00003838

The following example uses the constant kSuperSymbol as the value of the superSymbol slot. It is defined as follows in the Extend Notes Definition.f file:

```
constant kSuperSymbol := 'notes;// Note's SuperSymbol
```

Once you have created an NTK layout, named the template iouDataDef, and saved the file under the name iouDataDef, you may set the slots of the iouDataDef as follows:

- Set name to "IOU". This shows up in the New button's picker.
- Set superSymbol to the constant kSuperSymbol. This stationery can only be used by an application that has a matching value in the newtApplication base view's superSymbol slot.
- Set description to "An IOU entry". This string shows up in the information box that appears when the user taps the icon on the left side of the header, as shown in Figure 4-5 (page 4-9).
- Set symbol to kDataSymbol.
- Set version to 1. This is an arbitrary stationery version number set at your discretion.
- Remove the viewBounds slot; it's not needed since this object is not a view.

There are a number of methods defined within the newtStationery proto that you should override for your data type.

Defining DataDef Methods

The three methods MakeNewEntry, StringExtract, and TextScript are illustrated in this section. You use the method MakeNewEntry to define the soup entries for your dataDef; the method StringExtract is required by NewtApp overview scripts to return text for display in the overview; and TextScript is called by the routing interface to return a text description of your data.

The MakeNewEntry method returns a complete entry frame which will be added to some (possibly unknown) application soup. You should use MakeNewEntry, instead of the FillNewEntry method (which works in conjunction with the NewtApp framework's newtSoup.CreateBlankEntry), when your stationery is being defined as an auto part.

The example of MakeNewEntry used here defines the constant kEntryTemplate as a frame in which to define all the generic parts of the entry.

All the specific parts of the data definition are kept in a nested frame that has the name of the data class symbol, kDataSymbol. By keeping the specific definitions of your data grouped in a single nested frame and accessible by the class of the data, you are assuring that your code will be reusable in other applications.

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 193 of 200 PageID #: 28837


```
// Generic entry definition:
DefConst('kEntryTemplate, {
   class: kDataSymbol,
   viewStationery: kDataSymbol,// vestigial; for Notes
                               // compatibility
   title: nil,
   timeStamp: nil,
   height: 176,       // For page and paper roll-type apps
                      // this should be the same as height
                      // slot in dataDef and viewDefHeight
                      // slot in viewDef (if present)
});

// This facilitates writing viewDefs that can be reused
kEntryTemplate.(kDataSymbol) := {
   who: nil,
   howMuch: 0,
   dueDate: nil,
};

MakeNewEntry: func()
   begin
   local theNewEntry := DeepClone(kEntryTemplate);
   theNewEntry.title := ShortDate(time());
   theNewEntry.timeStamp := time();
   theNewEntry.(kDataSymbol).dueDate := time();
   theNewEntry;
   end;
```

The `StringExtract` method is called when an overview is generated and is expected to return a one or two-line description of the data. Here is an example of a `StringExtract` implementation:

```
StringExtract: func(item,numLines)
   begin
   if numLines = 1 then
      return item.title
   else
      return item.title&&item.(kDataSymbol).who;
   end;
```

The `TextScript` method is called by the routing interface to get a text version of an entire entry. It differs from `StringExtract` in that it returns the text of the item, rather than a description.

ARENDI-DEFS00003840

Here is an example:

```
TextScript: func(item,target)
   begin
   item.text := "IOU\n" & target.(kDataSymbol).who
               && "owes me" &&
               NumberStr(target.(kDataSymbol).howMuch);
   item.text;
   end;
```

Creating ViewDefs

ViewDefs may be based on any of the generic view protos. You could use, for instance, a `clView`, which has very little functionality. Or, if you wanted a picture to display behind your data, you could base your viewDef on a `clPictureView`.

Routing and printing formats are also implemented as viewDefs. You can learn more about using special protos to create routing and printing formats in Chapter 21, “Routing Interface.”

Note that these are just a few examples of views you may use as a base view in your viewDef. Your viewDef will function as expected, so long as the required slots are set and the resulting view template is registered, either in the `allviewDefs` slot of the `newtApplication` base view or through the `InstallScript` function of an auto part.

You may create the viewDef for the auto part that extends the Notes application by using a `clView` as the base view. Create an NTK view template, named `iouDefaultViewDef`, in which a `clView` fills the entire drawing area. Then save the view template file (using the Save As menu item) as `iouDefaultViewDef`.

You can now set the slots as follows:

- Set the `name` slot to `"IOU Info"`. This string appears in the Show button, if there is one.
- Set the `symbol` slot to `'default`. At least one of the viewDefs associated with a dataDef must have `'default` as the value of its `symbol` slot.
- Set the `type` slot to `'viewer`. The three system-defined types for viewDefs are `'editor`, `'viewer`, and `'routeFormat`. You may define others as you wish.
- Set the `viewDefHeight` slot to 176 (of the four slot views that will be added to this viewDef, each is 34 pixels high plus an 8-pixel separation between them and an 8-pixel border at the bottom).
- Set the `viewBounds` slot to 0, 0, 0, 0.
- Set the `viewJustify` slot to horizontal parent full relative and vertical parent full relative.

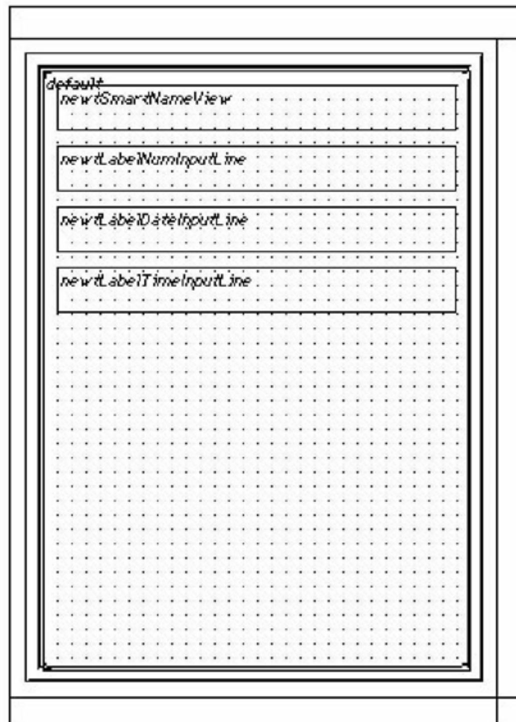
Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 195 of 200 PageID #: 28839

Add the protos that will display the data and labels to the working application. The protos used here include:

- `newtSmartNameView`
- `newtLabelNumInputLine`
- `newtLabelDateInputLine`
- `newtLabelTimeInputLine`

You can read more about these protos in Chapter 4, "NewtApp Applications." They should be aligned as shown in Figure 5-4.

Figure 5-4 The default viewDef view template



Set the slots of the `newtSmartNameView` as follows:

- Set the `label` slot to `"Who"`.
- Set the `path` slot to `[pathExpr: kDataSymbol, 'who]`. The path slot must evaluate to a slot in your data entry frame that contains a name (or a place to store one).
- Set the `usePopup` slot to `true`.

ARENDI-DEFS00003842

Set the slots of the `newtLabelNumInputLine` as follows:

- Set the `label` slot to `"How Much"`.
- Set the `path` slot to `[pathExpr: kDataSymbol, 'howMuch]`. This `path` slot must evaluate to a slot in your data entry frame that contains a number (or a place to store one).

Add a `newtLabelDateInputLine` at the top of the `default` template so that it is aligned as shown. Then set the slots as follows:

- Set the `label` slot to `"Date Due"`.
- Set the `path` slot to `[pathExpr: kDataSymbol, 'dueDate]`. This `path` slot must evaluate to a slot in your data entry frame that contains a date (or a place to store one).

Add a `newtLabelTimeInputLine` at the top of the `default` template so that it is aligned as shown. Then set the slots as follows:

- Set the `label` slot to `"Due Time"`.
- Set the `path` slot to `[pathExpr: kDataSymbol, 'dueDate]`. This `path` must evaluate to a slot in your data entry frame that contains a time (or a place to store one).

Registering Stationery for an Auto Part

When your stationery is implemented in an auto part, you are responsible for registering and removing it. The following code samples show `InstallScript` and `RemoveScript` functions that use the appropriate global functions to register and unregister the viewDef and dataDef files in your auto part as it is installed and removed, respectively. Note that the print format file is also registered as a viewDef with the system.

```
InstallScript: func(partFrame,removeFrame)
   begin
   RegDataDef(kDataSymbol, GetLayout("iouDataDef"));
   RegisterViewDef(GetLayout("iouDefaultViewDef"),
                                        kDataSymbol);
   RegisterViewDef(GetLayout("iouPrintFormat"),
                                        kDataSymbol);
   end;

RemoveScript: func(removeFrame)
   begin
   UnRegisterViewDef('default, kDataSymbol);
   UnRegisterViewDef('iouPrintFormat, kDataSymbol);
   UnRegDataDef(kDataSymbol);
   end;
```

Using the MinimalBounds ViewDef Method

The `MinimalBounds` method must be used in a viewDef when the size of the entry is dynamic, as it is in a paper-roll-style or page-style application. It's not necessary for a card-style application, which has a fixed height; in that case you should set a static height for your viewDef in the `viewDefHeight` slot.

The `MinimalBounds` method is used to compute the minimal size for the enclosing bounding box for the viewDef at run time. The following is an example of a `MinimalBounds` implementation where the viewDef contains a `newtEditView` whose `path` slot is set to `[pathExpr:kDataSymbol,'notes]`:

```
MinimalBounds: func(entry)
   begin
   local result := {left: 0, top: 0, right: 0,
                     bottom: viewDefHeight};

   // For an editView, make the bounds big enough to
   // contain all the child views.
   if entry.(kDataSymbol).notes then
      foreach item in entry.(kDataSymbol).notes do
         result := UnionRect( result, item.viewBounds );
   result;
   end;
```

Stationery Summary

Data Structures

ViewDef Frame

```
myViewDef := {
_proto: anyGenericView,
type: 'editor, // could also be 'viewer or a custom type
symbol: 'default, // required; identifies the view
name: string, // required; name of viewDef
version: integer, // required; should match dataDef
viewDefHeight: integer, // required, except in card-style
MinimalBounds:     // returns the minimal enclosing
   func(entry)..., //  bounding box for data
SetupForm:         // called by ViewSetupFormScript;
   func(entry, entryView)..., //   use to massage data
}
```

Protos

newtStationery

```
myDataDef := { // use to build a dataDef
_proto: newtStationery,
description: string,    ,  // describes dataDef entries
height: integer,  // required, except in card-style; should
               // match viewDefHeight
icon: resource,   // optional; used in header & New menu
name: string, // required; appears in New button picker
symbol: kAppSymbol, // required unique symbol
superSymbol: aSymbol, // identifies "owning" application
version: integer, // required; should match viewDef's version
FillNewEntry:      // returns a modified entry
   func(newEntry)...,
MakeNewEntry:      // used if FillNewEntry does not exist
   func()...,
StringExtract:     // creates string description
   func(entry, nLines)...,
TextScript:        // extracts data as text for routing
   func(fields, target)...,
}
```

newtStationeryPopupButton

```
aStatPopup := { // used to construct New and Show buttons
_proto: newtStationeryPopupButton,
form: symbol,     // 'viewDef or 'dataDef
symbols: nil,  // gathers all or specify:[uniqueSym,...]
text: string,     // text displayed in picker
types: [typeSym,...],// type slots of viewDefs
sorter: '|str<|,// sorted alphabetically by Sort function
shortCircuit: Boolean, // controls picker behavior
StatScript:         // called when picker item chosen
   func(stationeryItem)..., // define actions in this method
SetUpStatArray:// override to intercept picker items to
   func()...,   // be displayed
}
```

newtNewStationeryButton

```
aNewButton := { // the New button collects dataDefs
_proto: newtNewStationeryButton,
sorter: '|str<|,// sorted alphabetically by Sort function
shortCircuit: Boolean,// controls picker behavior
StatScript:         // called when picker item chosen
   func(stationeryItem)..., // define actions in this method
SetUpStatArray:// override to intercept picker items to
   func()...,   // be displayed
}
```

newtShowStationeryButton

```
aShowButton := { // the Show button collects viewDefs
_proto: newtShowStationeryButton,
types: [typeSym,...],// can specify type slots of viewDefs
sorter: '|str<|,// sorted alphabetically by Sort function
shortCircuit: Boolean,// controls picker behavior
StatScript:         // called when picker item chosen
   func(stationeryItem)..., // define actions in this method
SetUpStatArray:// override to intercept picker items to
   func()..., // be displayed
}
```

Case 1:13-cv-00919-LPS Document 311-5 Filed 03/10/21 Page 200 of 200 PageID #: 28844

newtRollShowStationeryButton

```
aRollShowButton := { // the Show button in paper roll apps
_proto: newtRollShowStationeryButton,
types: [typeSym,...],// can specify type slots of viewDefs
sorter: '|str<|,// sorted alphabetically by Sort function
shortCircuit: Boolean,// controls picker behavior
StatScript:       // called when picker item chosen
   func(stationeryItem)..., // define actions in this method
SetUpStatArray:// override to intercept picker items to
   func()...,   // be displayed
}
```

newtRollShowStationeryButton

```
anEntryShowButton := { // Show button in paperroll apps
_proto: newtEntryShowStationeryButton,
types: [typeSym,...],// can specify type slots of viewDefs
sorter: '|str<|,// sorted alphabetically by Sort function
shortCircuit: Boolean,// controls picker behavior
StatScript:       // called when picker item chosen
   func(stationeryItem)..., // define actions in this method
SetUpStatArray:// override to change entry displayed
   func()..., // can display different view for each
}
```

Functions

```
RegDataDef(dataDefSym, newDefTemplate)  // register dataDef
UnRegDataDef(dataDefSym)  // unregister dataDef
RegisterViewDef(viewDef, dataDefSym)// register viewDef
UnRegisterViewDef(viewDefSym, dataDefSym)//unregister viewDef
GetDefs(form, symbols, types)// returns view or data defs array
GetDataDefs(dataDefSym)// returns dataDef
GetAppDataDefs(superSymbol)// returns an app's dataDefs
GetEntryDataDef(soupEntry)  // returns the entry's dataDef
GetEntryDataView(soupEntry, viewDefSym)// returns the entry's
                                  // viewDef
GetViewDefs  (dataDefSym)   // returns viewDefs registered
                            // with the dataDef
GetDataView  (dataDefSym, viewDefSym)  // returns a specific
                  // viewDef of the dataDef
```


ARENDI-DEFS00003847