Newton Programmer's Guide: System Software

First Edition

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

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

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

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

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

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

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

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

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

11/95

Contents

Figures and Tables

About This Book

This two-volume set, *Newton Programmer's Guide: System Software*, is the definitive guide and reference for Newton programming. This set of volumes explains how to write Newton programs and describes the system software routines that you can use to do so. Note that communications programming topics are covered in a separate book, *Newton Programmer's Guide: Communications*.

Note

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

Audience

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

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*. Additionally, this set of volumes are a companion to the other volume in the set, *Newton Programmer's Guide: Communications*. You should refer to that book for details on NewtonScript communications programming topics.

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

Related Books

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

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

Sample Code

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

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

Conventions Used in This Book

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

Special Fonts

This book uses the following special fonts:

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

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

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

Tap Versus Click

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

Frame Code

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

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

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

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

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

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

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

Developer Products and Support

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

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

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

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

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

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.

CHAPTER 1

Overview

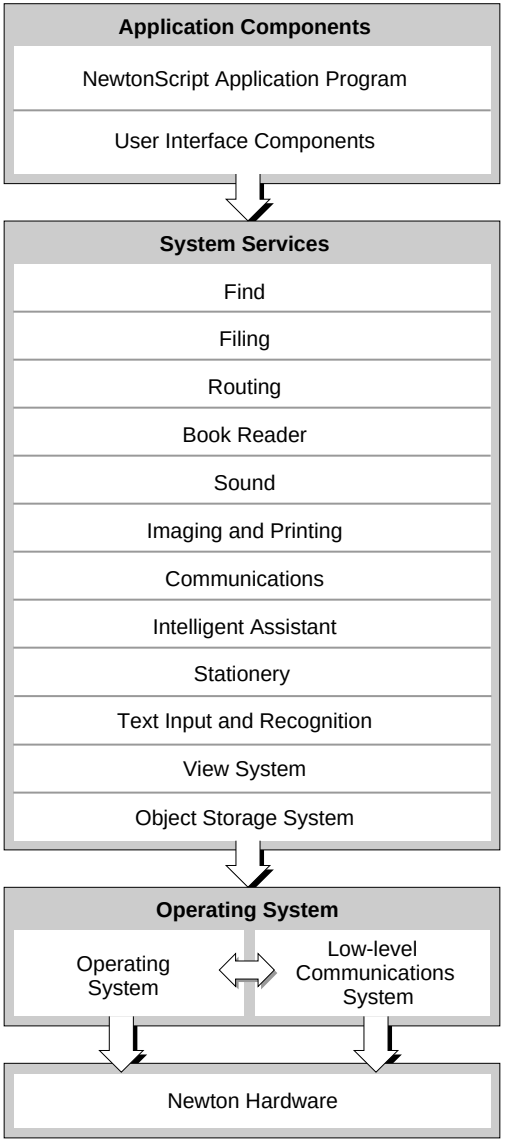
This chapter describes the general architecture of the Newton system software, which is divided into three levels, as shown in Figure 1-1.

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.

Figure 1-1 System software overview



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.

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.

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, routing data through a communication link, 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.

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

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 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.

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 10, “Recognition.”

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 17, “Intelligent Assistant.”

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 the chapter “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 2, "Routing Interface," in *Newton Programmer's Guide: Communications*.

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.

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 13, “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.

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 14, "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.”

Routing

The routing services allow users to send data and receive data from outside the system. These services include printing, faxing, mailing, beaming, duplicating, deleting, and moving data between internal and card stores. Users access the routing services through a standard user interface element called the Action button, which looks like a small envelope. Additionally, some routing services use the system Out Box to store and forward outgoing data. The Out Box and In Box provide a common user interface for all outgoing and incoming data in the system.

You may want to provide some or all of these standard routing capabilities in your application, or even add additional options that take advantage of special communication capabilities provided by your application. Also, you can add the capability to your application to automatically receive and operate on incoming data. For details on supporting the built-in routing services, refer to Chapter 2, “Routing Interface,” in *Newton Programmer’s Guide: Communications.*

For information about adding your own routing service, called a transport, to the system, refer to Chapter 3, “Transport Interface,” in *Newton Programmer’s Guide: Communications.*

Communications

The Newton provides a number of built-in communication services that are suited to real-time communication. They include:

- Synchronous and asynchronous serial connection

- Fax / data modem connection (V.34 with MNP / V.42 and fax Class 1—T30 / T4)
- Point-to-point infrared communication—called beaming (Sharp 9600 and Apple IR-enhanced protocols)
- AppleTalk ADSP protocol

The basis of the communications architecture is the **endpoint** object. The endpoint object encapsulates and manages the details of the specific connection. It includes methods that are executed in response to particular events, such as when a particular string is received.

A single endpoint prototype provides a standard interface to all communication services—serial, fax modem, beaming, and AppleTalk protocols, as well as third-party communication products. The endpoint object provides methods for

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

For details on how to use the Newton communications facilities, refer to Chapter 4, "Endpoint Interface," in *Newton Programmer's Guide: Communications*.

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 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 1-2. 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 1-2 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."

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" on page 1-10.

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 up to 7000% 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 pointers 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. 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 remote writing feature (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.

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.

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, and translate a number of minutes to duration strings ("16 hours", "40 minutes").

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.

CHAPTER 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 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` (page 2-11) or the proto `protoApp` (page 2-12) 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.

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, 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 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. 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 measure 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, including 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 reader. 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*.

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`). The software installed by an auto part remains in the Newton storage until it is explicitly removed by the user.

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 "Application Installation and Removal" beginning on page 2-5.) 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*.

Application Installation and Removal

When an application or auto part is installed on the Newton, the system executes a special function in the package: the `InstallScript` function. When a part is removed from the Newton, such as when a card is removed or the user deletes an application, the system executes another special function in the package: the `RemoveScript` function. In addition, the system may also execute a `DeletionScript` function, if the part was removed as a result of the user scrubbing it from the Extras Drawer. The following sections describe how to use these functions.

Installation

The `InstallScript` function (page 2-14) in a package is executed when an application or auto part is installed on the Newton or whenever the Newton is reset.

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

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 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. For many applications, this function is optional; for auto parts, this function is required. Applications built using the NewtApp framework require special `InstallScript` and `RemoveScript` functions. For details, see Chapter 4, "NewtApp Applications."

Removal

A package can be removed from the system in two ways: the user can delete it by scrubbing it from the Extras drawer, or if it is on a card, the card can be removed from the system. If the user deletes a package by scrubbing it from the Extras drawer, the system executes the `DeletionScript` function (page 2-14) in the package. Then, for applications and auto parts only, the system executes the `RemoveScript` function (page 2-15). If an application or auto part is removed as a result of card removal, only the `RemoveScript` function is executed.

For all types of parts, the `DeletionScript` function is optional. For applications, the `RemoveScript` function is optional; for auto parts, this function is required. However, note that automatically removed auto parts do not need the `RemoveScript` function since such auto parts are removed immediately after the `InstallScript` is executed—the `RemoveScript` is not executed.

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 3-36.

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 the application closes, it needs to clean up after itself as much as possible, removing its references to soups, entries or cursors. Again, this usually means setting to `nil` any references to these objects.

IMPORTANT

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

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.

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

Send this information to the e-mail address `NewtonSysOp@eworld.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 symbol 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 simply 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.

Reference

This section describes the protos and view classes mentioned in this chapter.

View Classes and Protos

clView

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

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

Slot descriptions

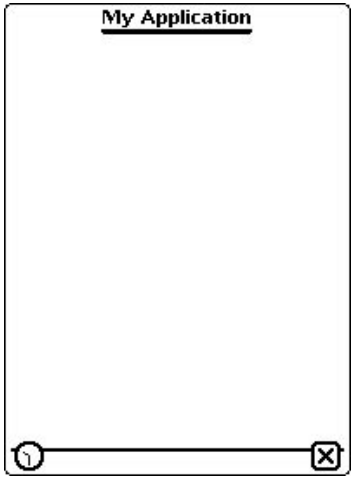
`viewBounds`	Set to the size and location where you want the view to appear.
`viewFlags`	The default setting is `vVisible`.
`viewFormat`	Optional. The default setting is `nil`.

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

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

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

protoApp

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



Slot descriptions

`title`	A string that is the title. This title appears in a title bar at the top of the view.
`viewBounds`	Set to the size and location where you want the view to appear. By default it is centered horizontally within its parent view.
`viewFlags`	The default setting is `vVisible + vApplication`. Do not change these flags, but you can add others if you wish.
`viewJustify`	Optional. The default setting is `vjParentCenterH`.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(1) + vfInset(1) + vfShadow(1)`.
`declareSelf`	Do not change. This slot is set by default to `'base`. This identifies the view to be closed when the user taps the close box.

The `protoApp` is based on a view of the `clView` class. It has the following two child views:

- A title, itself based on the `protoTitle` proto.
- A status bar, itself based on the `protoStatus` proto.

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

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

Functions and Methods

InstallScript

`InstallScript(`*partFrame*`, [`*removeFrame*`])`

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

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

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

DeletionScript

`DeletionScript()`

This function in the package is executed only when the part is deleted by the user (scrubbed out from the Extras drawer). This function applies to all types of package parts.

After the `DeletionScript` method is executed, the `RemoveScript` method is also executed.

RemoveScript

`RemoveScript(`*frame*`)`

This function in the application or auto part package is executed when the package is removed from the Newton (either deleted or if the PCMCIA card it is on is removed).

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

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

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 and Methods

Application-Defined Functions

```
InstallScript(partFrame, [removeFrame])
DeletionScript()
RemoveScript(frame)
```

CHAPTER 3

Views

This chapter describes the basic information that you need to know about views and provides information on 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 provides an introduction 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 on 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 3-2 on page 3-64.

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 is run on the Newton.

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 pieces of 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

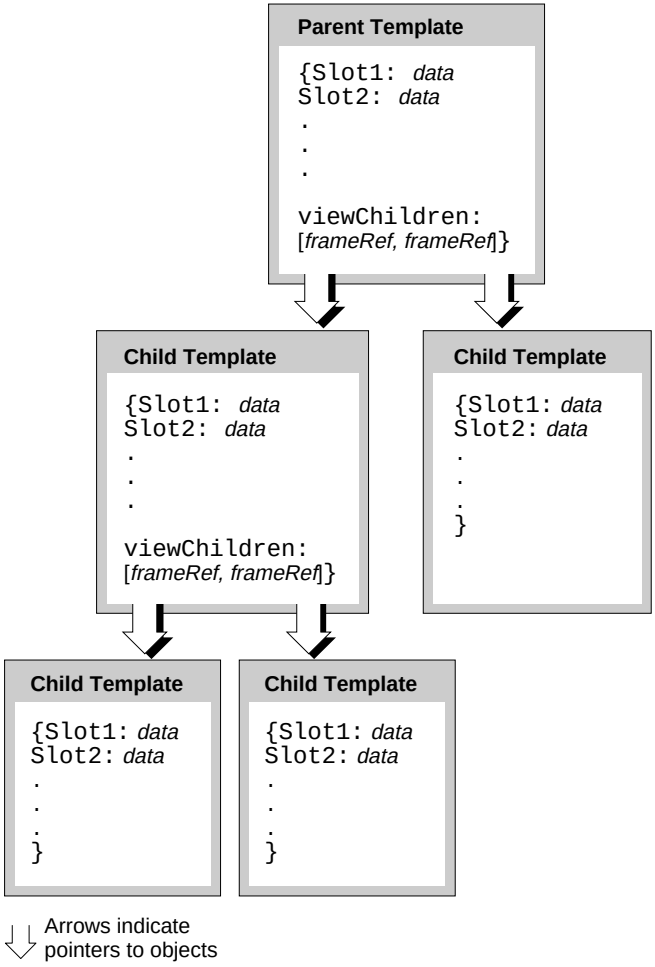
A template can also describe non-graphic objects like communication objects. Such objects have no visual representation and exist only as logical objects. For more information about communication objects, refer to the *Newton Programmer's Guide: Communications 2.0.* ◆

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 on page 3-4 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



Views

A template is a data description of an object. A **view** is the visual representation of that 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 like 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, any data written into them that needs to be saved permanently must be saved before the view disappears, since data is disposed of when the view is closed.

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

Figure 3-2 on page 3-7 shows the set of views that have been 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, often multiple views share a single template. Also, this example doesn't show templates that are built on other protos.

Figure 3-2 View hierarchy

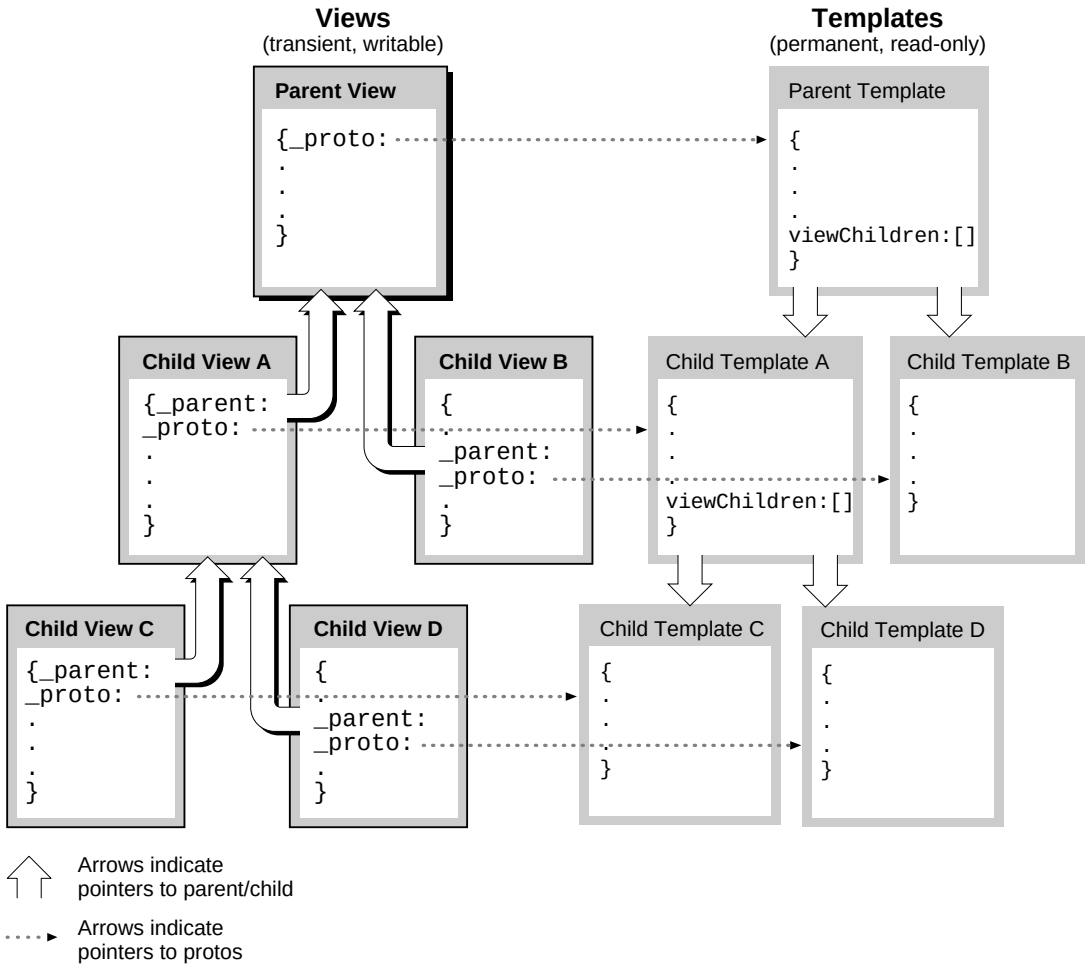
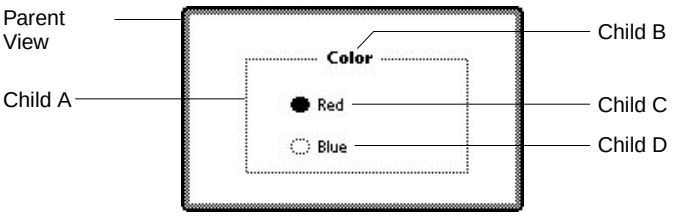


Figure 3-3 shows an example of what this view hierarchy might represent on the screen.

Figure 3-3 Screen representation of view hierarchy

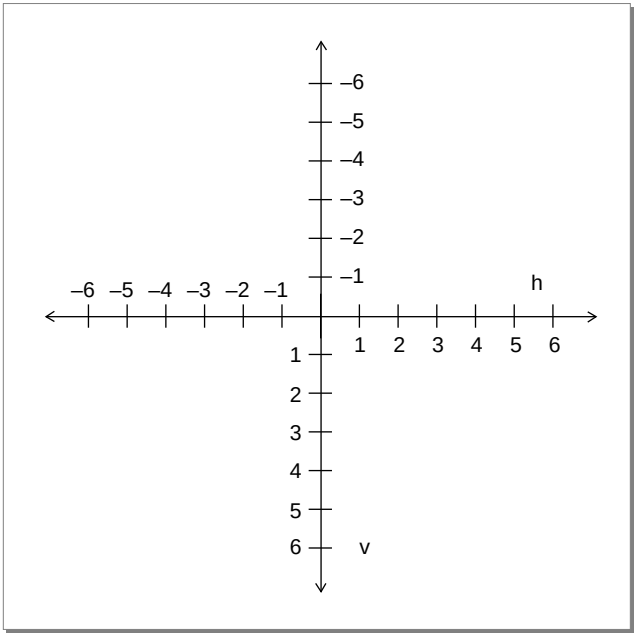


The application base view of each application exists as a child of the system **root view.** The root view is essentially the blank screen that exists before any other views are drawn. It is the ancestor of all other views that are instantiated.

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 on page 3-9 illustrates the view system coordinate plane.

Figure 3-4 View system coordinate plane

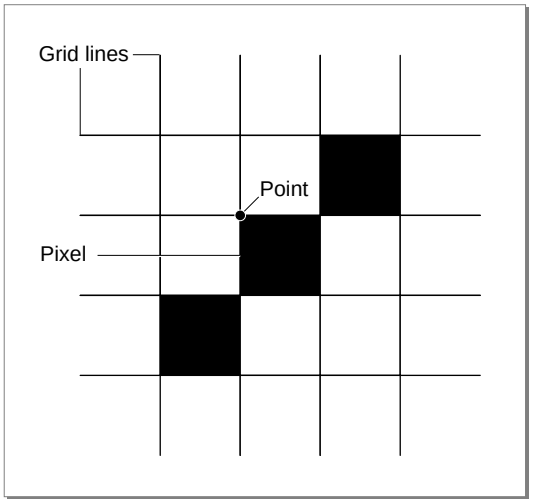


Views are defined by rectangular areas that are usually subsets of the screen. The origin of a view is usually its upper-left corner, though the origin of a view can be changed. The coordinates of a view are relative to the origin of its parent view—they are not screen coordinates.

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.

Figure 3-5 Points and pixels



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",

viewSetupDoneScript: func()
   :UpdateDisplay(),

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

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 3-2 on page 3-64.

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
```

Some of the `viewFlags` constants are listed and described in Table 3-3 on page 3-67. 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 Chapter 10, "Recognition."

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” on page 3-127.”

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 Chapter 10, “Recognition.” 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` which skips `ViewClickScript` but sends all other scripts.

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 Chapter 10, “Recognition.”

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 shapes 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
}
```

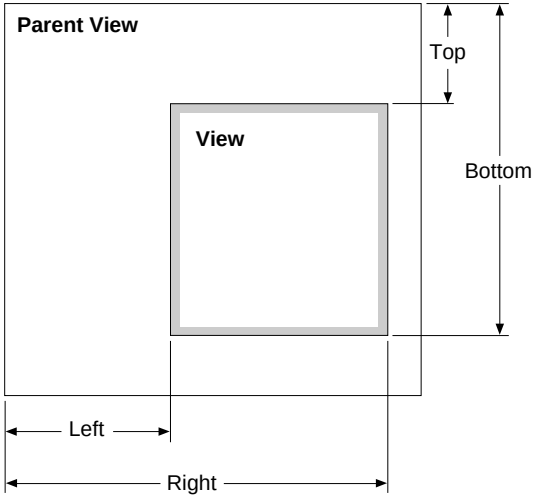
leftValue	The distance from the left origin of the parent view to the left edge of the view.
topValue	The distance from the top origin of the parent view to the top edge of the view.
rightValue	The distance from the left origin of the parent view to the right edge of the view.
bottomValue	The distance 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 how the `viewBounds` values are used. For more information, see the section "View Alignment" beginning on page 3-18. ◆

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 later in this chapter in the section "Finding the Bounds of Views" beginning on page 3-53.

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 should also 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, described on page 3-91. 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 Chapter 20, "Utility Functions," 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 size your application to the full extents of the screen. This might look odd if your application was run 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. If you work with global coordinates in your application, make sure to check its current location. To do this, send the `GlobalBox` (page 3-95) 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 its 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 that 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.

If you want to restrict a view to accepting a single character only, you can use `oneWordOnly` and use a custom dictionary of single letters.

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 is used 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.

continued

Table 3-1 `viewJustify` constants (continued)

Constant	Value	Description
`vjSiblingFullH`	1536	The left bounds value is used as an offset from the left edge of the sibling and the right bounds value is used as an offset from the right edge of the sibling (for example, specifying left = 10 and right = –10 indents the view 10 pixels on each side relative to its sibling).
Vertical alignment of the view relative to its parent or sibling view[3]		
`vjParentTopV`	0	The top and bottom view bounds are relative to the parent's top side (default).
`vjParentCenterV`	64	The difference between the top and bottom view bounds is used as the height of the view. If you specify zero for top, the view is centered in the parent view. If you specify any other number for top, the view is offset by that much from a centered position (for example, specifying top = –10 and bottom = height–10 offsets the view 10 pixels above a centered position).
`vjParentBottomV`	128	The top and bottom view bounds are relative to the parent's bottom side.
`vjParentFullV`	192	The top bounds value is used as an offset from the top edge of the parent and the bottom bounds value is used 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.

continued

Table 3-1 `viewJustify` constants (continued)

Constant	Value	Description
`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 is used as an offset from the bottom edge of the sibling (for example, specifying top = 10 and bottom = –10 indents the view 10 pixels on both the top and the bottom sides relative to its sibling).
Text limits		
`noLineLimits`	0	(Default) No limits, text wraps to next line.
`oneLineOnly`	8388608	Allows only a single line of text, with no wrapping.
`oneWordOnly`	16777216	Allows only a single word. (If the user writes another word, it replaces the first.)
Indicate that a bounds value is a ratio		
`vjNoRatio`	0	(Default) Do not use proportional alignment.
`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.

continued

Table 3-1 `viewJustify` constants (continued)

Constant	Value	Description
`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.

Figure 3-7 View alignment effects

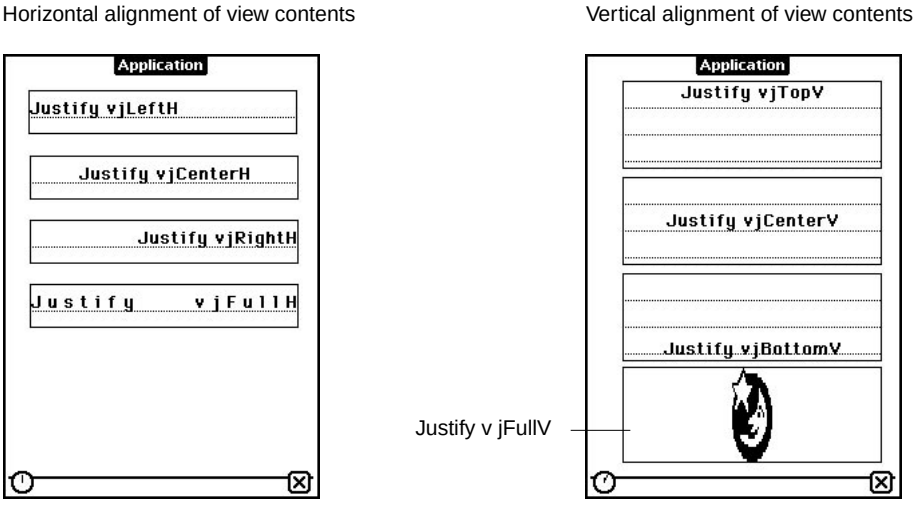


Figure 3-7 View alignment effects (continued)

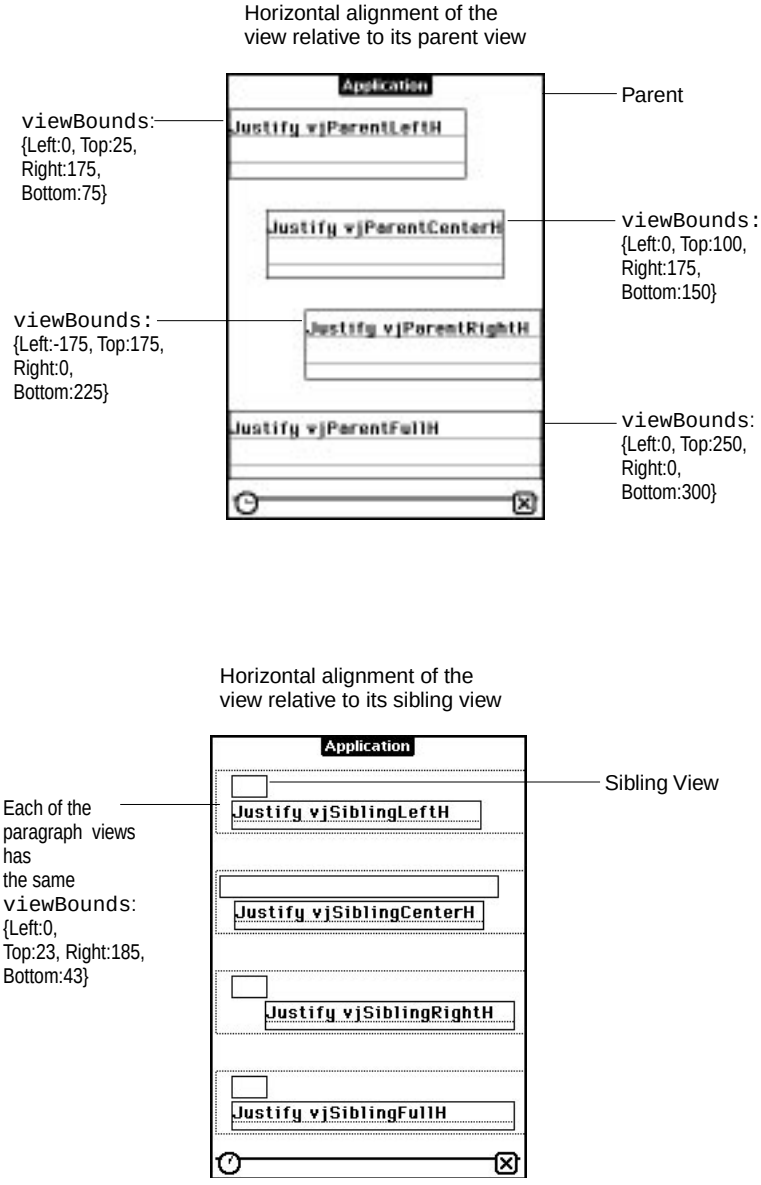
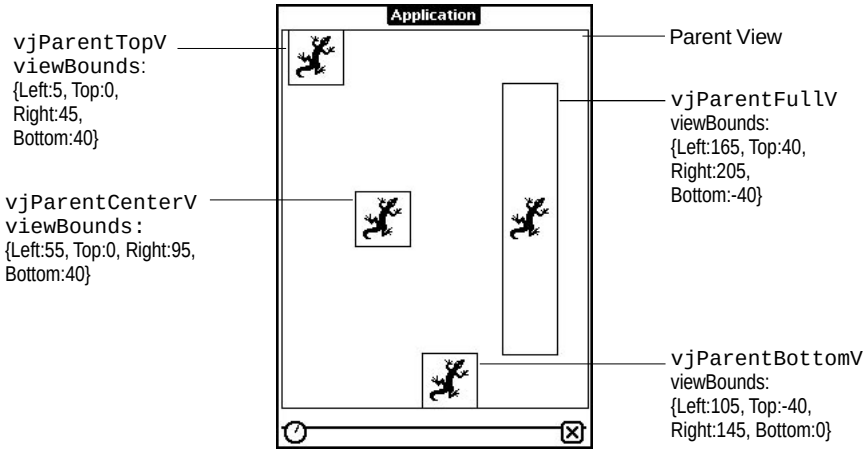
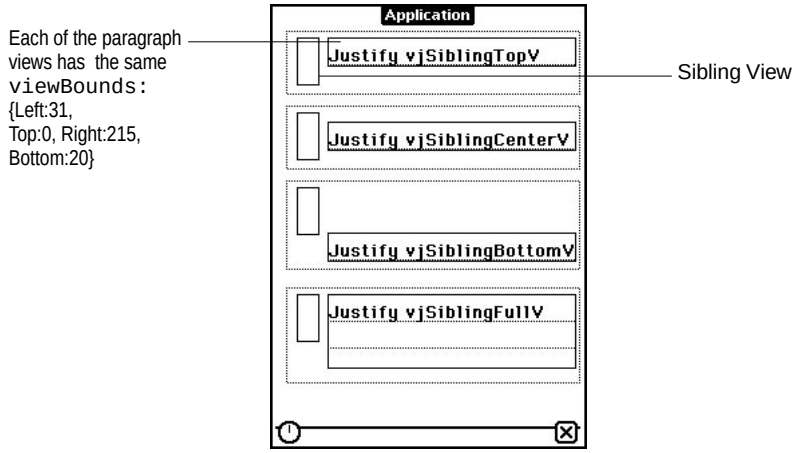


Figure 3-7 View alignment effects (continued)

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 the section "Scrolling View Contents" beginning on page 3-56.

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 view bounds.

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 are drawn and the frame is drawn after the contents are drawn.

IMPORTANT

Many views need no fill pattern, 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 the view may be explicitly dirtied (in need of redrawing) and if you don’t need a transparent view. This will increase 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 will automatically appear to be filled with white. ◆

The view format attributes are listed and described in Table 3-4 on page 3-69.

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 3-4 on page 3-69, 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”:

```
DefConst('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 using hexadecimal character codes whose binary representation is used to create the pattern. Each two-digit hex code creates one byte of the pattern.

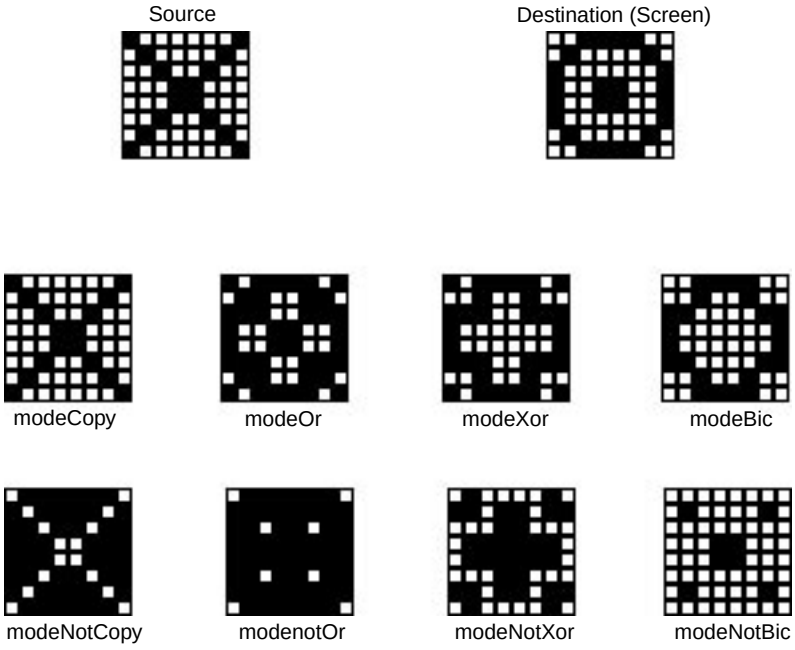
Drawing Transfer Mode for Views

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 3-5 on page 3-71.

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, `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.

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 the section "Animating Views" beginning on page 3-54.

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 3-6 on page 3-73. 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 this 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 3-6 on page 3-73 lists all of 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 different 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 “Specifying a Font” in Chapter 8, “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 simply 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.

`_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 this slot or create a new one with this name. If you do so, you may be inadvertently overriding the children of a system proto.

Here is the reason for the dual child view slots. 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 you change the `viewChildren` slot (for example, to add your own child templates programmatically), then

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 of 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 should use the `stepChildren` slot to do this. If there isn't a `stepChildren` slot, create one and put your views there.

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 and 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 the section "Getting References to Views" beginning on page 3-44. ▲

Application–Defined Methods

As your application executes, it receives messages from the system which 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 of the events 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” beginning on page 3-127.

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.

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 the view 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 to 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, the view 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 a slot in the place you declare the view. 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.

NTK always automatically declares the base view of your application 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.

For a more detailed technical description of the inner workings of declaring a view, see Appendix B, "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 and contains 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 non-declared 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. This is because it is declared in the root view, which is always open in the Newton system.
2. 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.
3. 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.
4. 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`.
5. The `ViewSetupDoneScript` message is sent to the view.
6. The view is displayed if its `vVisible` `viewFlags` bit is set.
7. 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.)
8. 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 of the 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, then 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.

New Drag and Drop API

A drag and drop API has been added. This API now lets users drag a view and drop it into another view. See 3-54 and 3-146 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. See 3-92 for details.
- `DirtyBox`– marks a portion of a view (or views) as needing redrawing. See 3-96 for details.
- `GetDrawBox`–returns the bounds of the area on the screen that needs redrawing. See 3-97 for details.
- `GlobalOuterBox`– returns the rectangle, in global coordinates, of the specified view, including any frame that is drawn around the view. See 3-95 for details.
- `ModalConfirm`– creates and displays a slip. See 3-92 for details.
- `MoveBehind`– moves a view behind another view, redrawing the screen as appropriate. See 3-86 for details.
- `StdButtonWidth`–returns the size that a button needs to be in order to fit some specified text. See 3-98 for details.

New Messages

The following message has been added.

- `ReorientToScreen`–This message is sent to each child of the root view when the screen orientation is changed. See 3-134 for details.
- `ViewPostQuitScript`– This message is sent to a view following the `ViewQuitScript` message and after all of the view's child views have been destroyed. See 3-130 for details.

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. See 3-19 for details.

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). See 3-19 for details.

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. See 3-88 for details.
- `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. See 3-85 for details.
- `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. See 3-95 for details.
- `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. See 3-96 for details.
- `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. See 3-129 for details.

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 plain a bad idea. See page 3-147 for details.

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 “View Reference” on page 3-63 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 should use the `ChildViewFrames` (page 3-78) and `Parent` (page 3-78) methods as well as the `GetRoot` (page 3-79) and `GetView` (page 3-79) functions to return references to these “related” views.

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 be later in the list. Views with the `vFloating` flag (which always appear above non-floating views) will be at the end of the array.

To test whether an application is open or not, you can use the `GetRoot` function, together with the application symbol:

```
GetRoot().appname:Visible()
```

Displaying, Hiding, and Redrawing Views

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

- `Open` (page 3-80)—to open the view
- `Toggle` (page 3-82)—to open or close the view

- `Show` (page 3-82)—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` (page 3-81)—to hide and delete it from memory
- `Toggle`—to close or open the view
- `Hide` (page 3-83)—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` (page 3-83)—flags the view as "dirty" so it is redrawn during the next system idle loop
- `RefreshViews` (page 3-84)—redraws all dirty views immediately
- `SetValue` (page 3-85)—sets the value of a slot and dirties the view
- `SyncView` (page 3-86)—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` (page 3-128) method.
- Create the template and the view at run time by using the `AddStepView` (page 3-87) function.
- Create the template and the view at run time by using the `BuildContext` (page 3-91) 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 will accomplish 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` (page 3-66) 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" beginning on page 3-36.

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` (page 3-127), `ViewSetupChildrenScript`, and `ViewSetupDoneScript` (page 3-128).

Adding to the stepChildren Array

If it is not possible to define the template for a view at compile time, then 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 would first define the view template as a frame. Then, in the `ViewSetupChildrenScript` method of its parent view, you would 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 if 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.

If at some point after the child views have been created you want to modify the contents of the `stepChildren` array and then build new child views from it, you can use the `RedoChildren` (page 3-118) 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 how you can reorder the child views of a view dynamically.

For details on an easy way to create a template dynamically, see the subsection "Creating Templates" beginning on page 3-49.

Using the AddStepView Function

If you need to create a template and add a view yourself at run time, use the function `AddStepView` (page 3-87). 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 in RAM, or `AddStepView` will fail. See the code in the previous section for an example of how to ensure that the `stepChildren` array is in RAM.

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` (page 3-83) 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` (page 3-82) 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, then you should instead be using 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.

To remove a view created by `AddStepView`, use the `RemoveStepView` (page 3-88) 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 the subsection "Creating Templates" beginning on page 3-49.

Using the BuildContext Function

Another function that is occasionally useful is `BuildContext` (page 3-91). It takes one parameter, a template. It makes a view from the template and returns it. The view's parent will be 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 would need to use `BuildContext` to create the slip.

`BuildContext` is also useful if you need to create 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 following subsection, "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.

An easy 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. At run time you can access the user proto from your code using the identifier `PT_`*filename*.

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`, then you could create the template you need using code like this:

```
template := {viewBounds: RelBounds(x, y, width, height),
            _proto: PT_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 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);
```

You should also note that for creating views arranged in a table, there is a function called `LayoutTable` which handles calculating 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` (page 6-138). 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 Chapter 6, “Pickers, Pop-up Views, and Overviews,” for information on `protoPicker` and `PopupMenu`.

Changing the Values in viewFormat

You can change the values in the `viewFormat` (page 3-69) slot of a view without closing and reopening a view. Use the `SetValue` (page 3-85) 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, 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 (page 3-115). You must call this function in combination with the HiliteOwner (page 3-114) function. When you call GetHiliteOffsets, it will return an array of arrays. Each item in the outer array represents selected subviews, as in:

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

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. clPolygonViews will not be 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 (page 3-64) 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.

Complex View Effects

If you have an application that uses `ViewQuitScript` (page 3-129) 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` (page 3-131) instead of `ViewCloseScript`. Second, remove the view's `ViewEffect` and manually force the effect at the end of `ViewQuitScript` using the `Effect` (page 3-98) 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. (The user can write outside the modal view, however.)

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` (page 3-92) to create and displays a slip that the user must dismiss before continuing. The slip is created at a deferred time, so the call to `AsyncConfirm` returns immediately allowing the currently executing NewtonScript code to finish. 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` (page 3-93) or `ModalDialog` (page 3-94) 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` (page 3-94)— Calculates the right and bottom values of a view and returns a a bounds frame.
- `SetBounds` (page 3-95)—Returns a frame when the left, top, right, and bottom coordinates are given.
- `GlobalBox` (page 3-95)—Returns the rectangle, in coordinates, of a specified view.
- `GlobalOuterBox` (page 3-95)—Returns the rectangle, in coordinates, of a specified view including any frame that is drawn around a view.
- `LocalBox` (page 3-96)—Returns a frame containing the view bounds relative to the view itself.
- `MoveBehind` (page 3-86)— Moves a view behind another view.
- `DirtyBox` (page 3-96)— Marks a portion of a view as needing redrawing.

Compile-time functions:

- `GetDrawBox` (page 3-97)— Returns the bounds of an area on the screen that needs redrawing.
- `ButtonBounds` (page 3-97)— Returns a frame when supplied with the width of a button to be placed in the status bar.
- `PictBounds` (page 3-98)— Finds the width and height of a picture and returns the proper bounds frame.

Animating Views

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

- `Effect` (page 3-98) (performs any animation that can be specified in the `viewEffect` slot)
- `SlideEffect` (page 3-100) (slides a whole view or its contents up or down)
- `RevealEffect` (page 3-102) (slides part of a view up or down)
- `Delete` (page 3-105) (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, and they do not 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` (page 3-106) message.

Dragging and Dropping a View

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 type of objects that can be dropped into a view, if any. Examples of types of objects include `'text` or `'picture`. See `dragInfo` on page 3-105 for a complete description of the slots. You then call `DragAndDrop`'s view method message and pass this frame to it.

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 will accept a drag and drop message, or 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` (page 3-142)— Sent to the destination view. It 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` (page 3-144)—Sent to the source view. It is sent when the destination view is found.
- `ViewDropScript` (page 3-145)— Sent to the destination view. You must add the object to the destination view.
- `ViewDropMoveScript` (page 3-146)— 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` (page 3-146)— 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` (page 3-141)—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` (page 3-142)—Sent to the source view. It draws the image that will appear behind the dragged image.
- `ViewFindTargetScript` (page 3-142)—Sent to the destination view. It lets the destination view change the drop point to a different view.

- `ViewDragFeedbackScript` (page 3-143)—Sent to the destination view. It provides visual feedback while items are dragged.
- `ViewDropDoneScript` (page 3-147)— 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` (page 3-108) method to perform this kind of scrolling. The `SetOrigin` 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` (page 3-111) 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 this latter kind of scrolling, the child views are simply redrawn within the parent view by changing their view bounds. 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 the subsection “Scrolling” beginning on page 3-62, and “Optimizing View Performance” beginning on page 3-59.

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 will be returned if you call `GetView('viewFrontMost)` (page 3-79).

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` (page 3-113) message.

To highlight or unhighlight a single view from a group, send the view the `HiliteUnique` (page 3-113) 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` (page 3-113) message. The view is unhighlighted when the pen moves outside the view bounds or when it is lifted. If the view is a button, you can send the view the `TrackButton` (page 3-114) message to accomplish the same task. The view is also sent the `ButtonClickScript` message if the pen is lifted inside the button.

To get the view containing highlighted data, you can call the global function `HiliteOwner` (page 3-114), and to get the highlighted text you can use `GetHiliteOffsets` (page 3-115).

To highlight some or all of the text in a paragraph, you can use the `SetHilite` (page 3-115) method.

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(GetDynamicValue(getroot(),'viewflags,nil),vSelected) <> 0` is true, the view is selected.

Creating View Dependencies

You can make one view dependent upon another by using the global function `TieViews` (page 3-117). 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. However, both views must be declared in a common view in their parent chain. This is required so that the views can access each other.

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 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` (page 3-120). 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` (page 3-121).

To lay out a view containing a vertical column of child views, send the view the message `LayoutColumn` (page 3-125).

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 you can use to 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 will increase drawing performance and reduce the system overhead used for each view you create. The drawing functions are described in Chapter 12, “Drawing and Graphics.”

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 the view may be individually dirtied and if you don't need a transparent view. This will increase 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.

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, for example, if you call the `SetValue` (page 3-85) function for a view. All dirty views are redrawn the next time the system idle loop executes. Often this redrawing speed is sufficient since the system idle 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` (page 3-84). 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 non-dirty 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` (page 3-83) 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` (page 3-131) or `ViewQuitScript` methods, and then reload these slots with data when the view is shown or opened again, using the `ViewShowScript` (page 3-131) 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, refer to the section "View Instantiation" beginning on page 3-36. For information on storing data in soups, refer to 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 the subsection "View Fill" beginning on page 3-60.

View Reference

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

Constants

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

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

Class Constants

The view class constants are listed and described in Table 3-2.

Table 3-2 View class constants

Constant	Value	Description
`clView`	74	The base view class. This class is used for a generic view that has no special characteristics. A view of this class is generally a container view that encloses other more specialized views. Such a high-level view would include global data and methods shared by its child views. See Chapter 2, "Getting Started," for more information on this constant.
`clPictureView`	76	Used for pictures. See Chapter 12, "Drawing and Graphics," for more information on this constant.
`clEditView`	77	Used for editing views that can accept both text and graphic user input. This view class typically has child views that are of class `clParagraphView` and `clPolygonView`. See Chapter 8, "Text and Ink Input and Display," for more information on this constant.
`clParagraphView`	81	A static or editable text view. When text is recognized, it is displayed in one of these views. Text is grouped into paragraphs so that many words can be shown in a single paragraph view. See Chapter 8, "Text and Ink Input and Display," for more information on this constant.

Table 3-2 View class constants (continued)

Constant	Value	Description
`clPolygonView`	82	A graphic view used in an edit view. When a shape is recognized, it is displayed in one of these graphic views. See Chapter 12, “Drawing and Graphics,” for more information on this constant.
`clKeyboardView`	79	Used to define keyboard-like arrays of buttons that can be tapped. No other forms of input recognition are available. See Chapter 8, “Text and Ink Input and Display,” for more information on this constant.
`clMonthView`	80	Used to define a calendar view of a month that lets the user select a date range. See Chapter 6, “Pickers, Pop-up Views, and Overviews,” for more information on this constant.
`clRemoteView`	88	Used for a view that displays another view as its contents. This can be used to show a page preview of a full-page view, for example. This view provides the scaling necessary to display the entire remote view. See Chapter 12, “Drawing and Graphics,” for more information on this constant.
`clPickView`	91	Used to display a list from which you can pick an item. The list can display both text and graphic items. This view class is supported through the `protoPicker` view proto. See Chapter 6, “Pickers, Pop-up Views, and Overviews,” for more information on this constant.

continued

Table 3-2 View class constants (continued)

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

viewFlags Constants

The following `viewFlags` constants are listed and described in Table 3-3. Several additional constants can be specified in the `viewFlags` slot that control what kinds of pen input (taps, strokes, words, letters, numbers, and so on) are recognized and handled by the view. These other constants are described in Chapter 8, “Text and Ink Input and Display.”

Table 3-3 `viewFlags` constants

Constant	Value	Description
`vVisible`	1	The view is visible. Usually used, except for invisible views. (Don't set this flag for your application base view, because you don't want it to be shown until the user taps its icon in the Extras Drawer.) If you Show, Hide, or Toggle a view, this flag is changed in the view by the system to reflect the current state of the view.
`vApplication`	4	Identifies a view that should receive scrolling and other high-level events. For example, when the user taps on the scroll arrows, the system searches all of the views to find the frontmost view that has this bit set, and then sends the scroll event to that view. Generally, this flag is set for the application base view. Views with this flag set can be found with the special view symbols `'viewFrontMost` or `'viewFrontMostApp`.
`vCalculateBounds`	8	The view bounds are not fixed, but are recalculated and will grow if the user enters more information than the view can hold. Used by views of the class `clParagraphView` and `clPolygonView` only, and only when they are enclosed in a view of the class `clEditView`.
`vClipping`	32	The view's contents, including child views, are clipped to its bounds when it is drawn. Note that the base view of all applications is automatically clipped, whether or not this flag is set.
`vFloating`	64	The view is a floating view; that is, it floats above its sibling views. A view without this flag will never come in front of a floating sibling view.

continued

Table 3-3 `viewFlags` constants (continued)

Constant	Value	Description
`vReadOnly`	2	The view cannot be changed, except that it can be scaled or distorted. It is read-only.
`vWriteProtected`	128	The same as `vReadOnly`, except that this flag propagates automatically to all of the view's child views. Additionally, scaling and distortion of the view are not allowed.
`vNoScripts`	134217728	Prevents the system from sending to the view any of the system messages described "Application–Defined Methods" on page 3-127 (except for the `ViewChangedScript, and ViewSetupFormScript` messages, which are still sent). Setting this flag speeds up the processing for a view if it has no application–defined handling methods, because the system won't bother trying to send it messages. This flag is set internally for views of the classes `clParagraphView` and `clPolygonView` that are created dynamically as the user writes in a `clEditView`.
`vClickable`	512	Allows the view to receive pen input. The system sends the `ViewClickScript` message to the view once for each pen tap (click) that occurs within the view. Refer to Chapter 8, "Text and Ink Input and Display," for more information.
`vNoFlags`	0	There are no flag attributes for the view.

viewFormat Constants

The view format constants are listed and described in Table 3-4.

Table 3-4 `viewFormat` constants

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

continued

Table 3-4 `viewFormat` constants (continued)

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

continued

Table 3-4 `viewFormat` constants (continued)

Constant	Value	Description
View line style (for `clEditView` and `clParagraphView` view classes only)		
`vfLinesWhite`	4096	Draw horizontal lines in white
`vfLinesLtGray`	8192	Draw widely dotted horizontal lines
`vfLinesGray`	12288	Draw dotted horizontal lines
`vfLinesDkGray`	16384	Draw dashed horizontal lines
`vfLinesBlack`	20480	Draw solid black horizontal lines
`vfLinesCustom`	57344	Use the custom line pattern specified in the `viewLinePattern` slot

viewTransferMode Constants

The constants that you can specify for the `viewTransferMode` slot are listed and described in Table 3-5.

Table 3-5 `viewTransferMode` constants

Constant	Value	Description
`modeCopy`	0	Replaces the pixels in the destination with the pixels in the source, “painting” over the screen without regard for what’s already there.
`modeOr`	1	Replaces screen pixels under the black part of the source image with black pixels. Screen pixels under the white part of the source image are unchanged.
`modeXor`	2	Inverts screen pixels under the black part of the source image. Screen pixels under the white part of the source image are unchanged.

continued

Table 3-5 `viewTransferMode` constants (continued)

Constant	Value	Description
`modeBic`	3	Erases screen pixels under the black part of the source image, making them all white. Screen pixels under the white part of the source image are unchanged.
`modeNotCopy`	4	Replaces screen pixels under the black part of the source image with white pixels. Screen pixels under the white part of the source image are made black.
`modeNotOr`	5	Screen pixels under the black part of the source image are unchanged. Screen pixels under the white part of the source image are made black.
`modeNotXor`	6	Screen pixels under the black part of the source image are unchanged. Screen pixels under the white part of the source image are inverted.
`modeNotBic`	7	Screen pixels under the black part of the source image are unchanged. Screen pixels under the white part of the source image are made white.
`modeMask`	8	This is a special transfer mode used for drawing views of the `clPictureView` class only. It causes the picture mask image to be erased first and then the picture bit image is drawn over it using the `modeOr` transfer mode.

viewEffect Constants

Table 3-6 lists all of the constants that you can use in the `viewEffect` slot to create custom animation effects.

Table 3-6 `viewEffect` constants

Constant	Value	Description
`fxSteps(`*x*`)`	(*x*-1)* 2097152	Sets the number of steps (*x*) that the animation should take to complete. Specify an integer from 1 to 15.
`fxStepTime(`*x*`)`	*x**4194304	Sets the amount of time (*x*) to take for each animation step, in ticks. There are 60 ticks per second, or 16.6 milliseconds per tick. Specify an integer from zero to 15.
`fxColumns(`*x*`)`	*x*-1	Sets the number (*x*) of columns in which to divide the view for animation purposes.
`fxRows(`*x*`)`	(*x*-1)*32	Sets the number (*x*) of rows in which to divide the view for animation purposes.
`fxMoveH`	65536	Indicates that you want the animation to include horizontal movement. (Note that you can also specify `fxMoveV`.)
`fxHStartPhase`	1024	If specified, indicates that you want the first column to begin moving towards the left. If not specified, the first column begins moving towards the right. This flag can be used only if `fxMoveH` is specified.
`fxColAltHPhase`	4096	If specified, the direction of horizontal movement alternates for each column in the view. If not specified, all columns move in the same direction (left or right) as the first column. This flag can be used only if `fxMoveH` is specified.

continued

Table 3-6 `viewEffect` constants (continued)

Constant	Value	Description
`fxRowAltHPhase`	16384	If specified, the direction of horizontal movement alternates for each row in the view. If not specified, all rows move in the same direction (left or right) as the first row. This flag can be used only if `fxMoveH` is specified.
`fxMoveV`	131072	Indicates that you want the animation to include vertical movement. (Note that you can also specify `fxMoveH`.)
`fxVStartPhase`	2048	If specified, indicates that you want the first row to begin moving upwards. If not specified, the first row begins moving downwards. This flag can be used only if `fxMoveV` is specified.
`fxColAltVPhase`	8192	If specified, the direction of vertical movement alternates for each column in the view. If not specified, all columns move in the same direction (up or down) as the first column. This flag can be used only if `fxMoveV` is specified.
`fxRowAltVPhase`	32768	If specified, the direction of vertical movement alternates for each row in the view. If not specified, all rows move in the same direction (up or down) as the first row. This flag can be used only if `fxMoveV` is specified.
`fxLeft`	66560	Indicates that motion should be towards the left. (This flag is the same as specifying `fxHStartPhase+fxMoveH`.)
`fxRight`	65536	Indicates that motion should be towards the right. (This flag is the same as specifying `fxMoveH` and not specifying `fxHStartPhase`.)
`fxUp`	133120	Indicates that motion should be towards the top. (This flag is the same as specifying `fxVStartPhase+fxMoveV`.)

continued

Table 3-6 `viewEffect` constants (continued)

Constant	Value	Description
`fxDown`	131072	Indicates that motion should be towards the bottom. (This flag is the same as specifying `fxMoveV` and not specifying `fxVStartPhase`.)
`fxRevealLine`	262144	If specified, causes a line to be drawn at the edge(s) from which the animation is being revealed. For some types of animation, this setting improves the effect.
`fxWipe`	524288	If specified, causes the view to be revealed in place rather than actually moved into place. In other words, the view is revealed just like a window is revealed by rolling a shade away. Without this flag, the view is actually moved into place.
`fxFromEdge`	1048576	If specified, causes the animation to begin at the edge of the screen, ending up at the ultimate view location. Without this flag, the entire animation occurs within the bounds of the view being animated.
`fxCheckerboardEffect`	155879	Reveals a view using a checkerboard effect, where adjoining squares move in opposite (up and down) directions.
`fxBarnDoorOpenEffect`	627713	Reveals a view from center towards left and right edges, like a barn door opening where the view is the inside of the barn.
`fxBarnDoorCloseEffect`	626689	Reveals a view from left and right edges towards the center, like a barn door closing where the view is painted on the doors.

continued

Table 3-6 `viewEffect` constants (continued)

Constant	Value	Description
`fxVenetianBlindsEffect`	131296	Reveals a view so that it appears behind venetian blinds that open.
`fxIrisOpenEffect`	1023009	Changes the size of an invisible "aperture" covering the view, revealing an ever-increasing portion of the full-size view as the aperture opens.
`fxIrisCloseEffect`	986145	Like `fxIrisOpenEffect`, except this decreases the size of an invisible "aperture" covering the view, as the aperture closes.
`fxPopDownEffect`	393216	Reveals a view as it slides down from its top boundary.
`fxDrawerEffect`	133120	Reveals a view as it slides up from its bottom boundary.
`fxZoomOpenEffect`	236577	Expands the image of the view from a point in the center until it fills the screen; that is, the entire view appears to grow from a point in the center of the screen.
`fxZoomCloseEffect`	199713	Opposite of `fxZoomOpenEffect`. This value shrinks the image of the view from a point in the center until it disappears or closes on the screen.

continued

Table 3-6 `viewEffect` constants (continued)

Constant	Value	Description
`fxZoomVerticalEffect`		
	165920	The view expands out from a horizontal line in the center of its bounds. The top half moves upward and lower half moves downward.

Functions and Methods

The following sections describe view functions and methods. This section contains the following topics:

- Functions and Methods to Get References to Views
- Functions and Methods to Display, Hide, and Redraw Views
- Functions to Dynamically Add Views
- Methods to Make Modal Views
- Methods and Functions to Find the Bounds of Views
- Methods to Animate Views
- Methods to Drag a View
- Methods and Messages to Drag and Drop a View
- Methods to Scroll View Contents
- Methods to Highlight a View
- Functions to Create View Dependencies
- Methods to Synchronize Views
- Methods to Lay Out Multiple Child Views
- Miscellaneous View Operations
- Application Defined Methods
- Drawing-Related Messages
- Scrolling and Overview Messages
- Other Messages

Getting References to Views

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

ChildViewFrames

view:`ChildViewFrames()`

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

IMPORTANT

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

Parent

view:`Parent()`

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

GetRoot

`GetRoot()`

Returns the system root view.

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

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

If the application is open, this function will return a non-`nil` value; otherwise, `nil` is returned. This reference is always present as long as a view is open, and never present when a view is closed.

GetView

`GetView(`*`view`*`)`

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

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

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

Displaying, Hiding, and Redrawing Views

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

Open

view:Open()

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

The view receives the following system messages: ViewSetupFormScript, ViewSetupChildrenScript, ViewSetupDoneScript, ViewDrawScript, and ViewShowScript. Note that these same system messages (except for ViewShowScript) are sent to all visible child views of the view as they are created and shown as well. For information about these system messages, refer to "Application–Defined Methods" on page 3-127.

This method always returns non-nil.

Note that this message must be sent to a view, not to a template. To ensure that a view exists for the template, you must have declared it. For details on declaring a view, see the section "View Instantiation" beginning on page 3-36.

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

view.viewCObject;

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

Close

view:Close()

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

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

If the view has already been closed, nothing happens.

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

Note

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

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

◆

Toggle

view:Toggle()

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

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

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

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

Note that this message must be sent to a view, not to a template. To ensure that a view exists for the template, you must have declared it. For details on declaring a view, see the section "View Instantiation" beginning on page 3-36.

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

Show

view:Show()

If the view is currently hidden, this method plays the "show" sound (stored in the showsound slot), brings the view to the front, shows it and all of its visible child views, and calls the ViewShowScript. Note that you must specify a view. This method always returns non-nil.

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

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

Hide

view`:Hide()`

If the view is currently shown, this method plays the "hide" sound (stored in the `hidesound` slot), calls the `ViewHideScript`, and hides the view and all of its child views. This method always returns `non-nil`.

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

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

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

Dirty

view`:Dirty()`

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

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

When a view is redrawn as a result of the `Dirty` method, the system does not necessarily reread all of the slots in the view. For example, slots

describing the view contents are not read—the contents are assumed to have not changed. If you were to directly change the `text` slot of a `clParagraphView` and then send it the `Dirty` message, you would not see the text in the view change.

Usually, you want a view to redraw with its new contents, if the contents change. To do this, use the global function `SetValue` to change the contents of slots in the view. The `SetValue` function causes the system to reread the changed slots in the view before it is redrawn, and it automatically dirties the view so you don't have to send it the `Dirty` message. You can find the description of `SetValue` on page 3-85.

If you change the bounds of a view directly, `Dirty` will not cause the view to be redrawn with new bounds. To do that, send the view the `SyncView` message. For more details, see the description of `SyncView` on page 3-86.

view:OffsetView

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

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

dx	The x coordinate of the view in which you want to change a slot value.
dy	The y coordinate of the view in which you want to change a slot value.

`OffsetView` does the redraw easier and faster then `SetValue` because you don't have to rejustify the bounds and run `SetupFormScript`.

RefreshViews

`RefreshViews()`

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

SetValue

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

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

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

This function always returns `nil`.

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

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

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

If the view exists, any dependent views (see the `TieViews` function on page 3-117) are notified, and the `ViewChangedScript` message is sent to the view.

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

Note

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

SyncView

view`:SyncView()`

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

MoveBehind

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

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

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

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

The return value of this method is undefined.

Dynamically Adding Views

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

AddStepView

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

Dynamically instantiates a new view based on the specified child template and adds it to the parent's `stepChildren` array. You must send the `Dirty` message to the new view or to its parent view to cause the new view to be drawn. See "Using the AddStepView Function" beginning on page 3-47 for information on using this function.

parentView The parent view to which you want to add the new view.

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

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

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

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

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

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

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

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

RemoveStepView

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

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

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

This function always returns `nil`.

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

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

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

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

Note

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

AddView

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

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

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

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

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

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

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

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

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

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

RemoveView

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

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

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

This function always returns `nil`.

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

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

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

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

BuildContext

`BuildContext(`*template*`)`

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

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

This function returns the view that it creates.

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

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

Making Modal Views

The following methods are used to make modal views.

AsyncConfirm

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

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

confirmMessage	A string to be displayed to the user
buttonList	A symbol (for one of the standard button sets) or an array of `buttonSpecs`
fn	A closure to be called when the slip is dismissed. It will be passed one argument, the value of the button tapped.

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

ModalConfirm

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

This method creates and displays a slip and does not return until the user dismisses the slip by tapping a button. Because this method causes a new task to be spawned, it is less efficient and takes more system overhead, so you should use `AsyncConfirm` in most cases.

For example

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

confirmMessage	A string to be displayed to the user
buttonList	A symbol (for one of the standard button sets) or an array of button specifications

`ModalConfirm` returns the value of the button (from buttonList) that the user tapped to dismiss a slip:

- OK = non-`nil`
- Cancel = `nil`
- Yes = non-`nil`
- No = `nil`

You can pass in one of the following two arrays of button specifications, a string or frame, for the *buttonList* parameter.

- A string specifies the button's value; the index of the string in the *buttonList* is its value
- A frame should be of the form:
  `{value:` *buttonValue*`, text` *string*`}`

FilterDialog

view`:FilterDialog()`

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

`FilterDialog` should be used instead of `ModalDialog` as it does not spawn a new task when it is used.

If you make a view that is not a child of the root view modal, its child-of-root-ancestor defines the area that is modal; that is, you can tap and write in the whole view.

Like `Open`, the `FilterDialog` method creates the graphic representation of the view. It then plays the "show" sound (stored in the `showsound` slot), brings the view to the front, and shows it (and all of its child views). The view receives the following system messages: `ViewSetupFormScript`, `ViewSetupChildrenScript`, `ViewSetupDoneScript`, `ViewDrawScript`, and `ViewShowScript`. For information about these system messages, refer to "Application–Defined Methods" on page 3-127.

Note that the `FilterDialog` message must be sent to a view, not to a template. To ensure that a view exists for the template, you must have declared it. For details on declaring a view, see the section "View Instantiation" beginning on page 3-36.

ModalDialog

view`:ModalDialog()`

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

This method always returns non-`nil`.

Finding the Bounds of Views

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

RelBounds

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

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

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

SetBounds

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

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

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

GlobalBox

view`:GlobalBox()`

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

Note

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

GlobalOuterBox

view`:GlobalOuterBox()`

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

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

Note

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

LocalBox

view`:LocalBox()`

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

Note

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

DirtyBox

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

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

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

The return value of this method is undefined.

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

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

GetDrawBox

view`:GetDrawBox()`

Returns the bounds of the area on the screen that needs redrawing (the area marked as dirty). This method returns a bounds frame containing global coordinates.

ButtonBounds

`ButtonBounds(`*width*`)`

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

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

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

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

Note

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

StdButtonWidth

`StdButtonWidth(`*str*`)`

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

str	A string that contains the button name

This function internally calls `strFontWidth`.

PictBounds

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

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

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

Note

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

Animating Views

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

Effect

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

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

effect	Specifies an animation effect. You can specify any of the effect constants that are used in the `viewEffect` slot (described in the section "Opening and Closing Animation Effects" beginning on page 3-31).
offScreen	Specifies whether or not the view should appear to animate off of or onto the screen. Specify non-`nil` to make the animation appear as if the view is moving off of the screen. Specify `nil` to make the animation appear as if the view is moving onto the screen.
sound	A sound frame containing a sound that you want played concurrently with the animation. (If you don't want a sound, specify `nil`.)
methodName	This method changes the state of your view (the two states that the effect transitions between). You must specify a symbol (for example, `'myScript`). Do not change the state of your view before calling `Effect`. This method must be accessible from the view to which the `Effect` message is sent; that is, this method must reside in that view or be accessible from that view through inheritance.
methodParameters	An array of parameters that are passed to your method.

The `Effect` method always returns `nil`.

Here is an example of using this method.

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

view1 := {...
```

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

SlideEffect

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

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

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

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

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

sound	A sound frame containing a sound that you want played concurrently with the animation. (If you don't want a sound, specify `nil`.)
methodName	The method that you want called before the animation occurs. You must specify a symbol (for example, `'myScript`). This method must be accessible from the view to which the `SlideEffect` message is sent; that is, this method must reside in that view or be accessible from that view through inheritance.
methodParameters	An array of parameters that are passed to your method.

The `SlideEffect` method always returns `nil`.

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

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

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

Here is an example of using this method.

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

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

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

RevealEffect

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

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

distance	The number of pixels to animate a portion of the view moving up or down on the screen. Specify a positive number to make the view portion appear to move upward on the screen this number of pixels. To make the view portion appear to move downward, specify a negative number. The *distance* parameter should be the height of the view content you want to reveal (or hide).
bounds	The partial area of the view that you want to animate moving up or down. You should specify a `viewBounds` frame using coordinates local to the view to which you are sending this message. The portion of the view that you specify is copied above or below its present position, depending on the setting of *distance*.
sound	A sound frame containing a sound that you want played concurrently with the animation. (If you don't want a sound, specify `nil`.)
methodName	The method that you want called before the animation occurs. You must specify a symbol (for example, `'myScript`). This method must be accessible from the view to which the `RevealEffect` message is sent; that is, this method must reside in that view or be accessible from that view through inheritance.
methodParameters	An array of parameters that are passed to your method.

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

The `RevealEffect` method always returns `nil`.

Here is an example of using this method.

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

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

Delete

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

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

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

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

The Delete method always returns nil.

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

Here is an example of using this method.

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

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

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.

Drag

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

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

unit	The current stroke unit passed by the `ViewClickScript` message.
dragBounds	A `bounds` frame describing the area, relative to the root view, within which the view can be dragged. If *dragBounds* is `nil`, the bounds of the entire screen limit the dragging area.

This method always returns non-`nil`.

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

Here is an example of using this view method:

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

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

Dragging and Dropping a Item

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

DragAndDrop

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

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

unit — The stroke unit received by the ViewClickScript method.

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

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

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

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

- types — An array of symbols of the types to which an item can be converted .
- view — A view object type if the dragged item is a view with a symbol type of 'paragraph, 'polygon, 'picture, and so on).
- dragRef — Any value that will be passed to other methods.

`label`	An optional string used when the drop is to the Clipboard; it is used as the Clipboard label. If this slot is missing and the item has a `'text` type, the text data will be used as the label; otherwise a default label is used.

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

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

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

The `DragAndDrop` method sends several messages to both the source view (the view to which `DragAndDrop` was sent) and the destination view (the view that will receive the items). These messages are documented "Application–Defined Methods" beginning on page 3-127.

Scrolling View Contents

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

SetOrigin

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

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

originX	The X coordinate of the new view origin.
originY	The Y coordinate of the new view origin.

This method always returns `nil`.

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

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

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

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

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

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

Figure 3-9 shows an example of a world map before and after it has been scrolled. The map is enclosed in a parent view which is the rectangle around the map. The map was scrolled to the right with this code:

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

Figure 3-9 `SetOrigin` example

Before Scrolling

After Scrolling Right

Here is an example of using this view method:

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

SyncScroll

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

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

what	You can specify either an array of view templates or a soup cursor, depending on what kind of data is contained in the view you want to scroll. If all of the view children are contained in an array, then specify the array. If your view data consists of child views created from soup entries, then specify the soup cursor.
index	Only used if you specify an array of view templates for *what*. This is the index of the child view template that is currently displayed at the top of the parent view.
upDown	Set to –1 to scroll up (visually, the views move downward on the screen), or set to 1 to scroll down (visually, the views move upward on the screen).

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

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

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

A slot named `index` is required in the view that receives the `SyncScroll` message (the parent view). Initialize the `index` slot to the index of the child template that is at the top of the parent view when the view is first displayed. Pass the `index` slot for the *index* parameter to `SyncScroll`. The `SyncScroll` method will modify this slot when it scrolls the views, so you

don't need to keep track of the index. On each subsequent call to `SyncScroll`, pass the `index` slot for the *index* parameter.

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

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

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

- This method may or may not move the cursor forward or backward in the soup. Scrolling does not always require advancing to the next or previous view, in which case the cursor would not be changed. For example, a single data item may be longer than the screen space allocated for it in a view, and so tapping the scroll arrow should scroll the view rather than advance to the next data item. In this case, the soup cursor would not be advanced since a new item need not be retrieved from the soup as a result of scrolling.
- Before the scrolling animation is done and the views are redrawn, the `ViewSetupChildrenScript` message is sent to the view that is being scrolled. The view being scrolled should use the `ViewSetupChildrenScript` method to recalculate its `stepChildren` array so that the correct views are displayed when they are redrawn by the `SyncScroll` method.

Working With View Highlighting

These methods and functions are used to highlight a view.

Hilite

view:`Hilite`(*on*)

Highlights or unhighlights a view.

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

This method always returns non-`nil`.

HiliteUnique

view:`HiliteUnique`(*on*)

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

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

This method always returns non-`nil`.

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

TrackHilite

view:`TrackHilite`(*unit*)

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

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

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

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

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

TrackButton

view:TrackButton(*unit*)

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

unit	The current stroke unit passed to the ViewClickScript method.

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

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

HiliteOwner

HiliteOwner()

Returns the view containing highlighted data. If there is more than one view containing highlighted data, the common parent of those views is returned. However, only one application at a time can have highlighted data. This function returns nil if no views contain highlighted data. See “Determining Which View Item is Selected” on page 3-51 for information on using this function.

This function works only with views of the class clEditView or clParagraphView. Views of the class clEditView can contain highlighted text or polygon data. Views of the class clParagraphView can contain only highlighted text.

GetHiliteOffsets

`GetHiliteOffsets()`

Returns an array of arrays containing information about views that have highlighted data. The return value looks like this:

`[[`*view*`, `*start*`, `*end*`], [`*view*`, `*start*`, `*end*`], ...]`

Each subarray contains information about one highlight. The first value, *view*, is the view containing the highlight. The second value, *start*, is the starting character position of the highlight. A character position of zero indicates the beginning of the view, a position of 1 is after the first character, and so on. The third value, *end*, is the ending character position of the highlight.

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

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

SetHilite

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

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

start	The starting character position of the highlighting. A character position of zero indicates the beginning of the view, a position of 1 is after the first character, and so on.
end	The ending character position of the highlighting.
unique	A Boolean value. Specify non-`nil` to make the specified text the only highlighted text in the view; any other highlighted text is unhighlighted. Specify `nil` to allow previously highlighted text to stay highlighted. In the later case, the highlighting is extended to include the

newly specified highlighted text. Discontiguous highlighting is not allowed.

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

Creating View Dependencies

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

TieViews

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

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

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

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

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

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

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

```
mainView := {...
viewClass: clParagraphView,
```

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

Synchronizing Views

The following two methods are used to synchronize views.

RedoChildren

view:RedoChildren()

Closes, then reopens and redraws, all of a view's child views. This method always returns non-nil.

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

1. The child views are sent ViewQuitScript messages, and then they are closed.
2. The parent view (the view to which you sent the RedoChildren message) is sent the ViewSetupChildrenScript message, and the child templates are reread from the viewChildren and stepChildren slots of the parent view.

3. The child views are reopened, and in this process are sent the following messages: `ViewSetupFormScript`, `ViewSetupChildrenScript`, `ViewSetupDoneScript`.
4. The parent view, and then the child views, are drawn and then are sent the `ViewDrawScript` message.

For more information about system messages, refer to "Application–Defined Methods" on page 3-127.

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

However, if a child view is declared in a view that is still open (typically the parent view), then, even though the child view is closed, its view memory object is not destroyed and any data stored in the view is preserved. This is the same as when you send the `Close` message to a declared view. For more information about declared views, see the section "View Instantiation" beginning on page 3-36.

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

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

```
{...
someFunction: func()
   self:RedoChildren();
...}
```

SyncChildren

view:SyncChildren()

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

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

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

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

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

Here is an example of using the SyncChildren method:

```
{...
addOneChild: func(childTemplate)
   begin
   // ensure that stepChildren array is in RAM
```

```
   if not HasSlot(self, 'stepChildren) then
      self.stepChildren := Clone(self.stepChildren);
   // add new template into the array
   AddArraySlot(self.stepChildren, childTemplate);
   // sync up the views
   self:SyncChildren();
   end
...}
```

Laying Out Multiple Child Views

The following methods are used to layout multiple child views.

LayoutTable

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

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

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

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

The viewBounds slots of the children are calculated so that the first child is placed in the upper-left corner of the parent view. You can use the

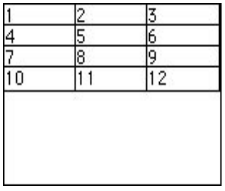
columnStart and *rowStart* parameters to change which child is the first child. By using these parameters to specify a different upper-left cell, you can display just a portion of the entire table.

For example, to generate templates for all of the cells in a table, specify 0, 0 for *columnStart* and *rowStart*. This places the top left cell in the table in the top left corner of the parent view. This is illustrated in the first view shown in Figure 3-10.

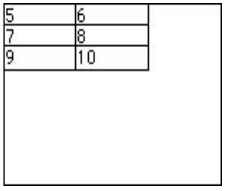
To offset the table upward and to the left, specify 1, 1. This places the second cell in the second row in the top left corner of the parent view. This is illustrated in the second view shown in Figure 3-10. Note, however, that cells are laid out sequentially beginning with the indicated cell. That is, cells 5 through 10 are all shown. The table isn't simply shifted up and to the right.

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

Figure 3-10 `LayoutTable` Results



The *œlumnStart* and *rowStart* parameters are set to 0, 0



The *columnStart* and *rowStart* parameters are set to 1, 1

TableDefinition slots

`tabAcross`	The number of columns in the table
`tabDown`	The number of rows in the table
`tabWidths`	An integer giving the fixed width of the columns, in pixels, or an array of column widths
`tabHeights`	An integer giving the fixed height of the rows, in pixels, or an array of row heights
`tabProtos`	A reference to a template used in creating the child views, or an array of references to templates. The array elements are mapped to the table of views beginning at the top-left cell of the table and continuing down the first column, and then down the second column, and so on. If there are fewer array elements than table cells, after the last array element is mapped, the mapping continues with the first element.
`tabValues`	A value that is used as the value of each of the child views. Alternately, an array of values that are mapped to table cells as above.
`tabValueSlot`	A symbol naming the slot in each of the child views where its view value (specified in `tabValues`) is stored. (Remember to quote the symbol; as with `'text`.) For example, if the table consists of child views based on the `clParagraphView` class, you would specify `'text` for this slot, since the value of a `clParagraphView` is stored in the `text` slot.
`tabSetup`	A method that is called before each of the child views is instantiated. It is passed three parameters: a reference to the child template, its column number in the table, and its row number in the table. This allows you to do special initialization operations to each child view before it is instantiated. This method must be passed the context with the call.

Here is an example of using the `LayoutTable` method. This example shows the code used to generate the first table in Figure 3-10.

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

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

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

LayoutColumn

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

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

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

This method returns a reference to an array of child views that fill the bounds of the main view, beginning with the view at *index* and containing as many subsequent views as it takes to fill the main view to the bottom.

Miscellaneous View Operations

This section describes other miscellaneous view methods and functions.

SetPopup

view:SetPopup()

After a view is shown, call this method to make the view a pop-up view (a picker); that is, a view that gets closed on the next pen tap (whether inside or outside of it). An example of where this feature is used is in the protoPicker view proto. See Chapter 6, “Pickers, Pop-up Views, and Overviews,” for details.

This method always returns nil.

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

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

GetViewFlags

`GetViewFlags(`*template*`)`

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

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

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

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

These symbols are evaluated at run time.

Visible

`Visible(`*view*`)`

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

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

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

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

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

These symbols are evaluated at run time.

Application–Defined Methods

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

ViewSetupFormScript

view`:ViewSetupFormScript()`

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

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

Here is an example of using this method:

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

Note

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

ViewSetupChildrenScript

view:`ViewSetupChildrenScript()`

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

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

Here is an example of using this method:

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

Note

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

ViewSetupDoneScript

view:`ViewSetupDoneScript()`

This message is sent after all of the child views of the view are instantiated, just before the view is displayed.

Here is an example of using this method:

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

Note

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

ViewQuitScript

view:`ViewQuitScript()`

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

Note that an undeclared view is destroyed when it is closed. A declared view still exists, if the view in which it is declared is still open. A view can get control after all of its children have been destroyed. If you return the symbol `'postQuit` from the `ViewQuitScript` method of a view, that same view will then be sent the `ViewPostQuitScript` message after all of its child views have been destroyed. This allows you an opportunity to do extra cleanup, if necessary. See `ViewPostQuitScript` (page 3-130) for additional details.

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

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

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

The key point to remember is that for these child views, you cannot count on their children still existing when they receive this message. So a view shouldn't do any processing involving data in child views unless it is the topmost view receiving the `ViewQuitScript` message.

IMPORTANT

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

Here is an example of using this method:

```
ViewQuitScript: func()
   begin
   RemoveSlot(GetGlobals().routing, appSymbol);
   RemoveSlot(GetRoot(), 'businessFormat);
   RemoveSlot(GetRoot(), 'myAuxFormat);
   end
```

Note

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

ViewPostQuitScript

view`:ViewPostQuitScript()`

This message is sent to a view following the `ViewQuitScript` message and after all of the view's child views have been destroyed. This message is not automatically sent to all views, but is sent only if the `ViewQuitScript`

method returns the symbol 'postQuit. See ViewQuitScript for more information (page 3-129).

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

ViewShowScript

view:ViewShowScript()

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

Here is an example of using this method:

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

ViewHideScript

view:ViewHideScript()

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

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

Here is an example of using this method:

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

ViewDrawScript

view:ViewDrawScript()

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

If you want to draw in a view other than when the ViewDrawScript message is sent, use the DoDrawing view method, documented in Chapter 12, “Drawing and Graphics.”

Here is an example of using this method:

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

ViewDrawScript: func()
   // draw an X in the view
   begin
```

```
   :DrawShape(self.lines, nil);
   end
```

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

ViewHiliteScript

view:ViewHiliteScript(*on*)

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

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

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

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

Here is an example of using this method:

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

Note

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

ReorientToScreen

view:ReorientToSccreen()

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

WARNING

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

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

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

```
ReorientToScreen: ROM_DefRotateFunc
```

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

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

ViewScrollDownScript

view`:ViewScrollDownScript()`

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

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

Here is an example of using this method:

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

ViewScrollUpScript

view:ViewScrollUpScript()

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

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

Here is an example of using this method:

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

ViewOverviewScript

view:ViewOverviewScript()

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

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

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

Here is an example of using this method:

```
ViewOverviewScript: func()
   begin
   if (cardPrefs.mode = modeCloseUp) then
      cardPrefs.mode := modeOverview
   else
      cardPrefs.mode := modeCloseUp;
   closeUp:Toggle();
   overView:Toggle();
   status:RedoChildren();
   end
```

ViewAddChildScript

view:ViewAddChildScript(*child*)

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

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

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

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

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

Here is an example of using this method:

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

```
AddView(myView, child);
end
```

Note

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

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

ViewChangedScript

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

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

slot — a symbol that is the name of the slot whose value changed

view — the view that slot resides in.

Here is an example of using this method:

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

ViewDropChildScript

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

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

child — the child view to remove.

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

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

Here is an example of using this method:

```
ViewDropChildScript: func(child)
   begin
   EntryRemoveFromSoup(child);
   base:RedoChildren();
   nil;
   end
```

Note

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

ViewIdleScript

view`:ViewIdleScript()`

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

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

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

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

Note

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

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

Here is an example of using this method:

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

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

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

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

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

Note

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

The `DragAndDrop` method sends the following messages to the source and destination views. It sends these message in the order in which they are listed.

Note that *sourceView* indicates that the message is sent to the source view, and *destView* indicates that the message is sent to the destination view.

Note also that when the source and destination views are the same (items are being moved within the same view) the `ViewGetDropDataScript` and `ViewDropScript` messages are not sent, but instead `ViewDropMoveScript` is called.

ViewDrawDragDataScript

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

bounds The bounds that were passed to `DragAndDrop`.

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

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

ViewDrawDragBackgroundScript

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

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

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

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

ViewGetDropTypesScript

destView`:ViewGetDropTypesScript(`*currentPoint)*

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

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

ViewFindTargetScript

destView:`ViewFindTargetScript(`*dragInfo*`)`

Lets the destination view redirect the drop to a different view. `ViewFindTargetScript` should return a view frame of the view that should get the drop messages. It is called right after the `ViewGetDropTypesScript`.

dragInfo	An array of frames (one frame per dragged item). See "DragAndDrop" on page 3-107 for a list of approved slots.

ViewDragFeedbackScript

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

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

dragInfo	The same parameter passed to `DragAndDrop`. See "DragAndDrop" on page 3-107.
currentPoint	The current pen position in global coordinates (a frame containing x and y slots).
show	A Boolean value indicating whether to show or hide the feedback. Specify non-`nil` to show the feedback or `nil` to hide it. Hiding the feedback means erasing any highlighting drawn when *show* is non-`nil`, so the view appears normally.

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

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

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

Note that XORing is not required as a draw mechanism. The view might be saving part of the screen to an offscreen bitmap, drawing feedback, and then

when asked to hide the feedback (show is set to `nil`), it could restore the original image from the offscreen bitmap.

ViewGetDropDataScript

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

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

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

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

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

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

dragType:	RequiredSlots	Optional slots
`'text`	`text`	`viewBounds` any other `clParagraphView` slots

dragType:	RequiredSlots	Optional slots
'polygon	points viewBounds	any other clPolygonView slots
'ink	ink viewBounds	any other clPolygonView slots
'picture	icon viewBounds	any other clPictureView slots

Note

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

ViewDropScript

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

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

dropType	One of the types that the destination view returned from the ViewGetDropTypesScript method.
dropData	The frame that the source view returned from the ViewGetDropDataScript method. If this frame had a viewBounds slot, this slot is converted to be in destination view coordinates before calling ViewDropScript.
dropPt	The last stroke point in global coordinates (a frame containing x and y slots).

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

Note that this method should post an undo action, if necessary.

ViewDropMoveScript

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

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

dragRef	The drag reference for this item (as passed to `DragAndDrop` in the *dragInfo* array).
offset	A frame with `x` and `y` slots indicating the horizontal and vertical offsets of the item.
lastDragPt	The last stroke point in global coordinates (a frame containing `x` and `y` slots).
copy	The *copy* parameter as passed to `DragAndDrop`.

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

Note that this method should post an undo action if necessary.

ViewDropRemoveScript

sourceView:`ViewDropRemoveScript`(*dragRef*)

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

This method should remove the item from the source view.

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

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

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

ViewDropDoneScript

destView:`ViewDropDoneScript()`

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

Warning Messages

The warnings in Table 3-7 are printed to the inspector when a NewtonScript application calls a view method in situations where the requested operation is unwise, unnecessary, ambiguous, invalid, or just plain a bad idea. The function or method will typically do nothing other than print this warning message, but they point out situations where code needs to be changed since these calls may very well not be supported in the future.

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

If the global variable `noEvilLiveOn` is set to `true`, a breakloop will be entered, which helps in finding out exactly which view is causing the problem. Setting `noEvilLiveOn` will also cause other “incompatibility” errors to enter a breakloop.

Table 3-7 View Warning Messages

Error Number	Message
4711	`Remove[Step]View was called while the parent view was being opened or closed`
4712	`Remove[Step]View was called with a template instead of a view frame`
4713	`Remove[Step]View was called on a view which was being opened or closed`

Error Number	Message
4714	Remove[Step]View was called with a read-only stepChildren array (i.e. the view wasn't previously added with AddView)
4715	Close() was sent to a view which was opening or closing
4716	Toggle() was sent to a view which was opening or closing
4717	Toggle() was sent to a view whose parent was being opened or closed
4718	Show() was sent to a view which was opening or closing
4719	Hide() was sent to a view which was opening or closing
4720	RedoChildren() was sent to a view which was opening or closing
4721	SyncChildren() was sent to a view which was opening or closing
4722	SetKeyView() was sent to a view that wasn't a clParagraphView

Summary of Views

Constants

viewJustify constants

vjLeftH
vjCenterH
vjRightH
vjFullH

```
vjTopV
vjCenterV
vjBottomV
vjFullV
vjParentLeftH
vjParentCenterH
vjParentRightH
vjParentFullH
vjSiblingNoH
vjSiblingLeftH
vjSiblingCenterH
vjSiblingRightH
vjSiblingFullH
vjParentTopV
vjParentCenterV
vjParentBottomV
vjParentFullV
vjSiblingNoV
vjSiblingTopV
vjSiblingCenterV
vjSiblingBottomV
vjSiblingFullV
noLineLimits
oneLineOnly
oneWordOnly
vjNoRatio
vjLeftRatio
vjRightRatio
vjTopRatio
vjBottomRatio
vjParentAnchored
```

Class Constants

```
clView
clPictureView
clEditView
```

clParagraphView
clPolygonView
clKeyboardView
clMonthView
clRemoteView
clPickView
clGaugeView
clOutline

viewFlags Constants

vNoFlags
vVisible
vReadOnly
vApplication
vCalculateBounds
vClipping
vFloating
vWriteProtected
vClickable
vNoScripts

viewFormat Constants

vfNone
vfFillWhite
vfFillLtGray
vfFillGray
vfFillDkGray
vfFillBlack
vfFillCustom
vfFrameWhite
vfFrameLtGray
vfFrameGray
vfFrameDkGray
vfFrameBlack
vfFrameCustom
vfFrameMatte
vfPen(*pixels*)

vfLinesWhite
vfLinesLtGray
vfLinesGray
vfLinesDkGray
vfLinesBlack
vfLinesCustom
vfInset(*pixels*)
vfShadow(*pixels*)
vfRound(*pixels*)

viewTransferMode Constants

modeCopy
modeOr
modeXor
modeBic
modeNotCopy
modeNotOr
modeNotXor
modeNotBic
modeMask

viewEffect Constants

fxHStartPhase
fxVStartPhase
fxColAltHPhase
fxColAltVPhase
fxRowAltHPhase
fxRight
fxMoveH
fxLeft
fxRowAltVPhase
fxDown
fxMoveV
fxUp
fxRevealLine
fxWipe
fxFromEdge

```
fxCheckerboardEffect
fxBarnDoorOpenEffect
fxBarnDoorCloseEffect
fxVenetianBlindsEffect
fxIrisOpenEffect
fxIrisCloseEffect
fxPopDownEffect
fxDrawerEffect
fxZoomOpenEffect
fxZoomCloseEffect
fxZoomVerticalEffect
fxColumns(x)
fxRows(x)
fxSteps(x)
fxStepTime(x)
```

Functions and Methods

Getting References to Views

```
view:ChildViewFrames()
view:Parent()
GetRoot()
GetView(view)
```

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*)
RemoveStepView(*parentView*, *childView*)
BuildContext(*template*)

Making Modal Views

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

Finding 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*)

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(`*RefArg dragType*`, `*RefArg 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()`

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. Using NewtApp is the best way to start programming for the Newton platform. However, if you have created Newton applications before, the process of putting together a NewtApp application will be familiar and fast.

Newton applications can be created by using either the NewtApp framework protos, described in this chapter, or by using the protos described in almost every other chapter of this book. Chapter 2, “Getting Started,” gives you an overview of the ways to approach the process.

Before reading this chapter you should already 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 14, “Find,” Chapter 15, “Filing,” and Chapter 2, “Routing Interface,” of the *Newton Programmer’s Guide: Communications.*

The examples in this chapter use the Newton Toolkit (NTK) development environment. Therefore, you should also be familiar with NTK or learn it before you try them. Consult the *"Newton Toolkit User's Guide"* for that information.

This chapter explains how to create a NewtApp-based application. It also includes reference information for all of the protos in the NewtApp framework.

The material in this chapter is still preliminary and will change in the future.

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—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 of write-protected cards. You basically set the values of a prescribed set of slots and the framework does the rest of the work.

When you use the NewtApp framework, you can create most kinds of applications. 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, like, for example, a calculator, or if you need to display more than one soup at a time, you should use the protos described in almost every other chapters of this book, to construct it. Specifically 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 12, "Drawing and Graphics," Chapter 13, "Sound," Chapter 11, "Data Storage and Retrieval," Chapter 14, "Find," and Chapter 15, "Filing."

Some of the NewtApp protos will work in non-framework applications. For instance, you may want to update an existing application to take advantage

of the functionality provided by the NewtApp slot view protos. This requires a bit of retrofitting but it can be done. See the example “Using Slot Views in Other Applications” beginning on page 4-28.

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 allows your application to present many different views of your data. The Show button is used to display different views of information. The New button is used to create 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, “Stationery,” though its use in a NewtApp application is demonstrated in an example in this chapter.

The NewtApp Protos

When you put the application protos together in a programming environment like the 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 instance, to include your application in system-wide date searches, all you need to do is set a slot in the base view of your application called `dateFindSlot`. (See “newtApplication” beginning on page 4-37.)

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 that 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 is used to implement slots that keep track of system details.

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

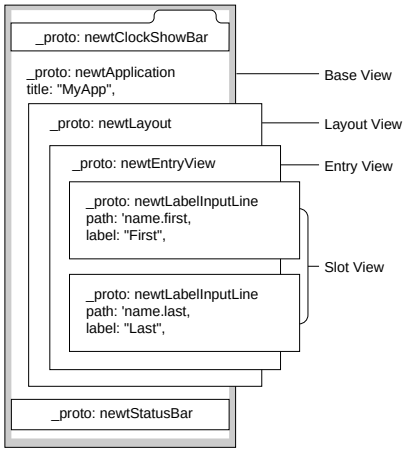
WARNING

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 will not be guaranteed to work in the future. ▲

You can see the four conceptual layers of NewtApp protos which you use to construct an application: the application base view, the layout view, the entry view, and the slot views. These are shown in Figure 4-1.

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



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 physical representation in a layout file.

- The `newtApplication` proto is the application's base view. As in the non-framework applications, the base view proto either physically contains or has references to all the other application parts.
- The `newtSoup` proto is used to create the data storage soup for your application; it is not displayed.
- The `newtLayout` protos govern the overall look of your data.
- The `newtEntryView` protos are the views associated with one soup entry and are contained in layout views. They do not display on the screen but instead, they manage operations on soups.
- 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

This proto serves as the base view for your application; it contains all the other protos of the application. In addition, 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 instance, the functionality for opening and registering soups, dispatching events, 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

There are several control protos which affect the entire application. Because of this they 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. The NewtApp controls that should also be in the `newtApplication` base view also include, the standard status bar, the folder tab, and the A-Z alphabet tabs. Note that the stationery buttons, which you use to tie viewDefs and dataDefs into your application, are defined in Chapter 5, “Stationery.”

About newtSoup

Application data is stored in persistent structures, known as soups, in any Newton application. In a NewtApp application, your soup definitions, which you write 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.

This `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 include:

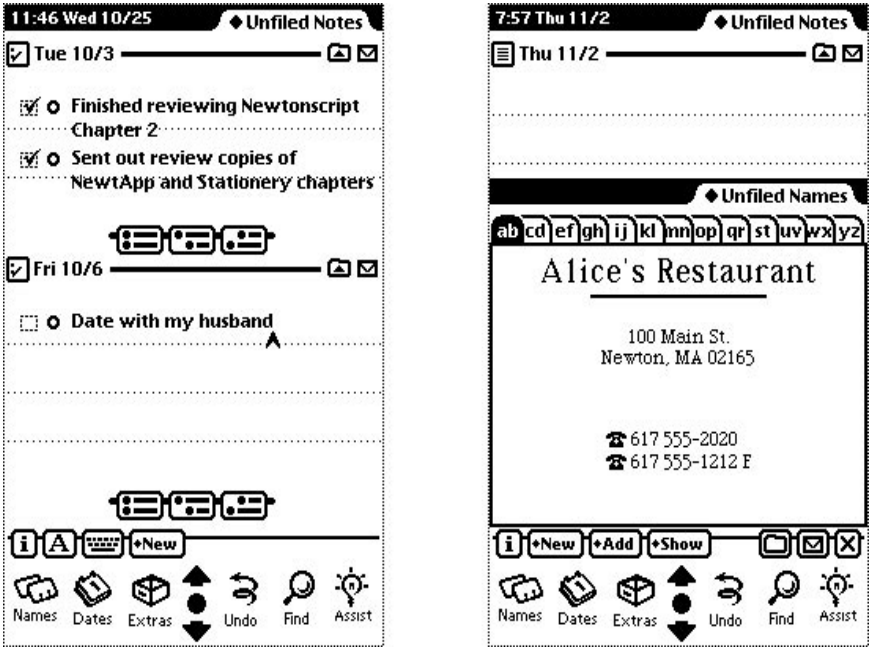
- an overview—seen when the Overview button is tapped
- a default view—seen when the application is first opened

There are three kinds of layouts corresponding to three basic application styles:

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

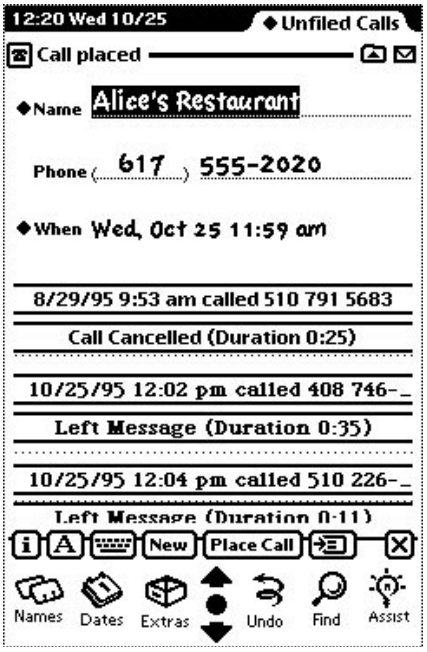
Card-based and paper roll -based applications differ in the number of entries each may have visible at one time, though, neither supports a display that is longer than the screen. The built-in Names application is a card-based application. For this type of application, only one entry is displayed at a time. In contrast, the built-in Notes application, which is a paper roll-based application, can have multiple entries visible at once. They must be separated by a header, which incorporates action and filing buttons, to make it obvious to which entry a button action should be applied. An example of both a card-based and a paper roll-based application is shown in Figure 4-2.

Figure 4-2 A paper roll-style application versus a card-based application.



The page-based application is a hybrid of the card-based and paper roll-based applications. Like the card-based application, only one entry is shown at a time. However, unlike the card-based application but like the paper roll -based application, the size of 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.

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, as well as, scrolling, and all the standard view functionality is implemented in the layout protos. A main (default) view layout and an overview layout are required to be declared in the `allLayouts` slot of the `newtApplication` base view. See the `newtApplication` documentation beginning on page 4-37.

Your layout can also control which buttons show on the status bar, if you set the `menuLeftButtons` and `menuRightButtons` slots of the layout proto—along with the `statusBarSlot` of the base view (`newtApplication` proto.) This becomes important when more than one entry is shown on the screen, as in a paper 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 must be on the header of each entry so the entry on which to perform an action is unambiguous.

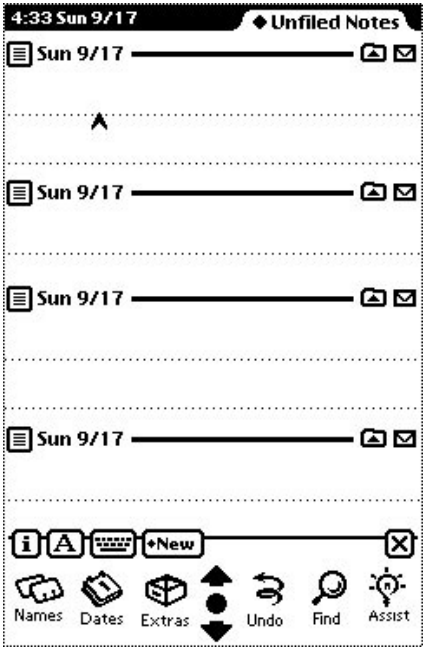
The Entry View Protos

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

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

The entry views also contains the user interface components that perform operations on one entry at a time. This includes the header bars, which are used as divider bars to separate multiple entries which are displayed simultaneously. This happens in the Notes application, and presumably in other paper roll-type applications. An example of the Notes application with multiple entries and header bars is shown in Figure 4-4.

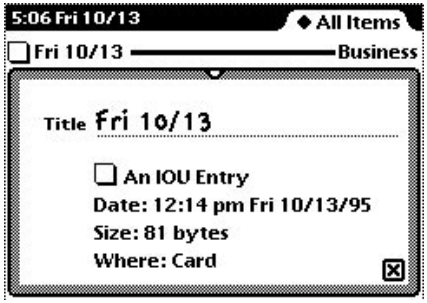
Figure 4-4 Multiple entries visible simultaneously



Note that the header bar contains the Action and Filing buttons at its far right side. These appear on the header bar to prevent any ambiguity in regards to the entry which is to be acted upon by those buttons.

In addition, the header bar contains a Title and icon at its far left. When the icon is tapped, the Information slip appears, as shown in Figure 4-5. (Note that this 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 which are used to access and display 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. In addition, the target view, which is the view in which the data in that entry will appear, and the data cursor, which accesses the slots in the entry, are set here.

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

About the Slot Views

These 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 will not work. This is because the entry views are responsible for setting references to the specific entry in which the slots, containing the data which a slot view will display, are to be found.

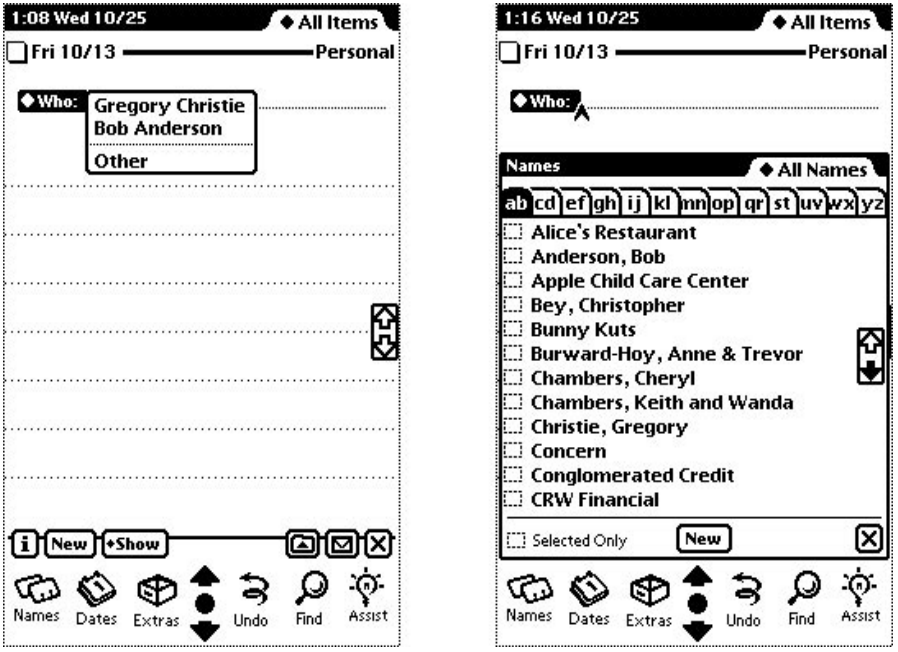
Slot views exist in two varieties: simple slot views and labelled input line slot views.

Both kinds of slot views are tailored to display and edit a particular kind of data which they format appropriately. For example, the number views (`newtNumberView` and `newtRONumberView`) format number data (according to the value of a `format` slot which you set.) The labelled input lines also format a particular kind of data, but they also allow you to set a label for the input line and to change that label into a picker (pop-up menu), if necessary.

In addition, there is one special slot view, called the `newtSmartNameView` proto, that allows the user to choose a name from the soup belonging to the built-in Names application. It adds a pop-up menu item, Other, to the picker, which when chosen from the `newtSmartNameView` proto, 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.

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



Stationery

Stationery, which is an extension you can add to any NewtApp application, is tightly integrated and works well 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. 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 4, “NewtApp Applications.”

NewtApp Compatibility

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

Some of the NewtApp protos are usable in your non-NewtApp applications. For instance, there is a `newtStatusBarNoClose` proto (see page 4-55) 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 only function within a simulated NewtApp environment. The mechanism for creating this setup is the `newtFalseEntryView` proto, described on page 4-70.

The slot views, which are documented in “The Slot View Protos” beginning on page 4-74, 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 which has one data soup and can be built as a data viewer or editor
- add functionality to existing non-NewtApp applications
- create and incorporate stationery extensions

When you use the set of protos which are the NewtApp application framework, you can quickly create an application that takes full advantage of the Newton system services. (Note that these were introduced in Chapter 1, “Overview.”)

In addition, many of the protos may be used in applications that are built without the framework. In particular, the slot views, which are used to display data, have functionality built into them 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, used in the following sections, uses of a single piece of stationery, which consists of a dataDef with two viewDefs. Note that stationery is documented fully in Chapter 5, "Stationery."

The example application is built using the Newton Toolkit (NTK) development environment. Again, details of using NTK are not explained here. Consult the *"Newton Toolkit User's Guide"* for more information.

Constructing a NewtApp Application

The sample "starter" application used here shows how to get a NewtApp application underway quickly. You may incorporate this sample code into your applications without restriction. It is being provided "as is," though, every reasonable effort has been made to make sure it is operable before publication. 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.

This sample is an application for gathering data that supports routing, filing, and finding. It presents two views of the data to be collected; a required default view "IOU Info," 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.

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 an its two corresponding two viewDefs:

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

Note that the creation of these files is shown in Chapter 5, “Stationery.”

In addition, a NewtApp application must include a standard install and remove script (the `InstallScript` and `RemoveScript`.) Any icons must be included with a resource file, ours is `CardStarter.rsrc`. There is also a text file, `Definitions.f`, in which application globals are defined. Neither the resource file nor the text file is required.

Note that the basic view slots, `viewBounds`, `viewFlags`, and `viewJustify`, are discussed in Chapter 3, “Views,” and will only be called out in the samples 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 name, `Definitions.f`.

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

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

Using newtApplication

This proto serves as 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 its appearance and disappearance
- layout slots
- stationery slots

Setting up the Application Base View

The application base view, `newtApplication`, should contain the basic application element protos. When you use NTK to create the layout for the `newtApplication` proto, you will add to it: a `newtStatusBar` proto (the menu 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. Draw out a `newtStatusBar` across the bottom of the layout.
3. Name the `newtStatusBar` proto `status`
4. Draw out 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 over 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.

- 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 the `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` implementation. Also, you may wish to add a `ReOrientToScreen` method to the `newtApplication` base view so that your application may be rotated to a landscape display. This message is sent to each child of the root view when the screen orientation is changed. See Chapter 3, “Views,” 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 which contains symbols for the application’s layout files. It must contain two slots, named `default` and `overview`, that refer to the two layout files which will be used for those respective views.

The section “Using the Layout Protos” beginning on page 4-21, shows how to use the NewtApp layout protos to construct these files. We will 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 one soup, though, a NewtApp application can contain more than one soup. Each soup defined for a NewtApp application must be based on the `newtSoup` proto. The slots, `soupName`, `soupIndices`, and `soupQuery`, are required to be defined within the `allSoups` soup definition frame.

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 two layouts; a default layout, which is displayed when the application is first opened, and an overview layout, which is displayed when the Overview button is pressed.

Depending upon which of the NewtApp framework's layout protos that you choose for your default view, you will set up your application as a card, paper roll, or page style application.

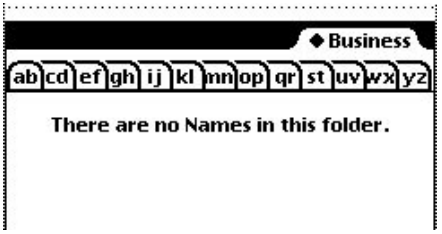
Unique slots in the layout protos include:

- `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 will get 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; where *items* is replaced by the value of the `appAll` slot set in the `newtApplication` base view. The an example of such a message, from the Names application, appears as shown in Figure 4-7.

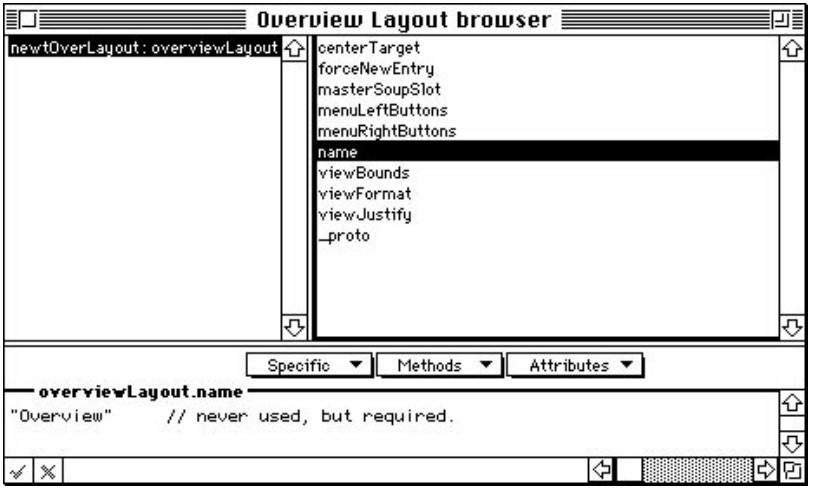
Figure 4-7 The message resulting from a `nil` value for `forceNewEntry`.



Using newtOverLayout

The slots that you must set for an overview are shown in the illustration of an 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`.
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 4-19.
4. Add the `forceNewEntry` slot. Leave it with the default value, `true`. This causes a new entry to be created if and when 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 thus 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, as needed, from the layout view.

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) which you wish to appear on the menu bar’s left and right sides, respectively. Note that when you use these arrays, the button protos listed in them are automatically placed correctly, according to the current human interface guidelines, on the status bar.

To appropriately set up the appearance of 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. Note that these are automatically laid out on the status bar starting from the far left and proceeding to the right.

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

Set the `menuRightButtons` slot to an array that includes the protos for the Action and Filing buttons. Note that these are automatically laid out on the status bar from right to left.

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

Be sure to add the Overview Layout template to your Project.

Creating the Default Layout

This is the view that you see on first 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. Note that the symbol in this slot must match the name of a soup declared in the `allSoups` slot of the `newtApplication` base view. If your recall, the name of the soup defined in the `allSoups` slot 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 4-19.
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 thus 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 it displays items as shapes.

Programmatically, the entry view sets value that are 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 which contains the stored data (the `target` slot) and the view that will be used to display data (the `targetView` slot.)

Follow these steps to create the 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. Note that this special view will not be visible. Its function is to provide a bounding box for the viewDef which will eventually be displayed there.

Registering DataDefs and ViewDefs

Several stationery slots are provided in the `newtApplication` base view, with which to identify the stationery belonging in your application. These slots include the `allViewDefs`, `allDataDefs`, and `superSymbol` slots.

Note

You may see how to create the stationery, which is used as part of this application, in 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, respectively. 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 of the `allDataDefs` and `allViewDefs` slots, contain frames that are required to contain slots with identical names, to indicate the dataDefs and viewDefs that work together. (Note that 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 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 dataDefs, 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 will only accept 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

There are both an install and remove script required to register your NewtApp application with the system for the various system services. They are strictly boilerplate scripts which you should copy, unaltered.

You should create a text file which you save as `Install&Remove.f` into which to copy the scripts.

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

RemoveScript := func(partFrame)
   begin

(partFrame.removeFrame):NewtRemoveScript(removeFrame);
   end;
```

Note that 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.) Following is the code listing.

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 Other Applications

The NewtApp slot view protos have a lot of functionality built in to 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 of a NewtApp application:

- The parent of the `newtFalseEntryView` must have the 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 `newtApplication` 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 will already have slots set which contain specifications for a soup name, soup indices, a soup query, and a soup cursor, (amongst others too numerous to list.)

Certain slots must be added to these `newtApplication` 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` and `targetView` slot, so that the false entry view can set them when an entry is changed. Second, you should include a method which sends the `Retarget` message to the false entry view when an entry is changed. As an example, you may wish to implement the following slot, or one like it:

```
baseView.DoReTargetting := { 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 `DoReTargetting` message, as defined above, in the `viewScrollUpScript` and the `viewScrollDownScript`. Following is an example of a `viewScrollUpScript` that scrolls through soup entries.

```
func()
   begin
      EntryChange(target);
      cardSoupCursor:Prev();
      :ResetTarget();
      :DoRetargetting();
   end
```

Other places where you may want to send the `Retarget` message include a delete action script, a `viewSetupDoneScript` (which executes immediately before a view is displayed or redisplayed), or even in the script of a button that generates new entries and thus changes the soup and the display of it.

Using a newtFalseEntryView

The example used here, in which the `newtFalseEntryView` is implemented, is a non-NewtApp application that basically supports the use of slot views. If you want to adopt slot views into an existing application, you may copy the code from a working `newtFalseEntryView` layer and put it directly into your application, as a container for the slot views.

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`. Note that this symbol should match a slot named 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`. Note that this symbol should match a slot named in your application base view. The slot may be 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`. Note that this symbol should match a slot named 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 that you declare the `newtFalseEntryView` to the application base view so that the base view will be able to send `Retarget` messages to the false entry view when data is changed.

For more information about the `newtFalseEntryView` see the "NewtApp Reference."

NewtApp Reference

The section describes the NewtApp prototypes (protos.) The protos can be divided into the following categories:

- data storage proto; `newtSoup`
- base view proto; `newtApplication`
- base view control protos
- the layout protos
- the entry view protos
- the slot view protos
- the labelled input line protos

In addition, the required install and remove scripts are described in “Data Structures.”

Data Structures

A NewtApp application has a required install and remove script which you must include in your application build if the application is to register properly for various system services. You may copy them directly from the following code.

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

RemoveScript := func(partFrame)
begin
```

```
   (partFrame.removeFrame):NewtRemoveScript(removeFrame);
end;
```

newtSoup

This is the abstract proto (in other words, it has no visible component) that contains soup-handling routines. Soups in a NewtApp application must be based on the `newtSoup` proto and are set up in the `newtApplication.allSoups` slot. The `allSoups` slot is used in an example on page 4-20 and described on page 4-38.

Slot descriptions

`soupName` Required. This should be a string that is unique to your application. If the application only has one soup, you can use a string version of your application symbol, for example, `"MyApp:SIG"`
For an application that uses more than one soup you can add a number prefix to a string version of the application symbol, so that the soup name becomes `"00:MyApp:SIG"`

`soupIndices` Required. An array in which you define the indices for your soup. Here is an example:

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

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

                { structure: 'slot,
                path: 'labels,
```

```
                                type: 'tags }
                                ]
```

`soupQuery` Required. A basic soup query. An example follows.

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

`soupDescr` Optional. A string describing the soup.

`defaultDataType`
Optional. (This slot pertains to applications that use stationery.) A unique symbol naming a data type for your soup. This is useful if your application uses several different kinds of soups. An example of an appropriate value is: `'paperroll`.

Following are several interesting methods, defined in the `newtSoup` proto.

AddEntry

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

Adds the entry to the specified store. If no store is given the entry is added to the default store.

entry The entry to which the cursor is moved. You cannot create an entry procedurally by creating a frame with the appropriate attributes. The only valid entries are those returned by the various cursor and entry methods.

store The result of a call to `GetDefaultStore` or `GetStores`—naming the device on which data is to be stored.

AdoptEntry

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

Returns a frame that is a soup entry frame. This new entry frame consists of the frame specified in the `CreateBlankEntry` method which you define in the `newtApplication.allSoups` slot and if your application has a dataDef, an entry defined in either a `FillNewEntry` or `MakeNewEntry`

method. Note that it `FillNewEntry` exists, `MakeNewEntry` will not be called.

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

entry — Required. If `nil`, a blank entry is created. The entry to which the cursor is moved. You cannot create an entry procedurally by creating a frame with the appropriate attributes. The only valid entries are those returned by the various cursor and entry methods.

type — Optional. Defaults to `nil`. If the value is `true` then the method looks for the value of the class slot of this entry. See the chapter “Stationery” for more information on the class slot.

CreateBlankEntry

yourNewtSoup`:CreateBlankEntry()`

Returns a blank entry. Override this method to create the necessary structure of your soup. You may or may not want to put a `class` slot in your soup. However, note that any routeable item must have one. (For more information about how the `class` slot is used, see Chapter 3, “Transport Interface,” in *Newton Programmer’s Guide: Communications.*

DeleteEntry

yourNewtSoup`:DeleteEntry(`*entry*`)`

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

entry — The entry to which the cursor is moved. You cannot create an entry procedurally by creating a frame with

the appropriate attributes. The only valid entries are those returned by the various cursor and entry methods.

DuplicateEntry

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

Clones and returns the specified entry. The new duplicate entry is stored on the specified storage device.*entry*The entry to which the cursor is moved. You cannot create an entry procedurally by creating a frame with the appropriate attributes. The only valid entries are those returned by the various cursor and entry methods.

store The result of a call to `GetDefaultStore` or `GetStores`– naming the device on which data is to be stored.

DoneWithSoup

yourNewtSoup`:DoneWithSoup(`*appSymbol*`)`

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

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

FillNewSoup

yourNewtSoup`:FillNewSoup()`

Called by `MakeSoup` to add soup values to a new soup. You should define this method with soup values appropriate to your application.

MakeSoup

yourNewtSoup:`MakeSoup`(*appSymbol*)

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

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

GetCursor

yourNewtSoup:`GetCursor()`

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

Query

yourNewtSoup:`Query`(*querySpec*)

Send this message to a `newtSoup` to perform a query on the soup. It returns a cursor referencing a set of soup entries.

The *querySpec* frame may include the slots: `structure`, `path`, `type`, and `tagSpec`. For more information see Chapter 11, "Data Storage and Retrieval."

GetAlias

yourNewtSoup:`GetAlias`(*entry*)

Returns an entry alias. These represent the specified soup entry—for fast access later—without holding onto the actual entry. The entry alias can be used later as input to the `GotoAlias` function to retrieve the soup entry. See "About Entry Aliases" beginning on page 11-51 for more information.

entry The soup entry to which this method creates a reference.

GetCursorPosition

yourNewtSoup:GetCursorPosition()

This returns the current cursor position.

GotoAlias

yourNewtSoup:GotoAlias(*alias*)

Returns the soup entry referenced by the specified alias. Returns nil if the entry cannot be retrieved, typically because the original store, the original soup or the original entry is not found.

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

newtApplication

This is the application base view definition for all NewtApp applications. In an application this proto, physically contains the application-wide elements like the show bar and status bar. It also contains references to all the layout protos and sets up the application soup.

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

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

Slot descriptions

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

If you use NTK as your development environment, the application symbol is constructed for you from values

you set in the Output Settings dialog box for that application.

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

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

`appAll` Required. A string that is used as the last item displayed in the folder tab picker (pop-up menu.) The string must be of the form: `All` *items*. For example, the value of the `appAll` slot in the built-in Notes application is `"All Notes"`

`allSoups` Required. Define the soup(s) for your application in this frame. Your soup definition should consist of a named frame that is a `newtSoup` proto and contains the slots `soupName`, `soupIndices`, `soupQuery`, and a `CreateBlankEntry` function. A `soupFilter` slot (for folder filtering) may be added but is not required.

Following is a sample `allSoups` frame.

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

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

more information about the `newtSoup` proto see page 4-32.

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

Suitable values for the all layouts frame are as follows.

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

`scrollingEndBehavior`

Optional. Defaults to `'beepAndWrap`. You may also set it to `'wrap`, `'stop`, or `'beepAndStop`.

The two values, `wrap` and `stop`, are used to select how scrolling is handled at the end of a view. `Wrap` causes scrolling to display from the last entry around to the first (or vice-versa). `Stop` means that scrolling just stops when the display reaches either end.

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

`scrollingUpBehavior`

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

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

`statusBarSlot` Optional. A symbol which is the declared name of the status bar. It is used by the layout to govern the

appearance / disappearance of buttons on the status bar. For this to work, the layouts must also have `menuLeftButtons` and `menuRightsButtons` slots. See “newtStatusBarNoClose” beginning on page 4-55 and “newtLayout” beginning on page 4-58, for more information.

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

Slot descriptions

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

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

The system automatically registers all the dataDefs in this frame when the application installs. See Chapter 5, “Stationery,” for more information about dataDefs.

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

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

```
                    notes:
                    GetLayout("iouNotesViewDef"),}}
```

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

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

This `superSymbol` slot's value must match the value of a `soupName` slot in the `allSoups` definition. Note that any would-be stationery extensions to this application must also have a `superSymbol` that matches this value.

Following are the routing, filing, and find-related slots.

Slot descriptions

`doCardRouting` Optional. Defaults to `true.` This enables the filing interface to allow moves to and from cards.

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

`routeScripts` Optional. Contains default route scripts for Delete and Duplicate.

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

Slot descriptions

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

Following are some interesting methods, defined in the `newtApplication` proto, which support adding stationery to your application.

AddEntryFromStationery

newtApplication`:AddEntryFromStationery(`*stationerySymbol*`)`

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

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

AdoptEntryFromStationery

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

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

adoptee	The entry being adopted.
stationerySymbol	A symbol referring to the value of the stationery's `symbol` slot. It is used to set a `class` slot for the new blank entry. An example of an appropriate value from the built-in Notes soup is `'paperroll`.
store	The storage device on which to keep the information. If none is specified, data is stored on the internal storage device.

AdoptSoupEntryFromStationery

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

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

adoptee	The entry being adopted.
stationerySymbol	A symbol referring to the value of the stationery's `symbol` slot. It is used to set a `class` slot for the new blank entry. An example of an appropriate value from the built-in Notes soup is `'paperroll`.
store	The storage device on which to keep the information. If none is specified, data is stored on the internal storage device.
soup	The symbol for one of the soups in the `allSoups` slot. Use `nil` to indicate the current soup.

Following are some interesting methods, defined in the `newtApplication` proto, which are used to support filing in your application.

FolderChanged

newtApplication:FolderChanged(*soupName*, *oldFolder*, *newFolder*)

This method changes the folder tab label to the new folder name, if the new folder name is different than the old folder name, and saves the new folder information for the soup.

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

FilterChanged

newtApplication:FilterChanged()

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

Any time the contents of a view are changed the following methods are used to update the affected view(s) and change the data target entry.

ChainIn

newtApplication:ChainIn(*chainSymbol*)

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

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

ChainOut

newtApplication:ChainOut(*chainSymbol*)

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

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

GetTarget

newtApplication:GetTarget()

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

GetTargetView

newtApplication:GetTargetView()

Returns the view in which the target soup entry is displayed. Note, however, the target view in the base application level is undefined.

Following are some interesting methods, defined in the newtApplication proto, which are used to add find support to your application. Note that you do not call any of these methods and therefore, none of the parameters are defined here. If you wish to know more about the Find system services, see Chapter 14, “Find.”

DateFind

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

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

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

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

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

Find

newtApplication:Find(*text, results, scope, findContext*)

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

text	Contains the user-specified string for which Find is to search.
results	This `Find` method appends a slot, `myResult`, to this the `results` array which is passed to the `Find` method by the system. The exact content of the `myResult` slot depends on the kind of finder proto used to create the frame returned by your search method. If you used the `soupFinder` proto, the frame contains a cursor used to iterate over a list of entries returned by your search method's query on the application data soup. If you used the ROM_CompatibleFinder proto, the frame contains an array of found items. If a global find is in progress, the results array may contain slots created by other applications' search methods.
scope	Either `'localFind` or `'globalFind`. Indicates whether the search is local or global, allowing you to handle these two cases differently if you prefer.
findContext	A frame to which the message `SetStatus` is sent. The `SetStatus` function accepts as its sole argument a string to display to the user while the search is in progress.

ShowFoundItem

newtApplication:ShowFoundItem(*entry, finder*)

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

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

The following method, defined in the `newtApplication` proto, is used to display a particular layout, at the appropriate time, in your application.

ShowLayout

newtApplication`:ShowLayout(`*layout*`)`

This method sets the current layout to the layout you specify. A parameter value of `nil` sets the value of the current layout to the value of the previous layout. You can use it to switch the display from one layout to a different layout; for example, from the main view to the overview.

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

Following are some interesting methods, defined in the `newtApplication` proto, which you can use to delete and duplicate data items.

NewtDeleteScript

newtApplication`:NewtDeleteScript(`*entry*, *view*`)`

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

entry	A symbol referring to the entry to be deleted.
view	A symbol referring to the view in which the item appears.

NewtDuplicateScript

newtApplication:NewtDuplicateScript(*entry*, *view*)

Duplicates the specified entry and removes it from the specified view. This method also contains an alert which appears if someone tries to use duplicate when nothing is selected.

entry	A symbol referring to the entry to be deleted.
view	A symbol referring to the view in which the item appears.

Following are some interesting methods, defined in the newtApplication proto, which you can use to obtain information about and save the state of your application.

GetAppState

newtApplication:GetAppState()

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

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

GetDefaultState

newtApplication:GetDefaultState()

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

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

SaveAppState

newtApplication`:SaveAppState()`

This method saves:

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

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

There are several control protos which affect the entire application. Because of this they are generally placed in the general application base view layer and managed by it. The descriptions of these protos follow. Note that the stationery buttons, which you use to tie viewDefs and dataDefs into your application, are defined in Chapter 5, “Stationery.”

newtInfoButton

This provides the standard “i” information button, which always appears to the far left of the status bar. It is based on `protoInfoButton`, which you can read about in Chapter 7, “Controls and Other Protos.”

Unlike the `protoInfoButton`, the `newtInfoButton` proto provides the default methods, `DoInfoAbout`, `DoInfoHelp`, and `DoInfoPrefs`, which are invoked when the user taps on About, Help, or Prefs in the picker, as shown in Figure 4-9.

Figure 4-9 The information button and picker.

Following are the methods which provide default handling for the items in the picker menu of the `newtInfoButton`.

DoInfoAbout

newtInfoButton`:DoInfoAbout()`

If an About view has been created, it is closed and set to `nil`. If no about view is open, one is created.

DoInfoHelp

newtInfoButton`:DoInfoHelp()`

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

DoInfoPrefs

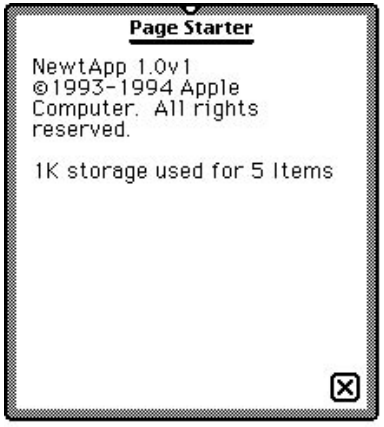
newtInfoButton`:DoInfoPrefs()`

If an Preferences view has been created, it is closed and set to `nil`. If no Preferences view is open, one is created.

newtAboutView

This is the view in which information about the application is stored in. The About view is displayed when the item, About, is chosen from the Info (“i”) button picker and the method `DoInfoAbout` is sent. It appears as shown in Figure 4-10.

Figure 4-10 The NewtApp About view.



newtPrefsView

This is the view in which information about the application is stored in. The About view is displayed when the item, About, is chosen from the Info (“i”) button picker and the method `DoInfoAbout` is sent. It appears as shown in Figure 4-11.

Figure 4-11 A NewtApp Preferences view.

newtActionButton

This proto provides the standard action button. If you have a card-style application and want routing, place this in the `menuRightButtons` (see page 4-56) and the framework will place it correctly on the status bar. The action button belongs next to the close box (to the left.) It appears as shown in Figure 4-12.

Figure 4-12 The Action button.

newtFilingButton

This proto provides the standard filing button but with added functionality. If you have a card-style application and want filing, place this in the `menuRightButtons` (see page 4-56) and the framework will place it correctly on the status bar. The filing button belongs next to the action box (to the left.) It appears as shown in Figure 4-13.

Figure 4-13 The Filing button.

newtAZTabs

This proto is used to include alphabetical tabs, arranged horizontally, in a view; it is based on the `protoAZTabs` but adds useful functionality to that base. (See `protoAZTabs` in Chapter 7, “Controls and Other Protos,”on page 7-30.) The `newtAZTabs` view appears as shown in Figure 4-14.

Figure 4-14 NewtApp A-Z tabs.

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

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

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

PickLetterScript

newtTabs`:PickLetterScript(`*letter*`)`

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

letter	The letter that was tapped.

newtFolderTab

This is the plain folder tab. If you want filing to operate correctly in your application, it must use either this proto or the `newtClockShowBar` proto. The `newtFolderTab` view is shown in Figure 4-15.

Figure 4-15 The plain folder tab.

newtClockShowBar

This folder tab incorporates a date and time indicator. It also is automatically updated if the current folder is deleted. If you want filing to operate correctly in your application, it must use either the `newtFolderTab` proto or the `newtClockFolderTab` proto, which is shown in Figure 4-16.

Figure 4-16 The digital clock and folder tab.

newtStatusBarNoClose

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

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

The status bar figures the correct size of the buttons in the `menuLeftButtons` and `menuRightButtons` arrays and places them correctly, when there is no `statusBarSlot` set in the `newtApplication` base view. It is recommended that you use these slots to ensure the correct justification of your status bar buttons with future enhancements. See the description of the `statusBarSlot` on page 4-39.

If the `statusBarSlot` in the base view (`newtApplication` proto) has been set, the appearance and disappearance of the buttons on the status bar

is governed by the values set for the `menuLeftButtons` and `menuRightButtons`, at the layout level of the application. See "newtLayout" beginning on page 4-58.

In essence, the buttons in the `menuLeftButtons` array are laid out from left to right (starting with the Info button) and vice versa, starting with the close box, for the buttons in the `menuRightButtons` array.

Slot descriptions

`menuLeftButtons`

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

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

`menuRightButtons`

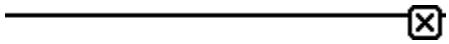
An array of standard text buttons. The elements in the array are laid out from right to left, so that the first element appears at the far right. An appropriate value is shown in the following code snippet.

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

newtStatusBar

This is based on the `newtStatusBarNoClose`. The only difference between the two is that this status bar includes a large close box at its far right side, as shown in Figure 4-17. Like the `newtStatusBarNoClose` proto you may use the `menuLeftButtons` and `menuRightButtons` arrays.

Figure 4-17 A Status Bar view.



Slot descriptions

`menuLeftButtons`

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

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

`menuRightButtons`

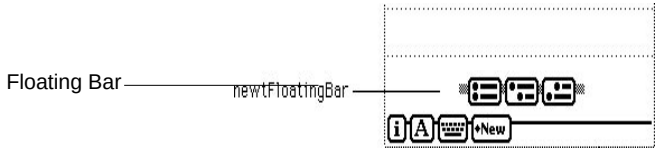
An array of standard text buttons. The elements in the array are laid out from right to left, so that the first element appears at the far right. An appropriate value is shown in the following code snippet.

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

newtFloatingBar

This is like a standard `newtStatusBar`, but it floats at the bottom of a view. It was originally designed for the Notes application where individual view types, like the Outline view, have their own menu buttons that are not necessary for the main application view. Like the `newtStatusBar` proto, it implements a `menuButtons` slot, in which you may enumerate the buttons to appear on the floating bar. A Floating Bar view appears as shown in Figure 4-18.

Figure 4-18 A Floating Bar view.



Slot descriptions

`menuButtons` An array of button protos. Buttons are laid out, an equidistant apart, on the status bar.

The layout protos and their methods follow.

newtLayout

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

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

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

Slot descriptions

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

`masterSoupSlot` Required. A symbol that refers to the soup in the `newtApplication.allSoups` frame that is the main soup of your application. It is used to set up the cursor

and soup query for your application. An appropriate value would be `'mySoup`.

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

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

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

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

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

Slot descriptions

`dataSoup` Set to the soup which contains the data this layout displays.

`dataCursor` The main cursor to the data soup; it points to the topmost visible entry.

Following are a few interesting methods defined in the `newtLayout` proto.

FlushData

myNewtLayout`:FlushData()`

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

NewTarget

myNewtLayout:`NewTarget()`

A handy utility function you can use to reset the view origin and redo the screen.

Retarget

myNewtLayout:`Retarget(`*setViews*`)`

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

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

ScrollCursor

myNewtLayout:`ScrollCursor(`*delta*`)`

Moves the cursor *delta* entries and resets it.

delta An integer which can be greater than 0 or less than or equal to 0, depending on the direction for the scroll and the amount to scroll.

If *delta* is not equal to 0 (and the cursor is valid) the cursor is moved that number of places.

A value less than or equal to 0 causes the cursor to reset to the end of the entries (for a scrolling end behavior of `'wrap` or `'beepAndWrap`) or to move to the next entry (for a scrolling end behavior of `'stop` or `'beepAndStop`.) A value greater than 0 causes the cursor to reset (for a scrolling end behavior of `'wrap` or `'beepAndWrap`) or to move to the previous entry (for a scrolling end behavior of `'stop` or `'beepAndStop`.)

SetUpCursor

myNewtLayout:`SetUpCursor()`

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

Scroller

myNewtLayout:`Scroller(`*numAndDirection*`)`

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

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

WARNING

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

ShowFoundItem

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

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

entry A valid soup entry.

finder A NewtApp-compatible finder.

ViewScrollDownScript

myNewtLayout:`ViewScrollDownScript()`

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

ViewScrollUpScript

myNewtLayout:ViewScrollUpScript()

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

newtRollLayout

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

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

IMPORTANT

Do not place the entry view of a paper roll -style application inside a layout view; instead, it must be in a layout file (in NTK) which is declared in an expression, like the following one, in the protoChild slot.

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

Slot description

protoChild	Required. Name of the layout file containing the view to be used to lay out the child views.The child view must be a newtRollEntryView. This is the most important newtRollLayout slot. Do not create the entry view within a layout view in a paper roll -style application. Instead create it in a separate layout file. An appropriate value for the protoChild slot of a newtRollLayout is: GetLayout("DefaultEntryView")

There are no new methods, specifically for the roll layout proto. However, it does have its own version of the `scroller` method, modified so it works with the long pages of the `newtRollLayout`. The `Scroller` method is documented on page 4-61.

newtPageLayout

Essentially, this layout allows one entry to be visible at a time; otherwise it acts the same as the roll layout. However, that one entry can be longer than the screen length.

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

IMPORTANT

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

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

Slot description

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

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

newtOverLayout

This is the default overview. It is based on `protoOverview`. (See page 6-104 to find out more about `protoOverview`.) It is singled out by the `newtApplication` proto so that overview events invoke it.

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

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

Slot descriptions

`masterSoupSlot` Required. A symbol that matches a value in the `allSoups` slot in the `newtApplication` base view.

`dataCursor` Required. Do not set this; value is inherited from the parent layout proto.

`name` Required. Set it to something meaningful, like "Overview."

`centerTarget` Optional. Defaults to `nil`. When set to `true`, the current entry is centered in the overview list.

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

`menuLeftButtons` Optional. If the `statusBarSlot` in the base view is set, this is used to replace the `menuLeftButtons` in the `newtStatusBar` in the main layout.

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

Several methods are defined in this proto.

Abstract

myNewtOverLayout:Abstract(*targetEntry*, *bbox*)

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

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

targetEntry	Required. The soup entry frame to be displayed.
bbox	Required. The bounding box defining the shape for the overview information. This includes a value for the left, right, top, and bottom.

A sample `Abstract` method example follows.

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

GetTargetInfo

myNewtOverLayout:GetTargetInfo(*targetType*)

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

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

This method returns a frame that has the following slots:

`target`	The value of the `target` slot in the view to which this message is sent.
`targetView`	The value of the `targetView` slot in the view to which this message is sent. If targetType is `'filing`, then this slot contains the value of the `targetApp` slot in the current view, instead.
`targetStore`	If the `target` slot is a soup entry, then the store on which the entry resides is returned in this slot.

HitItem

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

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

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

index	The index to the item in the list (the first one being zero).
x	The x coordinate of the tap, relative to the left edge of the item that was tapped.
y	The y coordinate of the tap, relative to the top edge of the item that was tapped.

newtRollOverLayout

Same as the `newtOverLayout` proto except that it is required to be used in a paper roll -style application. It is based on `newtOverLayout`. It is singled out by the `newtApplication` proto so that overview events invoke it.

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

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

Slot descriptions

`masterSoupSlot` Required. A symbol that matches a value in the `allSoups` slot in the `newtApplication` base view.

`dataCursor` Required. You do not set this, this value is inherited from the parent layout proto.

`name` Required. Set it to something easy to remember, like "Overview."

`centerTarget` Optional. Defaults to `nil`. When set to `true`, the current entry is centered in the overview list.

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

`menuLeftButtons` Optional. If the `statusBarSlot` in the base view is set, this is used to replace the `menuLeftButtons` in the `newtStatusBar` in the main layout.

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

The entry view protos, following, are the invisible container views for the protos that allow you to view and edit data (see page 4-74.) Entry views are essential because they set the `target` slot to refer to the soup entry which contains the data for the slot views to display.

newtEntryView

There are no unusual slots for you to set here—just the usual bounds and justify slots—and then only if you want to override the default settings.

The following slots are set automatically. Note that dataDefs and viewDefs are identified and used as target entries and target views in several `newtEntryView` slots.

WARNING

Do not change the values of any of the following slots or you application will not work correctly. ▲

Slot descriptions

`entryChanged`	When an entry is changed in a viewDef, this is set to true for flushing.
`entryDirtied`	If the targeted viewDef was changed once and a flush occurred, this is set to `true`. When the view is closed down, it checks this. If set, it does a broadcast soup change to other applications.
`target`	Set to the entry which is ready to display.
`viewJustify`	Optional. Defaults to parent full justify for horizontal and vertical `vjParentFullH + vjParentFullV`
`currentDataDef`	Set by the enclosed stationery view to the current dataDef. (See Chapter 5, “Stationery,”for more information. This is a convenient access point for things like the `newtEntryRollHeader` so it can pull out the appropriate icon from the `newtInfoBox`.
`currentViewDef`	Set by the enclosed stationery view to the current viewDef.
`currentStatView`	Set by the enclosed stationery view to the current context of the viewDef. If the target entry has a dataDef displayed, this points to it. Internal methods need to know the context for the view that contains the dataDef so that messages may be sent to it.

Following are some interesting methods that are defined for the `newtEntryView` proto and thus inherited by all the entry views which proto from it.

StartFlush

myEntryView:`StartFlush()`

This method is called to start the timer that flushes out the entry after a few seconds of inactivity. Normally this is called automatically by a dataDef, but if you have some other reason for causing an entry to be flushed, call this directly. Calling this sets the `entryChanged` slot and begins the timer.

EndFlush

myEntryView:`EndFlush()`

This method is called when the flush timer fires. Or, if you want an immediate flush, set `entryChanged` to `true` and call this method.

EntryCool

myEntryView:`EntryCool(`*report*`)`

Checks to see if the target entry is on read-only media. If the *report* parameter is set to `true` it displays the message “This is on a write protected card and cannot be changed.”

report If this is a non-`nil` value the notice, “This is on a write protected card and cannot be changed,” is displayed.

JamFromEntry

myEntryView:`JamFromEntry(`*otherEntry*`)`

This method looks for a `JamFromEntry` method in each child of the entry view and sends the message if one is found. It then retargets the view to display the changes. See the slot view’s redefinition of “JamFromEntry” beginning on page 4-76.

Retarget

myEntryView:Retarget()

This method changes the display for the viewDef(s) and dataDef(s) before conditionally sending the Retarget message to each child view. See the slot view's redefinition of Retarget on page 4-84.

newtFalseEntryView

This view, which is based on newtEntryView, allows for the use of the NewtApp framework's slot view protos without the rest of the NewtApp structure for updating entries. It is ideal for use in converting an existing non-NewtApp application to use the NewtApp slot view protos.

When you use slot views outside of a NewtApp application, you must put them in a newtFalseEntryView proto and make sure the target and targetView slots are set. This is accomplished by sending a Retarget message to the newtFalseEntryView whenever entries are changed.

Writing a changed entry back to the soup is the responsibility of the application. You may want to set up a flush timer, or at least write back changes when scrolling and closing.

Slot descriptions

targetSlot	Optional. Defaults to 'target. No need to reset it if the slot in the parent context of this view, which holds the current entry (or target), is named target. If not, set it to the symbol which refers to the slot in the parent context that holds the data from the target entry.
dataCursorSlot	Optional. Defaults to 'dataCursor. No need to reset it if the slot in the parent context of this view, which refers to the main soup cursor, is named dataCursor. If not, set it to the symbol which refers to the slot in the parent context that refers to the main soup cursor.
dataSoupSlot	Optional. Defaults to 'dataSoup. No need to reset it if the slot in the parent context of this view, which refers to the main soup, is named dataSoup. If not, set it to

the symbol which refers to the slot in the parent context that refers to the main soup.

`soupQuerySlot` Optional. Defaults to `'soupQuery`. No need to reset it if the slot in the parent context of this view, which refers to the soup query, is named `soupQuery`. If not, set it to the symbol which refers to the slot in the parent context that refers to the soup query.

```
cardSoupQuerySpec: {type: 'index,
                    indexPath: 'timeStamp}
```

The `soupQuerySlot` is then set to `'cardSoupQuerySpec`

Note that `newtFalseEntryView` inherits all the methods documented in the `newtEntryView` proto, though they have been altered slightly to provide a fake NewtApp application environment.

newtRollEntryView

This is based on the `newtEntryView` proto and is equivalent to it, except that it supports the paper roll-style of application (as implemented by the `newtRollEntryView` proto.) It dynamically sizes the entries depending on the size of the viewDef. Note that you must use stationery with this proto.

Slot descriptions

`target` Set by the system to point to the current entry.

`targetView` Refers to the `newtRollEntryView` proto, itself, so that routing and other system services can use it.

`bottomlessHeight`
Optional. Set to the constant `kEntryViewHeight`.

newtEntryPageHeader

This proto implements the standard header / divider bar for a page entry view. If this header is displayed in association with some stationery (a dataDef is the current target entry) and it has an icon assigned to its `icon`

slot (see page 5-21) then that icon is used at the far left of the header. Otherwise a default icon provided by the system is used there.

When you press the header icon on the left of the bar, the `newtInfoBox` proto is automatically opened. (See the `newtInfoBox` proto on page 4-73.) If your entry has a title slot, the title will be displayed in the area where the date is shown, otherwise the date is displayed. You can see all of these features, from the built-in Notes application.

Figure 4-19 A page header.

newtEntryRollHeader

This proto implements the standard header / divider bar in a roll entry view. If this header is displayed in association with some stationery (a dataDef is the current target entry) and if the dataDef has an icon assigned to its `icon` slot (see page 5-21) it is used at the far left of the header. Otherwise, a default icon provided by the system is used.

When you tap the header icon, a `newtInfoBox` proto is automatically displayed. (See the `newtInfoBox` proto on page 4-73.) If your entry has a `title` slot, the title is displayed, otherwise the date is displayed. You can see all of these features in the built-in Notes application. A roll header appears as shown in Figure 4-20.

Figure 4-20 A roll header.

Slot descriptions

`hasFiling`	Optional. Defaults to `true`. Set to `nil` for no filing or action buttons.
`resizable`	Optional. Defaults to `true`. Set to `nil` for no drag resizing.

newtEntryViewActionButton

This is the standard action button. It must be a child of the entry view. It handles the usual routing actions but in the entry view rather than the application base view context.

newtEntryViewFilingButton

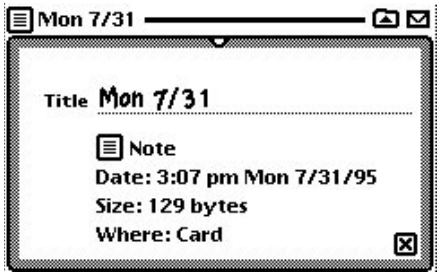
This is the standard filing button but it must be a child of the entry view. It handles the usual filing things but in the entry view rather than the application base view context.

newtInfoBox

This is a floating view which is based on `protoFloatNGo`. It displays informational text including the date, the size of the target entry, and the storage location of the entry. It also contains an input line, with the label "Title." If the text on that line is changed, the new text is saved automatically and displayed next to the icon on the title bar after the proto is closed.

If your application uses stationery, the icon you declared in the icon slot, is used next to its description, which is also taken from the dataDef. You need to add nothing to get a view that looks very similar to the one from the built-in Notes application shown in Figure 4-21.

Figure 4-21 A NewtApp information slip.



Slot descriptions

`icon`	Optional. An icon representing the object about which the information is provided.
`description`	Optional. A string describing the entry being displayed.

The Slot View Protos

The slot view protos include all the protos you use to view and edit the data which is held in the slots of a soup entry. The slot view protos usually have a one to one correspondence with soup slots.

There are two categories of slot views

- the simple read-only (RO) and edit views
- the labelled input line protos

All slot views assume a soup entry has been set by the parent proto as the value of the `target` slot. The `target` is a reference to the soup entry containing the data to be displayed in a slot view. This soup entry will also store the user-entered data.

This is set at runtime by the NewtApp framework, where `target` is a slot defined in the `newtApplication` base view. The `targetView` is the `newtEntryView` proto that contains the slot view in which the target data is to be displayed.

When slot views are used outside of a NewtApp application, the `target` and `targetView` slots must be set by you. In this case, the slot view protos must be contained by a `newtFalseEntryView` proto, which must be the view referred to by the `targetView` slot. (For a full description of the `newtFalseEntryView` proto see page 4-70.)

Slot views also require a `path` slot. Depending on the proto, this slot must be a path expression leading to a slot that holds a certain kind of data. For instance, the `path` slot of a `newtROTextDateView` proto must refer to a slot in an entry that contains dates.

Also included in this view category, is a lock proto (`newtEntryLockedIcon` on page 4-83) which you can use to indicate locked media or read-only views and a stationery placeholder proto (`newtStationeryView` on page page 4-83) which is used to provide a bounding box for your dataDef stationery component.

Slot description

`path`	Required for all slot views. A symbol that is a path expression to the slot in the target frame where the initial value for the input line resides, and in which the final value is to be stored. The slot identified by the path expression should contain the specified data for the specific slot view.

Also defined for the slot view protos, is a `TextScript` method which displays the text for the target entry and a `JamFromEntry` method that puts the path of a new entry into the `path` slot. These work for all simple slot views.

TextScript

aSlotView`:TextScript()`

Returns a text representation of the data at the specified path in the target soup entry for any slot view in your application.

JamFromEntry

aSlotView:JamFromEntry(*otherEntry*)

This method replaces the path expression in the path slot with a new path expression. The new path is formed by appending the value of the otherEntry parameter to the path expression that leads to the soup entry in which the slot resides, which it obtains from the jamSlot (if it's not nil.)

This essentially resets the target entry to a different entry and causes the display to change so that the user is looking at the new value.

For an example of when you might want to use this method, imagine you are coding an order entry system. You want the customer address stored in the order, but it's in the Names soup. To extract the data, you set the jamSlot to a path expression which leads to the address in the Names soup and send the JamFromEntry message with the Names soup entry as the value of the *otherEntry* parameter.

otherEntry	A soup entry. This is intended to be an entry other than the one to which the entryView is already targeted.

Following is the documentation for the simple slot view protos.

newtROTextView

This proto displays read-only text. It is the base proto for the rest of the simple slot views.

Slot descriptions

path	Required. The slot identified by this path expression is the slot from which the initial text to display in this view is gotten, and in which the final value is to be stored.
styles	Optional. Defaults to nil.
tabs	Optional. Defaults to nil.
jamSlot	Optional. Defaults to nil. If this view has a jamSlot that is not nil, the slots from an entry that is passed to

the `JamFromEntry` method will be placed (“jammed”) into the soup slot referred to by `path`.

The `jamSlot` may be set to a path expression that defines the path to use to extract data from a slot in an entry, when the entry is not the one already targeted by the entry view (which encloses the slot view.)

newtTextView

This is the other base class for the slot views; it is based on the read-only text view (`newtROTextView`.) Use it to display editable text that does not need a label.

Slot descriptions

`path`	Required. The slot identified by this path expression is the slot from which the initial text to display in this view is gotten, and in which the final value is to be stored.
`styles`	Optional. Defaults to `nil`.
`tabs`	Optional. Defaults to `nil`.
`jamSlot`	Optional. Defaults to `nil`. If this view has a `jamSlot` that is not `nil`, the slots from an entry that is passed to the `JamFromEntry` method will be placed (“jammed”) into the soup slot referred to by `path`. The `jamSlot` may be set to a path expression that defines the path to use to extract data from a slot in an entry, when the entry is not the one already targeted by the entry view (which encloses the slot view.)

newtRONumView

A read-only view for numbers which is based on the NewtApp read-only text view (`newtROTextView.`) It has functionality added for number formatting.

Slot descriptions

`path`	Required. The slot identified by this path expression is the slot from which the initial number to display in this view is gotten, and in which the final value is to be stored.
`format`	Optional. The format string for displaying the data defaults to `%.10g` and a 10 place decimal.
`integerOnly`	Optional. Defaults to `true`, signaling that conversion from text to number should result in an integer.

newtNumView

An editable number view which is based on the read-only number view (`newtRONumView`) and inherits its slots. Specify number formatting by assigning values to the `format` and `integerOnly` slots.

Slot descriptions

`path`	Required. The slot identified by this path expression is the slot from which the initial number to display in this view is gotten, and in which the final value is to be stored.
`format`	Optional. The format string for displaying the data defaults to: `%.10g` and a 10 place decimal.
`integerOnly`	Optional. Defaults to `true`, signaling that conversion from text to number should result in an integer.

newtROTextDateView

This proto is set to contain text and dates. Depending on which of the two slots, `longFormat` or `shortFormat`, is non-`nil`, this proto displays either long or short dates, such as, February 29, 1984, and 2/29/84, respectively.

Slot descriptions

`path`	Required. The slot identified by this path expression is the slot from which the initial text and/or date to

	display in this view is gotten, and in which the final value is to be stored.
`longFormat`	Optional. Defaults to `yearMonthDayStrSpec`, which is a format for use by the `LongDateStr` function. The `longdate` specification is defined by the system. Either this slot or the `shortFormat` slot should not be `nil` so the view can choose the format.
`shortFormat`	Optional. Defaults to `nil`. This is a format defined by the system for use by the `ShortDateStr` function.

newtTextDateView

This editable view protos from its read-only version (`newtROTextDateView`) and inherits its slots.

Slot descriptions

`path`	Required. The slot identified by this path expression is the slot from which the initial text and / or date to display in this view is gotten, and in which the final value is to be stored.
`longFormat`	Optional. Defaults to `yearMonthDayStrSpec`, which is a format for use by the `LongDateStr` function. The `longdate` specification is defined by the system. Either this slot or the `shortFormat` slot should not be `nil` so the view can choose the format. See Chapter 19, “Localizing Newton Applications,” for more information about this function.
`shortFormat`	Optional. Defaults to `nil`. This is a format defined by the system for use by the `ShortDateStr` function. See Chapter 19, “Localizing Newton Applications,” for more information about this function.

newtROTextTimeView

This proto is based on the `newtROTextView` but has functionality added to display and format a time string. The slot to be displayed must contain a time or text.

Slot descriptions

`path`	Required. The slot identified by this path expression is the slot from which the initial text and/or time to display in this view is gotten, and in which the final value is to be stored.
`format`	Optional. Defaults to `ShortTimeStrSpec which is a` format for use by the `TimeStr` function. See Chapter 19, "Localizing Newton Applications," for more information about this function.

newtTextTimeView

This editable view protos from its read-only version (`newtROTextTimeView`) and inherits its slots.

Slot descriptions

`path`	Required. The slot identified by this path expression is the slot from which the initial text and/or time to display in this view is gotten, and in which the final value is to be stored.
`format`	Optional. A format for use by the `TimeStr` function. Defaults to `ShortTimeStrSpec`. See Chapter 19, "Localizing Newton Applications," for more information about this function.

newtROTextPhoneView

This view, which is based on the `newtROTextView` proto, displays a telephone number from the application soup.

Slot description

`path`	Required. The slot identified by this path expression is the slot from which the initial phone number to display in this view is gotten.

newtTextPhoneView

This view, which is based on the `newtROTextView` proto, formats a number, which is entered into it by a user, as a telephone number.

Slot description

`path`	Required. The slot identified by this path expression is the slot from which the initial numbers to display in this view is gotten, and in which the final value is to be stored.

newtROEditView

This is a fixed size edit view which will display the application soup. It may also be set up to have its own scrollers. This is accomplished by setting the `optionFlags` slot.

Slot description

`doCaret`	Optional. Defaults to `true`, which autosets the caret.
`optionFlags`	Optional. Defaults to `kNoOptions` (which has a numeric value of 0) and sets the scrollers to not show. The constant `kHasScrollersOption` (which has a numeric value of 1<<0) sets them to show.
`viewLineSpacing`	Optional. Defaults to 28.

This proto also defines the method, `ScrollToWord`, for your convenience.

ScrollToWord

editView`:ScrollToWord(`*words, hilite*`)`

This method finds the specified word, scrolls the edit view to the found word, and highlights it—if the *hilite* parameter is true. If no match is found for the specified word in any of the view children of the edit view,

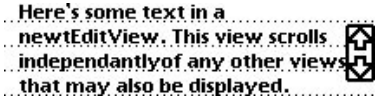
`ScrollToWord` does nothing. Note that this method does not work in paper roll layouts.

words	May be a string or an array of single words.
hilite	If `true`, the matching text of the paragraph view is highlighted.

newtEditView

This view protos from its read-only version (`newtROEditView`) and behaves simply, somewhat like a `clEditView`. (See "General Input Views" beginning on page 8-9.) Unlike the read-only version, this proto accepts user-entered text. A `newtEditView`, with scroll bars showing, appears as seen in Figure 4-22.

Figure 4-22 A newtEditView proto.



newtCheckBox

This view is based on the `protoCheckBox`, documented on page 7-26. Basically it works so that the check mark is on when the value of the `target.(path)` slot is equal to the value of the `assert` slot. If you want more complex behavior you may override the `ViewSetupFormScript` and the `ValueChanged` method.

Slot descriptions

`assert`	Optional. Defaults to `true`.
`negate`	Optional. Defaults to `nil`.

This proto implements the following two methods.

ViewSetupFormScript

`ViewSetupFormScript()`

Defined to simply check the value of `target.(path)` for equality against the value of the `assert` slot. Override this method for more complex behavior.

ValueChanged

`ValueChanged()`

If the equality check in the `ViewSetupFormScript` is non-`nil` then the slot `target.(path)` is set to the `assert` value otherwise it is set to the `negate` value. Override this method for more complex behavior.

newtStationeryView

This view holds nothing, its function is to give a viewDef its bounding box. It has an optional `path` slot which specifies where the soup entry exists that it will display. If the `path` slot is `nil` or missing, the entire `target` entry is used. Note that this proto is different from the `newtStationery` proto which you must use to create a dataDef. (For more information about the `newtStationery` proto, see page 5-21.)

Slot description

`path`	Optional. If `path` is missing or `nil`, the stationery view assumes it is to display the whole target entry.

newtEntryLockedIcon

You use this proto to show a lock icon if the slot is on locked media, a ROM card, or contained in a read-only view. The `newtEntryLockedIcon` proto is set to either show or not show when your view is opened. It is based on the `clIconView` proto.

Slot descriptions

`icon`	Optional. Defaults to `nil`, it may also have the value `lockedIcon`.

The following methods are defined internally to `newtEntryLockedIcon` and should not be changed or it will not work as documented.

Retarget

`Retarget()`

This method is defined so that it calls `SetIcon` to show either the locked or unlocked icon (according to whether the store is locked or in ROM) and redraws the screen.

SetIcon

`SetIcon()`

Checks the target soup entry to find out if it is or locked or in ROM. If it is, the locked icon is displayed.

The Labelled Input Line Slot Views

The NewtApp labelled input line protos function very similarly to the `protoLabelInputLine` family of protos. (If you are not familiar with those protos, you may look them up in Chapter 8, “Text and Ink Input and Display.”)

In addition to its label and popup menu capability these protos include the `flavor` and `access` slots, which are unique to the NewtApp label input line slot views. The `access` slot limits the type of access each label input line slot view allows. The `flavor` slots contain references to the NewtApp filter

protos. These protos assign the appropriate pickers and the correct formatting for the intended data type. They are enumerated in Table 4-1 .

Table 4-1 The NewtApp filters, used to set the `flavor` slot.

Filter	Description	Slots
`newtTextFilter`	This is the filter upon which the other filter protos are based. It allows the label input line proto, which uses it as the value of its `flavor` slot, to accept text input.	This proto contains no slots for you to set.
`newtIntegerFilter`	This filter is based upon the `newtTextFilter` proto. It is set to accept integers only as input and contains a `format` slot which you may set.	`format`: Optional. Defaults to `%.10g`. You should change this as needed.
`newtNumberFilter`	This filter is based upon the `newtIntegerFilter` proto. It is set to accept all numbers as input and contains a `format` slot which you may set.	`format`: Optional. Defaults to `%.10g`. You should change this as needed.
`newtDateFilter`	This filter is based upon the `newtTextFilter` proto. It is set to accept dates as input and contains two format slots which you may set. Of these slots, one must be set to a value that is not `nil`.This proto specifies that the DatePopup picker is to be used.	`shortFormat`: Optional. Defaults to `nil`. May be set to a format used by the `ShortDateStr` function. `longFormat`: Optional. Defaults to `yearMonthDayStrSpec`, which is a format used by the `LongDateStr` function.

Table 4-1 The NewtApp filters, used to set the `flavor` slot.

Filter	Description	Slots
`newtSimpleDateFilter`	This filter is based upon the `newtDateFilter` proto and, like that proto, is set to accept and format dates. This filter allows dates that look like 5/15/55 or 5/15 and is useful for birthday input lines. It also contains two format slots, one of which must be set to a value that is not `nil`.	`shortFormat`: Optional. Defaults to `nil`. May be set to a format used by the `ShortDateStr` function. `longFormat`: Optional. Defaults to `monthDayStrSpec`, which is the format used by the `LongDateStr` function to withhold the year.
`newtTimeFilter`	This filter is based upon the `newtTextFilter` proto. It contains a `format` and `increment` slot which you may set. If an input line of a `newtTimeFilter` flavor uses a popup menu, a TimePopup picker is specified by this proto.	`format`: Optional. Defaults to `shortTimeStrSpec`. You should change this as needed. `increment`: Optional. Defaults to 10.
`newtDateNTimeFilter`	This filter is based upon the `newtTextFilter` proto. It contains the slots, `format`, `longFormat`, and `shortFormat`, which you may set. Note that of the two slots, `longFormat` and `shortFormat`, one must be set to a value that is not `nil`. If an input line of a `newtDateNTimeFilter` flavor uses a popup menu, a DateNTimePopup picker is specified by this proto.	`shortFormat`: Optional. Defaults to `nil`. May be set to a format used by the `ShortDateStr` function. `longFormat`: Optional. Defaults to `yearMonthDayStrSpec`, which is the format used by the `LongDateStr` function to withhold the year. `format`: Optional. Defaults to `shortTimeStrSpec`. You should change this as needed.
`newtPhoneFilter`	This filter is based upon the `newtTextFilter` proto and is used to format numbers as phone numbers.	`kind`: Optional. Defaults to `nil`. The built-in types include `fax`, `home`, and `work` and is used to change the label for the input line.

Table 4-1 The NewtApp filters, used to set the `flavor` slot.

Filter	Description	Slots
`newtCityFilter`	This filter is based upon the `newtTextFilter` proto and is used to format text as cities.	This proto contains no slots for you to set.
`newtStateFilter`	This filter is based upon the `newtTextFilter` proto and is used to format text as state names or abbreviations. If an input line of a `newtStateFilter` flavor uses a popup menu, a LocationPopup picker is specified by this proto.	This proto contains no slots for you to set.
`newtCountryFilter`	This filter is based upon the `newtTextFilter` proto and is used to format text as country names or abbreviations. If an input line of a `newtCountryFilter` flavor uses a popup menu, a LocationPopup picker is specified by this proto.	This proto contains no slots for you to set.
`newtSmartNameFilter`	This filter is based upon the `newtTextFilter` proto and is used to present the Names soup to the user, who may chose a name which then appears on the input line. If an input line of a `newtSmartNameFilter` flavor uses a popup menu, a `protoPeoplePopup` picker is specified by this proto.	This proto contains no slots for you to set.

newtProtoLine

This is the base view for the input line protos. This proto inherits behavior from both the view class, `clView`, and the proto, `newtROTextView`. In addition, it contains a lot of code for setting up the pop-up label picker and interpreting menu item commands.

Most of the following slots are included for your information only. You should not change them for the built-in protos or they will not work as expected. The only slot you should change, for the built-in protos, is the `label` slot. If any of the other slots are changed they will not work as planned.

Slot descriptions

`label` Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input line label.

`labelCommands` Optional. An array of strings that should appear in a picker when the user taps on the label. If this slot is supplied, the picker feature is activated and the label is shown with a diamond to its left to indicate that it is a picker. The currently selected item in the list, if there is one, is marked with a check mark to its left. A good default value is

```
["picker option one", "picker option two"]
```

`curLabelCommand`

Optional. If the `labelCommands` slot is supplied, this slot specifies which item in that array should be initially marked with a check mark. Specify an integer, which is used as an index into the `labelCommands` array. If you omit this slot, no item is initially marked with a check

	mark. Note that you must update this value when a different value is chosen.
`usePopup`	Optional. Defaults to true. When set to true and you provide a `labelCommands` array, the input line label displays a diamond, indicating a picker (pop-up menu.)
`access`	Optional. Defaults to `'readWrite`. Valid values include `'readWrite`, `'readOnly`, `'pickOnly`. Do not change this value for the built-in protos or they will not work as expected.
flavor	Optional. Defaults to `newtFilter`. Do not change this value for the built-in protos or they will not work as expected.
memory	Optional. Defaults to `nil`. This keeps track of the most recent choices and displays them as items in the picker.

It also contains the following methods, defined for your convenience.

ChangePopup

`ChangePopup(`*item, entry*`)`

Assuming there is a picker menu, this method allows you to change a menu item before it is displayed. For example, if you do a name query, but want to display “Bob Johnson, Apple” instead of just “Bob”, use this method to do it. If `ChangePopup` isn’t defined, the menu just shows the original data.

item	An item to be displayed in the picker menu.
entry	The entry corresponding to the item selected from the picker menu.

UpdateText

`UpdateText(`*newText*`)`

A utility for updating text for an Undo action. It changes the old text to the text passed in as a parameter and posts that change to the Undo system service.

newText	A string to which the entry will be changed which is passed in as the parameter to this method.

newtLabelInputLine

This is like `protoLabelInputLine` and can use all the slots available to that proto. (See Chapter 8, “Text and Ink Input and Display.”) It also shares some behavior (like `jamSlot`, etc.) with the text view and is based on the `newtProtoLine` proto.

The `newtLabelInputLine` proto is a one-line input field that includes a text label at its left. When a `labelCommands` array is provided, a diamond appears to the left of the label and the contents of the array appear in a picker menu. Without `labelCommands`, the `newtLabelInputLine` proto appears as shown in Figure 4-23.

Figure 4-23 A NewtApp label input line.

Slot descriptions

`access`	`'readWrite`
`label`	Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input line label. An example is: `"Some Text:"`
`labelCommands`	Optional. An array of strings that should appear in a picker when the user taps on the label. If this slot is supplied, the picker feature is activated and the label is shown with a diamond to its left to indicate that it is a picker. The currently selected item in the list, if there is one, is marked with a check mark to its left. A good

	default value is: `["picker option one", "picker option two"]`
`usePopup`	Optional. Defaults to true. When set to true and you provide a `labelCommands` array, the input line label displays a diamond, indicating a picker (pop-up menu.)
`path`	Required. The path expression should identify the soup slot where the text will be saved. An example is: `[pathExpr: kAppSoupSymbol, 'someText]`
`flavor`	`newtTextFilter`

newtROLabelInputLine

This is the same as the editable label input line except that there is no dotted line and the text displayed is read-only.

Slot descriptions

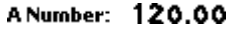
`access`	`'readOnly`
`label`	Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input line label. An example is: `"Some Text:"`
`labelCommands`	Optional. An array of strings that should appear in a picker when the user taps on the label. If this slot is supplied, the picker feature is activated and the label is shown with a diamond to its left to indicate that it is a picker. The currently selected item in the list, if there is one, is marked with a check mark to its left. A good default value is: `["picker option one", "picker option two"]`
`usePopup`	Optional. Defaults to true. When set to true and you provide a `labelCommands` array, the input line label displays a diamond, indicating a picker (pop-up menu.)
`path`	Required. The path expression should identify the soup slot where the text will be saved. An example is: `[pathExpr: kAppSoupSymbol, 'someText]`
`flavor`	`newtTextFilter`

newtROLabelNumInputLine

This (the read-only version and its editable counterpart) is the numeric equivalent of the `newtLabelInputLine` protos. It is based on the `newtProtoLine` proto but has a `newtNumberFilter` as the value of its `flavor` slot, which imparts number formatting to it.

The read-only display consists of the label designated in the `label` slot and the data stored in the location specified by the `path` slot but without a dotted line for the input line. Note that it is not possible to create a picker for a `newtROLabelInputLine`. An example is shown in Figure 4-24.

Figure 4-24 A NewtApp label display line for text.



Slot descriptions

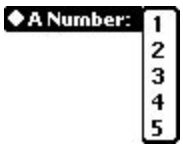
`access`	`'readOnly`
`label`	Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input line label. An example of a valid value is: `"A Number:"`
`labelCommands`	Optional. An array of strings that should appear in a picker when the user taps on the label. If this slot is supplied, the picker feature is activated and the label is shown with a diamond to its left to indicate that it is a picker. The currently selected item in the list, if there is one, is marked with a check mark to its left. A good

	default value is: `["picker option one", "picker option two"]`
`usePopup`	Optional. Defaults to true. When set to true and you provide a `labelCommands` array, the input line label displays a diamond, indicating a picker (pop-up menu.)
`path`	Required. A path expression of the form: `[pathExpr: yourSoupSymbol, 'aNumber]`
`flavor`	`newtNumberFilter`

newtLabelNumInputLine

This is the same as the read-only number input line except that data may be entered on the dotted input line and will be saved to the data location specified in the `path` slot. The proto, in which a `labelCommands` array with the specified value `["1","2","3","4","5"]` and a `true` value for the `usePopup` slot, is shown in Figure 4-25.

Figure 4-25 A NewtApp label number input line.



Slot descriptions

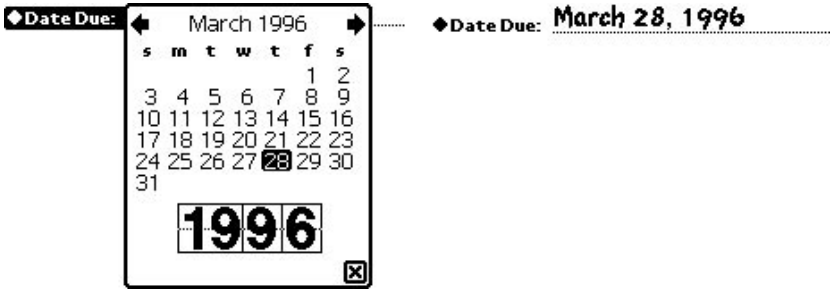
`access`	`'readWrite`
`label`	Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input line label. An example of a valid value is: `"A Number:"`
`labelCommands`	Optional. An array of strings that should appear in a picker when the user taps on the label. If this slot is supplied, the picker feature is activated and the label is shown with a diamond to its left to indicate that it is a

	picker. The currently selected item in the list, if there is one, is marked with a check mark to its left. A good default value is: `["picker option one", "picker option two"]`
`usePopup`	Optional. Defaults to true. When set to true and you provide a `labelCommands` array, the input line label displays a diamond, indicating a picker (pop-up menu.)
`path`	Required. A path expression of the form: `[pathExpr: yourSoupSymbol, 'aNumber]`
`flavor`	`newtNumberFilter`

newtLabelDateInputLine

Inputs dates through a system-provided picker or by directly entering it on the input line. A label date input line view is shown in Figure 4-26.

Figure 4-26 A NewtApp label date input line.



When a date is entered on the input line, the calendar changes to match. If the date is written in any other format than the one shown in Figure 4-26, it is accepted and recognized but is changed automatically to the date format shown in the figure.

Note that neither the `labelCommands` nor the `usePopup` slot is necessary with this proto. The pop-up menu is specified in the `newtDateFilter`.

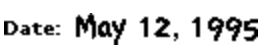
Slot descriptions

`access`	`'readWrite`
`label`	Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input line label.
`path`	Required. A path expression, that leads to a slot with a date in it, of the form: `[pathExpr: soupSymbol, 'aDate]`
`flavor`	`newtDateFilter`

newtROLabelDateInputLine

This is the same as the editable date label input line except that it is used to display, not edit, a date from a soup slot. As with all the read-only input line protos, the dotted line disappears when it is displayed. An example is shown in Figure 4-27.

Figure 4-27 A `newtROLabelDateInputLine` proto.



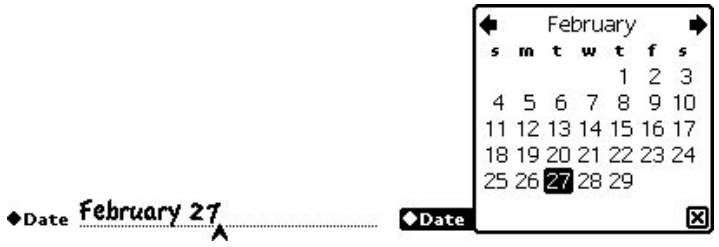
Slot descriptions

`access`	`'readOnly`
`label`	Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input line label.
`path`	Required. A path expression, that leads to a slot with a date in it, of the form: `[pathExpr: soupSymbol, 'aDate]`
`flavor`	`newtDateFilter`

newtLabelSimpleDateInputLine

This proto accepts simple dates; dates without the year, such as 7/24 and July 24, in addition to fully-specified dates, such as 7/24/88 and July 24, 1988. It is useful for birthday and anniversary fields. The `newtLableSimpleDateInputLine` proto is based on the `newtProtoLine` proto. It appears as shown in Figure 4-28.

Figure 4-28 The simple date input line.



Slot descriptions

`access`	`'readWrite`
`label`	Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input line label.
`path`	Required. A path expression, that leads to a slot with a date in it, of the form: `[pathExpr: soupSymbol, 'birthday]`
`flavor`	`newtSimpleDateFilter`

newtNRLabelDateInputLine

This proto is based on `newtProtoLine` and allows date input through a system-provided DatePopup picker. The initial display is simply the label with a diamond to its left and no input line following it. Once a date has

been displayed, any attempt to edit it will cause the date picker to display. It appears as shown in Figure 4-29.

Figure 4-29 Date input with picker only access.

Slot descriptions

`access`	`'pickOnly`
`flavor`	`newtDateFilter`
`label`	Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input line label.
`path`	Required. A path expression that leads to a slot with a date in it, of the form: `[pathExpr: yourSoupSymbol, 'date]`

newtROLabelTimeInputLine

This proto is based on `newtProtoLine` and is set to simply display a time. No input or editing are recognized.

Slot descriptions

`label`	Optional. A string which labels the input line.
`path`	Required. A path expression, that leads to a slot with a time in it, of the form: `[pathExpr: soupSymbol, 'time]`
`flavor`	`newtTimeFilter`
`access`	`'readOnly`

newtNRLabelTimeInputLine

This allows date input through a system-provided TimePopup picker only. The picker is specified by the `newtTimeFilter`, which is the value of its `flavor` slot. You should not change this or the proto will not work as expected. It is based on `newtProtoLine`. It appears as shown in Figure 4-30.

Figure 4-30 Time input with picker only access.

Slot descriptions

`label`	Optional. A string which labels the input line.
`flavor`	`newtTimeFilter`; do not change this or the proto will not work as expected.
`access`	`'pickOnly`

newtLabelTimeInputLine

This provides a labelled input line for time. When it initially displays the line is blank and a diamond appears to the left of the label. When the label is tapped a time picker displays. It appears as shown in Figure 4-31.

Figure 4-31 A `newtLabelTimeInputLine` proto.

Slot descriptions

`label`	Optional. A string which labels the input line.
`_proto`	`newtProtoLine`
`flavor`	`newtTimeFilter`

newtNRLabelDateNTimeInputLine

This proto is set up to contain times and dates and is based on `newtProtoLine`. Depending on which of the two slots, `longFormat` or `shortFormat`, is non-`nil`, this proto displays either long or short dates, such as, February 29, 1984, and 2/29/84, respectively. For more information about these formats, which are used in calls to `LongDateStr` and `ShortDateStr`, see "Formatting Date and Time Values" beginning on page 19-14.

Slot descriptions

`flavor`	`newtDateNTimeFilter`
`access`	`'pickOnly`
`path`	Required. Must be a soup slot that holds a date and time.
`longFormat`	Optional. Defaults to `yearMonthDayStrSpec`. The `longdate` specification as defined by the system. Either this slot or the `shortFormat` slot should be non-`nil` so the view can choose the format.
`shortFormat`	Optional. Defaults to `nil`. This is a `shortdate` specification as defined by the system. Either this slot or the `longFormat` slot should be non-`nil` so the view can choose the format.

newtLabelPhoneInputLine

This proto formats numbers as phone numbers just like the `newtTextPhoneView`, (see page 4-81) except that this proto has a label. It is based on `newtProtoLine`.

Slot description

`flavor`	`newtPhoneFilter`
`access`	Optional. Defaults to `'readWrite`. Valid values include `'readWrite`, `'readOnly`, `'pickOnly`. Do not change this value for the built-in protos or they will not work as expected.
`label`	Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input line label.
`labelCommands`	Optional. An array of strings that should appear in a picker when the user taps on the label. If this slot is supplied, the picker feature is activated and the label is shown with a diamond to its left to indicate that it is a picker. The currently selected item in the list, if there is one, is marked with a check mark to its left. A good default value is `["picker option one", "picker option two"]`
`curLabelCommand`	Optional. If the `labelCommands` slot is supplied, this slot specifies which item in that array should be initially marked with a check mark. Specify an integer, which is used as an index into the `labelCommands` array. If you omit this slot, no item is initially marked with a check mark. Note that you must update this value when a different value is chosen.
`usePopup`	Optional. Defaults to true. When set to true and you provide a `labelCommands` array, the input line label displays a diamond, indicating a picker (pop-up menu.)
memory	Optional. Defaults to `nil`. This keeps track of the most recent choices and displays them as items in the picker.

newtSmartNameView

This proto is able to pull names from the Names application soup. It is based on `newtProtoLine`, so it also implements a label. When you use it, a tap to the picker menu item, Other, displays `protoPeoplePopup` picker with the names from the Names soup. The `newtSmartNameFilter`, which is the value of the `flavor` slot, specifies it.

Slot description

`flavor`	`newtSmartNameFilter`
`access`	`'readWrite`
`label`	Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input line label.
`path`	Required. A path expression leading to the slot in the application soup where data changes should be stored.

Summary of the NewtApp Framework

Data Structures

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 basic 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

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()   ...

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)...

}
```

newtApplication

```
myNewtAppBaseView := {
_proto: newtapplication, // Application base view proto
   appSymbol: '|IOU:PIEDTS| //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;
               // 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 viewDef layout files.

   superSymbol: '|appName:SIG| //Unique symbol identifying
                        //superSet of application's soups

   doCardRouting:true //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
                     // 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
                     // 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.
            }
```

newtRollLayout

```
myRollLayout:= { // Dynamically lays out children 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
            }
```

newtPageLayout

```
myPageLayout:= { // Dynamically lays out children 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
            }
```

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.
   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)...,
            }
```

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
   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)...,
            }
```

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
   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
            }
```

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/divider bar for page-style apps
_proto: newtEntryPageHeader,
   // contains no additional slots or methods
            }
```

newtEntryRollHeader

```
aRollHeader:= {   // Header/divider bar for 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
_proto: newtEntryViewActionButton
            }
```

newtEntryViewFilingButton

```
anEntryFilingButton:= {// Filing button to use on headers
_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)..., //
            }
```

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)..., //
            }
```

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
ViewSetupForm: // Is target.(path)= assert?
   func()..., //
ValueChanged: // Changes target.(path) value to it's
   func()..., // opposite either true or false
            }
```

newtStationeryView

```
stationeryView := {    // Used as bounding box for viewDef
_proto: newtStationeryView
   path     // If missing or nil, it displays the whole
            // target entry.
            }
```

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 it's
   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
   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
   labelCommands: ["","",],// Picker options
   curLabelCommand: integer,// Integer for current command
   usePopup: true,// When true with labelCommands array
                  // picker is enabled
   access: 'readOnly,// Could be 'readWrite 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)..., //
            }
```

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)..., //
            }
```

newtROLabelNumInputLine

```
aDisplayLabelNumberInputLine:= {// Labelled number display line
_proto: newtROLabelNumInputLine,
   access: 'readOnly,// Could be 'readWrite or 'pickOnly
   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
   access: 'readOnly,// Could be 'readWrite or 'pickOnly
   flavor: newtDateFilter,//
   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
_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 or 'readWrite
   flavor: newtDateFilter,// Don't change
   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
   access: 'readOnly,// Could be 'readWrite or 'pickOnly
   flavor: newtTimeFilter,//
   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,//
   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 or 'readWrite
```

```
   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
   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: newtPhoneFilter,//
   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)..., //
            }
```

Stationery

You may add stationery, which consists of new data formats and different views of your data, to a Newton application, either as an extension, or as a built-in part of the application. These items are available through the pickers (pop-up menus) of the New or Show stationery buttons.

Stationery is tightly integrated into the NewtApp framework and works best when incorporated by a NewtApp application. You'll probably want to read this chapter if you are building applications using the NewtApp framework.

To use stationery 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 14, "Find," Chapter 15, "Filing," and Chapter 2, "Routing Interface," of the *Newton Programmer's Guide: Communications.*

The examples in this chapter use the Newton Toolkit (NTK) development environment. Therefore, you should also be familiar with NTK or learn it before you try them. Consult the "*Newton Toolkit User's Guide*" for that information.

This chapter documents:

- how to create stationery and tie it into an application
- how to create, register, and install an extension to the Notes application
- the stationery protos
- the stationery methods and global functions

The material in this chapter is still preliminary and will change in the future.

About Stationery

Stationery application extensions provide different ways of structuring data and various ways to view your data. To add stationery to your application, you must create a **data definition** called a dataDef and an adjunct **view definition** called a viewDef. Both of the stationery components are created as view templates, though only the viewDef displays as a view at runtime. Stationery always consists of at least one dataDef which has one or more viewDef associated with it.

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 the `superSymbol`, that identifies the application soup into which its data entries are to be subsumed. It also contains a `names` slot where the strings are defined that appear 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 “Stationery Reference.” Read about view protos in Chapter 3, “Views.”)The viewDef is primarily 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.

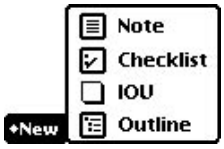
You may include or add stationery to any NewtApp application or any application that already uses stationery. The stationery components you create appear as items in the picker (pop-up menus) of the New and Show buttons.

The Stationery Buttons

These buttons are necessary to tie stationery definitions in with an application. They must be in the application which is to display your stationery components. Note that they are defined as part of the NewtApp framework and work best when they are included in a NewtApp application.

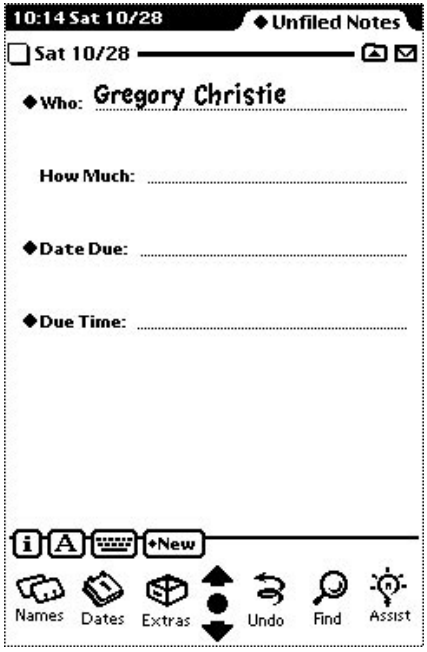
The New button offers new data formats generated from dataDefs. For example, the New button in the Calls application creates one new data entry form; if it contained more dataDefs there would be a New picker available. The New button of the built-in Notes application offers a picker whose choices create either a new Note, Checklist, or Outline format for entering notes. The example used in this chapter to illustrate using stationery, extends the built-in Notes application by adding the item, IOU, to the New menu, as shown in Figure 5-1.

Figure 5-1 The IOU extension in the New picker.



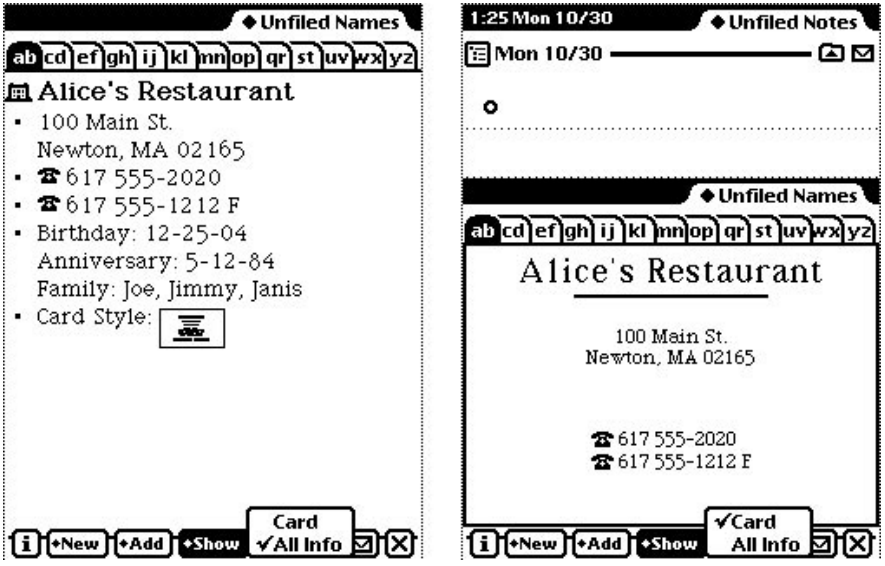
When you choose IOU from the New picker, an IOU entry displays, as shown in Figure 5-2.

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 from the viewDefs defined for your application. For example, the choices in the Show button of the built-in Names application includes a Card and All Info view of the data. These views appear as shown in Figure 5-3.

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 may exist outside of an application as purely NewtonScript code in an NTK auto part, must be registered with the system when an application is installed and removed. 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 just 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 of your application. (See “Stationery Reference” beginning on page 5-19.)

Once registered, the system determines the application to which the stationery belongs by checking the superSymbol slot in the dataDef and then finding an application, which has been registered for stationery, with a matching superSymbol slot. (See “Stationery Reference” beginning on page 5-19.)

Getting Information about Stationery

By using the appropriate global function, you may get information about all the dataDefs and viewDefs which have been registered and thus are part of the system registry. These functions include, GetDefs, GetDataDefs, GetAppDataDefs, GetViewDefs, and so on. For more information, see “Stationery Reference.”

You may also obtain application-specific stationery information. This should enable applications which are registered for stationery to be extended by third party developers.

Using Stationery

Stationery allows you to:

- Create discrete data definitions which may be accessed with a class slot you define.
- 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 programming situations in the future which 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 and thereby "hard-wiring" it. In addition, you should provide public interfaces to any values you want to share with the world 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 a discrete soup structures. (See "AdoptEntry" beginning on page 4-33 of Chapter 4, "NewtApp Applications.")

The dataDef component of your stationery should use a `FillNewEntry` method to define its own discrete soup entry structure. You may even choose to set a `class` slot within the entry, which you'll be able to use to access its parts. An example 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 and set to the same constant (`kDataSymbol`.) A `viewStationery` slot is then added and set to the same constant (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—which purely consists of NewtonScript code—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 dataDef, viewDef and Install & Remove files are used in examples. The resource file (`ExtendNotes.rsrc`) is the which contains the icon shown next to the dataDef 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 print format file is

Determining the SuperSymbol of the Host

Using stationery, requires the presence of a matching `superSymbol` slot in both the host application and in the dataDef component of your stationery. These value of the `superSymbol` slot, is a reference which names the soup in the host application to which will adopt the dataDef soup entry.

If you do not know the value of the `superSymbol` slot for an application which 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 the execution of 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>},
          ...]
```

This and other built-in global stationery functions are documented in the “Stationery Reference,” in this chapter.

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 which 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.

The following example uses the constant `kSuperSymbol` to define 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 picker of the New button.
- 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 which displays when the user taps the icon on the left side of the header, as shown in Figure 4-5 on page 4-12.
- Set `symbol` to `kDataSymbol`. Note that each item in the `allDataDefs` slot of the application base view matches the `symbol` slot of the corresponding dataDef file.
- Set `version` to `1`. This is an arbitrary version number and up to the discretion of the developer.
- Remove the `viewBounds` slot; it's not needed since this proto is not displayed.

There are a number of methods defined within the `newtStationery` proto which you should override for your data set.

Defining DataDef Methods

The three methods, `MakeNewEntry`, `StringExtract`, and `TextScript`, are illustrated with examples in this section. You use the method, `MakeNewEntry` to define the soup entries for your dataDef, while the method, `StringExtract`, is required by NewtApp overview scripts to return text for display in the overview, and `TextScript`, is called by routing (the Newton version of electronic mail) 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 (that 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 non-generic parts of the data definition, are kept in a nested frame which has the name of the data class symbol, `kDataSymbol`. By keeping the specific definitions of your data tucked away and accessible with the class of that data, you are assuring that you code will be reusable in other applications.

```
// 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,
};

func()
   begin
      local theNewEntry := DeepClone(kEntryTemplate);
      theNewEntry.title := ShortDate(time());
      theNewEntry.timeStamp := time();
      theNewEntry.(kDataSymbol).dueDate := time();
      theNewEntry;
   end
```

The `StringExtract` method is called by the Overview and expected to return a description of the data which is either one or two lines. Following is an example of a `StringExtract` implementation which returns a one line description.

```
StringExtract := {
      func(item,numLines)
      begin
         if numLines = 1 then
         return item.title
            else
            return item.title&&item.(kDataSymbol).who;
      end      }
```

The method, `TextScript`, is called by routing (the Newton system's form of email) 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. Following is an example.

```
TextScript:= {
   func(item,target)
   begin
      item.text := "IOU\n";
      item.text := item.text & target.(kDataSymbol).who
               && "owes me" &&
               NumberStr(target.(kDataSymbol).howMuch);
      item.text;
   end         }
```

Creating ViewDefs

ViewDefs may be based on any of 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`. For more information, see Chapter 3, "Views."

Printing formats are also implemented as viewDefs. You can learn more about `protoPrintFormat` in the *Newton Programmer's Guide: Communications*, in Chapter 2, "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` of an auto pat.

You may create the viewDef for the auto part which extends the Note 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.

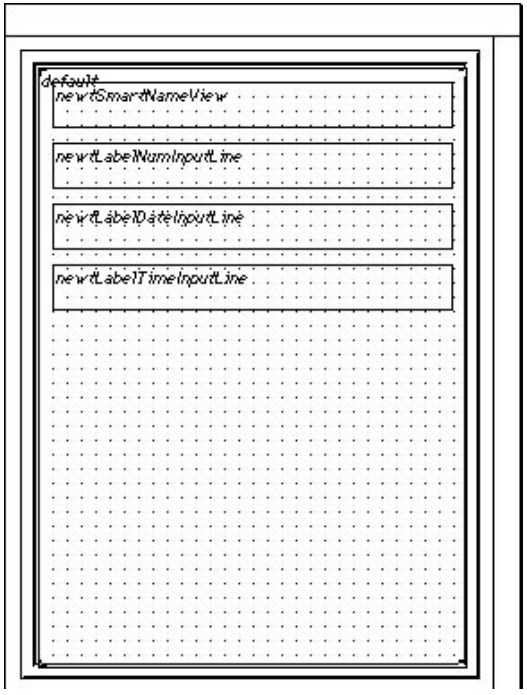
- Set the `name` slot to `"IOU Info"`. This appears in the Show button, if there is one.
- Set the `symbol` slot to `'default`. At least one of the viewDefs associated with a dataDef must have `'default` as the value of its `symbol` slot.
- Set the `type` slot to `'viewer`. The three system-defined types for viewDefs are `'editor`, `'viewer`, and `'printFormat`. 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 bottom.

Add the protos which will display the data and their labels in 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 which contains a name (or a place to store one.)
- Set the `usePopup` slot to `True`.

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 which 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 which 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 which 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 segment implements an `InstallScript` and `RemoveScript` which 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. Y

```
InstallScript := func(partFrame,RemoveFrame)
begin
   RegisterDataDef(GetLayout("iouDataDef"));
   RegisterViewDef(GetLayout("iouPrintFormat"),
                                          kDataSymbol);
   RegisterViewDef(GetLayout("iouDefaultViewDef"),
                                          kDataSymbol);
end;
```

```
RemoveScript := func(removeFrame)
begin
   UnRegisterViewDef('default,kDataSymbol);
   UnRegisterViewDef('iouPrintFormat,kDataSymbol);
   UnRegisterViewDef('targetFrameFormat,kDataSymbol);
end
```

Using the Minimal Bounds 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 and 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. Following is an example of a `MinimalBounds` implementation.

```
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 Reference

This section documents the proto for creating the dataDef stationery components. Note that there is no special purpose proto to be used for creating a viewDef. A viewDef is based on a general view template or class and has a specified data structure added. As such, it is documented in the "Data Structures" section.

Also contained in the "Stationery Reference" are:

- the proto used to create a dataDef (`newtStationery`)
- protos for stationery buttons
- global functions used to register and retrieve information about dataDefs and viewDefs.

Data Structures

You create a viewDef by basing it on a general view proto or class, such as a `clView`, and adding the slots specified here. Note that once the viewDef has been created it must be added to an application by using a `newtStationeryView` proto, as described on page 4-83, of Chapter 4, "NewtApp Applications."

Slot descriptions

`type`	Required. The view types, `'editor`, `'viewer`, and `'printFormat` are used by the system and the built-in applications to collect specific kinds of viewDefs. For instance, the Newton routing code collects viewDefs of type `'printFormat` and offers them as choices in the Format picker within the routing slip. You may also define custom types for your application.
`symbol`	Required. A symbol that names the view for the dataDef. One viewDef is required to have the `symbol`

	slot set to `'default`. This symbol is saved as a convenient reference by which to retrieve the view.
`name`	Required. A string that is used to build menus like the Show menu. An example of a suitable value is `"Note"`.
`version`	Required. This should match the version number of the dataDef.
`viewDefHeight`	Required; except in card-style applications. An integer which specifies a default height for applications which display data in a paper roll format. This value is not used by a card-style NewtApp application.

The following methods are for use with viewDefs.

MinimalBounds

myViewDef:`MinimalBounds(`*entry*`)`

Returns the minimal enclosing bounding box for the data in the given soup entry. In a viewDef you must use the `MinimalBounds` method if height of the entry is dynamic, as it is in a paper roll-style application. This method is not necessary for a card-style application which has a fixed height. If the entry size is static, use the `'viewDefHeight` slot.

entry	The specified soup entry.

SetupForm

targetViewDef:`SetupForm(`*entry*, *entryView*`)`

This function is called by the `ViewSetupFormScript` of the entry view containing the viewDef to be displayed. Override this method to massage the data before it's instantiated.

entry	The target soup entry.
entryView	The target view, in which the soup entry is about to be displayed.

Protos

This section contains the `newtStationery` proto, which is used to construct a dataDefs, and the stationery button protos.

Note

The viewDef stationery component is a data structure that is based on any basic container view. It is documented in the section "Data Structures." ◆

newtStationery

The proto you use as the base view when constructing a dataDef. Its basic function is to create the infrastructure for specified kinds of data. The `newtStationery` proto does not construct a visible runtime view.

Slot descriptions

`description` Optional. A string describing this dataDef's data entry. An example is: `"Lined note paper"`. This is used in the Information slip, (`newtInfoBox` proto) which is seen when the icon on the header bar is tapped.

`height` Required; except in card-style applications. This is default height used by viewers which display the data type in a paper roll format, like the built-in Notes application. This value should match the value in the `viewDefHeight` slot of the viewDef. It is not used by a card-style NewtApp application.

`icon` Optional. If an icon for this `dataDef` is provided, it is used in the New menu (the `newtNewStationeryButton` proto,) in the header / breaker bar (`newtEntryRollHeader,`) and in the Information slip (`newtInfoBox` proto,) which is seen when the icon on the header bar is tapped.

`name` Required. This string appears in the New button's picker. The New stationery button (the `newtNewStationeryButton` proto, described in this section) collects all the strings from the `name` slots of the

	registered dataDefs that have the same `superSymbol` slot value and displays them as items in the New picker. For example, the Notes application uses the string `"Note"` to identify one of its dataDefs.
`symbol`	Required. A unique symbol which identifies the data type of this application. The example in this chapter uses the variable `kDataSymbol`, which is set to the value of `kAppSymbol`. The symbol is used by a `newtStationery` view to select the viewDef and dataDef to use for a given entry.
`superSymbol`	Required. A unique symbol that identifies the set of soups in which the data definition provided in the `FillNewEntry` or `MakeNewEntry` slot of this dataDef is to be placed. Note that the value of this slot must match the value of a `superSymbol` slot in the host application, as well as, a soup name in the `newtApplication.allSoups` slot.
`version`	Required. This should match the version number of the viewDef.

Following are a few interesting methods that are defined in `newtStationery`.

FillNewEntry

myDataDef`:FillNewEntry(`*newEntry*`)`

This returns a modified soup entry. It takes a new entry, as returned by the `CreateBlankEntry` method in the `newtApplication.allSoups` slot, as its parameter.

You may use this method to add a `class` slot value and the other specific data structures you require. It is recommended that you put specific data structures in a slot, embedded within the entry. For an example of this, see

"Using FillNewEntry" beginning on page 5-7. See Chapter 4, "NewtApp Applications," for more information about the `CreateBlankEntry` method.

newEntry A frame that is a soup entry, as returned by the `CreateBlankEntry` method, which is defined in the `newtApplication.allSoups` slot of a NewtApp application.

MakeNewEntry

myDataDef`:MakeNewEntry()`

This returns a frame that will be added to some soup to make an entry. This method is used only if `FillNewEntry` does not exist. However, it is useful if you are creating stationery as an auto part instead of as part of a NewtApp application. Furthermore, if the NewtApp opening this dataDef has no `CreateBlankEntry` method this will be called.

StringExtract

myDataDef`:StringExtract(`*entry, nLines*`)`

This is called by overviews and find-like applications to get a string for display. You must create a version of this method that is tailored to extract a string from your soup entry.

entry A soup entry.

nLines Your method should return either 1 or 2 lines of text.

TextScript

myDataDef`:TextScript(`*fields, target*`)`

Write this method to extract text for use by routing (for example, in the IO box for email.)

fields The slots from your soup which you wish to coerce into text.

target The soup entry in which the data exists.

newtStationeryPopupButton

This button proto is used as the basis for both the `newtNewStationeryButton` and the `newtShowStationeryButton`, of which the former is for displaying a list of dataDefs, and the latter a list of viewDefs.

The `newtStationeryPopupButton` is based on the `protoPopupButton`, thus incorporating the necessary functionality for creating a pop-up menu for the stationery buttons. It also includes the `StatScript` method, which you must define to assign an action to a menu choice, and the `SetUpStatArray` method, which you may override to intercept or tweak the stationery items before they are displayed in the menu.

The methods, `BuildPopup`, and `ViewSetupFormScript`, are defined internally to `newtStationeryPopupButton`. If you need to use one of these methods, be sure to call the inherited method first (for example, `inherited:?viewClickScript()`), otherwise the proto may not work as expected.

WARNING

Do not override the internally defined methods: `buttonClickScript`, `pickActionScript`, and `pickCancelledScript`. ▲

Slot descriptions

`form`	Required. Can be either `'viewDef` or `'dataDef`. Determines which form of stationery is gathered.
`symbols`	Optional. Defaults to `nil`; which indicates that all unique symbols or `superSymbols`, for the specified stationery form, are to be gathered. If you don't want all the stationery, you can specify an array of unique symbols. A `superSymbol` slot is required for all

	dataDefs. See page 4-41 for the definition of the `superSymbol` slot.
`text`	Required. The text displayed in the `newtStationeryPopup` button. An example is: `"New"`
`types`	Required–when the stationery `form` slot is set to `'viewDef`. Must contain an array of type symbols. An example is: `['viewer, 'editor, 'symbolYouDefined]` This slot is ignored for the `'dataDef` form of stationery.
`sorter`	Optional. The default is the symbol `'|str<|` for sorting in alphabetical order. Set to `nil` to prevent sorting. This slot can be set to any of the string sort tests defined for the *test* parameter of the `Sort` global function. See Chapter 20, "Utility Functions." for the other sorter symbol options.
`shortCircuit`	Optional. Defaults to `true`. This slot controls the pop-up behavior of the button. When set to `true` and only one item is in the stationery popup array, the diamond, displayed to the left of the text in the button, disappears. Tapping on this button does not display a picker but instead causes the action to occur with the single item. Set this slot to `nil` if you prefer a picker list with one item.

The following methods are defined in this proto.

StatScript

newPopupButton`:StatScript(`*stationeryItem*`)`

This method is invoked when an item is chosen from a popup stationery menu button, It is called by the `pickActionScript` and should contain a defined action for the stationery item.

stationeryItem	The stationery which corresponds to the item chosen from the popup menu. It can be either a viewDef or a dataDef; depending on which is specified in the `form` slot.

SetUpStatArray

newPopupButton:`SetUpStatArray()`

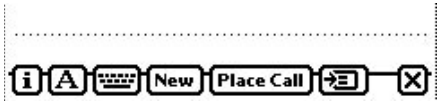
Override this method to change or intercept what will be displayed in the pop-up picker menu. This method must return the stationery array which is used in the menu item list, by calling the global method `GetDefs`, with the values you provide for the `form`, `symbols`, and `types` slots as its parameters. See “GetDefs” beginning on page 5-30.

newtNewStationeryButton

This button collects the dataDef stationery from your application and includes them as menu items in a popup menu whose button contains the text, New. If an icon exists for the given viewDef it is also displayed in the menu item list, next to the stationery name.

When there is only one dataDef in your application, the default behavior of this button is to display the text, New, without the diamond. If more than one dataDef exists for this application, the diamond that indicates a popup menu appears at the left of the New label. You can control this behavior by changing the `shortCircuit` slot (see page 5-25.) An example of this can be seen in the Calls application, where the New button is used to create a New entry and display a blank page. It is shown in Figure 5-5.

Figure 5-5 This New button, from the Calls application, creates a new entry.



When a menu item is chosen, the proto (through the `StatScript`) adds a new entry (defined by the dataDef) to the application soup and displays the blank entry. If you wish to add other actions to be executed when the user chooses a menu item, override the `StatScript` (inherited from the

newtStationeryPopupButton proto) and be sure to call the inherited method in your code.

The newtNewStationeryButton proto that appears in the built-in Names application is shown in Figure 5-6.

Figure 5-6 The newtNewStationeryButton proto.

This button protos from the newtStationeryPopupButton and thus inherits its methods and slots.

newtShowStationeryButton

The Show button collects the viewDef stationery from your application and displays it (if there is more than one and the shortCircuit slot is set appropriately) in a picker menu.The button displays the text, Show, and the pop-up button diamond. If an icon exists for the given viewDef it, too, is displayed in the menu, next to the stationery name.

You should use a Show button when you want to be able to extend your application with multiple views. For instance, you may wish to allow a choice between an informational view and an editable view, in which you can enter notes, as shown in Figure 5-7.

Figure 5-7 The newtShowStationeryButton proto.

When a menu item is chosen, the available viewDefs for the chosen item displays and a checkmark is placed next to the menu item to indicate which is the current viewDef. If you wish to add other actions to be executed when the user chooses a menu item you must redefine the `StatScript` (see page 5-25.) When you do this make sure to call the inherited method in your code.

Slot description

`types`	Optional. Defaults to: `['viewer, 'editor]`

newtRollShowStationeryButton

This has all the same slots and even the same appearance as the `newtShowStationeryButton`. However, it was built to function in a paper roll-style application. Again, if you wish to modify the `StatScript` method, (documented on page 5-25) make sure to call the inherited method.

newtEntryShowStationeryButton

This Show button also protos from the `newtShowStationeryButton;` this version is meant to be used within the entry view of an application. Like the `newtShowStationeryButton` it allows the user to change the viewDef being displayed. However, unlike that proto, this occurs for only the entry being displayed.

In a paper roll-style application it enables a different view for each entry. For instance, one entry might be a Note, while another might be an Information view.

Global Functions

The global functions, which are used to register stationery components and retrieve information about them, are documented in this section.

RegDataDef

`RegDataDef(`*symbol, newDefTemplate*`)`

Registers the given dataDef symbol with the system. It returns `true` if registration was successful or `nil` if the symbol was already registered.

If you build an application using the NewtApp framework protos, the base view proto, `newtApplication`, automatically registers any dataDefs you create, by using the values you put in its `allDataDefs` slot. (For more information see Chapter 4, "NewtApp Applications.")

symbol — The symbol which uniquely identifies the dataDef you wish to add to the system registry. The symbol is the value of the dataDef's `symbol` slot. An example of an appropriate value is '|IOU:PIEDTS|

newDefTemplate — The view template file defining the dataDef. This value may be obtained with a call like the one in the following line of code.

```
GetLayout("iouDataDef")
```

UnRegDataDef

`UnRegDataDef(`*symbol*`)`

Undoes the effect of `RegDataDef`. It should only be called by the system.

symbol — The symbol which uniquely identifies the dataDef you wish to remove from the system registry. The symbol is the value of the dataDef's `symbol` slot. An example of an appropriate value is '|IOU:PIEDTS|

RegisterViewDef

`RegisterViewDef(`*viewDef, dataDefSym*`)`

Registers the given viewDef view template and the unique identifying symbol of it's corresponding dataDef in the system registry. It returns `true` if it successfully registers or `nil` if it is already registered.

If you build an application using the NewtApp framework protos, the base view proto, `newtApplication`, automatically registers any viewDefs you create by using the values you put in its `allViewDefs` slot. (See Chapter 4, "NewtApp Applications." If you are building an auto part extension use a line of code like the following in its `InstallScript`:

```
RegisterViewDef(GetLayout("defaultViewDef"),kDataSymbol);
```

viewDef	The symbol identifying the viewDef.
dataDefSym	The symbol identifying the dataDef associated with this viewDef.

UnRegisterViewDef

`UnRegisterViewDef(`*viewDefSym, dataDefSym*`)`

Removes the viewDef from the system registry.

viewDefSym	The symbol identifying the viewDef.
dataDefSym	The symbol identifying the dataDef associated with this viewDef.

GetDefs

`GetDefs(`*form,symbols,types*`)`

Returns an array of stationery, depending on the value of the *form* parameter, of either viewDefs or dataDefs, that match the criteria you set up with the last two parameters.

form	Required. Can be either `'viewDef` or `'dataDef`. Determines which of the stationery forms is gathered into the array.
symbols	Required. Can be either `nil` or an array of `superSymbol` values. The value `nil` returns every registered `'dataDef` or `'viewDef`, whereas the array of superSymbols returns only the ones that match the array values. Note that the `superSymbol` is the symbol

of the soup to which a dataDef adds its soup. See the description of the `superSymbol` on page 5-22.

types Optional for dataDefs, but required when the stationery *form* parameter is set to `'viewDef`. Ignored when the stationery *form* parameter is set to `'dataDef`.

When the *form* parameter is set to `'viewDef` *types* can be `nil` or an array of types to match. The `nil` value returns all viewDefs. Symbols may be any of the built-in symbols (`'viewer`, `'editor`, or `'printFormat`) or they may include a symbol you define. An example of a *types* array follows:

```
['viewer, 'editor, 'symbolYouDefined]
```

GetDataDefs

`GetDataDefs(`*dataDefSym*`)`

Returns the corresponding dataDef, given the value of its symbol slot.

dataDefSym The value of the `symbol` slot of a dataDef.

The following example uses the symbol defined for the built-in Names application.

```
GetDataDefs('paperroll)

      {_proto: {symbol: NIL,
                superSymbol: NIL,
                name: "",
                description: "",
                icon: {@59},
                version: 0,
                height: 200,
                metadata: NIL,
                MakeNewEntry: <function, 0 arg(s) #4277B9,
                MinimalBounds: <function, 1 arg(s) #4277D9,
                SetupForm: <function, 2 arg(s) , #4277F9
```

```
        StringExtract: <function, 2 arg(s), #427819
        textScript: <function, 2 arg(s) #427839>},
    symbol: paperroll,
    name: "Note",
    superSymbol: notes,
    description: "Note",
    icon: {bits: <bits, length 76>,
           bounds: {#37B70D}},
    version: 1,
    metadata: NIL,
    MakeNewEntry: <function, 0 arg(s) #467251>,
    StringExtract: <function, 2 arg(s) #467271>,
    textScript: <function, 2 arg(s) #467291>}
```

GetAppDataDefs

`GetAppDataDefs(`*superSymbol*`)`

Returns an application's dataDefs when passed the value of the `superSymbol` slot of that application.

superSymbol The value of the `superSymbol` slot, as defined by the given application; for the built-in Notes application it is `'notes`.

The following example shows the dataDefs registered for the built-in Notes application.

```
GetAppDataDefs('notes)

    {list: {symbol: list,
            name: "Outline",
            description: "Outline",
            icon: {#73EF45},
            version: 1,
            height: 200,
            metaData: NIL,
```

```
            superSymbol: notes,
            mailBulletDone: "* ",
            mailBulletNotDone: "* ",
            MakeNewEntry: <function, 0 arg(s) #73ED1D>,
            SheetBounds: <function, 1 arg(s) #73ED45>,
            StringExtract: <function, 2 arg(s) #73ED85>,
            TextScript: <function, 2 arg(s) #73EF5D>,
            _proto: {@1807}},
      checkList: {_proto: {#750DFD},
            symbol: checkList,
            name: "Checklist",
            icon: {#73F0DD},
            description: "Checklist",
            superSymbol: notes,
            mailBulletDone: "+ ",
            mailBulletNotDone: "- ",
            MakeNewEntry: <function, 0 arg(s) #73F03D>},
      paperroll: {_proto: {@1807},
            symbol: paperroll,
            name: "Note",
            superSymbol: notes,
            description: "Note",
            icon: {@2871},
            version: 1,
            metadata: NIL,
            MakeNewEntry: <function, 0 arg(s) #467251>,
            StringExtract: <function, 2 arg(s) #467271>,
            textScript: <function, 2 arg(s) #467291>}}
```

GetEntryDataDef

`GetEntryDataDef(`*symbol, whichSoupEntry*`)`

Returns the dataDef for a given soup entry.

symbol	The symbol identifying the dataDef.
whichSoupEntry	The specified soup entry.

GetEntryDataView

`GetEntryDataView(`*entry, whichViewDef*`)`

Returns a specified viewDef for a given soup entry.

entry	The specified soup entry.
whichViewDef	The value of the `symbol` slot of the viewDef. All dataDefs are required to have at least a `'default` viewDef.

GetViewDefs

`GetViewDefs(`*dataDefSym*`)`

Returns the frame of viewDefs that are registered for the given dataDef symbol. If there are none it returns `nil`.

dataDefSym	A symbol referring to a dataDef which has been registered for an application.

GetDataView

`GetDataView`(*symbol, which*)

Returns a specific viewDef for a particular dataDef symbol. Use this if you know the symbols of the viewDefs and want to see the slots inherited by the viewDef from its proto. You may use a line like the following in the NTK Inspector window: `GetDataView('paperroll, 'default)`

symbol	The value of the `symbol` slot of the dataDef.
which	The value of the `symbol` slot of the viewDef. All dataDefs are required to have at least a `'default` viewDef.

Stationery Summary

Data Structures

ViewDef Structure

```
myViewDef := {
_proto: anyGenericView,
type: 'editor, //Could also be 'viewer, or 'printFormat
symbol: 'default, //One is required to be this
name: "", //Also required.
version: integer, //Required. Should match dataDef's version
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: "Note paper"  //Describes dataDef entries
height: integer,   //Required, except in card-style. Should
                   // match viewDefHeight.
icon: resource,   // Optional. Used in header & New menu
name:"", //Required. Appears in New button picker
```

```
symbol: kAppSymbol, //Required unique symbol.
superSymbol: aSoupSymbol, //Names superset of soups this
                          // dataDef's entry should join.
version: integer, //Required. Should match viewDef's version
FillNewEntry:     //Returns a modified entry
   func(newEntry)...,
MakeNewEntry:     //Used if FillNewEntry does not exist
   func()...,
StringExtract:
   func(entry, nLines)...,
TextScript        //Extracts text for email
   func (fields, target)...,
}
```

newtStationeryPopupButton

```
aStatPopup:= {   // Used to construct New and Show buttons
_proto: newtStationeryPopupButton,
form: 'statForm, //'viewDef or 'dataDef
symbols: nil,  //gathers all or specify:[uniqueSym,...]
text: "New",   //text displayed in picker
types: [typeSym,...],//Type slots of viewDefs
sorter: '|str<|,//Sorted alphabetically by Sort function
shortCircuit: true,//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
}
```

newtNewStationeryPopupButton

```
aNewButton:= {  // The New button collects dataDefs
_proto: newtNewStationeryPopupButton,
```

```
sorter: '|str<|,//Sorted alphabetically by Sort function
shortCircuit: true,//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
}
```

newtShowStationeryPopupButton

```
aShowButton:= {  // The Show button collects viewDefs
_proto: newtShowStationeryPopupButton,
types: [typeSym,...],//Can specify type slots of viewDefs
sorter: '|str<|,//Sorted alphabetically by Sort function
shortCircuit: true,//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
}
```

newtRollShowStationeryPopupButton

```
aRollShowButton:= { // The Show button in paper roll apps
_proto: newtRollShowStationeryPopupButton,
types: [typeSym,...],//Can specify type slots of viewDefs
sorter: '|str<|,//Sorted alphabetically by Sort function
shortCircuit: true,//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
}
```

newtRollShowStationeryPopupButton

```
anEntryShowButton:= { // The Show button in paper roll apps
_proto: newtEntryShowStationeryPopupButton,
types: [typeSym,...],//Can specify type slots of viewDefs
sorter: '|str<|,//Sorted alphabetically by Sort function
shortCircuit: true,//controls picker behavior
StatScript        //Called when picker item chosen
   func (stationeryItem)..., //Define actions in this method
SetUpStatArray//Override to change entry to be displayed
   func ()..., //Can display different view for each
}
```

Global Functions

```
RegDataDef(symbol, newDefTemplate) //Adds to System Registry
UnRegDataDef(dataDefSym) // Remove from System Registry
RegisterViewDef(viewDefSym, dataDefSym)// ViewDef Registry
                  //made to corresponding dataDef symbol
UnRegisterViewDef(viewDefSym, dataDefSym)// Removes ViewDef
                                          //from Registry
GetDefs(form,symbols,types)//Returns view or data defs array
GetDataDefs(dataDefSym)//Returns dataDef given its symbol
GetAppDataDefs(superSymbol)//Returns an app's dataDefs
                          //given it's superSymbol
GetEntryDataDef(symbol, whichSoupEntry) //Returns the entry's
                                          //dataDef
GetEntryDataView(entry, whichViewDef) //Returns the entry's
                                          //viewDef
GetViewDefs (dataDefSym)// Returns viewDefs registered
                          // with the dataDef symbol
GetDataView (dataDefSym, which)// Returns specific viewDef
                     //registered with the dataDef symbol
```

CHAPTER 6

Pickers, Pop-up Views, and Overviews

This chapter describes how to use pickers and pop-up views to present information and choices to the user. You should read this chapter if you are attempting to

- create your own pickers and pop-up views
- take advantage of built-in picker and pop-up protos
- present outlines and overviews of data

Before reading this chapter, you should be familiar with the information in Chapter 3, “Views.”

This chapter contains:

- a description of pickers and pop-up views
- pickers and pop-up views used to perform specific tasks
- picker and pop-up view reference information

About Pickers and Pop-up views

A **picker** or **pop-up view** is a view that pops up in response to a user tap or some other action and presents a list of items from which the user can make selections. With most picker protos, the view closes when the user taps an item in the list, or taps outside of the picker without making a selection. The more sophisticated picker protos allow multiple selections with a close box that dispatches the view.

Note

The distinction between a picker and a pop-up view is not particularly important, and has not been maintained in the naming of protos, so the terms are used somewhat interchangeably. In the discussion that follows, picker is used for both terms. ◆

A picker is a dynamic view, opened in response to a user action. The simplest picker protos handle all behaviors; you provide the items in the list. When the user taps a button or a label, the picker view opens automatically. When the user makes a selection, a message is sent to the picker view with the index of the chosen item in the list, and the view closes automatically.

With some picker protos, you must determine when and how the picker is displayed. For example, you might open a picker when the user taps a word, a button, or a “hot” spot in a picture, by sending the `Open` message to the picker view. You can also display a picker view by calling the `PopupMenu` function.

What you display in the picker view is up to you. Some protos present simple text lists, while others display two-dimensional grids with pictures and text.

The most sophisticated picker protos let you use your own soups as well as using built-in system soups. Much of the behavior of these protos is provided by data definitions that iterate through soup entries, display a list,

allow the user to see and modify the data, and even add new entries to the soup.

Pickers and Pop-up View Compatibility

The following new functionality has been added for the 2.0 release of Newton System Software.

New Pickers and Pop-up Views

Two general pickers have been added `protoPopupButton`, `protoPopInPlace`. See page 6-27–6-30 for details.

A new set of map pickers has been added. These protos display various maps from which a user can select a place and receive information about the selected location. The pickers include: `protoCountryPicker`, `protoProvincePicker`, `protoStatePicker`, and `protoWorldPicker`. See page 6-53–6-56 for details.

A set of text pickers have been added that display a pop-up view with a string of text a user can change by tapping on the string and entering a new string; for example, changing a date. These protos include: `protoTextPicker`, `protoDateTextPicker`, `protoDateDurationTextPicker`, `protoRepeatDateDurationTextPicker`, `protoDateNTimeTextPicker`, `protoTimeTextPicker`, `protoDurationTextPicker`, `protoTimeDeltaTimePicker`, `protoMapTextPicker`, `protoCountryTextPicker`, `protoUSstatesTextPicker`, `protoCitiesTextPicker`, and `protoLongLatTextPicker`. See page 6-58–6-80 for details.

Date, time, and location pop-up views have been added to let the user specify new information in graphical pop-up views; for example, changing the date on a calendar. These protos include: `protoDatePopup`, `protoDateNTimePopup`, `protoDateIntervalPopup`, `protoMulitDatePopup`, `protoYearPopup`, `protoTimePopup`, `protoAnalogTimePopup`, `protoTimeDeltaPopup`,

protoTimeIntervalPopup, and protoLocationPopup. See page 6-82–6-96 for details.

Two number pickers have been added to display pickers from which a user can select a number: protoNumberPicker and protoDigit. See page 6-97–6-98 for details.

A set of overview protos have been added to allow you to create overviews of data; some of the protos are designed to display data names from the Names soup. The protos include: protoOverview, protoSoupOverview, protoListPicker, protoNameRefDataDef, protoPeopleDataDef, protoPeoplePicker, and protoPeoplePopup. See page 6-104–6-127 for details.

Obsolete Functionality

The DoPopup global function, used in system software version 1.x is obsolete; however, is still supported for compatibility with older applications. You should use the PopupMenu function instead, as the DoPopup function will not be supported in future system software versions.

Using Pickers

This section describes how to use pickers to perform specific tasks. See "Reference" on page 6-27 for descriptions of the protos discussed in this section.

Handling Simple Picker Behavior

The simplest pickers handle all picker behaviors. You define the location and descriptive text for a button or label, as well as the items to be displayed when the user taps the button; the other behavior is handled automatically. These pickers are

- `protoPopupButton` (see page 6-27) is a text button, which, when tapped, pops up a picker. The text of the button is a descriptive word.
- `protoPopInPlace` (see page 6-30) is like `protoPopupButton` except that the text of the button is replaced by the text item that the user selects.
- `protoLabelPicker` (see page 6-32) provides a label, which, when tapped, pops up a picker. The current selected (or initial) item in the list is displayed next to the label.

Opening Pickers Dynamically

The following protos let you present pickers dynamically. The `protoPicker` proto, as well as the `PopupMenu` function, provide several different options for picker items. You specify these items in a `pickItem` array; see the following section "Specifying the PickItems Array" for details.

- `protoTextList` (see page 6-45) lets you present a scrollable list of items to the user. You specify an array, where each item in the array can consist of text, shapes, or a combination of the two.
- `protoTable` (see page 6-48) is similar to `protoTextList` in that it lets you present a scrollable list of items. With `protoTable`, however, each item is a separate view, while with `protoTextList` the entire list is maintained as a single view.
- `protoPicker` (see page 6-36) lets you dynamically present a picker that can be anything from a simple text list to a two-dimensional grid containing shapes and text.
- `protoGeneralPopup` (see page 6-42) is an all-purpose proto for creating a pop-up view.

Specifying the PickItems Array

The `pickItems` array contains the items that are to appear in the picker list. The list can include the following kinds of items:

simple string	A string. You can also provide additional options by specifying a frame. For details, see Table 6-1.

bitmap — A bitmap frame, as returned from the `GetPictAsBits` compile-time function. You can provide additional options by specifying a frame. For details, see Table 6-1.

icon with string — A frame, used to specify a string with an icon. For details, see Table 6-2.

separator line — To include a thin gray line, specify the symbol `'pickSeparator`. For a thicker black line, specify the symbol `'pickSolidSeparator`.

two-dimensional grid — A frame describing the grid item. This frame is described in Table 6-3.

Note that if all of the items in the picker list cannot fit into the view, the user can scroll the list to see more items.

Table 6-1 describes the frame used to specify simple string and bitmap items in the picker list.

Table 6-1 Item frame for strings and bitmaps

Slot Name	Description
`item`	The item string or bitmap reference.
`pickable`	Specify non-`nil` if you want the item to be pickable, or `nil` if you don't want the item pickable. Not-pickable items appear in the list but are not highlighted and can't be selected.
`mark`	Specify a dollar sign followed by the character you want to use to mark this item if it is chosen. For example, `$\uFC0B` specifies the check mark symbol in the Espy font. (You can use the constant `kCheckMarkChar` to specify the check mark character.)

Table 6-2 describes the frame used to specify a string with an icon in the picker list.

Table 6-2 Item frame for string with icon

Slot Name	Description
`item`	The item string.
`icon`	A bitmap frame, as returned from the `GetPictAsBits` compile-time function. The bitmap is displayed to the left of the text, and the text is drawn flush against it, unless the `indent` slot is specified.
`indent`	An integer that defines a text indent to be used for this item and subsequent icon/string items, until it is changed. This integer specifies the number of pixels to indent the text from the left side of the picker view, so that you can line up a number of text items that may have icons of varying width. Specify -1 to cancel the indent effect for the current and subsequent text items. Note that the icon is drawn centered within the indent width.

Table 6-3 describes the frame required to specify a two-dimensional grid item in the picker list.

Table 6-3 Item frame for two-dimensional grid

Slot Name	Description
`bits`	A binary object representing the bitmap, such as one from the `bits` slot in the frame returned by the `GetPictAsBits` compile-time function. The bitmap is a complete picture of the grid item, including the lines between cells and the border around the outside of the cells. There must be no extra white space outside the border. Each cell must be the same size, and must be symmetrical.
`bounds`	The bitmap bounds frame, from the `bounds` slot in the frame returned by `GetPictAsBits`.
`width`	The number of columns in the grid (must be nonzero).
`height`	The number of rows in the grid (must be nonzero).
`cellFrame`	Optional. The width of the separator line between cells, used for highlighting purposes. If you don't specify this slot, the default is 1 pixel.
`outerFrame`	Optional. The width of the border line around the cells, used for highlighting purposes. If you don't specify this slot, the default is 2 pixels.
`mask`	Optional. A binary object representing the bits for a bitmap mask. This mask is used to restrict highlighting, or for special hit-testing. The mask must be exactly the same size as the bitmap. Cells in the grid are highlighted only if the position tapped is "black" in the mask.

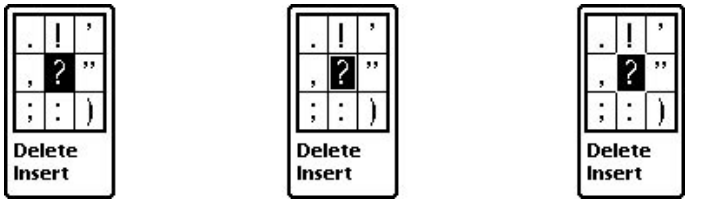
Note

Picker items can include 1.x bitmaps but not 2.0 shapes. ◆

When a cell is highlighted in a two-dimensional picker item, only the part of the cell inside the cell frame lines is inverted. You can vary the highlighting effect by changing the values of the `cellFrame` and `outerFrame` slots,

which control how much unhighlighted space to leave for the cell frame lines. An example of how these values affect cell highlighting is shown in Figure 6-1.

Figure 6-1 Cell highlighting example for protoPicker



`cellFrame = 1`
`outerFrame = 2`

`cellFrame = 3`
`outerFrame = 3`

`cellFrame = 0`
`outerFrame = 0`

Map Pickers

A variety of map picker protos are provided for letting the user specify cities, U.S. states, Canadian provinces, and countries. These pickers are:

- `protoCountryPicker` (See page 6-53) lets you display a picker of the world from which a user can select a country.
- `protoProvincePicker` (See page 6-54) lets you display a picker of Canada from which a user can select a Canadian province.
- `protoStatePicker` (See page 6-55) lets you display a picker of the United States from which a user can select a U.S. state.
- `protoWorldPicker` (See page 6-56) lets you display a picker of the world from which a user can select a continent.

Date, Time, and Location Pickers and Pop-up Views

A number of text picker protos let the user specify dates, times, durations, and geographical information. These protos all display a label picker with a string showing the currently selected (or initial) data. For example, `protoDurationTextPicker` (See page 6-70) might have a label of “When” followed by a time duration of the form “8:26 a.m. – 10:36 p.m.”

Tapping the picker causes a pop-up view to be displayed that allows the user to enter new information. The user dispatches the pop-up view by tapping a close box. The text picker protos are

- `protoTextPicker` (See page 6-58) displays a label picker with a text representation of an entry.
- `protoDateTextPicker` (See page 6-59) lets you display a picker with a text representation of the date.
- `protoDateDurationTextPicker` (See page 6-61) lets you display a picker with a text representation of a range of dates.
- `protoRepeatDateDurationTextPicker` (See page 6-64) lets you display a label picker with a text representation of a date range.
- `protoDateNTimeTextPicker` (See page 6-66) lets you display a picker with a text representation of a date and time.
- `protoTimeTextPicker` (See page 6-68) lets you display a picker with a text representation of a time.
- `protoDurationTextPicker` (See page 6-70) lets you display a picker with a text representation of a range of times.
- `protoTimeDeltaTextPicker` (See page 6-72) lets you display a picker with a text representation of a time delta.
- `protoMapTextPicker` (See page 6-74) lets you display a picker with a text representation of a country.
- `protoCountryTextPicker` (See page 6-76) same as “protoMapTextPicker”.
- `protoUSstatesTextPicker` (See page 6-76) lets you display a picker with a text representation of a US state.

- `protoCitiesTextPicker` (See page 6-78) lets you display a picker with a text representation of a city.
- `protoLongLatTextPicker` (See page 6-80) lets you display a picker with a text representation of longitude and latitude values.

The pop-up views the text pickers use are also available for use on their own, but you'll need to display them by sending a `New` message to the pop-up proto. (Basically, the text picker protos simply provide you an easy way to display the pop-up view by linking it to a label picker.) These pop-up views are described in the section "Date, Time, and Location Pop-up Views" beginning on page 6-82.

Creating Text Lists

The `protoTextList` (See page 6-45) is similar to the `protoTable` (See page 6-48) in that both can be used to present a scrolling list of text items. However, `protoTextList` uses only a single view to hold all of the text items, while `protoTable` uses separate views for each item. Each proto has its advantages and disadvantages. The `protoTextList` may be faster because it uses just a single view, but the `protoTable` may be more versatile since each item is a separate view.

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

```
theList := {...
_proto: protoTextList,
viewLines: 6,
listItems: nil,
ViewSetupFormScript: func()
   begin
   // list of soups in internal store
   listItems := getstores()[0]:getsoupnames();
   :SetupList(); // Always must call this internal method
   end,
viewfont: simpleFont10,
ButtonClickScript: func(textIndex)
   begin
   print("Selected the following index =" &&textIndex);
```

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

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

```
theTable := {...
_proto: protoTable,
def: protoTableDef,
ViewSetupFormScript: func()
   begin
   def := Clone (def);
   def.tabValues := ["foo", "bar", "baz", "qux", "4",
                     "5", "6", "7", "8", "9"];
   // tabWidths must be =< the view width-2
   def.tabWidths := self; LocalBox() .right -2;
   def.tabDown := 10;
   end,
selectThisCell: func(viewTapped)
   begin
   // first you MUST call the inherited method
   inherited:selectThisCell(viewTapped);
   // here you can do your own things
   ...
   end,
// Use this method to fix scroll-down problem
ViewScrollDownScript: func()
   begin
   local biggestTop;
   local temp := (self; LocalBox() bottom) /
                  def.tabHeights);
   temp := if (temp - Floor(temp)) < 0.5 then Floor(temp)
               else Floor(temp) + 1 ;
   biggestTop := Max(0, def.tabDown - temp);
```

```
   if vOrg < biggestTop then
      begin
      vOrg := vOrg + scrollAmount;
      if vOrg > biggestTop then
         vOrg := biggestTop ;
      tabbase:RedoChildren();
      :updateSelection();
      end;
   end,
...};
```

`protoTableDef` defines the format of the table. You use it by setting the `protoTable` slot `def` to `protoTableDef`. You change individual items in the `ViewSetupFormScript` method. Here's an example of `protoTableDef`:

```
protoTableDef := {
tabAcross: 1,
tabDown: 0,
tabWidths: 50,
tabHeights: 0,
tabProtos: protoTableEntry,
tabValues: nil,
tabValueSlot: 'text,
tabSetup: func(childView, hIndex, vIndex)
   begin
   childView.hIndex := hIndex - 1;// Save for selection
   childView.vIndex := vIndex - 1;
   end,
tabUniqueSelection: true, //use false for mult. selection
indentX: 2,
}
```

Creating Overviews

These protos create overviews of data:

- `protoOverview` (page 6-104) provides a framework for presenting an overview of data in an application. It handles pen tracking, highlighting, and hit-testing. You can you use `protoErsatzCursor` to work with non-soup data.
- `protoSoupOverview` (page 6-107) is based on `protoOverview`, and operates on soups using a provided cursor. This proto also adds checkboxes.
- `protoListPicker` (page 6-110) adds the concept of columns and has a built-in (yet suppressible) close box, alphabetical tabs, scrollers, folder tabs. Any column can have a built-in pop-up view or tap action.
- `protoPeoplePicker` (page 6-126) and `protoPeoplePopup` (page 6-127) are two protos based on `protoListPicker` that are specifically designed to display overviews of data from the "Names" soup.

Using protoOverview

Here is some sample code showing how to use `protoOverview`:

```
constant kNumItems := 20;
overtest := {
   _proto: protoOverview,
   viewBounds: RelBounds(20,20,200,150),
   viewFlags: vApplication+vClickable,
   saveIndex: 0,
   hiliteIndent: 20,
   items: nil,

   ViewSetupChildrenScript: func()
      begin
      local index := saveIndex;
      local array;
      if items then
```

```
      array := items
   else
      begin
      array := Array(kNumItems, nil);
      foreach i,x in array do
         array[i] := GetRandomDictionaryWord(5,15);
      self.items := Sort(array, '|str<|, nil);
      self.hilited := SetLength(Clone(""),kNumItems);
      end;
   local curs := {
      Entry: func()
         begin
         if index < Length(array) then
            index;
         end,
      Next: func()
         if index < Length(array)-1 then
            begin
            index := index + 1;
            end,
      };
   :SetupAbstracts(curs);
   end,

Abstract: func(item, bbox)
   begin
   local result := [MakeText(items[item],
      bbox.left+hiliteIndent, bbox.top, bbox.right,
      bbox.bottom-8),
      MakeRect(bbox.left,bbox.top,bbox.right,bbox.bottom)];
   if ExtractByte(hilited,item) <> 0 then
      AddArraySlot( result, MakeLine(bbox.left,
         bbox.bottom, bbox.left+hiliteIndent, bbox.top) );
```

```
      result;
      end,

   HitItem: func(i,x,y)
      begin
      i := i + saveIndex;
      print("hit item" && i & ":" && items[i]);
      :Dirty();
      end,

   Scroller: func(dir)
      begin
      saveIndex := Min(Max(saveIndex + dir, 0), kNumItems-1);
      :RedoChildren();
      :Dirty();
      end,
};
```

Using protoSoupOverview

Here is some sample code showing how to use `protoSoupOverview`. It is a simple implementation of an overview of the “Names” soup.

```
overtest := {
   _proto: output.protoSoupOverview,
   viewBounds: RelBounds(10,25,220,195),
   appSymbol: 'OverTest,
   lineHeight: 15,
   selectIndent: 20,
   items: nil,
   ViewSetupChildrenScript: func()
      begin
      if not cursor then
```

```
      cursor := Query(GetUnionSoup("Names"), '{type: index,
         indexPath: sortOn});
   self.prevLtr := nil;
   inherited:ViewSetupChildrenScript();
   end,

Abstract: func(entry, bbox)
   begin
   result := [
      begin
         local txt := if entry.name then entry.name.first &&
            entry.name.last else Clone(entry.Company);
         MakeText(txt,30,bbox.top,150,bbox.bottom-3);
      end,
      begin
         txt := Clone(if entry.phones and length(entry.phones)>0
            then entry.phones[0] else "");
         MakeText(txt,150,bbox.top,bbox.right,bbox.bottom-3);
      end,
   ];
   txt := begin local ltr := Upcase(entry.sortOn[0]);
      local sh := if ltr <> prevLtr then ltr&"" else nil;
      prevLtr := ltr; sh; end;
   if txt then AddArraySlot(result, MakeText(txt,
      selectIndent,bbox.top,30,bbox.bottom-3));
   if :IsSelected(entry) then
      AddArraySlot(result, ScaleShape( MakeShape(@311), nil,
         RelBounds(bbox.left+2,bbox.top+2,10,10) ));
   result;
   end,
};
```

Using protoListPicker

The `protoListPicker` (See page 6-110) proto provides a number of controls for finding specific entries, including folder tabs, azTabs, and scrolling arrows; any of these controls can be suppressed if desired.

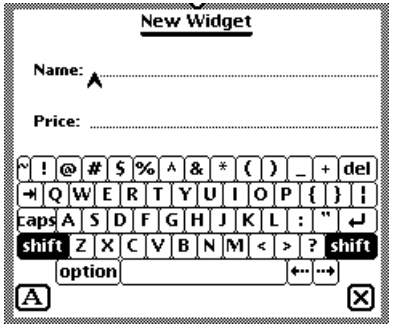
This proto manages an array of selected items (as does `protoOverview`); however, the items in the array are not simply aliases but frames containing the alias and various other data. Any soup that can be queried by a cursor can be displayed, or elements from an array can be displayed.

Figure 6-2 shows a full-featured example of `protoListPicker` that displays a two-column list: the first column is used to select or deselect members, and the second column provides additional information that can be edited in place.

Figure 6-2 **`ProtoListPicker`** example

The checkbox at the bottom-left of the slip is used to either show every eligible item or to trim all unselected elements from the list. The “New” button at the bottom allows the immediate creation of another entry to be displayed. See Figure 6-3:

Figure 6-3 Creating a new name entry



When the pen comes down in any column, the row / column cell inverts as shown in Figure 6-4:

Figure 6-4 Highlighted row

When the pen is released, if it is within the first column, the item is either checked to show that it is selected or unchecked to show that it is not. See Figure 6-5:

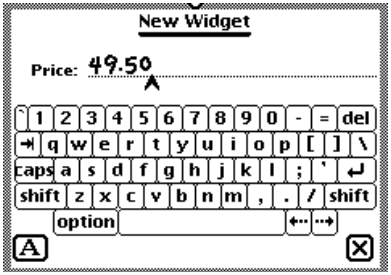
Figure 6-5 Selected row

When the pen tapped is released within the second column, what happens next depends on the underlying data. If there are many options already available, a pop-up view is displayed to allow the user to select any option, or enter a new option. See Figure 6-6:

Figure 6-6 Pop-up view displayed over list

If the user selects “Add new price” (or if there were one or no options already available to them), the user can enter a new price as shown in Figure 6-7:

Figure 6-7 Slip displayed for gathering input



The proto is driven by a frame contained in the `pickerDef` slot. This frame may or may not come from the data definition registry; however, the functionality it provides is similar to that of any data definition, in that it provides all the hooks the proto needs to interpret and display the data without the proto itself knowing what the data is.

The chosen items are collected into an array (described on page 6-23), which then can be stored separately from the original entries. Each selection is represented in the array by a name reference that contains all information needed to display or operate on the entries. The name reference is stored as part of the selection, along with an alias to the original entry, if there is an original entry.

The data definition (described in the next section) is responsible for providing the routines to create a name reference from an entry, an alias, from another name reference, from a straight frame, or just to create a canonical empty name reference (if no data is provided). It also retrieves the data from a name reference. Finally, it provides some information about the name reference to support actions like tapping and highlighting.

You also need to define the soup to query. Both this and the query specification can be defined either in the data definition or in the list picker.

Data Definitions

The data definition “Widgets” for widget names is used as an example.

```
pickerDef:= {
   _proto:     protoNameRefDataDef,

   name:       "Widgets",
   class:      '|nameRef.widget|,
   entryType:  'widget,
   soupToQuery:"Widgets",
   querySpec:  {indexPath: 'name},
   columns:    kColumns,
   validationFrame:kValidationFrame,
   editTitle:  "New Widget"
};
```

Specifying Columns

The list is defined by an array of column specifications; each specification is a frame with the following slots:

`tapWidth` Required. Specifies the width of the column. If this value is zero or negative, it’s interpreted as a distance from the right margin of the view; if positive, it’s considered a true width.

`fieldPath` Required. A symbol uniquely identifying the field that should be displayed in this column. This list picker uses this symbol to retrieve the data, and (in most cases including the default case) is the actual path in the entry to the data field desired. However, it is possible to use the symbol purely as a marker— for example if the particular data required is a calculated aggregate of a number of data fields— as long as all the routines in the data definition that use this symbol are overridden to recognize this usage.

`optional`	Optional. This slot tells the list picker that the contents of this field must be non-`nil` before the item may be selected. If `optional` is set and the data specified by the `fieldPath` is `nil`, when an attempt to select the item is detected, the user will be given the opportunity to fill in this field.

Here's an example that illustrates how to specify columns:

```
columns:= [{
   fieldPath:'name,// path for field to display in column
   optional:true,// not required -- unnamed widget

   tapWidth:155},// width for checkbox & name combined

{
   fieldPath:'price,// path for field to display
                     in column
   optional:nil,// price is required

   tapWidth:0}];// width -- to right end of view
```

Name References

A name reference is a wrapper for a soup entry. A name reference has the following structure:

```
local aNameRef := {
   class: dataClass, usually just 'nameRef
   _alias: <entryAlias> or nil,
   _entryClass:  class of the underlying entry or nil,
   _unselected: true or nil,
   _fakeID: integer or nil,
   <application defined slots>,
};
```

A name reference has the following slots:

`class`	A symbol specifying a registered data definition that can interpret the name reference.
`_alias`	A reference to the underlying entry, if any, from which this name reference was created. This is usually just `'nameRef` or a subclass of a name reference.
`_entryClass`	A symbol identifying the class of the underlying entry. This information is useful when you want to know what the entry is without having to read the entire entry in from the soup.
`_unselected`	A Boolean that determines whether an item is displayed as selected (in other words, checked) or not. By default, adding a name reference to the array of selections causes the name reference to be displayed as selected. However, if this slot is present and non-`nil`, the name reference is displayed with its checkbox unchecked. This slot is useful if items are displayed for which no entry exists in the soup, and which should not be selected. An example of how this is used in the system is for the potential locations for a meeting, to which an item is added to the list for each person who is going to attend the meeting, but only one site can be chosen (`singleSelect` is true) —that is, "My Place," "Sue Anderson's, and so on. See the description of the `selected` slot on page 6-127 for more details.

Name references also have several global functions:

- The `IsNameRef` function determines whether a given item is a name reference.
- The `AliasFromObj` function returns an alias to an entry.
- The `EntryFromObj` function returns the entry.
- The `ObjEntryClass` function returns the class of an entry (returned by the `EntryFromObj` function).

All of these functions can be passed an alias, an entry, or a name reference. (If you pass any other type of object, the result will be `nil`.)

To make a name reference, you can use the `MakeNameRef` method; as shown in the following example:

```
pickerDef.MakeNameRef := func(item, dataClass) begin

   local nameRef := :MakeCanonicalNameRef(item,
dataClass);

   if IsFrame(item) AND IsArray(nameRef.otherPrices) then
      Sort(nameRef.otherPrices, '|<|, nil);

   nameRef;
end;
```

Handling Taps

When the user taps in the picker, the data definition method `HitItem(`*tapInfo, context*`)` is called. The *tapInfo* parameter is a frame with the following slots:

`nameRef`	The name reference that was tapped.
`tapIndex`	The visible index of the name reference, or `nil` for `'new`.
`bbox`	The bounding box for the cell that was tapped.
`fieldPath`	The field path for the column tapped, or `'new` if the "New" button was tapped.
`editPaths`	All columns for this list.
`_reqPaths`	Used internally; do not modify.
`popup`	Used in pop-up view processing.

Note

The `context` parameter of the `HitItem` method is not the list picker view, but a child of it. This view is for callback purposes only. To get the list picker view, call `GetVariable(`*context, pickBase*`)`. ◆

`HitItem` should return either a reference to a view opened as a result of the tap, or `nil`. If a view is opened, all tap processing by the list picker is

suspended until the data definition passes control back to the picker by calling `context:Tapped(`*action*`)`. The *action* parameter can contain `'select` which will cause the item to be selected, `'toggle` which will toggle between selected and unselected items or `nil` which causes nothing to be done.

If the `validationFrame` slot is provided in the data definition (containing a frame appropriate for passing to the `ValidateConfig` method), a label input line slip will be opened for any item tapped (similar to that created by a call to `ValidateConfig`).

You can proto from `protoNameRefValidationFrame` to get the validation behavior for names (including company and work site), phone numbers, and email addresses.

When the "New" button is tapped, the data definition method `New(`*tapInfo, context*`)` is called. If the `validationFrame` slot is provided, the default `New` method will open a label input line slip, allowing editing of a new entry with one line for each column in the picker. New entries are handled by the data definition `New` method. The `DefaultOpenEditor` method is the bottleneck for the default behavior.

A tap may be for editing an existing entry, or for creating a new one. In both cases, by default the opening of an edit view is accomplished by calling the `DefaultOpenEditor` method. You may call this method as desired, or you may provide an `OpenEditor` method, which if present, will be used instead.

The `DefaultOpenEditor` method calls either `DefaultEditDone` or `DefaultNewDone` when the edit view is closed.

If a soup is available, the default tap handling routines will call either the `NewEntry` or `ModifyEntry` methods. You may override these methods. By default, these methods strip off the name reference-specific slots, add the item to the soup using the `AddEntryXmit(`*strippedNameRef*`)` method, and then "rewire" the name reference so that it maintains an alias to the entry.

For your convenience, `protoNameRefDataDef` provides the method `PrepareToAdd(`*nameRef*`)` which returns a stripped copy of the name reference, and the `FixupPostAdd(`*nameRef, entry*`)` method, which rewires the name reference appropriately.

Also, the `context` parameter to the various tap methods has a method `GetStoreSoup` which returns the store-based soup, if any is in use.

Validation of Entries

During an attempt to select an item, the list picker calls the data definition method `Validate(`*target*, *pathArray*`)`. This method should determine if the *target* contains information for each path specified in the *pathArray* that would allow this item to be selected. It returns an array of every invalid path in the path array; that is, a valid target would result in an empty array, and a completely empty target would return a clone of the *pathArray*. By default, this routine merely checks that each path specified contains non-`nil` data; however, if a `validationFrame` slot is present, the system global function `ValidityCheck` is called.

Reference

General Pickers

This section describes general purpose pickers and pop-up views.

protoPopupButton

This proto is a text button displays a picker when it is tapped. The button is highlighted and the picker appears to the right of it. Unlike `protoPopInPlace`, the text of the button does not change when an item is chosen from the picker.

Here is an example:

Figure 6-8 Pop-up button and picker

Button

After button is tapped, it is highlighted and picker is shown to the right of it

Note that the `ViewClickScript` method is used internally in the `protoPopupButton` and should not be overridden.

The `protoPopupButton` uses the `protoTextButton` as its proto; and `protoTextButton` is based on a view of the `clTextView` class.

Slot descriptions

`viewFlags`	The default is `vVisible + vReadOnly + vClickable`.
`viewBounds`	Set to the size and location where you want the button to appear. If you do not set this slot, the button appears seven pixels to the right of its sibling. It is designed to be placed next to another button, for example in the status bar.
`viewJustify`	Optional. The default setting is `vjSiblingRightH + vjCenterH + vjCenterV + oneLineOnly`.
`text`	A string that is the text inside the button.
`popup`	An array of items to be displayed in the picker list. Usually this is an array of strings to appear in the list. To include a separator line between two items, specify the symbol `'pickseparator` where you want to place the line.

To provide additional options for an item in the list, specify the item as a frame containing the following slots:

`item`	The item string.
`pickable`	Specify non-`nil` if you want the item to be pickable, or `nil` for not pickable. Non-pickable items appear in the list but are not highlighted and can't be selected.
`mark`	Specify a dollar sign followed by the character you want to use to mark this item. For example, `$\uFC0B` specifies the check mark symbol in the Espy font. (You can use the constant `kCheckMarkChar` to specify the check mark character.)

ButtonClickScript

picker`:ButtonClickScript()`

This method is called when the button is tapped. You can use this method if you want to construct the `pop-up` array dynamically. After setting the value of the `pop-up` slot, call the inherited `buttonClickScript` to preserve the pop-up behavior of the view. For example:
`inherited:buttonClickScript()`.

PickActionScript

picker`:PickActionScript(`*index*`)`

This method is called when an item is selected from the picker list.

index	The index of the item that was chosen from the `pop-up` array.

If you don't supply this method, the button is simply unhighlighted. If you do supply this method, call the inherited method to unhighlight the button. For example: `inherited:PickActionScript(`*arg*`)`.

If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

PickCancelledScript

picker:`PickCancelledScript()`

This method is called if the picker is cancelled by a tap outside of it. If you don't supply this method, the button is simply unhighlighted. If you do supply this method, also call the inherited method to unhighlight the button. For example: `inherited:PickCancelledScript()`.

protoPopInPlace

This proto is a text button that displays a picker when it is tapped. When an item is chosen from the picker, the text of the chosen item appears in the button.

Figure 6-9 Example of a text button

Button

After button is tapped, picker is shown on top of it

After item is chosen from picker, it is shown in button

Note that the `ViewSetupFormScript` is called multiple times; use the `ViewSetupDoneScript` to provide the initial text.

Also note that the `ViewClickScript` and `ButtonClickScript` methods are used internally; if you need to use one of these methods, be sure to also call the inherited method.

The `protoPopInPlace` uses the `protoTextButton` as its proto; and `protoTextButton` is based on a view of the `clTextView` class.

Slot descriptions

`viewBounds` Set to the size and location where you want the button to appear. Note that the right bounds value is set automatically, based on the length of the text.

`viewFlags` The default is `vVisible + vReadOnly + vClickable`.

`viewJustify` Optional. The default setting is `vjCenterH + vjCenterV + noLineLimits`.

`text` A string that is the text inside the button. This string must not begin with a space. Note that this string will be modified.

`popup` An array of items to be displayed in the picker list. Usually this is an array of strings to appear in the list. To include a separator line between two items, specify the symbol `'pickseparator` where you want the line to separate.

To provide additional options for an item in the list, specify the item as a frame containing the following slots:

- `item` The item string.
- `pickable` Specify non-`nil` if you want the item to be pickable, or `nil` for not pickable. Non-pickable items appear in the list but are not highlighted and can't be selected.
- `mark` Specify a dollar sign followed by the character you want to use to mark this item. For example, `$\uFC0B` specifies the check mark symbol in the Espy font. (You can use the constant `kCheckMarkChar` to specify the check mark character.)

PickActionScript

picker`:PickActionScript(`*index*`)`

This method is called when an item is selected from the picker list.

index The index of the item that was chosen from the `pop-up` array.

If you don't supply this method, the button is simply unhighlighted. If you do supply this method, call the inherited method to unhighlight the button. For example: `inherited:PickActionScript(`*arg*`)`.

If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

PickCancelledScript

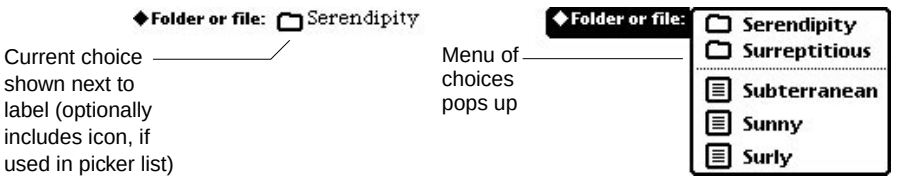
picker`:PickCancelledScript()`

This method is called if the picker is cancelled by a tap outside of it. If you don't supply this method, the button is simply unhighlighted. If you do supply this method, also call the inherited method to unhighlight the button. For example: `inherited:PickCancelledScript()`.

protoLabelPicker

This proto is a label that displays a picker when it is tapped. The picker list can consist of simple strings, or icons with strings. If the items are simple strings, the currently selected item is shown with a check mark next to it. The user can select a different item from the picker and that choice is then shown next to the label. Here is an example:

Figure 6-10 Picker displayed when tapped



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

Note that inking is automatically turned off when the view based on this proto is tapped.

The `protoLabelPicker` uses `protoStaticText` as its proto; and `protoStaticText` is based on a view of the `clParagraphView` class. The `protoLabelPicker` itself implements the label portion of the proto. It has one child view, also a `protoStaticText` view, that implements the text value portion of the proto. This child view is named `entryLine`.

Slot descriptions

`viewBounds` Set to the size and location where you want the label with its item to appear. Note that if you use horizontal sibling-relative justification, you normally would specify relative values for the left and right bounds. For this proto, however, you must specify left and right bounds values whose difference equals the actual view width. The `bounds` values are used to calculate the width of the view that holds the text item.

`text` A string that is the text label. The label is drawn with a diamond to its left, to indicate to the user that this is a picker.

`labelCommands` An array of items that are the choices to be displayed in the picker. You can specify an array of strings, or you can specify an array of frames if you want the list items to appear as icons with strings. In the latter case, each frame represents one list item, and should be specified as indicated in Table 6-2 on page 6-7.

To include a thin gray separator line, specify the symbol `'pickSeparator`. For a thicker black line, specify the symbol `'pickSolidSeparator`.

`iconBounds` Optional. Provide this bounds frame if you want the icon to appear next to the chosen item when the picker is not popped up. (As in Figure 6-10.) Specify the bounds of the largest icon in the list.

`iconIndent` Optional. The distance between the icon and the text when an icon / string item is shown next to the label. If you don't specify this slot, the default is 3 pixels.

`checkCurrentItem` Optional. If non-`nil`, the currently selected item in the list, if there is one, is marked with a check mark to its left. If `nil`, check marks are not shown. Note that check marks are not shown for list items that are icons with strings.

`indent` Optional. The distance, in pixels, to indent the picker from the beginning of the line (the beginning of the text label). If you don't include this slot, the picker is placed 6 pixels to the right of the text label by default.

`textIndent` The distance from the left side of the view to the text in the picker list. This is set in the `ViewSetupChildrenScript` method of the proto.

`viewFont` Optional. The font for the text label. The default is `ROM_fontSystem9Bold`.

`entryLine.viewFont` Optional. This is the `viewFont` slot in the `entryLine` child of the `protoLabelPicker`. It sets the font for the text field to the right of the label. The default font is `userFont10`.

LabelActionScript

picker`:LabelActionScript(`*index*`)`

When the user chooses an item from the picker, the new item is displayed next to the label and this method is called to allow additional processing.

index The index of the item that was chosen from the `labelCommands` array.

TextSetup

picker`:TextSetup()`

This method is called to get the initial choice that should be shown next to the label when the view is being created. This method is passed no parameters and should return a string. It you don't include this method, the first item from the `labelCommands` array is used as the initial item.

TextChanged

picker`:TextChanged()`

This method is called whenever the value of the item is changed. If you don't supply this method, no default action occurs.

UpdateText

picker`:UpdateText(`*newItem*`)`

You can call this method to programmatically change the value of the text item. Note that you don't normally need to call this method; the text item is updated automatically when the user makes a selection from the picker.

newItem A string that is the new value for the text item.

PickerSetup

picker`:PickerSetup()`

This method is called when the user taps the label; it gives you a chance to do your own processing, including setting up the `labelCommands` array.

This method should return non-`nil` if you want the default action to occur; that is, for the picker to pop up. If you return `nil`, the picker is not popped up. You must use this method or something else on your own. If you omit this method, non-`nil` is returned and the default action occurs.

Popit

picker`:Popit(`*position*`)`

You can send this message to programmatically pop up the picker.

position The horizontal position of the picker; you should pass (`indent`–2) for this parameter.

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

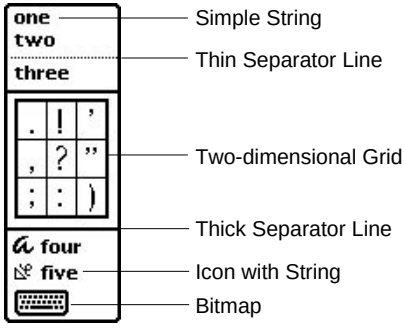
```
myPicker := {...
_proto: protoLabelPicker,
viewBounds: RelBounds(10, 60, screenWidth-100, 16),
text: "Folder or file:",
labelCommands: [
   {item:"Serendipity", icon:folder, indent:25},
   {item:"Surreptitious", icon:folder},
   'pickSeparator,
   {item:"Subterranean", icon:doc},
   {item:"Sunny", icon:doc},
   {item:"Surly", icon:doc} ],

textSetup: func()
   lastChoice, // retrieve the last choice

LabelActionScript: func(index)
   lastchoice:=labelCommands[index].item,//store choice
...}
```

protoPicker

This proto is used to create a picker in which you define the action that causes the picker to be displayed, sending an `Open` message in response to that action.

The picker is a list of items (simple strings, bitmaps, two-dimensional grids, icons with strings, and separator lines) from which the user can choose one item by tapping on it. Here is an example:

Figure 6-11 Selection of items to choose



The `ViewSetupDoneScript` method is defined internally. If you need to use this method, be sure to call the inherited method also (for example, `inherited:?ViewSetupDoneScript()`), otherwise the proto may not work as expected.

The `protoPicker` is based on a view of the `clPickView` class.

Note that there can be only one open picker view at a time in the system. If one is open and the user opens another, the first one closes.

Slot descriptions

`bounds` Must contain a `viewBounds`-like frame specifying a rectangle. The picker view will be created so that one of its corners corresponds to one of the corners of the rectangle you specify. However, the system figures out exactly where to position the view, depending on how large it is and how much space is available around it. For example, it would normally be positioned so that its top-left corner corresponds to the top-left corner of the rectangle you specify. However, if you specify a location in the lower-right corner of the screen, there won't be enough room for the picker; instead it will be positioned

	with its lower-right corner corresponding to the lower-right corner of the rectangle you specify. Generally, a picker view is shown as a result of a tap on a button, word, or some other visible element. In most cases, simply specify the `viewBounds` slot of that element as the value of the `bounds` slot.
`viewBounds`	This slot is ignored. Any value you place here will be overwritten by the system, which calculates the value of this slot when the view is opened. The `bounds` slot is used to control the position of the view. The size of the view is determined by the width of the widest item and the total height of all items.
`viewFlags`	The default is `vFloating + vReadOnly + vClickable`.
`viewFormat`	Optional. The default setting is `vfFrameBlack + vfPen(2) + vfRound(4)`.
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV`.
`viewFont`	Optional. The default font for text items in the list is `ROM_fontSystem10Bold`.
`viewEffect`	Optional. The default view effect is `fxPopDownEffect`.
`pickItems`	An array of items to be displayed in the picker list. There are many options, so this topic is described in the section "Specifying the PickItems Array" beginning on page 6-5.
`pickTextItemHeight`	Optional. The height in pixels that should be reserved for each text item in the picker list. Note that each text item may actually occupy a height that is less than this amount. In this case, the item is vertically centered within the space. The default setting is 13 pixels.
`pickLeftMargin`	Optional. The margin of blank space, in pixels, between the list entries and the view boundary on the left side. The default is 4.
`pickRightMargin`	Optional. The margin of blank space, in pixels, between

the list entries and the view boundary on the right side. The default is 5.

`pickTopMargin` Optional. The margin of blank space, in pixels, above each bitmap item in the list. The default is 2.

`pickBottomMargin`
Optional. The margin of blank space, in pixels, below each bitmap item in the list. The default is 2.

`pickAutoClose` Optional. If the value of this slot is non-`nil` (the default), the picker is automatically hidden after the user selects an item by tapping it. If this slot is `nil`, the picker is not hidden after a selection is made. If you want to hide the view in this case, you must explicitly send it the `Hide` message. Regardless of the setting of this slot, the picker is automatically closed if the user cancels the list by clicking outside of it.

`pickItemsMarkable`
Optional. If the value of this slot is non-`nil`, space for marks is reserved at the left side of the list. If this slot is `nil` (the default), no space for marks is reserved. Note that space is reserved for marks if any of the list items has a mark specified, regardless of the setting of this slot.

`pickMarkWidth` Optional. The number of pixels of space to reserve for marks at the left side of the list. If you don't specify this value, and marks are used, the space defaults to 10 pixels. All items are indented this amount.

`callbackContext`
Optional. The name of the view containing the `PickActionScript` and `PickCancelledScript` methods. If this slot is omitted, the picker view looks in itself for these methods.

PickActionScript

picker`:PickActionScript(`*itemPicked*`)`

This method is called when an item is selected from the picker list. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method

is called instead. Note that the `PickActionScript` method can be in the picker view itself or in a different view. If this method is in a different view, the name of that view should be stored in the `callbackContext` slot.

itemPicked For a simple list, an integer that is the index of the selected item in the `pickItems` array is passed as a parameter to this method. For two-dimensional grids, a frame with three slots:

`index`	The index of the grid item in the `pickItems` array.
`x`	The column index (zero-based) of the selected cell in the grid.
`y`	The row index (zero-based) of the selected cell in the grid.

PickCancelledScript

picker`:PickCancelledScript()`

This method is called if the picker is cancelled by a tap outside of it and `pickAutoClose` is set to non-`nil`. If you don't supply this method, there is no default action. Note that the `PickCancelledScript` method can be in the picker view itself or in a different view. If this method is in a different view, the name of that view should be stored in the `callbackContext` slot.

SetItemMark

picker`:SetItemMark(`*index*, *mark*`)`

You can call this method to set the mark character for an item in the list.

index The integer index of the item whose mark you want to set.

mark The character you want to set as the mark. Do not specify a string; you must specify a character (for example, $>). To set no mark for an item, specify `nil` for the character.

GetItemMark

picker:GetItemMark(*index*)

You can call this method to get the mark character for an item in the list. This method returns the character, or `nil` if the item has no mark character set.

index The index of the item whose mark you want to get.

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

```
picker := {...
   _proto: protoPicker,
   bounds: {left:34, top:66, right:96, bottom:96},
   viewFlags: vFloating+vReadOnly+vClickable,
   viewFormat: vfPen(2)+vfRound(4)+vfFrameBlack+
               vfFillWhite,
   pickItems: ["one",
      "two",
      'pickseparator,
      "three",
      'picksolidseparator,
      {bits:punctpict.bits, bounds:punctpict.bounds,
         width:3,height:3,cellFrame:1,outerFrame:2},
      'picksolidSeparator,
      {item:"four", icon:icon1, indent:15},
      {item:"five", icon:icon2},
      {item:keys}],
   PickActionScript: func(item)
      begin ...end,
   PickCancelledScript: func()
      Print("PickCancelledScript");
...}
```

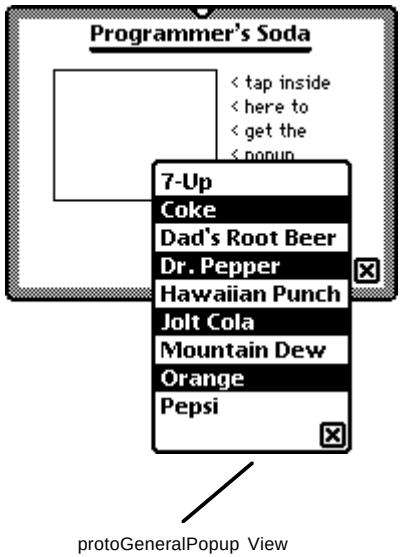
protoGeneralPopup

This proto provides a way to display a pop-up view that has a close box. The pop-up view goes away (cancels) if a user taps outside of it or taps the close box thereby executing a `ViewQuitScript`.

`protoGeneralPopup` must have a `viewBounds` frame that is set to 0 width and 0 height. In addition, the `protoGeneralPopup` can have an Affirmative script that gets called if the pop-up view is closed but not cancelled. The script takes no arguments.

Figure 6-12 shows an example of `protoGeneralPopup`. Notice that the close box is included by `protoGeneralPopup`.

Figure 6-12 Example of a pop-up view with a close box



Slot descriptions

`viewFlags`	The default value is `vClickable + vFloating + vClipping`.
`cancelled`	Internal use only. A Boolean indicating whether the user has cancelled the pop-up view; the default value is non-`nil`.
`context`	Internal use only. The callback context. You do not need to change this slot; instead let `inherited?:New` handle the call back.
`viewBounds`	The location of the pop-up global coordinates (root). You can update `viewBounds` in *popup:*`New` or in *callback:*`viewSetupFormScript`.

New

popup`:New(`*bbox*`,` *callbackContext*`)`

This method is called to open the pop-up view. You can pass in anything you want to from the parent. Usually the parent passes at least two arguments: `bbox` and `callbackContext`.

bbox	A bounding box for the pop-up view. This box is a suggestion only; generally it will be `:GlobalBox()`.
callbackContext	The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the initiating pop-up view.

PickActionScript

callbackContext`:PickActionScript(`*itemSelected*`)`

This method is called when the user taps the pop-up view. `PickActionScript` must be called by `callback:ViewClickScript`, which is described below. This method lets you track what is and is not selected. When the user taps the close box, your `callback:ViewQuitScript` method is called.

ItemSelected	The selected delta time. You can pass in any object (single value, frame, array, and so on) to this method.

PickCancelledScript

callbackContext:PickCancelledScript()

This method is called if the pop-up view is cancelled by a tap outside of it. Take care when accessing your data. The callback ViewQuitScript is executed first before callbackContext:PickCancelledScript.

ViewClickScript

callbackContext:ViewClickScript(*unit*)

This method is called when the user taps the pop-up box. In your callback function ViewClickScript, you can decode what the user tapped on and then call callback:pickActionScript.

unit The stroke unit passed to the ViewClickScript and ViewStrokeScript methods when the user taps the pen. Use GetPoint to extract a coordinate.

ViewDrawScript

callbackContext:viewDrawScript(*unit*)

This method is called when the user taps the pop-up box. In your callback:ViewClickScript, you can decode what user tapped on and then call callback:pickActionScript.

unit The stroke unit passed to the ViewClickScript and ViewStrokeScript methods when the user taps the pen. Use GetPoint to extract a coordinate.

ViewQuitScript

callbackContext:ViewQuitScript()

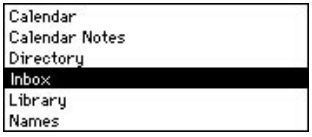
This method is called when the user taps the pop-up close box. You must do an appropriate action based on the information accumulated by callback:pickActionScript.

protoTextList

This proto creates a scrollable list of items from which the user can choose one or more items by tapping them. The selected items are highlighted in the list. The user scrolls the list by tapping the optional scroll arrows or tapping and dragging the pen either above or below the list.

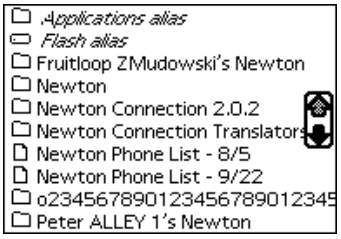
You can specify an array of strings as shown in Figure 6-13.

Figure 6-13 Scrollable list of items



or you can specify an array of shapes that include both shapes and text as shown in Figure 6-14.

Figure 6-14 Scrollable list of shapes and text



The following methods are defined internally: `ViewClickScript`, `ViewSetupChildren`, `ViewScrollDownScript`, `ViewScrollUpScript`, `DoScrollScript`, `HiliteLine`, `DrawHilite`, `SetupList`, `SetChild`, `GetTotalLines`, `GetVisibleLines`,

GetViewHeight, GetViewWidth, GetLineHeight, ShowScrollers, SetViewHeight, SetupList, HiliteLine, DrawHilite, InvertLine, and ButtonClickScript. If you need to use one of these methods, be sure to call the inherited method (for example, inherited:?ViewClickScript()), otherwise the proto may not work as expected.

The protoTextList is based on a view of the clView class. The protoTextList has a single child view, based on a view of the clTextView class (or clPictureView if shapes are shown), that displays the items in the list.

You can add or remove items from the list during run time by adding or removing items from the listItems array and then sending the view the SetupList and RedoChildren messages, in that order.

Slot descriptions

viewBounds	Set to the size and location where you want the list to appear. The value you set for the bottom bound is ignored. The bottom bound setting is calculated based on viewLines and viewFont unless viewLines is 0.
viewFont	Optional. The default font is ROM_fontSystem9.
viewFormat	Optional. The default setting is vfFillWhite + vfFrameBlack + vfPen(1).
viewLines	The number of lines to show in the list. This controls the height of the list view. If you don't specify viewLines, or if you specify 0, the number of lines that will fit in the bounds rectangle are calculated for you.
selection	Optional. This slot controls what is highlighted when the list is first displayed. On input, if you set selection to nil or -1, nothing is highlighted. You can set selection to the index of an item in the listItems array to highlight that item. The default setting is zero, highlighting the first item. On output, and while the protoTextList is displayed, selection contains the current selection. If the user doesn't select anything, selection is left as whatever the default was.

`selectedItems`	Optional. An array of selected items if multiple selection is enabled. Also contains the selected items when the user is done making the selection.
`listItems`	An array of strings or an array of shapes that are the list items. Each item in the array corresponds to one line in the list. If you specify an array of shapes, each shape must be the same size. For shapes, the size of the selection highlight is based on the height of the shape. For text, the size of the selection highlight is based on the line height of the text.
`lineHeight`	The height of each line in pixels. Set by `setupList`.
`isShapeList`	Optional. Set to non-`nil` if using picts instead of text.
`useMultipleSelections`	Optional. Set to non-`nil` to allow multiple selections.
`useScrollers`	Optional. Set to non-`nil` to include scrollers.
`scrollAmounts`	If `useScrollers` is non-`nil`, you can specify an array of three integers representing lines, pages and double-clicks.

The `protoTextList` scrolls using the `SetOrigin` method. Therefore, the slot `viewOriginY` contains the number of pixels the view is scrolled (and `viewOriginY DIV lineHeight` specifies the line number of the top displayed line). In addition, the `DoScrollScript` method scrolls the list by a specified offset.

DoScrollScript

list`:DoScrollScript(`*offset*`)`

This method scrolls the list by the specified offset.

offset	The offset, in pixels, by which to scroll.

ViewSetupFormScript

list`:ViewSetupFormScript()`

In this method, you must do two things: set the value of the `listItems` slot and call the internal method `SetupList`.

ButtonClickScript

list:ButtonClickScript(*index*)

This method is called after the pen is placed down and then lifted within the list. It is not called if the pen is lifted outside the bounds of the list.

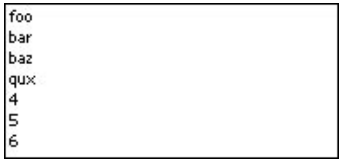
index The index of the selected item in the `listItems` array.

Note that the selected item is kept in the `selection` slot. If `multipleSelection` is enabled, the selected items are stored in the `selectedItems` slot; so, you may not need to supply a `ButtonClickScript`.

protoTable

This proto is used to create a simple one-column table of text. Each of the table items can be selected (highlighted) by tapping it. Figure 6-15 shows an example:

Figure 6-15 One-column table of text



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

The `protoTable` includes `ViewScrollUpScript` and `ViewScrollDownScript` methods to handle scrolling. However, a

view based on `protoTable` won't receive these system messages directly. To support scrolling, your application base view (which typically receives these messages from the system) should simply pass them along to the `protoTable` view.

Note

Because of the way scrolling is handled in the proto, it is possible that the last item in the table may not show up in the view when the table is scrolled to the bottom. You can fix this problem by overriding the `ViewScrollDownScript` method in the proto with a new method. ◆

Slot descriptions

`viewBounds`	Set to the size and location where you want the table to appear.
`def`	The table definition frame. Initially, you should set this to `protoTableDef`, which is the proto frame. Then in the `ViewSetupFormScript` method, you can change individual items. An example of the `protoTableDef` frame is shown on page 6-13.
`scrollAmount`	Optional. The table will scroll one row at a time when the user taps a scroll button. If you want it to scroll more rows at a time, specify the number of rows here.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfPen(1) + vfFrameBlack`.
`currentSelection`	Contains a string that is the text of the currently selected cell. If multiple selections are allowed, this string is the text of the last cell selected.
`selectedCells`	An array of indexes of selected cells. These are indexes into the `def.tabValues` array.
`declareSelf`	Do not change. This slot is set by default to `'tabbase`. This symbol identifies the view for scrolling and other internal purposes.

ViewSetupFormScript

table:`ViewSetupFormScript()`

Use this method to clone the table definition frame, `def`, if you want to change any of the values in the frame at run time.

SelectThisCell

table:`SelectThisCell(`*cell*`)`

This method is defined internally and is called when the user taps a cell in the table. If you want to be notified whenever the user taps a cell, you can override this method. However, you must call the inherited method before doing anything else in your own method. The example for `protoTable` beginning on page 6-12 shows how to do this.

cell The child view representing the cell that was tapped.

protoTableDef

This proto is used to define the format of the table. You use it by setting the `protoTable` slot `def` to `protoTableDef`. You change individual items in the `ViewSetupFormScript` method. See `protoTable` for details.

Here is a description of the slots in `protoTableDef`:

`tabAcross` The number of columns in the table. This must be set to one (1). Multi-column tables are not supported by `protoTable`.

`tabDown` The number of rows in the table.

`tabWidths` An integer giving the width of the single table column, in pixels.

`tabHeights` An integer giving the height of a row, in pixels (constant for all rows).

`tabProtos` Each row in the table is child view of the table. This slot holds either a reference to a template used to create the child views, or an array of references to templates. The

	slots for the default, `protoTableEntry` appear on page 6-52.
`tabValues`	A value that is used as the value of each of the child views. Alternately, an array of values that are mapped to table cells.
`tabValueSlot`	A symbol naming the slot in each of the child views where that child's view value (specified in `tabValues`) is stored. (Remember to quote the symbol; for example, `'text`.) For example, if the table consists of child views based on the `clParagraphView` class (the default), you would specify `'text` for this slot, since the value of a `clParagraphView` is stored in its `text` slot.
`tabUniqueSelection`	A Boolean value. Set to non-`nil` to allow only a single cell to be selected. Set to `nil` to allow multiple cells to be selected.
`indentX`	Reserved for internal use. Do not change.

IMPORTANT

If you allow multiple cell selection, your program will fail unless you ensure that the `selectedCells` slot is in RAM, since the proto attempts to add to this array. To make sure the slot is in RAM, use the following code in the `ViewSetupFormScript` method:
`self.selectedCells:=Clone(selectedCells);`. ◆

TabSetup

table`:TabSetup(`*view*`, `*column*`, `*row*`)`

This method is called before each of the child views is instantiated. It allows you to do special initialization operations to each child view before it is instantiated. If you choose to override this method, call the inherited method also: `inherited:?TabSetup(childView, hIndex, vIndex)`.

view	A reference to the child view.
column	The column number of the child within the table.

row The row number of the child within the table.

protoTableEntry

This proto controls how the text in each row of the table appears; for example text justification and type of text selection. You use it by setting the `tabProtos` slot to `protoTableEntry` in `protoTableDef`. You change individual items in the `ViewSetupFormScript` method. See `protoTable` and `protoTableDef` for details.

Here is a list of the important slots in `protoTableEntry`:

`viewClass` `clTextView` This view is a read-only `clParagraphView` and it supports no tabs or multi-styled text.

`viewFlags` `vVisible + vClickable + vReadOnly`

`viewJustify` `vjLeftH + vjCenterV + oneLineOnly`

`viewTransferMode` `modeOr`

`text` Holds the text shown in this view.

ViewClickScript

entry`:ViewClickScript()`

This method sets `currentSelection` in the parent view (the table) to the value of the `text` slot. It also sends the `SelectThisCell` message.

ViewHiliteScript

entry`:ViewHiliteScript()`

This method highlights self.

Map Pickers

These protos display various maps, let the user select a place, and return information about the location selected.

protoCountryPicker

This proto displays a picker from which a user can select a country, as shown in Figure 6-16:

Figure 6-16 Example of a country picker

You specify a `viewBounds`; the proto scales the picture to fit within it.

The picker behavior is automatic. On a tap, a picker listing nearby countries pops up; if the user selects one, the `PickWorld` message is sent to your country picker view with one parameter: a frame containing information about the country picked.

Slot descriptions

`autoClose`	Optional. Set to non-`nil` to force the `protoCountryPicker` view to close when the user chooses an item from a picker on the map. Set to `nil` to disable this auto-closing behavior. The default is `nil`.
`listLimit`	Optional. Set to the maximum number of items to be listed in one of the pickers that pops up when a user taps the map. The default value is 12.

PickWorld

picker`:PickWorld(`*info*`)`

This message is sent when the user picks a country.

info A frame describing the country picked. The following example shows the information returned (from the Inspector output):

```
{name:"Guatemala", outgoing: s00, countryCode: 502,
latitude: 23363826, longitude: 401907529, continent:
'centralAmerica, currency: "Quetzal" },
```

protoProvincePicker

This proto is used to display a picker from which a user can select a Canadian province, as shown in Figure 6-17:

Figure 6-17 Example of a province picker

You specify a `viewBounds`, and the proto scales the picture to fit within it.

The picker behavior is automatic. On a tap, a picker listing nearby provinces pops up; if the user selects one, the `PickWorld` message is sent to your province picker view with one parameter: a frame containing information about the province picked.

Slot descriptions

`viewFlags` Optional. Should you override this slot, you must set `vClipping` because this proto draws outside of its bounds.

`autoClose` Optional. Set to non-`nil` to force the `protoProvincePicker` view to close when the user

	chooses an item from a picker on the map. Set to `nil` to disable this auto-closing behavior. The default is `nil`.
`listLimit`	Optional. Set to the maximum number of items to be listed in one of the pickers that pops up when a user taps the map. The default value is 12.

PickWorld

picker`:PickWorld(`*info*`)`

This message is sent when the user picks a province.

info	A frame describing the province picked. The following is an example of the information returned (from the Inspector output):

```
{name: "Nova Scotia", latitude: 67357415, longitude:
442918502},
```

protoStatePicker

This proto is used to display a picker from which a user can select a U.S. state, as shown in Figure 6-18:

Figure 6-18 Example of a state picker

You specify a `viewBounds`, and the proto scales the picture to fit within it.

The picker behavior is automatic. On a tap, a picker listing nearby states pops up; if the user selects one, the `PickWorld` message is sent to your state

picker view with one parameter: a frame containing information about the state picked.

Slot descriptions

`viewFlags`	Optional. Should you override this slot, you must set `vClipping` because this proto draws outside of its bounds.
`autoClose`	Optional. Set to non-`nil` to force the `protoStatePicker` view to close when the user chooses an item from a picker on the map. Set to `nil` to disable this auto-closing behavior. The default is `nil`.
`listLimit`	Optional. Set to the maximum number of items to be listed in one of the pickers that pops up when a user taps the map. The default value is 12.

PickWorld

picker`:PickWorld(`*info*`)`

This message is sent when the user picks a state.

info	A frame describing the state picked. Here's an example of the information returned (from the Inspector output):

```
{name: "Florida", latitude: 42502280, longitude:
414583648},
```

protoWorldPicker

This proto is used to display a picker from which a user can select a continent, as shown in Figure 6-19:

Figure 6-19 Example of a world picker

You specify a `viewBounds` frame, and the proto scales the worldmap picture to fit within it.

The picker behavior is automatic. On a tap, a picker listing nearby continents pops up. If the user selects one, the `PickWorld` message is sent to your world picker view with one parameter: a frame containing information about the continent picked.

Slot descriptions

`autoClose`	Optional. Set to non-`nil` to force the `protoWorldPicker` view to close when the user chooses an item from a picker on the map. Set to `nil` to disable this autoclosing behavior. The default is `nil`.
`listLimit`	Optional. Set to the maximum number of items to be listed in one of the pickers that pops up when a user taps the map. The default value is 12.

PickWorld

picker`:PickWorld(`*info*`)`

This message is sent when the user picks a continent.

info	A frame describing the continent picked. Here's an example of the information returned (from the Inspector output):

```
{ name: "Europe", topLatitude: 104391566, leftLongitude:
499588209, bottomLatitude: 49213166, rightLongitude:
59652323},
```

Text Pickers

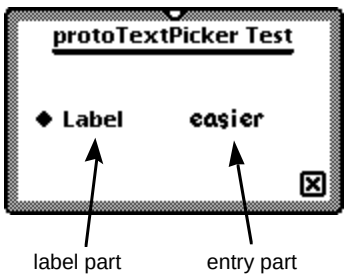
This section includes various text pickers.

protoTextPicker

This proto displays a label picker with a text representation of an entry. When the user taps on the picker, the `PopIt` method, which allows a customized picker to be displayed, is executed. If the user picks an item, the `pickActionScript` is called. If you provide a customized picker, you must call `pickActionScript` with a right `itemSelected` number.

Figure 6-20 shows an example of a slip that contains a `protoTextPicker` with its label preceded by `kPopChar`:

Figure 6-20 Example of a text picker



Slot descriptions

`label` The constant `kPopChar` & is a string to be displayed as the picker label.

`indent`	You can specify an indent; otherwise, it's calculated for you.
`labelFont`	Optional. The font for the label; the default setting is `tsSize(9) + tsBold`.
`entryFont`	Optional. The font for the text picker line; the default setting is `editFont10`.

Popit

`picker:PopIt(`*x*`)`

This method is called when the user taps the picker.

x	A value equal to (indent - 2).

PickActionScript

picker`:PickActionScript(`*item*`)`

This method is called after the user picks an item from the view displayed in `PopIt`.

item	The item passed by `PopIt`.

PickCancelledScript

picker`:PickCancelledScript()`

This method is called if the pop-up view is cancelled by a tap outside of it; if you don't supply this method, there is no default action.

TextSetup

`picker:TextSetup()`

This method returns a text string to be displayed in the entry part of the picker display.

protoDateTextPicker

This proto displays a label picker with a text representation of a date; for example "June 22, 1995." When the user taps the picker, the

`protoDatePopup` is displayed, allowing the user to specify a different date. When the user taps the close box of the pop-up view, the text next to the label is updated with the new date. Figure 6-21 shows an example:

Figure 6-21 Example of a date text pop-up view

The `Popit` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoDateTextPicker` uses the `protoTextPicker` as its proto; and `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	A string to be displayed as the picker label.
`date`	An initial date to display (as returned by the `Time` function). If you don't specify a date, the current date appears by default. This slot is also updated with the new date when the user closes the pop-up view.
`longFormat`	A symbol specifying the format in which to display the date; the default is `'yearMonthDayStrSpec`.
`shortFormat`	A symbol specifying the format in which to display the date. `ShortFormat` is used only if you have a `nil` value for `longFormat`.

See Table 19-1 in Chapter 19, “Localizing Newton Applications,” for a complete list of symbols for `longFormat` and `shortFormat`.

Notes

Both `longFormat` and `shortFormat` must be present if you plan to use `shortFormat`. If you use `shortFormat`, `longFormat` must be set to `nil`.

If you implement `pickActionScript`, the parameter `newDate` is an array containing a single element of integer; it’s the selected date in terms of minutes passed since 1/1/1904 12:00 am.

The slot `Date` always contains the selected date (in term of minutes passed since 1/1/1904 12:00 am). ◆

PickActionScript

picker`:PickActionScript(`*newDate*`)`

This method is called when the user taps the close box of the pop-up view. If you don’t supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

newDate The new date selected by the user.

PickCancelledScript

picker`:PickCancelledScript()`

This method is called if the pop-up view is cancelled by a tap outside of it; if you don’t supply this method, there is no default action.

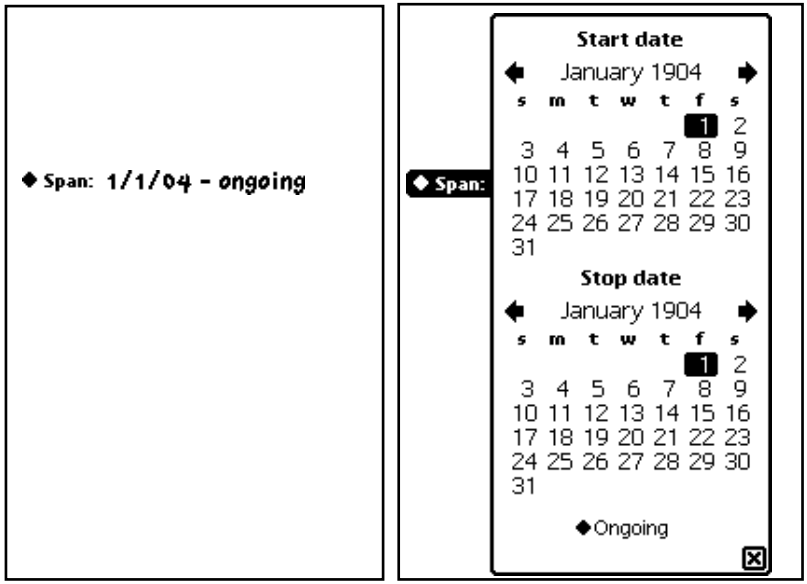
protoDateDurationTextPicker

This proto displays a label picker with a text representation of a date range; for instance “January 5, 1974 – February 7, 1975.” When the user taps the picker, the `protoDateIntervalPopup` is displayed, allowing the user to

specify a different range. When the user taps the close box of the pop-up view, the text next to the label is updated with the new date range.

Figure 6-22 shows an example of a `protoDateDurationTextPicker` with slot `shortFormat = 'numericDateStrSpec`. Notice the label is proceeded by `kPopChar`.

Figure 6-22 Example of date picker before and after it is tapped



Before Tap After Tap

The `Popit` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoDateDurationTextPicker` uses the `protoTextPicker` as its proto; `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	A string to be displayed as the picker label.
`labelFont`	Optional. Font used to display the label; the default is `ROM_fontsystem10bold`.
`entryFont`	Optional. Font used to display the picked entry; the default is `10243 (= editFont10 ?)`
`indent`	Optional. If not supplied, `protoDateDurationTextPicker` will calculate the indent based on the length of label.
`startTime`	An initial start date to display (as returned by the `Time` function).
`stopTime`	An initial end date to display (as returned by the `Time` function).
`longFormat`	A symbol specifying the format in which to display the time; the default is `'yearMonthDayStrSpec`.
`shortFormat`	A symbol specifying the format in which to display the time; the default is `nil`.

Notes

Both `longFormat` and `shortFormat` must be present if you plan to use `shortFormat`. If you use `shortFormat`, `longFormat` must be set to `nil`.

You can provide a value for either a `longFormat` slot or a `shortFormat` slot, but not both, to specify the format in which to display the date range. ◆

PickActionScript

picker`:PickActionScript(`*startTime, stopTime*`)`

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

startTime	The new starting time selected by the user.
stopTime	The new ending time selected by the user.

PickCancelledScript

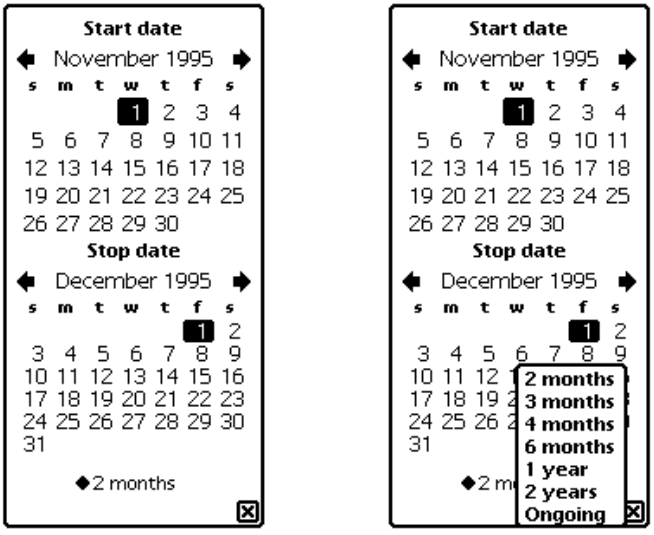
picker:`PickCancelledScript()`

This method is called if the pop-up view is cancelled by a tap outside of it; if you don't supply this method, there is no default action.

protoRepeatDateDurationTextPicker

This proto displays a label picker with a text representation of a date range; for example, "January 5, 1974 - February 7, 1975." When the user taps on the picker, the `protoDateIntervalPopup` is displayed, allowing the user to specify a different range. When the user taps on the close box, the text next to the label is updated with the new date range. Figure 6-23 shows an example,

Figure 6-23 Example label picker with text representation



Unlike `protoDateDurationTextPicker`, `protoRepeatDateDurationTextPicker`'s `protoDateIntervalPopup`'s duration picker shows choices that are appropriate for the `repeatType` slot, and the duration displayed when the user taps on a duration or stop date is given in units of the `repeatType`. For example, if the `repeatType` slot specifies monthly, the duration picker shows the choices two months, three months, and so on, and the duration value string is in units of months. In contrast, a `protoDateDurationTextPicker` would always show the choices one week, two weeks, and so on and would display the duration value in units of weeks and days.

The `viewSetupFormScript`, `Popit`, `TextSetup`, and `GetDuration` methods are defined internally; you shouldn't need to override them. If you do override them, make sure to call the inherited method.

`protoRepeatDateDurationTextPicker` uses `protoDateDurationTextPicker` as its proto; `protoDateDurationTextPicker` is based on `protoTextPicker`, which in turn is based on a view of the `clView` class.

Slot descriptions

`label`	A string to be displayed as the picker label.
`startTime`	An initial start date to display (as returned by the `Time` function).
`stopTime`	An initial ending date to display (as returned by the `Time` function).

You can provide a value for either a `longFormat` slot or a `shortFormat` slot, but not both, to specify the format in which to display the date range.

`longFormat`	A symbol specifying the format in which to display the time; the default is `nil`.
`shortFormat`	A symbol specifying the format in which to display the time; the default is `'numericDateStrSpec`.

`repeatType`	Used for repeating meetings and events. `repeatType` contains one of the following constants that describe how often the meeting repeats: `kDayofWeek`, `kWeekInMonth (1)`, `kDateInMonth (2)`, `kDateInYear(3)`, `kPeriod(4)`, `kNever(5)`, `kWeekInYear(7)`.
`mtgInfo`	Used for repeating meetings and events. An immediate value containing packed repeating meeting information. This slot is interpreted differently, depending on the value of the `repeatType` slot. For a complete list of values, see the description of this slot on page 18-75.

PickActionScript

`picker:PickActionScript(`*startTime*`, `*stopTime*`)`

This method is called when the user taps on the close box of the popup. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

startTime	The new starting time selected by the user.
stopTime	The new ending time selected by the user.

PickCancelledScript

`picker:PickCancelledScript()`

This method is called if the popup is cancelled by a tap outside of it; if you don't supply this method, there is no default action.

protoDateNTimeTextPicker

This proto displays a label picker with a text representation of a date and time; for example, "6/22/95 2:11 p.m." When the user taps the picker, the `protoDateNTimePopup` is displayed, allowing the user to specify a different date and time. When the user taps the pop-up's close box, the text next to the label is updated with the new date and time.

Figure 6-24 shows an example of a date and time label picker before and after it is tapped.

Figure 6-24 Example of a date and time pop-up view

The `Popit` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoDateNTimeTextPicker` uses the `protoTextPicker` as its proto; and `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	Optional. A string to be displayed as the picker label.
`date`	Optional. An initial date/time to display (as returned by the `Time` function). If you don't specify a date, the current date and time are used by default.
`format`	Optional. A symbol specifying the format in which to display the time; for example, "2:15 p.m." The default value is `'shortTimeStrSpec`.
`longFormat`	Optional. A symbol specifying the format in which to display the date; for example, "September 27, 1995." The default is `nil`.

`shortFormat`	Optional. A symbol specifying the format in which to display the date; for example, “9/27/95”. The default is `'numericDateStrSpec`.
`increment`	Optional. An integer representing the increment by which to change the time when the user taps the time picker portion of the pop-up view; a value of 15, for example, will cause the time to change in 15 minute increments.

Notes

You can provide a value for either a `longFormat` slot or a `shortFormat` slot, but not both, to specify the format in which to display the date and time. Because the default value of `longFormat` is `nil`, you can use `shortFormat` without providing a `longFormat` slot.◆

PickActionScript

picker`:PickActionScript(`*newDate*`)`

This method is called when the user taps the pop-up’s close box. If you don’t supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

startTime	The new date and time selected by the user.

PickCancelledScript

picker`:PickCancelledScript()`

This method is called if the pop-up view is cancelled if the user taps outside of it; if you don’t supply this method, there is no default action.

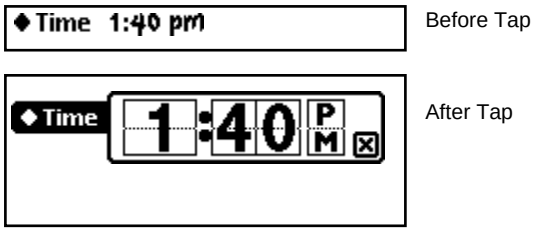
protoTimeTextPicker

This proto displays a label picker with a text representation of a time; for example, “2:56 p.m.” When the user taps the picker, the `protoTimePopup` is displayed, allowing the user to specify a different time. When the user taps

the pop-up's close box, the text next to the label is updated with the new time.

Figure 6-25 shows an example of a `protoTimeTextPicker` before and after it has been tapped:

Figure 6-25 Example of a label picker with a text representation of a time



The `Popit` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoTimeTextPicker` uses the `protoTextPicker` as its proto; and `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	The constant `kPopChar &` is a string to be displayed as the picker label.
`labelFont`	Optional. Font used to display the label. Default is `ROM_fontsystem10bold`.
`entryFont`	Optional. Font used to display the picked entry. Default is `10243 (= editFont10 ?)`
`indent`	Optional. If not supplied, `protoDateDurationTextPicker` will calculate the indent based on the length of label.
`time`	The initial time (in number of minutes since 1/1/1904 12:00 am). This value is updated by the picker as the user picks a new value.

`format`	Optional. A symbol specifying the format in which to display the time; the default is `'shortTimeStrSpec`.
`increment`	Optional. An integer representing the increment by which to change the time when the user taps the pop-up view; the default value is 12, meaning that the time will change twelve minutes for each tap.

PickActionScript

picker`:PickActionScript(`*newTime*`)`

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the pop-up view, the `PickCancelledScript` method is called instead.

newTime	The new time selected by the user.

PickCancelledScript

picker`:PickCancelledScript()`

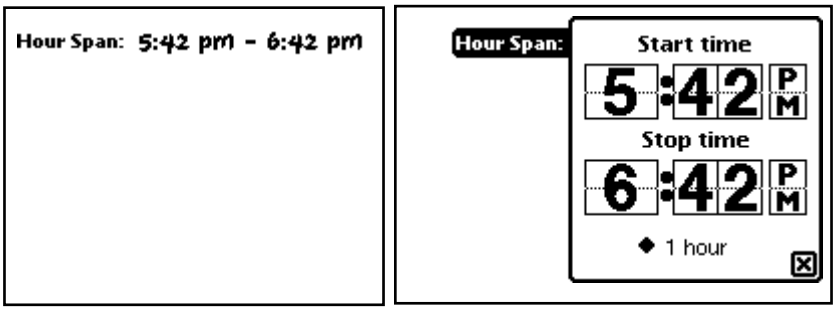
This method is called if the pop-up view is cancelled by a tap outside of it; if you don't supply this method, there is no default action.

protoDurationTextPicker

This proto displays a label picker with a text representation of a time range; for example, "2:33 p.m – 5:54 a.m." When the user taps the picker, the `protoTimeIntervalPopup` is displayed, allowing the user to specify a different range. When the user taps the pop-up's close box, the text next to the label is updated with the new time range.

Figure 6-26 shows an example `protoDurationTextPicker` before and after the user taps on the picker:

Figure 6-26 Example label picker with a text representation of a time range



Before Tap After Tap

The `Popit` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoDurationTextPicker` uses the `protoTextPicker` as its proto; and `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	A string to be displayed as the picker label.
`startTime`	An initial start time to display (as returned by the `Time` function).
`stopTime`	An initial ending time to display (as returned by the `Time` function).
`format`	A symbol specifying the format in which to display the time; the default is `'shortTimeStrSpec`. If you provide a format slot, see Table 19-1 in Chapter 19, "Localizing Newton Applications," for a complete list of symbols.
`increment`	An integer representing the increment by which to change the time when the user taps the pop-up view; the default value is 1, meaning that the time will change one minute for each tap.

PickActionScript

picker:`PickActionScript(`*startTime, stopTime*`)`

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the pop-up view, the `PickCancelledScript` method is called instead.

startTime	The start time selected by the user.
stopTime	The end time selected by the user.

PickCancelledScript

picker:`PickCancelledScript()`

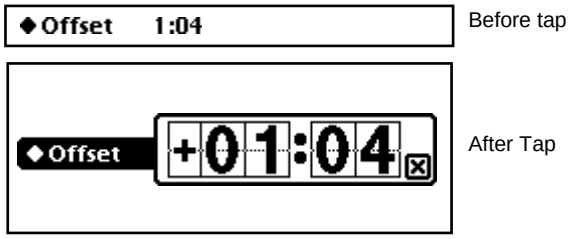
This method is called if the pop-up view is cancelled if a user taps outside of it; if you don't supply this method, there is no default action.

protoTimeDeltaTextPicker

This proto displays a label picker with a text representation of a time delta. When the user taps the picker, the `protoTimeDeltaPopup` is displayed, allowing the user to specify a new time delta. When the user taps the pop-up's close box, the text next to the label is updated with the new time delta.

Figure 6-27 shows an example of a `protoTimeDeltaTextPicker` before and after it is tapped:

Figure 6-27 Example of a label picker with a text representation of a time delta



The `Popit` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoTimeDeltaTextPicker` uses the `protoTextPicker` as its proto; and `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	The constant `kPopChar & is` a string to be displayed as the picker label.
`time`	An initial time (in number of minutes) which is then updated by the picker as a new value has been picked.
`labelFont`	Optional. Font used to display the label. Default is `ROM_fontsystem10bold`.
`entryFont`	Optional. Font used to display the picked entry. Default is `10243 (= editFont10 ?)`
`indent`	Optional. If not supplied, `protoDateDurationTextPicker` will calculate the indent based on the length of label.
`minValue`	Optional. An integer specifying a minimum delta value.

PickActionScript

picker:PickActionScript(*newDuration*)

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the PickCancelledScript method is called instead.

newDuration The number of minutes the user picked.

PickCancelledScript

picker:PickCancelledScript()

This method is called if the pop-up is cancelled by a tap outside of it; if you don't supply this method, there is no default action.

protoMapTextPicker

This proto displays a label picker with a text representation of a country; for example, "Afghanistan." When the user taps the picker, the protoLocationPopup is displayed, allowing the user to select a new country from an alphabetical list. When the user taps the pop-ups close box, the text next to the label is updated with the new country name.

Figure 6-28 shows an example of before and after a protoMapTextPicker is tapped on.

Figure 6-28 Example of a Map Label picker

Before Tap After Tap

The `Popit` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoMapTextPicker` uses the `protoTextPicker` as its proto; `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	A string to be displayed as the picker label.
`labelFont`	Optional. Font used to display the label; the default font is `ROM_fontsystem10bold`.
`entryFont`	Optional. Font used to display the picked entry; the default font is `10243 (= editFont10 ?)`
`indent`	Optional. If not supplied, the proto will calculate it based on the length of label.
`params`	A frame with the following slots:
	`spots` The name of the soup containing the country data; the default value is `CountrySoupName`.
	`result` The default value is `'name`.

PickActionScript

picker:`PickActionScript(`*newName*`)`

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `pickCancelledScript` method is called instead.

newName The new country name selected by the user.

PickCancelledScript

picker:`PickCancelledScript()`

This method is called if the pop-up view is cancelled by a tap outside of it; if you don't supply this method, there is no default action.

protoCountryTextPicker

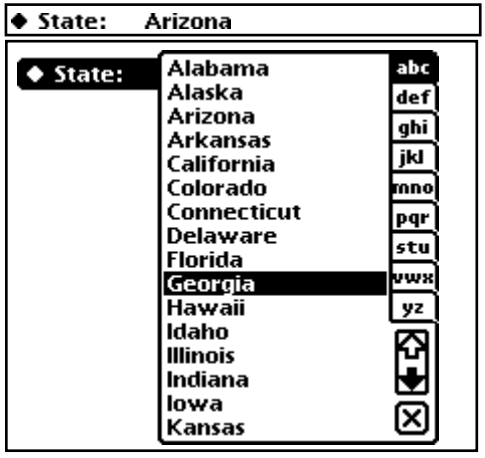
The `protoCountryTextPicker` is the same as `protoMapTextPicker` (which it uses as its proto).

protoUSstatesTextPicker

This proto displays a label picker with a text representation of a U.S. state; for example, "Ohio." When the user taps the picker, the `protoLocationPopup` is displayed, allowing the user to select a new state from an alphabetical list. When the user taps the pop-up's close box, the text next to the label is updated with the new state name.

Figure 6-29 shows an example of `protoUSstatesTextPicker` before and after it has been tapped:

Figure 6-29 Example of a label picker with a text representation of a U.S. state



The `Popit` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoUSstatesTextPicker` uses the `protoMapPicker` as its proto.

Slot descriptions

`label`	A string to be displayed as the picker label.
`labelFont`	Optional. Font used to display the label; the default font is `ROM_fontsystem10bold`.
`entryFont`	Optional. Font used to display the picked entry; the default font is `10243 (= editFont10 ?)`
`indent`	Optional. If not supplied, `protoDateDurationTextPicker` will calculate the indent based on the length of label.

`params`	A frame with the following slots:	
	`spots`	The name of the soup containing the state data; the default value is `ROM_usstatesoupnam`.
	`result`	The default value is `'name`.

PickActionScript

picker`:PickActionScript(`*newName*`)`

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

newName	The new state name selected by the user.

PickCancelledScript

picker`:PickCancelledScript()`

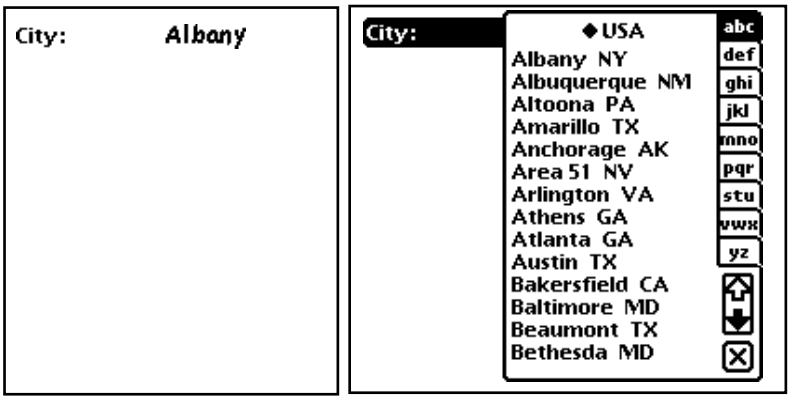
This method is called if the pop-up is cancelled by a tap outside of it; if you don't supply this method, there is no default action.

protoCitiesTextPicker

This proto displays a label picker with a text representation of a city; for example, "Albany." When the user taps the picker, the `protoLocationPopup` is displayed, allowing the user to select a new city from an alphabetical list. When the user taps the pop-up's close box, the text next to the label is updated with the new city name.

Figure 6-30 shows an example of a `protoCitiesTextPicker` before and after a tap was made:

Figure 6-30 Example of a label picker



Before Tap After Tap

The `Popit` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoCitiesTextPicker` uses the `protoMapPicker` as its proto.

Slot descriptions

`label` A string to be displayed as the picker label.

`labelFont` Optional. Font used to display the label. Default is `ROM_fontsystem10bold`.

`entryFont` Optional. Font used to display the picked entry. Default is `10243 (= editFont10 ?)`

`indent` Optional. If not supplied, `protoCitiesTextPicker` will calculate it based on the length of label.

`params` A frame with the following slots:

- `spots` The name of the soup containing the city data; the default value is `CitySoupName`.
- `result` The default value is `'name`.

Do not change the `params` slot as unexpected results will occur.

PickActionScript

picker:`PickActionScript(`*newName*`)`

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

newName The new city name selected by the user.

PickCancelledScript

picker:`PickCancelledScript()`

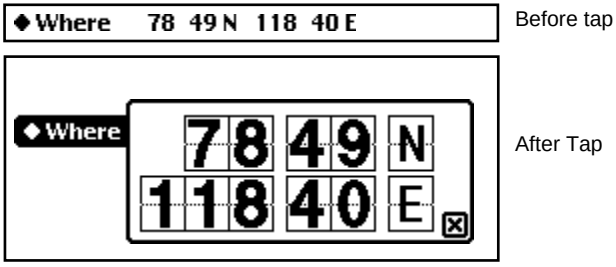
This method is called if the pop-up is cancelled when the user taps outside of it; if you don't supply this method, there is no default action.

protoLongLatTextPicker

This proto displays a label picker with a text representation of longitude and latitude values. When the user taps the picker, the `longLatPicker` is displayed, allowing the user to select new values for longitude and latitude. When the user taps the pop-up's close box, the text next to the label is updated with the new values.

Figure 6-31 shows an example of `protoLongLatTextPicker` before and after it has been tapped.

Figure 6-31 Example of a text representation of longitude and latitude values



The `Popit` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoLongLatTextPicker` uses the `protoTextPicker` as its proto; `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	The constant `kPopChar` followed by a string to be displayed as the picker label; it's "Where" by default.
`latitude`	An integer specifying the latitude to display initially.
`longitude`	An integer specifying the longitude to display initially.
`labelFont`	Optional. The default font is `systemfont10bold`.
`entryFont`	Optional. The default font is `editFont10`.
`indent`	Optional. The distance, in pixels, to indent the picker from the beginning of the line (the beginning of the text label). If you don't include this slot, the picker is placed 6 pixels to the right of the text label by default.
`worldClock`	A boolean, must be non-`nil`.

See page 18-28 for a code example on how to calculate longitude and latitude.

PickActionScript

picker:`PickActionScript(`*long*, *lat*`)`

This method is called when an item is selected from the pop-up view. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the pop-up view, the `PickCancelledScript` method is called instead.

long	The new longitude selected by the user.
lat	The new latitude selected by the user.

When the user picks new longitude or latitude value, the slots `longitude` and `latitude` are automatically updated.

PickCancelledScript

picker:`PickCancelledScript()`

This method is called if the pop-up view is cancelled when the user taps outside of it; if you don't supply this method, there is no default action.

Date, Time, and Location Pop-up Views

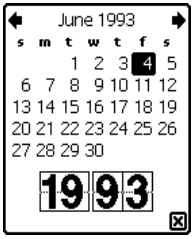
These protos let the user specify dates, times, and locations using graphical pop-up views.

protoDatePopup

This proto lets the user choose a single date. To provide selection of multiple dates, use the `protoMultiDatePopup` proto. The user confirms the selected date by tapping the close box; tapping outside the pop-up view cancels the pop-up view.

Figure 6-32 shows the result of a single-date selection.

Figure 6-32 Example of a single date selection



New

popup`:New(`*initialDate*`,` *bbox*`,` *callbackContext*`)`

This method is called to open the pop-up view.

initialDate	An array containing one element, an integer representing the initial date to display selected (as returned by the `Time` function).
bbox	A bounding box for the pop-up view. This box is a suggestion only; generally it will be `:GlobalBox()`.
callbackContext	The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view

PickActionScript

callbackContext`:PickActionScript(`*selectedDate*`)`

This method is called when the user taps the close box.

selectedDate	An array containing a single date.

PickCancelledScript

callbackContext`:PickCancelledScript()`

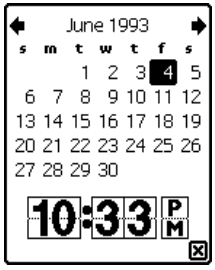
This method is called if the pop-up view is cancelled by a tap outside of it.

protoDateNTimePopup

This proto lets the user choose a single date and time. The user confirms the selection by tapping the close box; tapping outside the pop-up view cancels it.

Figure 6-33 shows the result of a date and time selection:

Figure 6-33 Example of a single date and time selection



New

popup:New(*dateNTime*, *increment*, *bbox*, *callbackContext*)

This method is called to open the pop-up view.

dateNTime	An array containing one element, an integer, representing the initial date and time to display as selected (as returned by the `Time` function).
increment	An increment value that determines the minimal granularity for the pop-up view. The value “1” specifies one minute; try “15”, “30”, and “60”.
bbox	A bounding box for the pop-up view. This box is a suggestion only; generally it will be `:GlobalBox()`.

callbackContext The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view.

NewTime

callbackContext`:NewTime(`*dateNTime*`)`

This method is called whenever the time is changed.

dateNTime The new date and time.

PickActionScript

callbackContext`:PickActionScript(`*dateNTime*`)`

This method is called when the user taps the close box.

dateNTime The selected date and time.

PickCancelledScript

callbackContext`:PickCancelledScript()`

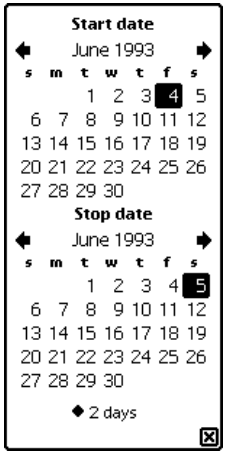
This method is called if the pop-up view is cancelled when the user taps outside of it; if you don't supply this method, there is no default action.

protoDateIntervalPopup

This proto lets the user specify an interval of dates by selecting a start and stop date. The user confirms the selection by tapping the close box; tapping outside the pop-up view cancels it.

Figure 6-34 shows the result of selecting a start and stop date:

Figure 6-34 Example of a date interval pop-up view



The `protoDateIntervalPopup` is based on the `protoGeneralPopup` proto. It has the following two child views declared in itself:

- `start`. This child view uses the `protoDatePicker` proto and implements the starting date section of the pop-up view.
- `stop`. This child view uses the `protoDatePicker` proto and implements the ending date section of the pop-up view.

New

popup`:New(`*initialDates*`,` *bbox*`,` *callbackContext*`)`

This method is called to open the pop-up view.

initialDates	An array with two values (as returned by the `Time` function) specifying the initial range of dates to display as selected.
bbox	A bounding box for the pop-up view. This box is a suggestion only; generally it will be `:GlobalBox()`.

callbackContext The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view.

NewTime

callbackContext`:NewTime(`*startDate*`, `*stopOrMax*`)`

This method is called each time the user changes the selection.

startDate The new start date.

stopOrMax The new stop date, or the maximum time if ongoing. Note that the maximum time is defined as the constant `kMaximumTime := 0x1FFFFFFF`.

PickActionScript

callbackContext`:PickActionScript(`*startDate*`, `*stopOrMax*`)`

This method is called when the user taps the close box.

startDate The new start date.

stopOrMax The new stop date, or the maximum time if ongoing. Note that the maximum time is defined as the constant `kMaximumTime := 0x1FFFFFFF`

PickCancelledScript

callbackContext`:PickCancelledScript()`

This method is called if the pop-up view is cancelled by a tap outside of it.

protoMultiDatePopup

This proto lets the user specify a range of dates. To provide selection of a single date, use the `protoDatePopup` proto. The user confirms the selected range by tapping the close box; tapping outside the pop-up view cancels it.

Figure 6-35 shows the result of selecting a range of dates:

Figure 6-35 Example of a multidate pop-up view

New

popup`:New(`*initialDates*`,` *bbox*`,` *callbackContext*`)`

This method is called to open the pop-up view.

initialDates	An array specifying a range of dates to display as selected.
bbox	A bounding box for the pop-up view. This box is a suggestion only; generally it will be `:GlobalBox()`.
callbackContext	The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view.

PickActionScript

callbackContext`:PickActionScript(`*selectedDates*`)`

This method is called when the user taps the close box.

selectedDates	An array containing the selected range of dates.

PickCancelledScript

callbackContext`:PickCancelledScript()`

This method is called if the pop-up view is cancelled when the user taps outside of it; if you don't supply this method, there is no default action.

protoYearPopup

This proto lets the user specify a year. The user confirms the selected range by tapping the close box; tapping outside the pop-up view cancels it.

Figure 6-36 shows the result of selecting a year.

Figure 6-36 Example of a year pop-up view

New

popup:New(*initialYear*, *bbox*, *callbackContext*)

This method is called to open the pop-up view.

initialYear	The year to display initially, specified as an integer (for example, “1995”).
bbox	A bounding box for the pop-up view. This box is a suggestion only; generally it will be :GlobalBox().
callbackContext	The name of the view to which callback messages should be sent. Specify self if you define these methods in the pop-up view.

NewYear

callbackContext:NewYear(*year*)

This method is called each time the user changes the selection.

year	The new year, specified as a year.

DoneYear

callbackContext:`DoneYear(`*year*`)`

This method is called when the user taps the close box.

year The selected year, specified as a year.

PickCancelledScript

callbackContext:`PickCancelledScript()`

This method is called if the pop-up view is cancelled when the user taps outside of it; if you don't supply this method, there is no default action.

protoTimePopup

This proto lets the user choose a time using a digital clock. The user confirms the selection by tapping the close box; tapping outside the pop-up view cancels it.

Figure 6-37 shows how the digital clock appears.

Figure 6-37 Example of a time pop-up view

New

popup:`New(`*time*`,` *increment*`,` *bbox*`,` *callbackContext*`)`

This method is called to open the pop-up view.

time An array containing one element, an integer representing the initial time to display as selected (as returned by the `Time` function).

increment	An increment for the pop-up view that determines the minimal granularity for the pop-up view. The value “1” specifies one minute; try “15”, “30”, and “60”.
bbox	A bounding box for the pop-up view. This box is a suggestion only; generally it will be `:GlobalBox()`.
callbackContext	The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view.

NewTime

callbackContext`:NewTime(`*time*`)`

This method is called whenever the time is changed.

time	The new time.

PickActionScript

callbackContext`:PickActionScript(`*time*`)`

This method is called when the user taps the close box.

time	The selected time.

PickCancelledScript

callbackContext`:PickCancelledScript()`

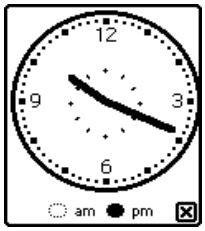
This method is called if the pop-up view is cancelled by a tap outside of it.

protoAnalogTimePopup

This proto lets the user choose a time using an analog clock. The user confirms the selection by tapping the close box; tapping outside the pop-up view cancels it.

Figure 6-38 shows how the analog clock appears:

Figure 6-38 Example of an analog time pop-up view



New

popup`:New(`*time*`, `*increment*`, `*bbox*`, `*callbackContext*`)`

This method is called to open the pop-up view.

time	An array containing one element, an integer representing the initial time to display as selected (as returned by the `Time` function).
increment	An increment for the pop-up view that determines the minimal granularity for it. The value “1” specifies one minute; try “15”, “30”, and “60”.
bbox	A bounding box for the pop-up view. This box is a suggestion only; generally it will be `:GlobalBox()`.
callbackContext	The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view.

NewTime

callbackContext`:NewTime(`*time*`)`

This method is called whenever the time is changed.

time	The new time.

PickActionScript

callbackContext:PickActionScript(*time*)

This method is called when the user taps the close box.

time The selected time.

PickCancelledScript

callbackContext:PickCancelledScript()

This method is called if the pop-up view is cancelled when the user taps outside of it; if you don't supply this method, there is no default action.

protoTimeDeltaPopup

This proto lets the user choose a time period, or delta. The user confirms the selection by tapping the close box; tapping outside the pop-up view cancels it.

Figure 6-39 illustrates this time choice option.

Figure 6-39 Example of a time delta pop-up view

New

popup:New(*initialDelta*, *params*, *bbox*, *callbackContext*)

This method is called to open the pop-up view.

initialDelta An integer representing the initial delta time. A value of "1" specifies one minute, and the sign of the value specifies whether the delta is positive (+) or negative (–).

params	A frame containing the following slots: `increment` An increment value that determines the minimal granularity for the pop-up view. The value “1” specifies one minute; try “15”, “30”, and “60”. `minValue` Optional. A minimum delta value. `maxValue` Optional. A maximum delta value.
bbox	A bounding box for the pop-up view. This box is a suggestion only; generally it will be `:GlobalBox()`.
callbackContext	The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view.

PickActionScript

callbackContext`:PickActionScript(`*delta*`)`

This method is called when the user taps the close box.

delta	The selected delta time.

PickCancelledScript

callbackContext`:PickCancelledScript()`

This method is called if the pop-up view is cancelled by a tap outside of it.

protoTimeIntervalPopup

This proto lets the user choose a time interval by specifying a start and stop time. The user confirms the selection by tapping the close box; tapping outside the pop-up view cancels it.

Figure 6-40 illustrates a time interval selection.

Figure 6-40 Example of a time interval pop-up view

New

popup:New(*dateNTime*, *increment*, *bbox*, *callbackContext*)

This method is called to open the pop-up view.

initialTimes	An array with two values (as returned by the Time function) specifying the initial range of start and stop times.
increment	An increment value that determines the minimal granularity for the pop-up view. The value “1” specifies one minute; try “15”, “30”, and “60”.
bbox	A bounding box for the pop-up view. This box is a suggestion only; generally it will be :GlobalBox().
callbackContext	The name of the view to which callback messages should be sent. Specify self if you define these methods in the pop-up view.

PickActionScript

callbackContext:PickActionScript(*startTime*, *stopTime*)

This method is called when the user taps the close box.

startTime	The selected start time.
stopTime	The selected stop time.

PickCancelledScript

callbackContext:PickCancelledScript()

This method is called if the pop-up view is cancelled by a tap outside of it.

protoLocationPopup

An abstract prototype for general pop-up views driven by the worldclock data (for example, cities, states, countries — any soup that has name, latitude and longitude slots can use this pop-up proto.

New

popup:New(*params*, *bbox*, *callbackContext*)

This method is called to open the pop-up view.

params — A frame containing the following slots:

spots	name of the soup containing the locations (CitySoupName)
default	the actual entry that should default showing (nil)
returnSlot	the slot symbol to be passed to pickAction nil = whole entry (nil)

bbox — A bounding box for the pop-up view. This box is a suggestion only; generally it will be :GlobalBox().

callbackContext — The name of the view to which callback messages should be sent. Specify self if you define these methods in the pop-up view.

PickActionScript

callbackContext:PickActionScript(*location*)

This method is called when the user taps the close box.

location — Unless overridden by the returnSlot parameter, the value of location is the record from the soup.

PickCancelledScript

callbackContext:`PickCancelledScript()`

This method is called if the pop-up view is cancelled by a tap outside of it.

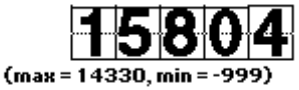
Number Pickers

This section describes the protos used to display pickers with numbers.

protoNumberPicker

This proto is used to display a picker from which the user can select a number. Figure 6-41 shows an example:

Figure 6-41 Example of a number picker



The following slots are of interest:

`minValue` Required. The minimum value in the list.

`maxValue` Required. The maximum value in the list.

`value` Required. The initial and currently selected value.

`showLeadingZeros`
Optional. Set this slot to non-`nil` to show leading zeros; for example, to show “007” with the two leading zeros.

This proto is based on a view of the `clPictureView` class. It has one child view, based on the `protoDigit` proto, for each digit in the number; these views implement the picker functionality of the proto.

PrepareForClick

picker:`PrepareForClick()`

This method is called before a click on an individual digit is processed. The `value` slot is updated accordingly.

ClickDone

picker:`ClickDone()`

This method is called after a click on an individual digit is processed. The `value` slot is updated accordingly. You can override this method and check the `value` slot to determine the selected value.

protoDigit

This proto allows you to display and navigate among a sequence of equal-sized graphics stored side-by-side in a single bitmap. Used in a cascade, this proto can display multiple digits. Figure 6-42 shows a multiple digit display.

Figure 6-42 Example of a multiple digit display

The user can tap on the top half of the displayed graphic to advance to the next graphic in the sequence, or can tap on the bottom of the graphic to display the previous graphic in the sequence. If the user taps and holds the pen down on the graphic, it will advance repeatedly, after an initial delay.

To make this proto automatically advance periodically, you can use a `ViewIdleScript` method as follows:

```
ViewIdleScript: func()
   begin
   self:Advance(1);
```

```
return kDelay; //delay in milliseconds to next Advance
end
```

Note that the `ViewSetupFormScript`, `ViewClickScript`, and `ViewDrawScript` methods are used internally in the `protoDigit` and should not be overridden.

The `protoDigit` is based on a view of the `clPictureView` class.

Slot descriptions

`viewFlags`	The default is `vVisible + vClickable`.
`viewBounds`	Set the frame to the size and location where you want the list to appear.
`digits`	The bitmap object. This must contain a sequence of equal-size graphics arranged in a horizontal strip.
`totalFrames`	An integer representing the number of individual graphics (or "frames") stored in the `digits` bitmap. Note that you must set this value before the `ViewSetupFormScript` is executed, because it is used therein.
`increment`	Optional. Set this slot to an integer indicating the number of "frames" to advance or back up when the user taps the displayed graphic. The default setting is 1.
`firstTapDelay`	Optional. An integer indicating the number of ticks to delay before advancing the graphic when the user taps it. The default setting is 30 ticks.
`repeatingTapDelay`	Optional. An integer indicating the number of ticks to delay before advancing the graphic each increment after the first graphic when the user taps and holds the pen down on it. The default setting is 15 ticks.
`digitFrame`	Optional. A rectangle that specifies the size and shape of the individual slides in the strip. If not provided, it is

	calculated from the size of the `digits` and `totalFrames` slots.
`copyMode`	Optional. A constant specifying the transfer mode to be used in drawing the graphic. Specify one of the standard view transfer modes described in the section "viewTransferMode Constants" in the "Chapter 3, "Views." The default setting is `modeCopy`.
`value`	Required. This slot holds the index of the graphic currently being displayed. This is the index in the `digits` bitmap of the current graphic, counting from the left and beginning with zero.

PrepareForClick

`PrepareForClick()`

This method is called when the user taps the view, before any processing is done by the proto. It allows you to take some action before the graphic is advanced. This method is passed no parameters and its return value is ignored.

ClickDone

`ClickDone()`

This method is called when the user taps the view, after all processing is done by the proto (the graphic is changed). `ClickDone` gives you a chance to take some action after the graphic is advanced. This method is passed no parameters and its return value is ignored.

Advance

`Advance(`*increment*`)`

Send this message to programmatically change the displayed graphic to a different graphic.

increment	An integer representing the increment to advance from the current position in the strip of graphics. For example, specify "1 "to advance to the next graphic to

the right in the strip; specify “-1” to move back to the previous graphic to the left in the strip.

If you advance past the end of the strip, the proto wraps around to the other end of the strip and continues in the same direction. This method returns non-`nil` if advancing caused a wrap to the other end of the strip; otherwise it returns `nil`.

To make this proto automatically advance periodically, you can use a `ViewIdleScript` method like this:

```
ViewIdleScript: func()
   begin
   self:Advance(1);
   return kDelay; //delay in milliseconds to next Advance
   end
```

GetIndex

`GetIndex(`*value*`)`

This method returns the current digit based on the value. By default, this method returns the value `mod 10`.

value An integer representing the index of the current digit.

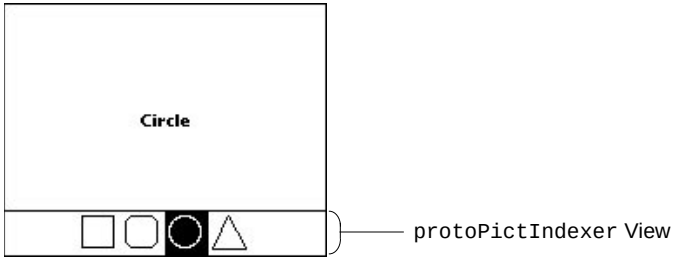
Picture Picker

This section describes the protos used to create a picture as a picker.

protoPictIndexer

This proto is used to create a view with a horizontal array of pictures, one of which the user can tap. When the user taps a picture, it is highlighted, and the system sends the `IndexClickScript` to signal which picture was selected. Figure 6-43 shows a typical array of pictures from which a user might make a selection.

Figure 6-43 Example of an indexed array of pictures



The following methods are defined internally: `ViewSetupDoneScript`, `ViewDrawScript`, `ViewClickScript`, `Hilite`, `Unhilite`, and `TrackPictHilite`. If you need to use one of these methods, be sure to call the inherited method also; for example,

```
inherited:?ViewSetupDoneScript()
```

or the proto may not work as expected.

The `protoPictIndexer` is based on a view of the `clPictureView` class.

Slot descriptions

`viewBounds`	Set to the size and location in which you want the view to appear.
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV + vjParentFullH + vjParentBottomV`.
`viewFormat`	Optional. The default setting is `vfFillWhite`.
`icon`	The bitmap serving as the picture. This picture should be a single bitmap containing multiple objects or symbols that are all of the same width and arranged next to each other in a vertical row.
`iconBBox`	A bounds frame giving the bounds of the bitmap within the view. (The view can be bigger than the bitmap.) The width is the only important dimension calculated from

	the bounds frame. The width is used to calculate the size of the active rectangle; that is, the rectangle used to differentiate and highlight each of the objects in the bitmap. The width of the bitmap is divided into (width÷`numIndices`) equal rectangles that extend the full height of the view.
`numIndices`	The number of objects or symbols in the bitmap.
`curIndex`	This slot is set to the index of the currently selected item in the bitmap. Note that the first item has an index of zero. This slot must initially be set to an integer.

IndexClickScript

`IndexClickScript()`

This method is called whenever the user taps the bitmap.

index	The index of the item that was chosen from the pop-up array. Note that the first item has an index of zero.

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

```
indexView := {...
_proto: protopictindexer,
viewBounds: {top: -25, left: 0, right: 0, bottom: 0},
viewJustify: vjCenterH+vjCenterV+vjParentFullH+
             vjParentBottomV,
viewFormat: vfFillWhite+vfPen(1)+vfFrameBlack,
icon: shapesBitmap, //square, roundrect, circle, triangle
iconBBox: {top: 0, left: 0, right: 100, bottom: 0},
numIndices: 4,
IndexClickScript: func(currIndex)
   begin
   SetValue(theText, 'text, shapeArray[currIndex]);
   end,
// set highlight to first item on entry
curIndex: 0,
...}
```

Overview Protos

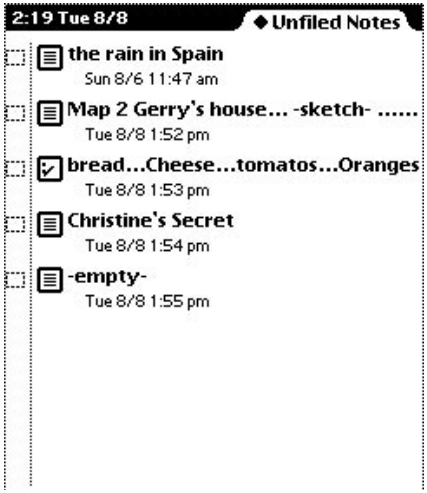
The protos in this section are used to create overviews of data; they include some protos specifically designed to display names from the Names soup.

protoOverview

This proto provides a framework for doing an overview view of data in an application. Each item in the overview has one line; the user can scroll the list and pick individual items or multiple items in the list.

Each entry in the list is a set of shapes that is created by the client application. Figure 6-44 is an example:

Figure 6-44 Example of an overview list



Note that the `ViewClickScript` and `ViewDrawScript` methods are used internally in the `protoOverview` and should not be overridden.

Slot descriptions

`viewBounds` Set to the size and location where you want the overview to appear.

`viewFlags` The default is `vVisible + vApplication + vClickable`.

`lineHeight` Optional. The default is 32, which specifies the height in pixels of each item in the overview.

`selectIndent` Optional. Specifies the left margin within which selection highlighting and behavior occur. If an item is tapped within this margin, the default `HitItem` method calls the `SelectItem` method with the item index. The default, if you don't supply this slot, is 18.

`selected` Required. Initially set to `nil`; it is modified by `protoSoupOverview` as the user selects and deselects overview items. This proto is an array of aliases to the selected items when the overview is closed. For example:

```
[[alias: NIL, 66282812, 84, "Names"],
 [alias: NIL, 66282812, 85, "Names"]]
```

`viewFont` Optional. The default setting is `systemFont10Bold`.

`nothingCheckable` Optional. If you don't want checkboxes at all, set this slot to non-`nil`. None of the list items will be indented and the vertical line down the left side of the list will be removed.

SelectItem

overview`:SelectItem()`

This method must be provided if `SelectIndent` is greater than 0 and should implement a way of remembering selected items, so the user can do filing or routing of the selected items at a later point.

SetupAbstracts

overview:SetupAbstracts(*cursor*)

A method that should be called from the ViewSetupChildrenScript as the instantiator.

cursor — A cursor to the soup or a protoErstazCursor if it is a cursor or soup.

Abstract

overview:Abstract(*entry, bounds*)

This method should return a shape or shape list representing an item in the overview. It is passed two parameters, the first an entry obtained from the cursor passed to SetupAbstracts, the second a bounds frame within which the returned shape should be placed.

entry — The entry returned from the cursor that was passed to the SetupAbstracts method.

bounds — A bounds frame within which the returned shape should be placed.

HitItem

overview:HitItem(*index, x, y*)

A method that is called when an item is tapped. The default method returns non-nil if it handled the tap; that is, if it determined it should select the item (if the tap was within the selectIndent margin). In general, you should first call inherited:HitItem, and only handle the tap yourself if the inherited method returns nil.

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

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

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

CheckState

overview:CheckState(*entry*)

If the nothingCheckable method is set to nil, the SetUpAbstracts method calls CheckState for each entry. You can override CheckState to return one of the following values:

Value	Meaning
'notCheckable	Cannot be checked; don't put a checkbox here.
nil	Can be checked, but isn't.
true	Can be checked, and is.

CheckState returns nil by default (checkable, but not checked). If checkboxes are specified, they are centered vertically based on the value in the lineHeight slot.

entry A soup entry.

Scroller

overview:Scroller(*numItems*)

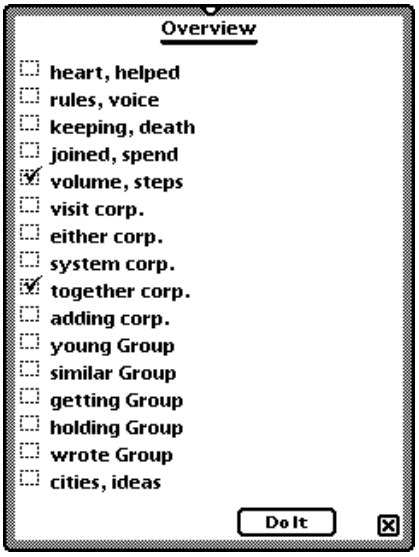
Scrolls the contents of the overview. The default method does nothing. If overridden, Scroller should cause the SetupAbstracts method to be called again, for example, by calling RedoChildren.

numItems The number of items to scroll, with a negative value meaning to scroll in an upwards direction.

protoSoupOverview

This proto is similar to protoOverview, but is designed to work with data that consists of soup entries. It expects each overview item to be a soup entry (whether or not the cursor itself is an ordinary soup cursor). Figure 6-45 shows an example of this proto:

Figure 6-45 Example of a soup entry proto



A default `ViewSetupChildrenScript` method calls `SetupAbstracts`, passing the cursor in the `cursor` slot. If you override this method, you should call the inherited method once you've set up the cursor slot.

Slot description

`cursor`	Required. Set this slot to a cursor describing your entries. Initialized in `ViewSetupFormScript`; soup cursor.

All other slots are the same as in `protoOverview`.

Scroller

overview`:Scroller(`*numItems*`)`

This optional method scrolls the overview, with respect to the specified cursor (in the `cursor` slot). The default is to take an integer and move the

cursor forward (positive) or backward (negative). If you try to move the cursor forward past the end, the last item is returned. If you try to move the cursor backward before the first item, the first item is removed.

numItems The number of items to scroll, with a negative value meaning to scroll in an upwards direction.

SelectItem

overview`:SelectItem()`

This method remembers selected items, doing the right thing with respect to the specified cursor (in the `cursor` slot). It keeps a list of the selected items by getting an alias to them, hence the need for the items to be real soup entries.

Abstract

overview`:Abstract(`*entry*`,` *bounds*`)`

This required method should return a shape or shape list representing an item in the overview.

entry The entry returned from the cursor that was passed to the `SetupAbstracts` method.

bounds A bounds frame within which the returned shape should be placed.

IsSelected

overview`:IsSelected(`*entry*`,` *fieldPath*`,` *format*`)`

This method returns non-`nil` if the specified entry is currently selected.

entry An entry from the cursor.

ForEachSelected

overview:ForEachSelected(*function*)

This method calls the specified function once for each entry that is currently selected. The function is passed one argument: the entry.

function A function object.

protoListPicker

This proto provides a scrollable list of items, from either a soup or an array (or both), from which one or many may be chosen. The list is built from a soup, using a cursor. By default, this proto queries the "Names" soup, but you can change it to query a different soup.

The selections are intended to be persistent, so enough information from soup entries is maintained to allow the selection to be displayed even if the soup is removed.

The protoListPicker is based on a view of the clView class.

The viewFlags, viewBounds, viewJustify, and viewFormat slots can be overridden at will. The ViewScrollUpScript and ViewScrollDownScript methods are provided for the developer to invoke.

The following slots and methods are used internally: ViewSetupChildrenScript, ViewDrawScript, ViewQuitScript, fOpenEditView, nowShowing, fBorder, cursor, myQuerySpec, fCurrentKey, MarkCursorPosition, filterLabels, SetupFiltering, SetupCursor, RedoCursor, GetTargetInfo, FilterChanged, folderTabs, AZtabs, and listBase.

Slot descriptions

defaultJustification
The default is vjParentFull + vjParentTopV

viewFlags The default is vVisible + vApplication +vClickable.

`viewBounds`	Set to the size and location where you want the list of scrollable items to appear.
`lineHeight`	Optional. Set to the height, in pixels, of each line in the list. The default setting is the maximum of the font height and the checkmark height.
`listFormat`	Optional. Specify `viewFormat` flags to be used for the `viewFormat` slot of the list child view. The default setting is `vfFrameGray + vfPen(1)`.
`pickerDef`	Required. A frame used to determine the overall behavior of the list picker. This frame should proto from `protoNameRefDataDef`, `protoPeopleDataDef`, or at least support the required slots. For details, see the section "Using protoListPicker" beginning on page 6-18.
`selected`	Required. An array of references. Set this slot in the `ViewSetupFormScript` method if you want the list to be displayed with one or more items preselected. Note that the name reference data definition contains the `_unselected` slot, which can be used to override the preselection of individual items (even though they are in the `selected` array). While the list picker is open, the selected list is not valid until the picker's `ViewQuitScript` has run. Any operations on the data should be postponed, either by using the `'postQuit` deferral mechanism, or by calling the inherited `ViewQuitScript` method *before* your own operations.
`soupToQuery`	Optional. A string specifying the union soup to query, or a function that returns a soup. This slot overrides any soup specified in the data definition. By default, no soup is queried.
`querySpec`	Passed to the query routine. The `tagSpec` slot is replaced internally, and the `validTest` may be enhanced internally to allow the checkbox filtering and folder support. This slot overrides any `querySpec` specified in the data definition.

`suppressNew` Optional. If this slot is present and its value is non-`nil`, the "New" button is not drawn.

`suppressScrollers`

Optional. If this slot is present and its value is non-`nil`, the up and down scroll arrows are not drawn.

`suppressAZTabs` Optional. If this slot is present and its value is non-`nil`, the a-z tabs are not drawn.

`suppressFolderTabs`

Optional. If this slot is present and its value is non-`nil`, the folder tabs are not drawn.

`suppressSelOnlyCheckbox`

Optional. If this slot is present and its value is non-`nil`, the "Selected Only" checkbox is not drawn.

`suppressCloseBox`

Optional. If this slot is present and its value is non-`nil`, the close box is not drawn.

`suppressCounter`

Optional. Suppresses the text at the bottom right indicating how many items are selected

`reviewSelections`

Optional. If present and non-nil, and if `singleSelect` is `nil`, when the picker is opened with pre-selected items, the "Selected Only" checkbox is checked.

`readOnly` Optional. If present and non-`nil`, constrains the interface so that the currently selected list can be viewed but not changed. All taps in the body of the picker are ignored, the "New" button and "Selected Only" checkbox are hidden, and the checkboxes are suppressed.

`dontPurge` Optional. If present and non-`nil`, prevents unselected name references from being stripped out of the selected array when the picker is closed. You may also specify this slot in the data definition.

`soupChangeSymbol`

The symbol to use in the `RegSoupChange` message; by default, its `'listpicker`.

The list picker automatically registers to be notified for soup change notification on the soup it will be querying. By default, only the `SoupEnters` and `SoupLeaves` messages are handled. To handle any other messages, or to override the default behavior for the `SoupEnters` or `SoupLeaves` change types, add a slot whose name is the `changeType` you wish to support, and make its value a function of a `soupName` and the `changeData`. This function will be called when the soup notification is received with that `changeType`.

SoupEnters

picker`:SoupEnters(`*soupName*`,` *changeData*`)`

Called when the list picker is notified that the soup has changed and the *changeType* is `'soupEnters`. By default, the list picker just calls `:redoChildren` to make sure that any items in the newly arrived soup that should be showing are displayed.

SoupLeaves

picker`:SoupLeaves(`*soupName*`,` *changeData*`)`

Similar to `SoupEnters`, synchronizes the cursor in case it was pointing to an entry removed with a soup and then refreshes the list.

ChangePickerDef

picker`:ChangePickerDef(`*newPickerDef*`)`

You can use this method to specify a new data definition; the behavior is updated immediately.

newPickerDef A new data definition.

SetNowShowing

picker:SetNowShowing(*value*)

Sending this message is equivalent to tapping the “Selected Only” button

value A symbol, where 'all means show all entries and 'selected means show only selected entries.

AddFakeItem

picker:AddFakeItem(*item*)

This method adds the specified item to the selected array and updates the screen.

item A new entry to display.

GetSelected

picker:GetSelected(*activeOnly*)

This method returns a clone of the selected array.

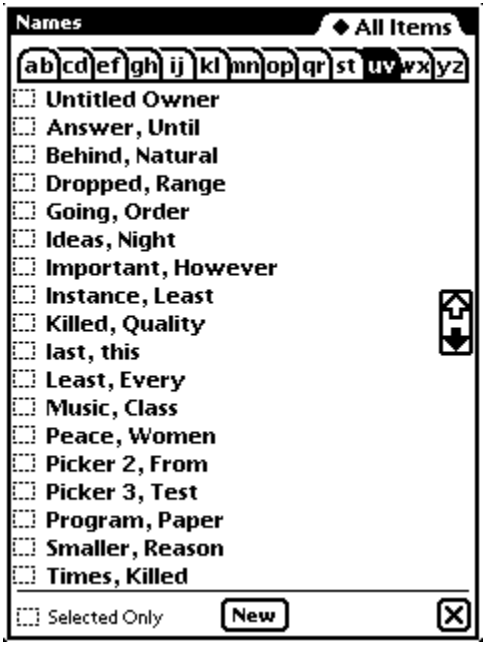
activeOnly A Boolean which, if non-nil, returns an array that is stripped of any _unselected name references.

protoNameRefDataDef

The protoListPicker proto is driven in large part by the data definition specified in the pickerDef slot. The protoNameRefDataDef proto is provided for creating your own data definitions.

Figure 6-46 shows an example of a protoListPicker with the slot pickerDef set to protoPeopleDataDef.

Figure 6-46 Example of a `protoListPicker` proto



All calls to methods in the `pickerDef` slot are handled by sending the message to the frame itself, so the methods described below can use inherited functions and store data in the frame itself as needed.

Slot descriptions

`name` The name that is shown in the top left corner of the picker: “E-Mail addresses” in the examples pictured “Using protoListPicker” on page 6-18. The default value (in English) is “Names.”

`class` A symbol specifying the class to which all name references should be set; the default value is `'nameRef`.

`entryType`	When a soup entry is created, its class should be set to this type. The `makeNameRef` routine should respect this slot.
`columns`	An array of column specifications; for details, see the section "Specifying Columns" beginning on page 6-22. The default is a single, full-width column whose `fieldPath` is `'name`.
`singleSelect`	Optional. If this slot is present and its value is non-`nil`, only a single item at a time can be selected from the list. (Selecting additional items deselects the original.) Do not pre-load the `selected` slot with multiple selected name references and then specify `singleSelect`; the results will be unpredictable.
`soupToQuery`	A string specifying the union soup to query or a function returning a soup. All data displayed is retrieved from this soup.
`querySpec`	Passed to the query routine. The `tagSpec` slot is replaced internally, and the `validTest` may be enhanced internally to allow the checkbox filtering and folder support. By default all "Names" entries are displayed.
`validationFrame`	A validation frame acceptable to the `ValidityCheck` system global function. Used by the default `ValidityCheck` method. The default value is `nil`.

MakeCanonicalNameRef

dataDef`:MakeCanonicalNameRef(`*object*`,` *dataClass*`)`

Creates and returns a name reference with no application-specific slots. This method should not be overridden, but can be called if needed.

object	An entry, an alias to an entry, a name reference, a frame, or `nil`.
dataClass	Optional. The class of the entry.

MakeNameRef

dataDef:`MakeNameRef(`*object*`,` *dataClass*`)`

Creates a name reference with one additional slot `name` (by calling `MakeCanonicalNameRef`). Overrides of this method should generally call `MakeCanonicalNameRef` and fill in the slots that are needed.

If you are using `protoListPicker` to browse an array, this method should be overridden to add the slots returned by `MakeCanonicalNameRef` to the items in the array. To remove these slots, use the `PrepareToAdd` method.

object — An entry, an alias to an entry, a name reference, a frame, or `nil`.

dataClass — Optional. The class of the entry. If this is not specified, it's taken from the data definition.

Get

dataDef:`Get(`*object*`,` *fieldPath*`,` *format*`)`

Returns a value from the specified object, retrieved from the column specification.

object — An entry, an alias to an entry, a name reference, a frame, or `nil`.

fieldPath — A symbol uniquely identifying the field that should be displayed in this column. This symbol is used by the list picker to retrieve the data, and (in most cases and certainly the default case) is the actual path in the entry to the data field desired. However, it is possible to use the symbol purely as a marker—for example if the particular data required is a calculated aggregate of a number of data fields—as long as all the routines in the data definition that use this symbol are overridden to recognize this usage.

format — Determines the value returned; possible values are: `'text`, `'sortText`, or `nil`. If `nil`, the actual field is desired. If `'text`, a text representation is requested. The

value'sortText should be used only for the first column in the columns array. For example, assume the following is defined:

```
local aName := '{first: "Cindy", last: "Peters"};
```

The result of calling the default Get method

```
[:Get(aName, 'name, format)]
```

depends on the value of the format parameter:

'text	"Cindy Peters"
'sortText	"Peters, Cindy"
nil	{first: "Cindy", last: "Peters"}

If the first column specification has fieldPath = 'fruitType, the overridden Get function should support 'sortText for 'fruitType, but all other fields need only support nil and 'text.

GetPrimaryValue

dataDef:GetPrimaryValue(*object*, *format*)

Called by the default Get method to retrieve the data as described by the primaryPath and primaryPathMapper slots. Default method returns nil.

object	An entry, an alias to an entry, a name reference, a frame, or nil.
format	Determines the value returned; see the Get method (page 6-117) for details.

HitItem

dataDef:HitItem(*tapInfo*, *context*)

Called when the user taps in the picker. This method should return either a reference to a view opened as a result of the tap, or nil. If a view is opened, all tap processing by the list picker will be suppressed until the data

definition passes control back to the list picker by calling `context:Tapped(action)`, which is described below.

tapInfo A frame containing the following slots:

`nameRef`	The name reference that was tapped.
`tapIndex`	The visible index of the name reference, or `nil` for `'new`.
`bbox`	The bounding box for the cell that was tapped.
`fieldPath`	The `fieldPath` for the column tapped, or `'new` if it was the "New" button.
`editPaths`	All columns for this list.
`_reqPaths`	Used internally; do not modify.
`popup`	Used in pop-up processing.

context The view handling the tap.

MakePopup

dataDef`:MakePopup(`*object*`,` *fieldPath*`)`

Returns `nil` or an array suitable to be passed to the `PopupMenu` function. If the value of an item in the pop-up view is different from the `item` slot the slot `value` should hold the proper value. If the item is to open the editor, the value `slot` should be the symbol `'openeditor`. This method is called by the list picker to determine when to precede a column with a diamond character. If you override the default `HitItem` method, this method should return non-`nil` to get the diamond character.

If an array is returned, it is popped up by `PopupMenu`. If `nil` is returned, the `HandleTap` method is called.

object An entry, an alias to an entry, a name reference, a frame, or `nil`.

fieldPath A symbol uniquely identifying the field that should be displayed in this column. This symbol is used by the list picker to retrieve the data, and (in most cases and certainly the default case) is the actual path in the entry to the data field desired.

Tapped

context:Tapped(*action*)

Call this method from the HitItem method to indicate that a tap has been handled.

action — A symbol indicating what action to take in response to a tap. The following values can be specified:

'select	Select the item.
'toggle	Toggle between selected and unselected state.
nil	Do nothing.

New

dataDef:New(*tapInfo*, *context*)

Called when the user taps the "New" button. This method should return either a reference to a view opened as a result of the tap, or nil. If a view is opened, all tap processing by the list picker will be suppressed until the data definition passes control back to the list picker by calling context:Tapped(action), which is described below.

If a validationFrame slot is provided, the default New method opens a label input line slip (as in the default editing for an item) allowing editing of a new entry with one child view for each column in the picker.

tapInfo — A frame containing the following slots:

nameRef	The name reference that was tapped.
tapIndex	The visible index of the name reference, or nil for 'new.
bbox	The bounding box for the cell that was tapped.
fieldPath	The field path for the column tapped, or 'new if it was the "New" button.
editPaths	All columns for this list.
_reqPaths	Used internally; do not modify.

`popup`	Used in pop-up processing.
context	The view handling the tap.

DefaultOpenEditor

dataDef:`DefaultOpenEditor(`*tapInfo*`,` *context*`,` *why*`)`

You can call this method to open an edit view, for editing an existing record or in response to a tap on the "New" button. You may also provide an `OpenEditor` method, which if present, is called instead of `DefaultOpenEditor`.

The `DefaultOpenEditor` method causes a call to either `DefaultEditDone` or `DefaultNewDone` when the edit slip is closed.

tapInfo	A frame containing the following slots:	
	`nameRef`	The name reference that was tapped.
	`tapIndex`	The visible index of the name reference, or `nil` for `'new`.
	`bbox`	The bounding box for the cell that was tapped.
	`fieldPath`	The field path for the column tapped, or `'new` if it was the "New" button.
	`editPaths`	All columns for this list.
	`_reqPaths`	Used internally; do not modify.
	`popup`	Used in pop-up processing.
context	The view handling the tap.	
why	A symbol which can be either `'edit` or `'new`.	

NewEntry

dataDef:`NewEntry(`*nameRef*`,` *label*`)`

Returns a new soup entry, filled in as much as possible from the name reference passed in, and with the tags slot set appropriately so that the entry is in the current folder. The new entry's class slot is given the value specified by the `cardType` slot in the data definition.

In addition, `NewEntry` strips out the four name reference utility slots (`_alias`, `_entryClass`, `_unselected`, and `_fakeID`) from the entry if they are present. It sets the `_alias` slot in the original name reference to point to the new entry, the `_entryClass` slot to `ClassOf(entry)`. Finally, it calls `EntryChangeXmit(entry)`.

nameRef	Holds the new soup information.
context	The view handling the tap.

Note
If the soup doesn't exist, this method fails silently. ◆

ModifyEntry

dataDef`:ModifyEntry(`*nameRef*`,` *fieldPath*`)`

Returns the modified entry. Sets the field named by *fieldPath* in the underlying soup entry for the name reference. It then calls `EntryChangeXmit` on the entry.

nameRef	The actual entry that underwent the modifications.
fieldPath	The array of the paths into the *nameRef* that changed.

protoPeopleDataDef

The `protoPeopleDataDef`, which is based on the `protoNameRefDataDef`, is the basis of the built-in data definitions used by `protoPeoplePicker` and `protoMeetingplacePicker`.

Slot descriptions

`entryType`	When a soup entry is created, its class should be set to this type. The `makeNameRef` routine should respect this slot. The default value is `'person`.
`soupToQuery`	A string specifying the union soup to query or a function returning a soup. All data displayed is retrieved from this soup. By default the "Names" soup is queried.

`primaryPath` Optional. Symbol used to specify that a specific column is the primary path. The primary path is treated specially in that the data displayed can be retrieved from multiple source slots; that is, the primary path for a card is the `name`, but for a company card the "name" data comes from the `company` slot. The mapping of where the data comes from is specified by the `primaryPathMapper`.

`primaryPathMapper`

Optional. A frame where each slot maps an entry class to the slot from which the data for the primary path should be retrieved. So, for example, the `primaryPathMapper` for the cardfile is:

```
{person: name,
owner: name,
company: company,
group: group,
worksite: place,}
```

`superSymbol` (Used exclusively to support routing.) Used as usual for `dataDefs`. However, if the `superSymbol` is `'groupTransport`, the list picker type defined by this `nameRef` will be available as one of the routing choices in the "group" card in the "Names" application. The name displayed in that application is the value of the `name` slot in the data definition.

`routePath` (Used exclusively to support routing.) Used by the `GetRoutingInfo` function to determine which `nameRef` slot contains the routing information.

`protoPeopleDataDef` uses the following methods. The methods `GetRoutingInfo`, `GetItemRoutingFrame`, `GetRoutingTitle`, and `PrepareForRouting` are used exclusively to support routing. They can be ignored if the data definition is not intended for routing; they can be overridden if necessary.

Equivalent

dataDef:`Equivalent(`*nameRef1, nameRef2, pathArray*`)`

This method compares the data in two name references and returns an array of all paths that contain nonequivalent (in terms of what is displayed) values. This method, which overrides a method in `protoNameRefDataDef`, handles name equivalence.

The default method handles strings and immediates; anything more sophisticated should be overridden here.

nameRef1 A name reference.

nameRef2 A name reference.

pathArray An array of paths.

If you are using the default editing methods with a slot containing a frame, you need to override this method as well as provide a `validationFrame` (or override the `Validate` method). The `ModifyEntry` method is not responsible for deciding if an entry should be modified; when it is called, all the paths specified in the `fieldPath` parameters have been changed and should be entered properly in the appropriate "Names" soup entry.

Validate

dataDef:`Validate(`*nameRef, pathArray*`)`

This method returns an array of invalid paths.

nameRef2 A name reference.

pathArray An array of paths.

ModifyEntryPath

dataDef:`ModifyEntryPath(`*nameRef, entry, path*`)`

This method handles the modification of currently defined "Names" soup entries. For nonprimary paths, it sets `entry.(path) := nameRef.(path)`. For the primary path (phone numbers, email addresses and so on), it sets the `sortOn` and `class` slots correctly.

In other words, you should override `ModifyEntry` as appropriate, iterate across the paths, write through those for which `entry.(path) := nameRef.(path)` isn't sufficient (other than the primary path), and call the inherited `ModifyEntryPath` for all the others.

nameRef A name reference.

entry A "Names" soup entry.

nameRef A field path.

GetRoutingInfo

dataDef:`GetRoutingInfo(`*object*`)`

This method retrieves all the routing information for an item. By default, this method just calls `GetRoutingFrame` on the item. However, if the item is a group, this method iterates across each member, as returned by `Get(`*item*, *routePath*, *nil*), and recursively calls `GetRoutingInfo` for each member.

object An entry, an alias to an entry, a name reference, a frame, or `nil`.

GetItemRoutingFrame

dataDef:`GetItemRoutingFrame(`*item*`)`

This method is required for transport name references. It is called by the `GetRoutingInfo` method to convert the specific routing information into a form acceptable by the transport.

entry The name reference of the entry from which to get the routing information.

GetRoutingTitle

dataDef:`GetRoutingTitle(`*objects*`, `*width*`, `*font*`)`

Similar to the `GetRoutingInfo` method, `GetRoutingTitle` is called by the transport code to create a string to display as the target of the transport. The string that is displayed is retrieved from the `primaryPath` slot.

objects A name reference, an array of name references, or `nil`.

width	The maximum length of the string, as specified in number of pixels.
font	The font in which the string is rendered.

PrepareForRouting

dataDef:`PrepareForRouting(`*nameRef*`,` *fieldPath*`,` *format*`)`

This method is called to strip any information that is context-specific (aliases, for instance) from the specified name reference.

object	A name reference.

protoPeoplePicker

This proto implements a picker showing names from the "Names" application, along with associated phone numbers, fax numbers, and email addresses. In cases where several choices are possible, the picker allows selection using a pop-up selector. The proto also allows the user to add new entries, or additional information for existing entries.

This proto works with the data definition registry, using predefined data definitions and data views to implement the picker behavior.

Slot descriptions

`class` — A symbol specifying the type of data to display, and the data definition used to display it. You can specify the following values:

`|nameRef.people|`	Names.
`|nameRef.phone|`	Phone numbers.
`|nameRef.fax|`	Fax numbers.
`|nameRef.email|`	Email addresses.

`selected` — This slot is inherited by `protoListPicker` and contains an array of name references for selected items. These items may have been selected from the picker, or added by the user. Note that some clean-up is conducted when the `ViewQuitScript` of `protoListPicker` is called, so the `selected` array

should only be used after this executes (in other words, in a deferred send).

An array of name references may be passed into the picker when it is first opened to establish defaults for the current item selections.

All other behavior is provided by the data definition; see the description of `protoNameRefDataDef` on page 6-114 for details.

protoPeoplePopup

This proto is similar to `protoPeoplePicker`, but opens a pop-up view containing the picker (instead of having the picker embedded in the application.)

Slot descriptions

`class` — A symbol specifying the type of data to display, and the data definition used to display it. You can specify the following values:

`|nameRef.people|`	Names.
`|nameRef.phone|`	Phone numbers.
`|nameRef.fax|`	Fax numbers.
`|nameRef.email|`	Email addresses.

`selected` — This slot is inherited by `protoListPicker` and contains an array of name references for selected items. These items may have been selected from the picker, or added by the user. Note that some cleanup is conducted when the `ViewQuitScript` of `protoListPicker` is called, so the `selected` array should only be used after this executes (in other words, in a deferred send or `'postQuit` operation).

An array of name references may be passed into the picker when it is first opened to establish defaults for the current item selections.

`context` — Optional. The name of the view containing the `pickActionScript` method.

`options`	All slots in this frame are copied to the `protoListPicker` view, so anything that can be specified to `protoListPicker` can be specified in the `options` slot. You can override any slot in the pop-up view; the `suppressNew` slot, for instance.

PickActionScript

picker`:PickActionScript(`*selected*`)`

This method is called when the pop-up view is closed.

selected	The `selected` array.

All other behavior is provided by the data definition; see the description of `protoNameRefDataDef` on page 6-114 for details.

Roll Protos

These protos are used to implement roll views. A roll view is a view that consists of several discrete subviews, arranged vertically, one above the other. The roll can be viewed in overview mode, where each subview is represented by a single-line description. Any single view or all views can be expanded to full size.

protoRoll

This proto is used to create a roll-like view that includes a series of individual items (other views) that the user can see either as a collapsed list of one-line overview descriptions or as full-size views. When one of the overview lines is tapped, all of the full-size views are displayed, with the one that was tapped shown at the top of the `protoRoll` view. Each view occupies the full width of the `protoRoll` and the views are arranged one above the other.

The user can then scroll through all of the expanded views by using the universal scrollers (up and down arrows). The user can also tap the Overview button (the dot between the up and down arrows) to get back to the overview list to select another item.

In the collapsed view, the items in the overview list are preceded by bullets. Figure 6-47 shows an example of this type of view:

Figure 6-47 Example of a rolled list of items

The following `protoRoll` methods are defined internally: `ViewSetupChildrenScript`, `ViewScrollUpScript`, `ViewScrollDownScript`, `ViewOverviewScript`, `GetOverview`, and `ShowItem`. If you need to use one of these methods, be sure to call the inherited method also (for example, `inherited:?ViewSetupChildrenScript()`), otherwise the proto may not work as expected.

The `protoRoll` is based on a view of the `clView` class. It has no predefined child views, though they are dynamically created at run time from the view templates you place in the `items` slot.

Slot descriptions

`viewFlags`	The default setting is `vApplication + vClipping`.
`viewBounds`	By default, the bounds are set to the entire screen, beginning 16 pixel lines down from the top. This would leave room for a title at the top if the `protoRoll` was placed inside of a `protoApp`.
`items`	An array of templates that correspond to the items in the list. Each of these should use `protoRollItem` as its proto. For details, see the section “protoRollItem” beginning on page 6-134. Because this slot cannot usually

	be set until run time, you should set it in the `ViewSetupFormScript` method.
`allCollapsed`	Optional. If this slot is set to a non-`nil` value, the roll is initially displayed in a collapsed state; that is, only the list of one-line overviews is displayed. If this slot is `nil`, the roll is initially displayed in an expanded state. The default is `nil`.
`index`	This slot is used only when `allCollapsed` is set to `nil`; that is, when the roll is initially displayed in an expanded state. Items from the `items` array are displayed in the roll beginning with the item at this index.
`declareSelf`	Must be set to `'roll`. This identifies the view that should receive scroll and overview events. This view must also be immediately enclosed by a parent view that has the `vApplication` view flag set, in order for scrolling and overview handling to operate properly.

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

```
myRoll := {...
_proto: protoRoll,
declareSelf: 'roll,
allCollapsed: true,
index: 0,
items: [
{_proto:protoRollItem,
height:50,
overview:"Overview of item 1",
viewBounds:{left:0,top:0,right:0,bottom:50},
stepChildren:[{_proto:protoStaticText,
            text:"This is the first test roll item",
            viewJustify: vjParentFullH + vjParentFullV,
            viewBounds:{left:0,top:0,right:0,bottom:0},
            viewfont:ROM_fontSystem12 }],
},
{_proto:protoRollItem,
```

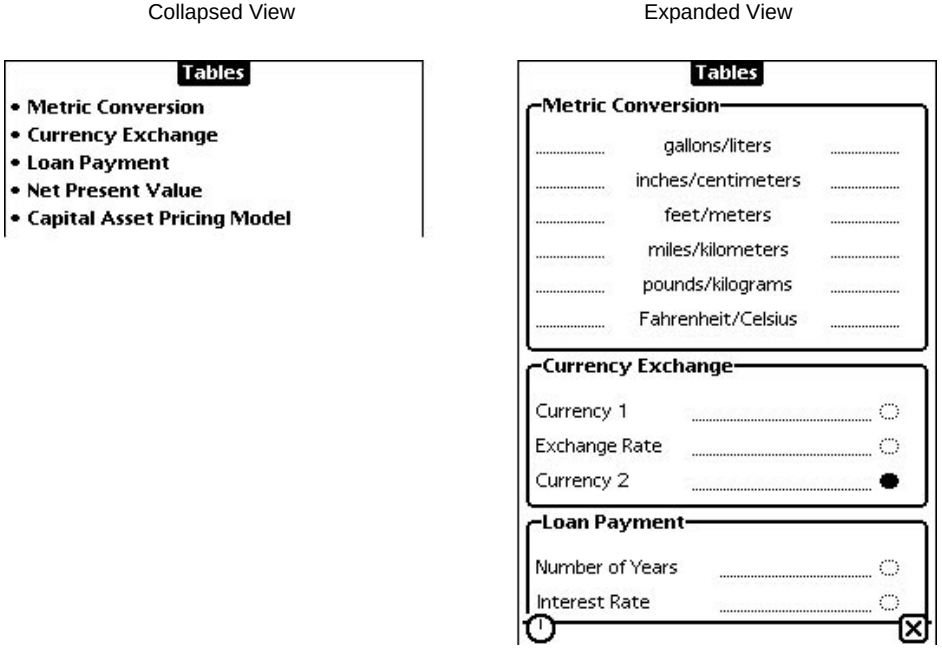
```
height:200,
overview:"Overview of item 2",
viewBounds:{left:0,top:0,right:0,bottom:200},
stepChildren:[{_proto:protoStaticText,
            text:"This is the second test roll item",
            viewBounds:{left:0,top:0,right:0,bottom:0}
}],
},
{_proto:protoRollItem,
height:200,
overview:"Overview of item 3",
viewBounds:{left:0,top:0,right:0,bottom:200},
stepChildren:[{_proto:protoStaticText,
            text:"This is the third test roll item",
            viewBounds:{left:0,top:0,right:0,bottom:0}
}],
},
{_proto:protoRollItem,
height:50,
overview:"Overview of item 4",
viewBounds:{left:0,top:0,right:0,bottom:50},
stepChildren:[{_proto:protoStaticText,
            text:"This is the fourth test roll item",
            viewBounds:{left:0,top:0,right:0,bottom:0}
}],
}],
...};
```

protoRollBrowser

This proto is similar to `protoRoll`, except that the `protoRollBrowser` is an entirely self-contained application. It is based on the `protoApp` proto, so it has a title and a status bar. Also, it need not be contained in another view.

The `protoRollBrowser` works exactly like the `protoRoll` in other respects. Figure 6-48 shows an example of a `protoRollBrowser` view in its collapsed and expanded states:

Figure 6-48 Example of a collapsed and expanded rolled list of items



The `protoRollBrowser` uses the `protoApp` proto. The `protoRollBrowser` has the following three child views:

- A roll view, based on the `protoRoll` proto. This view occupies most of the parent view, except for the title and status bar areas.
- A title, based on the `protoTitle` proto.
- A status bar, based on the `protoStatus` proto.

Slot descriptions

`viewBounds`	Set to the size and location where you want the roll to appear. By default it is centered horizontally within its parent view.
`viewJustify`	Optional. The default setting is `vjParentCenterH`.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(1) + vfInset(1) + vfShadow(1)`.
`title`	A string that is the title. This title appears in a title bar at the top of the roll. (It uses `protoTitle` to create the title.)
`rollItems`	An array of templates that correspond to the items in the list. Each of these should use `protoRollItem` as its proto. For details, see the section "protoRollItem" beginning on page 6-134. Because this slot cannot usually be set until run time, you should set it in the `ViewSetupFormScript` method.
`rollCollapsed`	Optional. If this slot is set to a non-`nil` value, the roll is initially displayed in a collapsed state; that is, only the list of one-line overviews is displayed. If this slot is `nil`, the roll is initially displayed in an expanded state. The default is non-`nil`.
`rollIndex`	This slot is used only when `rollCollapsed` is set to `nil`; that is, when the roll is initially displayed in an expanded state. Items from the `items` array are displayed in the roll beginning with the item at this index.
`declareSelf`	Do not change. This slot is set by default to `'base`. This identifies the view to be closed when the user taps the close box.

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

```
myRollBrowser := {...
_proto: protoRollBrowser,
title: "My RollBrowser",
rollCollapsed: true,
```

```
declareSelf: 'base,
rollitems: [
{_proto:protoRollItem,
height:50,
overview:"Overview of item 1",
viewBounds:{left:0,top:0,right:0,bottom:50},
stepChildren:[{_proto:protoStaticText,
            text:"This is the first test roll item",
            viewJustify: vjParentFullH + vjParentFullV,
            viewBounds:{left:0,top:0,right:0,bottom:0},
            viewfont:ROM_fontSystem12 }],
},  // ... and so on
],
...};
```

protoRollItem

This proto is used for one of the views in a roll (based on `protoRoll` or `protoRollBrowser`). You should specify an array containing one or more views based on `protoRollItem`. Each item in the array represents one of the views in the roll.

The `protoRollItem` is based on a view of the class `clView`.

Note that the `protoRollItem` proto is not used by picking it from the view palette in NTK. You use this proto by writing a textual description of your template, referring to this proto in the `_proto` slot of your template frame.

You write one template frame for each item to be shown in the roll, and place them all in an array in the `items` slot of the roll. See `protoRoll` (page 6-128) for an example.

Slot descriptions

`viewBounds` Typically, you set the bounds to 0, 0, 0, `height`. The first three bounds parameters are not needed because the view is positioned below the previous child and fully horizontally justified within the roll. However,

	you can specify values other than zero to indent the view from the sides of its parent or to separate it from its preceding sibling, but keep in mind how the `viewJustify` setting affects the interpretation of the `viewBounds` values. For more information on the `viewJustify` slot, see the section "View Alignment" beginning on page 3-18.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(1)`.
`viewJustify`	Optional. The default setting is `vjSiblingBottomV + vjParentFullH`.
`overview`	A string that is the one-line overview to be displayed for this view when the roll is collapsed and only the overview list is shown.
`height`	Set to the height of the view, in pixels.
`stepChildren`	An array containing one or more child views that belong to the view that is this particular roll item. These are shown when this item is expanded (tapped by the user, or scrolled to after the roll has already been expanded). Typically, each child view uses a proto and can include whatever slots are important for use with its particular proto.

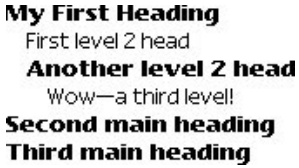
View Classes

The following view class is used to display an expandable text outline.

Outline View (clOutline)

The `clOutline` view class is used to display an expandable text outline. Figure 6-49 shows an example:

Figure 6-49 Example of an expandable text outline



The `clOutline` view class includes these features:

- Multilevel outline (up to 15 levels), with each outline level indented from the previous one.
- Headings that can be expanded (those that contain subheadings) are shown in bold automatically.
- Headings that the user can expand to show its subheadings by tapping on the heading. Another tap on the heading collapses it, hiding its subheadings.
- Only one main heading can be expanded at a time. If the user taps a different heading, any other expanded heading is automatically collapsed, and the new heading is expanded.

Slot descriptions

`viewBounds` Set to the size and location where you want the view to appear.

`browsers` An array containing one frame item, `List`, which is itself an array of the items to be shown in the outline. Each outline item is a frame containing these slots:

- `level` The outline level of this item. “1” specifies a top-level heading, “2” specifies a second-level heading, and so on. This slot can be omitted for top-level items; it defaults to level 1. You can use up to 15 levels.

	`name`	A string that is the text to be shown in the outline. Tabs are not allowed in the text.
`viewFont`		Specify the font to be used for the text in the outline. It's best not to specify a bold font since bold is added automatically for headings that have subheadings. If you specify bold, all the text will be bold. The default font is `ROM_fontSystem10`.
`viewFlags`		The default setting is `vVisible+vClickable+vReadOnly`.
`viewFormat`		Optional. The default setting is `nil`.
`clickSound`		Optional. Specify a sound frame. This sound is played when the user taps any item in the outline.

OutlineClickScript

outline`:OutlineClickScript(`*index*`, `*unused*`)`

This method is called whenever the user taps an item in the outline. This function must return non-`nil`.

index	The index of the outline item in the `List` array (inside the `browsers` slot).
unused	Unused.

Here is an example of a view definition of the `clOutline` class:

```
myOutline := {...
   viewclass: clOutline,
   viewFlags: vVisible+vClickable+vReadOnly,
   viewBounds: {left: 25, top: 56, right: 220,
               bottom: 232},
   viewFont: ROM_fontsystem12,
   clickSound: ROM_flip,
   browsers: [{list: [
      {level:1, name:"My First Heading"},
      {level:2, name:"First level 2 head"},
```

```
      {level:2, name:"Another level 2 head"},
      {level:3, name:"Wow—a third level!"},
      {level:1, name:"Second main heading"},
      {level:2, name:"Section 2 subhead 1"},
      {level:2, name:"Section 2 subhead1"},
      {level:1, name:"Third main heading"},
      {level:2, name:"Last subhead"},
      ] }],
   OutlineClickScript: func(index, dummy)
      begin
      Print("You picked browser item " & index);
      true;
      end,
...};
```

Pop-up Functions and Methods

The following functions and methods are used in creating pop-up views.

PopupMenu

`PopupMenu(`*list*`, `*options*`)`

Creates a dynamic pop-up list view, or picker, from which one item can be selected. Items can include strings with icons, and that include two-dimensional grids in which individual cells can be selected.

`PopupMenu` returns the picker view that it creates.

list — An array of items that you want to appear in the picker list. The elements in the array will be shown with the first item at the top of the list, continuing down to the last item. If the list contains more items than can be shown on the screen at one time, the user can scroll it to see more items. For more details on the items you can

specify in the picker list, see the section “Specifying the PickItems Array” beginning on page 6-5.

options There are a number of possible options. For button pop-up views, specify `nil`; the pop-up view is placed adjoining the view to which the `PopupMenu` message is sent (not obscuring the button).

You can also specify a frame with two slots—`left` and `top` —which define a rectangle that is inset within the bounds of the view from within which `PopupMenu` is called (the parent). The left and right edges of the new rectangle are inset `left` pixels from the sides of the parent, and the top and bottom edges of the rectangle are inset `top` pixels from the top and bottom edges of the parent.

You can also provide a `'bounds` slot that is a bounds frame that specifies, in local coordinates, the rectangle next to which the pop-up view should appear.

If you list items that include icons, you should be aware that `PopupMenu` scales the items to the maximum of the icon height and text height. You can force this to a desired value (for all items, except separators) by adding this option slot to the *first* item:

```
PopupMenu([{item: "first one", fixedheight: 22}...)
```

If you use icons in a list that can become large enough to scroll, you should specify the `fixedHeight` slot for *every* item:

Finally, if you find the indentation and placement of your icons and text are ragged, you can provide an indent slot for the first item which forces every item to be indented correctly; for example:

```
PopupMenu([{item: "first one", indent: 28}...)
```

When an item in a picker is selected, the system sends the `PickActionScript` message to the view identified by `self` (the view from which `PopupMenu` was called). You must define `PickActionScript` as a method that accepts one parameter. The parameter passed to `PickActionScript` is the array index of the item number selected in the list (the first item has an index of zero).

If no item is selected—that is, if the user taps outside the picker to close it—the `PickCancelledScript` message is sent to the view identified by `self`. If you want to handle this message, define a method that accepts no parameters, since none are passed.

`SetItemMark` and `GetItemMark` are two methods provided for picker views. You can use them within the `PickActionScript` method (or elsewhere) to set and get the mark for an item. You call these methods as follows:

popupView`:SetItemMark()`
popupView`:GetItemMark()`

where *popupView* is the view returned by the `PopupMenu` function. For details on these methods, see the description of the `protoPicker` proto (page 6-36).

The picker view created by `PopupMenu` is automatically closed after the user selects an item or taps outside the view.

Name Reference Functions

The following global routines are provided for working with name references.

IsNameRef

`IsNameRef(`*item*`)`

This function returns non-`nil` if the specified item is a name reference (as determined by the presence of an `_alias` slot).

AliasFromObj

`AliasFromObj(`*item*`)`

This function returns an alias to an entry if possible. If the item is an alias, it is simply returned. If the item is a soup entry, an alias to it is created and returned. If the entry is a name reference, the contents of the `_alias` slot are returned. In all other cases, `nil` is returned.

EntryFromObj

`EntryFromObj(`*item*`)`

This function returns an entry if possible. Basically, it looks for an alias and then tries to resolve the entry from it.

ObjEntryClass

`ObjEntryClass(`*item*`)`

This function returns the class of the entry returned by the `EntryFromOb` function.

Summary

Protos

General Pickers

protoPopupButton

```
aProtoPopupButton := {
_proto : protoPopupButton,
viewFlags : flags,
viewBounds : boundsFrame,
```

```
viewJustify : justificationFlags,
text : string, // text inside button
popup : array, // items in list
ButtonClickScript : function, // called when button tapped
PickActionScript : function, // returns item selected
PickCancelledScript : function, // user cancelled
...
}
```

protoPopInPlace

```
aProtoPopInPlace := {
_proto : protoPopInPlace,
viewBounds : boundsFrame,
viewFlags : constant,
viewJustify : justificationFlags,
text : string, // text inside button
popup : array, // items in list
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled
...
}
```

protoLabelPicker

```
aProtoLabelPicker := {
_proto : protoLabelPicker,
viewBounds : boundsFrame,
viewFont : constant,
labelCommands : array, // items in list
iconBounds : boundsFrame, // bounds of largest icon
iconIndent : integer, // indent of text from icon
checkCurrentItem : Boolean, // true to check selected item
indent : integer, // indent of picker from label
```

```
textIndent : integer, // indent of text from left of view
entryline.viewFont : constant, // font for label text field
LabelActionScript : function, // returns selected item
TextSetup : function, // gets initial item for top of list
TextChanged : function, // called when value of item
                        changes
UpdateText : function, // call to change selected item
PickerSetup : function, // called when user taps label
Popit : function, // call to programmatically pop up picker
...
}
```

protoPicker

```
aProtoPicker := {
_proto : protoPicker,
bounds : boundsFrame,
viewBounds : boundsFrame, // ignored
viewFlags : constant,
viewFormat : constant,
viewJustify : justificationFlags,
viewFont : constant,
viewEffect : constant,
pickItems : array, // items in list
pickTextItemHeight : integer, // height reserved for items
pickLeftMargin : integer, // margin from left of view
pickRightMargin : integer, // margin from right of view
pickTopMargin : integer, // margin above each item in list
pickAutoClose : Boolean, // true to close list after pick
pickItemsMarkable : Boolean, // true to reserve space for
                             // check mark before item
pickMarkWidth : integer, // space to reserve for marks
callbackContext : viewTemplate, // view with pick scripts
PickActionScript : function, // returns selected item
```

```
PickCancelledScript : function, // user cancelled
SetItemMark : function, // set character for check marks
GetItemMark : function, // set character for check marks
...
}
```

protoGeneralPopup

```
aProtoGeneralPopup := {
_proto : protoGeneralPopup,
viewBounds : boundsFrame,
viewFlags : constant,
cancelled : Boolean, // true if user cancelled pop-up view
context : viewTemplate, // view with pick scripts
New : //open pop-up view
PickActionScript : function, //user taps pop-up view
PickCancelledScript : function, //called in pop-up view
                                cancelled
ViewClickScript : function, //user taps pop-up box
ViewDrawScript : function, //user taps pop-up box
ViewQuitScript : function, //user taps pop-up close box
...
}
```

protoTextList

```
aProtoTextList := {
_proto : protoTextList,
viewBounds : boundsFrame,
viewFont : constant,
viewFormat : constant,
viewLines : integer, // number of lines to show in list
selection : integer, // index of item to show as selected
selectedItems : arrary, // items in list
```

```
listItems : array, // strings or shapes in list
lineHeight : array, // height of lines in list
isShapeList : Boolean, // true if picts instead of text
useMultipleSelections : Boolean, // true for multiple
                        select
useScroller : Boolean, // true to include scrollers
scrollAmounts : array, // units to scroll
DoScrollScript : function, // scrolls list by offset
ViewSetupFormScript : function, // set up list
ButtonClickScript : function, // returns selected item
...
}
```

protoTable

```
aProtoTable := {
_proto : protoTable,
viewBounds : boundsFrame,
viewFormat : constant,
def : frame, // table definition frame - protoTableDef
scrollAmount : integer, // number of rows to scroll
viewFormat :
currentSelection : string, // text of selected item
selectedCells : array, // indexes of selected cells
declareSelf : symbol, // 'tabbase; do not change
ViewSetupFormScript : function, // set up table
SelectThisCell : function, // called when cell is selected
...
}
```

protoTableDef

```
aProtoTableDef := {
_proto: protoTableDef,
```

```
tabAcross : integer, // number of columns - must be 1
tabDown : integer, // number of rows in table
tabWidths : integer, // width of table
tabHeight : integer, // height of rows
tabProtos : frame, // reference(s) to template(s) for rows
tabValues : integer/array, // value/array of values for rows
tabValueSlot : symbol, // slot where tabValues are stored
tabUniqueSelection : Boolean, // true for single selection
indentX : integer, // used internally - do not change
TabSetUp : function, // called before each row is set up
...
}
```

protoTableEntry

```
aProtoTableEntry := {
_proto: protoTableEntry,
viewClass : clTextView,
viewFlags : flags,
viewJustify : justificationFlags,
viewTransferMode : modeOr,
text : string, // text inside table
ViewClickScript : function, // sets current selection
ViewHiliteScript : function, // highlights selection
...
}
```

Map Pickers

protoCountryPicker

```
aProtoCountryPicker := {
_proto : protoCountryPicker,
```

```
viewBounds : boundsFrame,
autoClose : Boolean, // true to close picker upon selection
listLimit : function, //maximum items listed
PickWorld : function, // called when selection is made
...
}
```

protoProvincePicker

```
aProtoProvincePicker := {
_proto : protoProvincePicker,
viewFlags : constant,
autoClose : Boolean, // true to close picker upon selection
listLimit : function, //maximum items listed
PickWorld : function, // called when selection is made
...
}
```

protoStatePicker

```
aProtoStatePicker := {
_proto : protoStatePicker,
viewFlags : constant,
autoClose : Boolean, // true to close picker upon selection
PickWorld : function, // called when selection is made
...
}
```

protoWorldPicker

```
aProtoWorldPicker := {
_proto : protoWorldPicker,
viewBounds : boundsFrame,
autoClose : Boolean, // true to close picker upon selection
listLimit : function, //maximum items listed
```

```
PickWorld : function, // called when selection is made
...
}
```

Text Pickers

protoTextPicker

```
aProtoTextPicker := {
_proto : protoTextPicker,
label : string, // picker label
indent : integer, // indent
labelFont : constant, // font for label
entryFont : constant, // font for picker line
Popit : function, // user tapped picker
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
TextSetup : function, // returns text string
...
}
```

protoDateTextPicker

```
aProtoDateTextPicker := {
_proto : protoDateTextPicker,
label : string, // picker label
date : integer, // initial and currently selected date
longFormat : symbol, // format to display date
shortFormat : symbol, // format to display date
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoDateDurationTextPicker

```
aProtoDateDurationTextPicker := {
_proto : protoDateDurationTextPicker,
label : string, // picker label
labelFont : constant, // display font
entryFont : constant, // picked entry font
startTime : integer, // initial start date
stopTime : integer, // initial end date
longFormat : symbol, // format to display date
shortFormat : symbol, // format to display date
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoRepeatDateDurationTextPicker

```
aProtoRepeatDateDurationTextPicker := {
_proto : protoRepeatDateDurationTextPicker,
label : string, // picker label
startTime : integer, // initial start date
stopTime : integer, // initial end date
longFormat : symbol, // format to display date
shortFormat : symbol, // format to display date
repeatType : constant, //how often meeting meets
mtgInfo : constant, // repeating meetings
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
```

```
}
```

protoDateNTimeTextPicker

```
aProtoDateNTimeTextPicker := {
_proto : protoDateNTimeTextPicker,
label : string, // picker label
date : integer, // initial date/time
format : symbol, // format to display time
longFormat : symbol, // format to display date
shortFormat : symbol, // format to display date
increment : integer //amount to change time
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoTimeTextPicker

```
aProtoTimeTextPicker := {
_proto : protoTimeTextPicker,
label : string, // picker label
labelFont : constant, // label display font
entryFont : constant, // picked entry font
indent : integer, //amount to indent text
time : integer, // initial start time
format : symbol, // format to display time
increment : integer, // increment to change time for taps
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoDurationTextPicker

```
aProtoDurationTextPicker := {
_proto : protoDurationTextPicker,
label : string, // picker label
startTime : integer, // initial start time
stopTime : integer, // initial end time
format : symbol, // format to display time
increment : integer, // increment to change time for taps
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoTimeDeltaTextPicker

```
aProtoTimeDeltaTextPicker := {
_proto : protoTimeDeltaTextPicker,
label : string, // picker label
time : integer, // initial time
labelFont : constant, // label display font
entryFont : constant, // picked entry font
indent : integer, //amount to indent text
increment : integer, // increment to change time for taps
minValue : integer, // minimum delta value
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoMapTextPicker

```
aProtoMapTextPicker := {
_proto : protoMapTextPicker,
```

```
label : string, // picker label
labelFont : constant, // label display font
entryFont : constant, // picked entry font
indent : integer, //amount to indent text
params : frame, // name of soup containing map data
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoCountryTextPicker

```
aProtoCountryTextPicker := {
_proto : protoCountryTextPicker,
label : string, // picker label
labelFont : constant, // label display font
entryFont : constant, // picked entry font
indent : integer, //amount to indent text
params : frame, // name of soup containing map data
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoUSstatesTextPicker

```
aProtoUSstatesTextPicker := {
_proto : protoUSstatesTextPicker,
label : string, // picker label
labelFont : constant, // label display font
entryFont : constant, // picked entry font
indent : integer, //amount to indent text
params : frame, // name of soup containing map data
PickActionScript : function, // returns selected item
```

```
PickCancelledScript : function, // user cancelled picker
...
}
```

protoCitiesTextPicker

```
aProtoCitiesTextPicker := {
_proto : protoCitiesTextPicker,
label : string, // picker label
labelFont : constant, // label display font
entryFont : constant, // picked entry font
indent : integer, //amount to indent text
params : frame, // name of soup containing map data
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoLongLatTextPicker

```
aProtoLongLatTextPicker := {
_proto : protoLongLatTextPicker,
label : string, // picker label
latitude : integer, // initial latitude
longitude : integer, // initial longitude
labelFont : constant, // label display font
entryFont : constant, // picked entry font
indent : integer, //amount to indent text
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

Date, Time, and Location Pop-up Views

protoDatePopup

```
New: function, // creates pop-up view
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoDateNTimePopup

```
New : function, // creates pop-up view
NewTime : function, // called when time changes
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoDateIntervalPopup

```
New : function, // creates pop-up view
NewTime : function, // called when time changes
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoMultiDatePopup

```
New : function, // creates pop-up view
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoYearPopup

```
New : function, // creates pop-up view
NewYear : function, //called when user changes selection
DoneYear : function, //called when user taps close box
PickCancelledScript : function, // user cancelled picker
...
}
```

protoTimePopup

```
New : function, // creates pop-up view
NewTime : function, // called when time changes
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoAnalogTimePopup

```
New : function, // creates pop-up view
NewTime : function, // called when time changes
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoTimeDeltaPopup

```
New : function, // creates pop-up view
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoTimeIntervalPopup

```
New : function, // creates pop-up view
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

protoLocationPopup

```
New : function, // creates pop-up view
PickActionScript : function, // returns selected item
PickCancelledScript : function, // user cancelled picker
...
}
```

Number Pickers

protoNumberPicker

```
aProtoNumberPicker := {
_proto : protoNumberPicker,
minValue : integer, // minimum value in list
maxValue : integer, // maximum value in list
value : integer, // initial or currently selected value
showLeadingZeros : Boolean, // true to show leading zeros
prepareForClick : function, //called after click is
processed
ClickDone : function, //called after click is processed.
...
}
```

protoDigit

```
aProtoDigit := {
_proto : protoDigit,
viewFlags  : constant,
viewBounds  : boundsFrame,
digits : bitmap, // sequence of graphics to display
totalFrames : integer, // number of graphics in digits
increment : integer, // # of frames to advance or back-up
firstTapDelay : integer, // # of ticks to delay on tap
repeatingTapDelay : integer, // # of ticks before repeat
digitFrame : frame, // rect for individual frames
nextDigit : symbol, // next digit in sequence
copyMode : symbol, // transfer mode for drawing
value : integer, // index of currently displayed graphic
PrepareForClick : function, // called upon tap
ClickDone : function, // called after tap is processed
Advance : function, // call to change graphic
GetIndex : function, // returns current digit based on
value
...
}
```

Picture Picker

protoPictIndexer

```
aProtoPictIndexer := {
_proto : protoPictIndexer,
viewBounds  : boundsFrame,
viewJustify : justificationFlags,
viewFormat  : formatFlags,
icon : bitmap, // bitmap with objects arranged vertically
```

```
iconBBox : boundsFrame, // bounds of bitmap within view
numIndices : integer, // # of objects in bitmap
curIndex : integer, // index of currently selected item
IndexClickScript : function, // user taps bitmap
...
}
```

Overview Protos

protoOverview

```
aProtoOverview := {
_proto : protoOverview,
viewBounds  : boundsFrame,
viewFlags  : constant,
viewFont  : fontFlags,
lineHeight : integer, // height of items in pixels
selectIndent : integer, // specifies left margin
selected : array, //modified as user selects and
               deselects item
nothingCheckable : Boolean, // true for no checkboxes
SelectItem : function, // to record selected items
SetupAbstracts : function, // set up entry
Abstract : function, // return shape given entry
HitItem : function, // called when item is tapped
CheckState : function, // determines if selectable
Scroller : function, // implement scrolling here
...
}
```

protoSoupOverview

```
aProtoSoupOverview := {
_proto : protoSoupOverview,
```

```
cursor : cursor, // cursor describing the entries
Scroller : function, // implement scrolling here
SelectItem : function, // records selected items
Abstract : function, // return shape given entry
IsSelected : function, // returns true if selected
ForEachSelected : function, // called for each selected
item
...
}
```

protoListPicker

```
aProtoListPicker := {
_proto : protoListPicker,
viewFlags  : constant,
viewBounds  : boundsFrame,
lineHeight : integer, // height of items in pixels
listFormat : formatFlags,
pickerDef : frame, // defines list behavior
selected : array, // references to selected items
soupToQuery : string, // union soup to query
querySpec : frame, // query to use
singleSelect : Boolean, // single selection if non-nil
suppressNew : Boolean, // suppress New button if non-nil
suppressScrollers : Boolean, // suppress scroller if
non-nil
suppressAZTabs : Boolean, // suppress tabs if non-nil
suppressFolderTabs : Boolean, // suppress if non-nil
suppressSelOnlyCheckbox : Boolean, // suppress if non-nil
suppressCloseBox : Boolean, // suppress if non-nil
suppressCounter : Boolean, // suppress if non-nil
reviewSelections : Boolean, // Selected Only if non-nil
readOnly : Boolean, // items are read-only if non-nil
dontPurge : Boolean, // keep unselected refs if non-nil
```

```
soupChangeSymbol : symbol, // for RegSoupChange method
SoupEnters : function, // syncs up changed soup
SoupLeaves : function, // syncs up changed soup
ChangePickerDef : function, // specify new pickerDef
SetNowShowing : function, // set Selected Only
AddFakeItem : function, // add item to array; update screen
GetSelected : function, // returns clone of selected array
...
}
```

protoNameRefDataDef

```
aProtoNameRefDataDef := {
_proto : protoNameRefDataDef,
name : string, // name to identify picker in top left
corner
class : symbol, // specify class for new name references
entryType : symbol, // class for new soup entries
columns : array, // column specifications
singleSelect : Boolean, // single selection if non-nil
soupToQuery : string, // union soup to query
querySpec : frame, // query to use
validationFrame : frame, //checks validity of entry
MakeCanonicalNameRef : function, // make blank name ref
MakeNameRef : function, // make name reference
Get : function, // returns data from specified object
GetPrimaryValue : function, // retrieves data from object
HitItem : function, // called when item tapped
MakePopup : function, // called before making pop-up view
Tapped : function, // called when tap has been handled
New : function, // called when tap on New button
DefaultOpenEditor : function, // open an edit view
NewEntry : function, // returns a new soup entry
ModifyEntry : function, // returns a modified soup entry
```

```
...
}
```

protoPeopleDataDef

```
aProtoPeopleDataDef := {
_proto : protoPeopleDataDef,
entryType : symbol, // class for new soup entries
soupToQuery : string, // union soup to query
primaryPath : symbol, // specify column as primary path
primaryPathMapper : frame, // maps entry class to data
Equivalent : function, // compares data in two name refs
Validate : function, // returns array of invalid refs
ModifyEntryPath : function, // entry modification of Names
GetRoutingInfo : function, // retrieves routing info
GetItemRoutingFrame : function, // converts routing info
GetRoutingTitle : function, // creates target string
PrepareForRouting : function, // strips extra info
...
}
```

protoPeoplePicker

```
aProtoPeoplePicker := {
_proto : protoPeoplePicker,
class : symbol, // type of data to display
selected : array, // references to selected items
...
}
```

protoPeoplePopup

```
aProtoPeoplePicker := {
_proto : protoPeoplePicker,
class : symbol, // type of data to display
```

```
selected : array, // references to selected items
context : symbol, // view with PickActionScript method
options : array, // options for protoListPicker
PickActionScript : function, // called when pop-up view is
closed
...
}
```

Roll Protos

protoRoll

```
aProtoRoll := {
_proto : protoRoll,
viewFlags : constant,
viewBounds : boundsFrame,
items : array, // templates for roll items
allCollapsed : Boolean, // roll collapsed if non-nil
index : integer, // index of item to start display at
declareSelf : symbol, // 'roll — do not change
...
}
```

protoRollBrowser

```
aProtoRollBrowser := {
_proto : protoRollBrowser,
viewBounds : boundsFrame,
viewJustify : justificationFlags,
viewFormat : formatFlags,
title : string, // text for title at top of roll
rollItems : array, // templates for roll items
rollCollapsed : Boolean, // roll collapsed if non-nil
rollIndex : integer, // index of item to start display at
```

```
declareSelf : symbol, // 'base – do not change
...
}
```

protoRollItem

```
aProtoRollItem := {
_proto : protoRollItem,
viewBounds : boundsFrame,
viewJustify : justificationFlags,
viewFormat : formatFlags,
overview : string, // text for one-line overview
height : array, // height of the view in pixels
stepChildren : Boolean, // child views for this roll item
...
}
```

Outline View

clOutlineView

```
myOutline:= {...
viewClass : clOutline,
viewBounds : boundsFrame,
browsers: array, // frame with array of outline items
viewFont : constant,
viewFlags  : constant,
viewFormat : formatFlags,
clickSound : frame, // sound frame for taps
OutlineClickScript : function, //called when user taps item
```

Functions

`PopupMenu(`*list*`, `*options*`)`
`IsNameRef(`*item*`)`
`AliasFromObj(`*item*`)`
`EntryFromObj(`*item*`)`
`ObjEntryClass(`*item*`)`

CHAPTER 7

Controls and Other Protos

This chapter describes how to implement a wide variety of controls and other protos, including:

- horizontal and vertical scrollers
- boxes and buttons
- alphabetical tabs
- status indicators
- date and time protos
- various other useful protos

Controls Compatibility

The following new functionality has been added for the 2.0 release of Newton System Software.

Four new scroller protos have been added: `protoHorizontal2DScroller`, `protoLeftRightScroller`, `protoUpDownScroller`, and `protoHorizontalUpDownScroller`. See page 7-5 to page 7-8 for details.

A `protoInfoButton` has been added to the button and boxes protos. See page 7-13 for details.

Two tab protos have been added: `protoAZTabs` and `protoAZVertTabs`. See page 7-30 to page 7-31 for details.

Four new date and time protos have been added: `protoDatePicker`, `protoDigitalClock`, `protoNewSetClock`, `protoAMPMCluster`. See page 7-40 to page 7-47 for details.

Two miscellaneous protos have been added: `protoDragger`, `protoDragNGo`. See page 7-53 to page 7-55 for details.

Scrollers

This section describes the protos that implement horizontal and vertical scrollers within a view. All of the scroller protos are implemented in the same way; that is, they use the same methods and slots. These scrollers are not linked or related to the scroll arrows on the built-in button bar. The protos include the following:

- `protoHorizontal2DScroller` implements both horizontal and vertical scroll arrows, centered at the bottom of a view
- `protoLeftRightScroller` implements horizontal scroll arrows, centered at the bottom edge of a view
- `protoUpDownScroller` implements vertical scroll arrows, centered at the right of a view

- `protoHorizontalUpDownScroller` implements vertical scroll arrows, centered at the bottom of a view

How Scrollers Work

This section describes how scrollers work and the slots that controls particular scrolling behavior. Specifically it covers:

- How to implement a basic scroller
- How to advance the scrollable area
- How to advance the scrollers
- Advanced scrolling techniques

A Simple Scroller

Creating a simple scroller requires including only the `ViewScroll2DScript` method in the scroller template. Keep in mind that the scroller protos do not perform the actual scrolling; they simply display and maintain the arrows as the user taps them. The `ViewScroll2DScript` message is sent when the user taps a scroll arrow, giving the direction in which the user wants to scroll and the frame containing the number of units to scroll. Definition of these units is up to you; they can be lines, cells, pixels, and so on.

Automatic Feedback

In order to display arrows in scrollers properly, you must specify three rectangle slots: `scrollRect`, `viewRect`, and `dataRect`. The scrolling proto automatically takes care of highlighting the arrows at the appropriate time.

The `scrollRect` slot specifies the amount of space, the “scrollable area,” that a user can scroll over. For example, if you are working with lines of text as the units to scroll, a `scrollRect` value of `SetBounds(0,0,0,20)` would specify a scrollable area of 20 lines.

The `viewRect` slot specifies how much of the scrollable area is shown. For example, if you want the user to see only six lines of text at a time, you would specify the `viewRect` slot as `SetBounds(0,0,0,6)`.

The `dataRect` slot is used to properly highlight the arrows so that they indicate where you can scroll to see more information. If you are in the lower-right corner of the scrolling area, the up and left arrows will be black, while the right and down arrows will be hollow.

The `dataRect` slot is useful in cases where you have data that only shows up in only a portion of the scrollable area. For example the “Dates” application has a scrollable area of 24 hours, but it might, for example, have only a single meeting at 9:00 a.m. To show only the single meeting, the `scrollRect` would be specified as `SetBounds(0,0,0,24)` and the `dataRect` value would be `SetBounds(0,9,0,10)`.

To see how these slots are used, let’s look at a more complete example. Say you’re working with pixels as the units to scroll. In this case, you want to move the view origin each time the user taps the arrows, which will cause the view to move over a bit. You would set the following rectangles:

```
viewRect:     SetBounds(0, 0, 50, 50),
scrollRect:   SetBounds(0, 0, 200, 200),
dataRect:     SetBounds(50, 50, 60, 60),
```

This example shows two-dimensional scrolling. The area you want to scroll over is 200x200 pixels, but you can only view a 50x50 area at a time. In addition, an object of importance is located at (50, 50, 60, 60).

Advancing Scrollers

The user advances scrollers in one of three ways: you can tap, tap and hold, and double-tap on the scroller. The number of lines scrolled is specified in `scrollAmounts` slot. The default values are set to [1, 1, 1]; that is, each tap represents one unit. However, if you have a long list to scroll through, you might want scroll values of [1, 3, 20], which would scroll three lines at a time if the user holds the arrow down and 20 lines if they double-tap on the arrow.

Keep in mind that if you set `scrollAmounts` to values other than the default, you must check the value passed to it and scroll that amount.

Note
In general, the use of double-tap should be discouraged as the user may inadvertently tap twice causing a double-tap to occur. ◆

Advanced Usage

If you want more control over the arrow feedback, don't use the rectangle slots; instead, use the `SetArrow` and `GetArrow` methods. `SetArrow` sets the direction in which the lines will scroll. `GetArrow` returns the direction of the current arrow.

protoHorizontal2DScroller

This proto is used to include both left / right and up / down scrollers, centered at the bottom of a view. Note that most units are expressed in terms of scrollable items (cells, lines, and so on) rather than pixels. The following graphic shows the possible scrolling directions.

The `protoHorizontal2DScroller` automatically places itself in the center view; the `viewBounds` and `viewJustify` slots are set up for you. These slots are very precise, so you usually won't want to change them.

Slot descriptions

`scrollView`	Optional. Messages are sent to this view; the default is the template. This slot is usually set in the `ViewSetupForm` script.
`scrollRect`	Optional. Extent of scrollable area, in units to scroll (lines, pixels, and so on).
`dataRect`	Optional. Extent of data in the view.

`viewRect`	Optional. Extent of visible area.
`scrollAmounts`	Optional. An array of three numbers passed to you for scrolling: [line, page, double-click]. The default is [1, 1, 1].
`pageThreshold`	Optional. The number of lines scrolled before scrolling in pages; the default is 5.

The following slots represent the current offset from the scrollable area. For example, if you scroll to the right, `xPos` is a positive value.

`xPos`	Current horizontal coordinate in the `scrollRect`.
`yPos`	Current vertical coordinate in the `scrollRect`.

The `protoHorizontal2DScroller` scroll arrows are handled for you, provided you specify `scrollRect`, `dataRect`, and `viewRect` correctly. If you want to get and set the arrows state, though, you can use the `GetArrow` and `SetArrow` methods, described on page 7-7.

ViewScroll2DScript

scroller`:ViewScroll2DScript(`*direction*`,` *extras*`)`

Required. This method is called when the user taps the scroll arrows.

direction	A symbol indicating the direction to scroll; values can be `'left` , `'right`,`'up`, or `'down`.
extras	A frame with the following slots:

`count`	Number of calls to this method.
`amount`	Number of units.
`axis`	The axis of scrolling (`'horizontal` or `'vertical`). This is a convenience value.
`unit`	Click unit in which to scroll.

While the pen is held down, the extras frame information is reused. This lets you attach state specific slots to the extras frame, which can be referenced in subsequent calls to the method.

Note

You usually will call `RefreshViews` in `ViewScroll2DScript` to force the view to be redrawn while the user has the pen down. For applications that have complicated scrolling, you may want to also include the `ViewDoneScript` method. This method forces a redraw only when the user lets the pen up.◆

ViewScrollDoneScript

scroller`:ViewScrollDoneScript()`

This method is called after the scroll is finished.

SetArrow

scroller`:SetArrow(`*direction*`,` *state*`)`

This method is called when the user taps the scroll arrows.

direction	A symbol indicating the direction to scroll; values can be `'left`, `'right`, `'up`, or `'down`.
state	A symbol indicating the state of the scroll; values can be `'normal`, `'more`, or `'hilite`.

GetArrow

scroller`:GetArrow(`*direction*`)`

This method returns the current state of the arrow direction (in this case, `'left`, `'right`, `'up`, or `'down`.

direction	A symbol indicating the direction to scroll; values can be `'left`, `'right`, `'up`, or `'down`.

protoLeftRightScroller

`protoLeftRightScroller` is used to include left / right scrollers, centered at the bottom of a view. Here is an example:

The `protoLeftRightScroller` automatically places itself in the bottom edge of the view; the `viewBounds` and `viewJustify` slots are set up for you.

See the previous section,“protoHorizontal2DScroller,” for a description of `protoLeftRightScroller`’s slots and methods.

protoUpDownScroller

`protoUpDownScroller` is used to include up / down scrollers, centered at the right side of a view. Here is an example:

The `protoUpDownScroller` automatically places itself centered at the right of the view; the `viewBounds` and `viewJustify` slots are set up for you.

See the section,“protoHorizontal2DScroller” on page 7-5, for a description of `protoUpDownScroller`’s slots and methods.

protoHorizontalUpDownScroller

`protoHorizontalUpDownScroller` is used to include horizontal up / down scrollers, centered at the right side of a view. Here is an example:

The `protoHorizontalUpDownScroller` automatically places itself centered and the bottom of the view; the `viewBounds` and `viewJustify` slots are set up for you.

See the section,“protoHorizontal2DScroller” on page 7-5, for a description of `protoUpDownScroller`’s slots and methods.

Buttons and Boxes

This section describes the protos that implement text and picture buttons, checkboxes, and radio buttons. The protos include the following:

- `protoTextButton` creates a rounded text button with text inside of it.
- `protoPictureButton` creates a picture that is a button.
- `protoInfoButton` includes an information button in a view.
- `protoOrientation` changes the screen orientation so that data on the screen can be displayed facing different directions.
- `protoRadioCluster` groups a series of radio buttons into a cluster where only one can be “on” at a time.
- `protoRadioButton` creates a radio button child view of a radio button cluster (based on `protoRadioCluster`).
- `protoPictRadioButton` creates a child view of a radio button cluster (based on `protoRadioCluster`).
- `protoCloseBox` shown in a view as a mechanism for the user to close the view.
- `protoLargeCloseBox` creates a picture button that contains an “X” icon.
- `protoCheckBox` creates a checkbox.
- `protoRCheckBox` creates a checkbox where the text is to the left of the checkbox.

protoTextButton

This proto is used to create a rounded rectangle button with text inside it. The text is centered vertically and horizontally, as shown in the following graphic.

Note that the `ViewClickScript` method is used internally in the `protoTextButton` and should not be overridden. You should use the `ButtonClickScript` method to handle a tap event. (The `ButtonClickScript` message is sent from within the `ViewClickScript` method specifically to provide a mechanism for you to handle the event.)

Note that inking is automatically turned off when the button is tapped.

The `protoTextButton` is based on a view of the `clTextView` class. This view is a `clParagraphView` that is read-only and it supports no tabs or multistyled text.

Slot descriptions

`viewBounds` Set to the size and location where you want the button to appear.

`viewFlags` The default setting is `vVisible + vReadOnly + vClickable`.

`text` A string that is the text inside the button.

`viewFont` Optional. The default font for the text is `ROM_fontSystem9Bold`.

`viewFormat` Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(2) + vfRound(4)`.

`viewJustify` Optional. The default setting is `vjCenterH + vjCenterV + oneLineOnly`. To make a button with multiple text lines, instead of `oneLineOnly`, use the `noLineLimits` flag.

`viewTransferMode` Optional. The default transfer mode is `modeOr`.

ButtonClickScript

button:ButtonClickScript()

This method is called when the button is tapped; it is passed no parameters and the return value is ignored.

ButtonPressedScript

scroller:ButtonPressedScript()

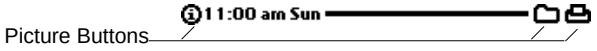
This method is called repeatedly as long as the button is pressed (the pen is held down within it). This method is passed no parameters and the return value is ignored.

The following example is a template using protoTextButton:

```
aButton := {...
_proto: protoTextButton,
viewFont: ROM_fontSystem12Bold,
text: "Hit me",
ButtonClickScript: func()
   Print("ouch!");

// a handy way to fit a button around a string
ViewSetupFormScript: func()
   viewbounds := RelBounds( 150, 60,
                  StrFontWidth( self.text, viewFont )+12,
                  FontHeight( viewFont ) );
...}
```

protoPictureButton

This proto is used to create a picture that is a button; that is, the picture can be tapped by the user to cause an action to occur. The following figure illustrates the use of a `protoPictureButton`:



Note that the `ViewClickScript` method is used internally in the `protoPictureButton` and should not be overridden. You should use the `ButtonClickScript` method to handle a tap event. (The `ButtonClickScript` message is sent from within the `ViewClickScript` method specifically to provide a mechanism for you to handle the event.)

Note that inking is automatically turned off when the button is tapped.

The `protoPictureButton` is based on a view of the `clPictureView` class.

Slot descriptions

`viewBounds`	Set to the size and location where you want the button to appear.
`viewFlags`	The default is `vVisible + vReadOnly + vClickable`.
`icon`	The bitmap to be used as the button.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(2) + vfRound(4)`. (The examples in the picture above have `viewFormat` set to zero.
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV`.

ButtonClickScript

button`:ButtonClickScript()`

This method is called when the button is tapped; it is passed no parameters and the return value is ignored.

ButtonPressedScript

scroller:ButtonPressedScript()

This method is called repeatedly as long as the button is pressed (the pen is held down within it). This method is passed no parameters and the return value is ignored.

The following example is a template using protoPictureButton:

```
pictButton := {...
_proto: protoPictureButton,
icon: namesBitmap,
viewBounds: SetBounds( 2, 8, 34, 40 ),
ButtonClickScript: func()
   cardfile:Toggle()
...}
```

protoInfoButton

This proto is used to include the information button in a view. Tapping the button causes a picker containing information items to appear. Available items include About, Help, and Prefs. The user can tap an item to see information of the type selected.

The following graphic illustrates the information button, with and without its picker displayed.

Information Button

Picker displayed when button is tapped

The ViewClickScript, ViewQuitScript, PickActionScript, and PickCancelledScript methods are used internally in the protoInfoButton and should not be overridden.

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

Slot descriptions

`viewFlags`	The default is `vVisible + vReadOnly + vClickable`.
`viewBounds`	Optional. Set to the size and location where you want the information button to appear. If you do not set this slot, the information button appears five pixels to the right of its sibling in a 13 x 13 view. It is designed to be placed next to another button, for example in the status bar.
`viewJustify`	Optional. The default setting is `vjParentLeftH + vjParentTopV + vjSiblingRightH + vjSiblingTopV + vjCenterH + vjCenterV`.

DoInfoAbout

button`:DoInfoAbout()`

This message is sent to the information button view if the user selects the About menu item. The message has no parameters and the return value is ignored. The About item appears on the picker only if you provide this method in the information button view.

DoInfoHelp

button`:DoInfoHelp()`

This message is sent to the information button view if the user selects the Help menu item. The message has no parameters and the return value is ignored. The Help item appears on the picker only if you provide this method in the information button view. You can also call `ShowManual` to bring up the system Help book. See Chapter 20, “Utility Functions,” for more information. Others methods that open help books include `OpenHelpTo`, `OpenHelpBook`, and `OpenHelpBookTo`; see the *Newton Book Maker User's Guide* for more information.

DoInfoPrefs

button:`DoInfoPrefs()`

This message is sent to the information button view if the user selects the Prefs menu item. The message has no parameters and the return value is ignored. The Prefs item appears on the picker only if you provide this method in the information button view.

GenInfoAuxItems

button:`GenInfoAuxItems()`

You can override this method to provide application-specific items that will appear in the picker below the three standard items. This method takes no parameters and should return an array of items suitable for display in the picker. For details, see the section "Specifying the PickItems Array" beginning on page 6-5.

DoInfoAux

button:`DoInfoAux(`*items*`,` *index*`)`

This message is sent to the information button view if the user selects one of the auxiliary items (included by `GenInfoAuxItems`).

items	The array of auxiliary items, returned by the `GenInfoAuxItems` method.
index	The index of the selected item in the *items* array.

protoOrientation

This proto is available on Newton platforms that support changing the screen orientation so that data on the screen can be displayed can be displayed facing different ways.

The appearance and operation of this proto varies depending on the type of Newton ROM. On Newton devices with two available orientations—landscape and portrait— this proto presents a `protoTextButton` with the label "Rotate" which lets the user change between the two modes, while on

other devices it presents a protoPopupButton offering a list of possible orientations.

If you override the default viewBounds or viewJustify values, you should check the protoOrientation.viewBounds value in your ViewSetupFormScript method to ensure that the height and width are correct.

When the user changes the orientation, the screenOrientation slot of the system userConfiguration entry is updated with the selected orientation. In addition, the ReOrientToScreen message is sent to all children of the root view; this message is described in Chapter 3, "Views."

Note that the ButtonClickScript method is used internally in the protoOrientation and should not be overridden.

The protoOrientation uses the protoTextButton as its proto; and protoTextButton is based on a view of the clTextView class.

Slot descriptions

viewFlags	The default is vVisible + vReadOnly + vClickable.
viewBounds	Set to the size and location where you want the orientation button to appear.
viewJustify	Optional. The default setting is vjCenterH + vjParentBottomV + vjParentCenterH.

protoRadioCluster

This proto is used to group a series of radio buttons into a cluster where only one can be "on" at a time. The individual radio buttons should be added as child views to the radio cluster view.

There is no visual representation of a protoRadioCluster view by itself. It serves only as a container for child views based on protoRadioButton or protoPictRadioButton. See protoRadioButton (page 7-19) for an example of what this proto looks like.

The `protoRadioCluster` is based on a view of the `clView` class. The proto itself is provided with no child views, though you add child views to it by placing radio buttons inside of it. The individual radio buttons use one of the following protos: `protoRadioButton` or `protoPictRadioButton`. These two protos are described on pages 7-19 and 7-20.

Slot descriptions

`viewBounds`	Set to the size and location where you want the radio button cluster to appear.
`clusterValue`	Optional. You can specify the button initially selected by storing its `buttonValue` in this slot. During execution, this slot holds the current value of the radio button cluster by storing the `buttonValue` of the selected radio button. The default initial value is `nil` (no button selected).
`declareSelf`	Do not change. This slot is set by default to `'cluster`. This symbol is used internally to identify the cluster view.

InitClusterValue

cluster`:InitClusterValue(`*buttonValue*`)`

This method initializes the cluster. You can pass a button value to set a particular button, or `nil` to initialize the cluster with no buttons set. This method does not send the `ClusterChanged` method.

buttonValue	The button value, or `nil` for no buttons set.

ViewSetupFormScript

cluster`:ViewSetupFormScript()`

If you want to set an initially selected button but need to calculate its value at run time, calculate the value and set `clusterValue` from within this method.

ClusterChanged

cluster:ClusterChanged()

This method is called whenever the value of the radio cluster changes (that is, when a different radio button is "turned on") to give you a chance to do any processing. This method is passed no parameters and the return value is ignored.

SetClusterValue

cluster:SetClusterValue(*buttonValue*)

You can call this method to programmatically change the selected radio button in a cluster. This method does several tasks, including giving the user "undo" capability for the change, updating the screen appropriately, and calling the clusterChanged method.

buttonValue The button value of the button you want to change.

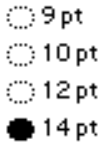
The following example is a template using protoRadioCluster and the three radio buttons included in the cluster:

```
textFaceCluster := {...
_proto: protoRadioCluster,
viewBounds: SetBounds( 70, 17, 130, 77),
ViewSetupFormScript: func()
   clusterValue := userConfiguration.userFont.face,
clusterChanged: func()
   userConfiguration.userFont.face := clusterValue,
...}

child1 :={
_proto: protoRadioButton,
viewBounds: SetBounds( 0, 0, 60, 20),
text: "Bold",
buttonValue: 1
...}
```

```
child2 := {...
_proto: protoRadioButton,
viewBounds: SetBounds( 0, 20, 60, 40),
text: "Underline",
buttonValue: 4
...}

child3 := {...
_proto: protoRadioButton,
viewBounds: SetBounds( 0, 40, 60, 60),
text: "Plain",
buttonValue: 0
...}
```

protoRadioButton

This proto is used to create a radio button child view of a radio button cluster (based on `protoRadioCluster`). A radio button is a small oval bitmap that is either empty or contains a solid bull's-eye when it is selected. It is labeled to the right with a text label as illustrated in the following figure.



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

Note that inking is automatically turned off when the radio button is tapped.

The protoRadioButton uses protoCheckbox as its proto, and protoCheckbox itself has a proto and a child view. See the description of protoCheckbox (page 7-26) for details.

A radio button based on protoRadioButton must be a child view of a view based on protoRadioCluster. You can't create stand-alone radio buttons using this proto.

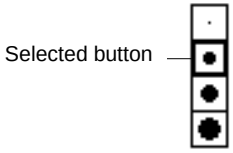
Slot descriptions

viewBounds	Set to the size and location where you want the radio button to appear.
viewFormat	Optional. The default setting is vfNone. Typically you wouldn't want any frame or fill since the radio button provides all of the visual content necessary.
text	A string that is the radio button text label.
buttonValue	Set to the value of the view when the radio button is selected. Each radio button in the cluster should have a different buttonValue. This value is stored in the radio button cluster clusterValue slot if this button is the one selected in the cluster. It's best to use a symbol for this value, since strings and other structured objects may fail an equivalence test even though their contents are equivalent to the comparison value. This is because internal comparisons on pointer objects are done using pointer equality, not content equality.
viewValue	Holds the current value of the radio button. It is set to nil when it is unselected and to the value specified in buttonValue when it is selected. The proto initializes this slot to nil.

protoPictRadioButton

This proto is used to create a child view of a radio button cluster (based on protoRadioCluster). The protoPictRadioButton creates a small boxed view containing a picture. Typically, several of these views are placed adjacent to each other (four of them are placed vertically in the example

shown). The one that is selected is indicated by some kind of highlighting that you must supply. The following figure illustrates the vertical radio button view.



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

The `protoPictRadioButton` uses `protoPictureButton` as its proto, and `protoPictureButton` itself is based on a view of the class `clPictureView`.

A radio button based on `protoPictRadioButton` must be a child view of a `protoRadioCluster`. You can't create stand-alone picture radio buttons using this proto.

Slot descriptions

`viewBounds`	Set to the size and location where you want the picture radio button to appear.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(2) + vfRound(4)`. To simply frame the view, as shown in the example illustration, use this setting: `vfFillWhite + vfFrameBlack + vfPen(1)`.
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV`.
`icon`	The bitmap to be used as the button picture.
`buttonValue`	Set this slot to the value of the view when this picture radio button is selected. Each radio button in the cluster should have a different `buttonValue`. This value is

	stored in the radio button cluster `clusterValue` slot if this button is the one selected in the cluster. It's best to use a symbol for this value, since strings and other structured objects may fail an equivalence test even though their contents are equivalent to the comparison value. This is because internal comparisons on pointer objects are done using pointer equality, not content equality.
`viewValue`	Holds the current value of the radio button. It is set to `nil` when it is unselected and to the value specified in `buttonValue` when it is selected. The proto initializes this slot to `nil`. If you want one radio button initially selected, set this slot to `buttonValue` for that radio button.
`declareSelf`	Do not change. This slot is set by default to `'base`.

ViewDrawScript

button`:ViewDrawScript()`

You must supply this method to highlight the radio button in some manner when it is selected. You could do this by drawing an inner black border, as shown in the example illustration. Your code should begin like this:

```
if viewValue then // do drawing here...
```

The following example is a template using `protoPictRadioButton`:

```
pictRadio := {...
_proto: protoPictRadioButton,
//override frame
viewFormat: vfFillWhite+vfFrameBlack+vfPen(1),
icon: myPict,
buttonValue: 3,
ViewDrawScript: func()
   begin  //if button is selected then highlight it
   if viewValue then
      :DrawShape(MakeRect(0,0,15,15), nil);
```

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

protoCloseBox

This is the close box typically shown in a view as a mechanism for the user to close the view. When it is tapped, the view is closed. This proto is very similar to `protoLargeCloseBox`. The differences are that `protoCloseBox` is a slightly smaller icon and the frame is part of the bitmap. The following graphic is an example:

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

Slot descriptions

`viewFlags`	The default is `vVisible + vReadOnly + vClickable`.
`viewBounds`	Set to the size and location where you want the close box to appear. If you do not set this slot, the close box defaults to the lower right corner of its instantiator's view. (The bitmap is placed at -14, -14 from the lower-right corner.)
`viewJustify`	Optional. The default setting is `vjParentBottomV + vjParentRightH`.
`viewFormat`	Optional. The default setting is `vfNone`. Typically you wouldn't want any frame or fill since the close box picture provides the visual content.

Additionally, the view that is to be closed by the close box must contain the following slot:

```
declareSelf: 'base
```

Usually this would be the application base view.

ButtonClickScript

box:ButtonClickScript()

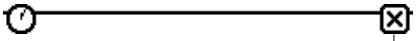
This method is defined in the proto to send the Close message to the view identified as base. You don't need to redefine this method unless you want to perform additional operations, such as disconnecting a communications link, before the view is closed. If you do redefine this method, you should send the message inherited:?ButtonClickScript() at the end of your method to preserve the view closing behavior.

The following example is a template using protoCloseBox:

```
printerPicker := {...
declareSelf: 'base,
...}

child := {...  // child of printPicker
_proto: protoCloseBox,

ButtonClickScript: func()
   begin
   :closeNetwork();
   inherited:?ButtonClickScript();
   end,
...}
```

protoLargeCloseBox

This is a picture button that contains an “X” icon. When tapped, it closes the view identified as base (usually the view the button appears in). This proto is very similar to protoCloseBox. The differences are that protoLargeCloseBox is a slightly bigger icon and the frame is not part of the bitmap, but is controlled by the viewFormat flags. The following graphic illustrates a large close box.



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

Slot descriptions

`viewFlags`	The default is `vVisible + vReadOnly + vClickable`.
`viewBounds`	Set to the size and location where you want the close button to appear. If you do not set this slot, the close button defaults to the lower right corner of its instantiator's view. (The bitmap is placed at -18, -18 from the lower-right corner.)
`viewJustify`	Optional. The default setting is `vjParentBottomV + vjParentRightH + vjCenterH + vjCenterV`.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(2) + vfRound(4)`.

Additionally, the view that is to be closed by the close button view must contain the following slot:

```
declareSelf: 'base
```

Usually this would be the application base view.

ButtonClickScript

box`:ButtonClickScript()`

This method is defined in the proto to send the `Close` message to the view identified as `base`. You don't need to redefine this method unless you want to perform additional operations, such as disconnecting a communications link, before the view is closed. If you do redefine this method, you should

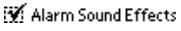
send the message `inherited:?ButtonClickScript()` at the end of your method to preserve the view closing behavior.

The following example is a template using `protoLargeCloseBox`:

```
closer := {...
_proto: protoLargeCloseBox,
// no need to define anything else
...}
```

protoCheckbox

This proto is used to create a checkbox. This is a small dotted box that may include a check mark. It is labeled to the right with a text label. When the user taps the checkbox, a check is drawn in it. If the user taps a checked box, the check is removed. Here following figure illustrates a checked box.



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

Note that inking is automatically turned off when the checkbox is tapped.

The `protoCheckbox` uses an internal proto called `protoCheckBoxIcon` as its proto; `protoCheckBoxIcon` is based on a view of the `clPictureView` class. The `protoCheckbox` itself implements the checkbox icon portion of the proto. It has one child view, a `clTextView`, that implements the text label portion of the proto. A `clTextView` is simply a `clParagraphView` that is read-only and it supports no tabs or multistyled text.

The `protoCheckBoxIcon` proto identifies itself as the `base` view (`declareSelf: 'base`).

Slot descriptions

`viewBounds`	Set to the size and location where you want the checkbox to appear.
`viewFormat`	Optional. The default setting is `vfNone`. Typically you wouldn't want any frame or fill since the checkbox provides all of the visual content necessary.
`viewFont`	Optional. The default font for the text label is `ROM_fontSystem9`.
`text`	A string that is the checkbox text label.
`buttonValue`	Optional. Set this slot to the value that you want the view to have when the checkbox is "checked." The default, if you don't provide this slot, is `non-nil`. It's best to use a symbol for this value, because strings and other structured objects may fail an equivalence test even though their contents are equivalent to the comparison value. This is because internal comparisons on pointer objects are done using pointer equality, not content equality.
`viewValue`	Holds the current value of the checkbox. This slot is set to `nil` when it is unchecked and to the value specified in `buttonValue` when it is checked.

ValueChanged

checkBox`:ValueChanged()`

This method is called whenever the value of the checkbox changes, to allow you to do additional processing. It is passed no parameters and the return value is ignored.

ToggleCheck

checkBox`:ToggleCheck()`

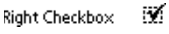
You can call this method to programmatically toggle the check mark in the checkbox. If the check is shown, it is removed, and vice versa. The checkbox is redrawn appropriately. This method takes no parameters and always returns `non-nil`.

The following example is a template using protoCheckBox:

```
notifier :={
_proto: protoCheckBox,
viewBounds: SetBounds( 40, 30, 200, 45),
buttonValue: true,
text: "Play Notify Sound"
...}
```

protoRCheckbox

This proto is used to create a checkbox where the text is to the left of the checkbox. Otherwise, this proto is identical to protoCheckbox. The following graphic illustrates a checkbox with text to its left:



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

Note that inking is automatically turned off when the checkbox is tapped.

The protoRCheckbox uses an internal proto called protoCheckBoxIcon as its proto; protoCheckBoxIcon is based on a view of the clPictureView class. The protoRCheckbox itself implements the checkbox icon portion of the proto. It has one child view, a clTextView, that implements the text label portion of the proto. A clTextView is simply a clParagraphView that is read-only and it supports no tabs or multistyled text.

The protoCheckBoxIcon proto identifies itself as the base view (declareSelf: 'base).

Slot descriptions

`viewBounds`	Set to the size and location where you want the checkbox to appear.
`viewFormat`	Optional. The default setting is `vfNone`. Typically you wouldn't want any frame or fill since the checkbox provides all of the visual content necessary.
`viewFont`	Optional. The default font for the text label is `ROM_fontSystem9`.
`text`	A string that is the checkbox text label.
`indent`	Optional. The number of pixels to indent the checkbox to the right of the text. The default indent is 16.
`buttonValue`	Optional. Set this slot to the value that you want the view to have when the checkbox is "checked." The default, if you don't provide this slot, is non-`nil`. It's best to use a symbol for this value, since strings and other structured objects may fail an equivalence test even though their contents are equivalent to the comparison value. This is because internal comparisons on pointer objects are done using pointer equality, not content equality.
`viewValue`	Holds the current value of the checkbox. This slot is set to `nil` when it is unchecked and to the value specified in `buttonValue` when it is checked.

ValueChanged

checkBox`:ValueChanged()`

This method is called whenever the value of the checkbox changes, to allow you to do additional processing. It is passed no parameters and the return value is ignored.

ToggleCheck

checkBox`:ToggleCheck()`

You can call this method to programmatically toggle the check mark in the checkbox. If the check is shown, it is removed, and vice versa. The checkbox

is redrawn appropriately. This method takes no parameters and always returns non-`nil`.

The following example is a template using `protoRCheckBox`:

```
rightCheckView :={
_proto: protoRCheckBox,
viewBounds: SetBounds( 40, 30, 200, 45),
buttonValue: true,
text: "Right Checkbox"
...}
```

Tabs

This section describes protos that implement tabs with letters of the alphabet. The protos include the following:

- `protoAZTabs` includes alphabetical tabs arranged horizontally, in a view.
- `protoAZVertTabs` includes alphabetical tabs, arranged vertically, in a view.

protoAZTabs

This proto is used to include alphabetical tabs, arranged horizontally, in a view; for example:

PickLetterScript

tabs`:PickLetterScript(`*letter*`)`

This method is called when the user taps a tab.

letter	The letter that was tapped.

The following example shows a `pickLetterScript` method:

```
pickLetterScript: func(theLetter)
   begin
   setValue(theText,'text);
   end
```

protoAZVertTabs

This proto is used to include alphabetical tabs, arranged vertically, in a view; for example:

PickLetterScript

tabs:`PickLetterScript(`*letter*`)`

This method is called when the user taps a tab.

letter The letter that was tapped.

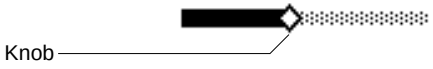
Status Indicators

This section describes protos that implement status. The protos include the following:

- `protoSlider` creates a user-settable gauge view.
- `protoGuage` creates a read-only gauge view.
- `protoLabelBatteryGuage` creates a read-only gauge view that graphically shows the amount of power remaining in the system battery.
- `clGuageView` displays objects that look like analog bar gauges.

protoSlider

This proto is used to create a user-settable gauge view, which looks like an analog bar gauge with a draggable diamond-shaped knob, as shown in the following graphic:



If you want to have a read-only gauge, use the `protoGauge` proto.

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

Note that you cannot dynamically change the value of the `minValue` and `maxValue` slots at run time, except within the `ViewSetupFormScript` method. If you need to change the value of these slots, you must close the view, change the values, then reopen the view.

The `protoSlider` uses the `protoGauge` as its proto; `protoGauge` is based on a view of the `clGaugeView` class.

Slot descriptions

`viewBounds`	Set to the size and location where you want the gauge to appear.
`viewValue`	Set this slot to give the gauge an initial value, if you know it ahead of time. If you need to calculate the initial value at run time, set this slot within the

	`ViewSetupFormScript`. This value must be an integer between `minValue` and `maxValue`, inclusive. During execution, the `viewValue` slot stores the current gauge setting by interpolating between `minValue` and `maxValue`.
`minValue`	Optional. This specifies the minimum gauge value; the number set as the `viewValue` when the gauge reads fully to the left side. The default is zero, which you can change if you wish.
`maxValue`	Optional. This specifies the maximum gauge value; the number set as the `viewValue` when the gauge reads fully to the right side. The default is 100, which you can change if you wish.
`gaugeDrawLimits`	Optional. The default is non-`nil`, which displays the gray background. If you set this slot to `nil`, the gray background is not displayed.

ViewSetupFormScript

slider`:ViewSetupFormScript()`

Use this method to set the initial value of the `viewValue` slot at run time, or to do any other processing necessary before the view is instantiated. If you do not want to do anything in this method, you must still supply it, but just define it as `nil`.

ChangedSlider

slider`:ChangedSlider()`

This method is called once each time the gauge is moved, after the pen is lifted from it. The current gauge setting is available in the `viewValue` slot. If the gauge was moved but reset back to its original value before the pen was lifted, this method is not called. This method is passed no parameters and the return value is ignored. If you do not want to do anything in this method, you must still supply it, but just define it as `nil`.

TrackSlider

slider:`TrackSlider()`

This method is called whenever the value of the `viewValue` slot changes. It is provided so you can dynamically track the changes as the slider is moved and take action based on the current value. `TrackSlider` is called repeatedly as the gauge knob is dragged; it is passed no parameters and the return value is ignored.

The following example is a template using `protoSlider`:

```
SoundSetter := {...
_proto: protoSlider,
viewBounds: RelBounds( 12, -21, 65, 9),
viewJustify: vjParentBottomV,
maxValue: 4,

ViewSetupFormScript: func()
   self.viewValue := userConfiguration.soundVolume,

ChangedSlider: func()
   begin
   SetVolume(viewValue);
   :SysBeep();
   end,
...}
```

protoGauge

This proto is used to create a read-only gauge view, which looks like an analog bar gauge is shown in the following graphic.

If you want to have a gauge that the user can set, use the `protoSlider` proto.

Note that you cannot change the value of the `minValue` and `maxValue` slots at run time, except within the `ViewSetupFormScript` method. If you need to change the value of these slots, you must close the view, change the values, then reopen the view.

The `protoGauge` is based on a view of the `clGaugeView` class.

Slot descriptions

`viewBounds`	Set to the size and location where you want the gauge to appear.
`viewValue`	Set this slot to give the gauge an initial value, if you know it ahead of time. If you need to calculate the initial value at run time, set this slot within the `ViewSetupFormScript`. This value must be an integer between `minValue` and `maxValue`, inclusive. During execution, the `viewValue` slot stores the current gauge setting by interpolating between `minValue` and `maxValue`.
`minValue`	Optional. This specifies the minimum gauge value; the number set as the `viewValue` when the gauge reads fully to the left side. The default is zero.
`maxValue`	Optional. This specifies the maximum gauge value; the number set as the `viewValue` when the gauge reads fully to the right side. The default is 100.
`gaugeDrawLimits`	Optional. The default is non-`nil`, which displays the gray background. If you set this to `nil`, the gray background is not displayed.

ViewSetupFormScript

gauge`:ViewSetupFormScript()`

Use this method to set the initial value of the `viewValue` slot at run time, or do any other processing needed before the view is instantiated. If you do not

want to do anything in this method you must still supply it, but just define it as `nil`.

The following example is a template using `protoGauge`:

```
BatteryGauge := {...
_proto: protoGauge,
viewBounds: RelBounds( 157, -21, 55, 9),
viewJustify: vjParentBottomV,
ViewSetupFormScript: func()
   self.viewvalue := BatteryLevel(0),
...}
```

protoLabeledBatteryGauge

This proto is used to create a read-only gauge view that graphically shows the amount of power remaining in the system battery. The battery level is checked every 10 seconds and the gauge is updated, if necessary. If the Newton is plugged in and the battery is charging, a charging symbol is shown instead of the gauge. The following graphic shows the battery gauge and battery charging symbol.

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

The `protoLabeledBatteryGauge` uses an internal proto, `protoBatteryGauge`, as its prototype; `protoBatteryGauge` is based on a view of the `clView` class. The `protoBatteryGauge` has two children, the gauge or charging symbol and the label.

Slot description

`viewBounds`	Set to the size and location where you want the gauge to appear. The gauge fills the entire width of the view.

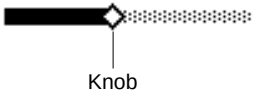
The following example is a template using `protoLabeledBatteryGauge`:

```
BatteryGauge := {
viewBounds: {left: 58, top: 106, right: 186,
            bottom: 130},
_proto: protolabeledbatterygauge,
// no other slots needed
};
```

clGaugeView

The `clGaugeView` view class is used to display objects that look like analog bar gauges. Gauges are useful for graphically displaying information, such as the amount of battery power remaining in the system. They can also be used as interactive controls.

For interactive gauges, the end of the gauge indicator bar contains a small diamond-shaped active area called the knob. The user can click on the knob and drag the indicator bar to a new position. The following graphic is an example of a typical gauge view:



The following slots are of interest for a view of the `clGaugeView` class:

`viewBounds`	Set to the size and location where you want the view to appear.
`viewValue`	Set this slot to give the gauge an initial value, if you know it ahead of time. If you need to calculate the initial value at run time, set this slot within the `ViewSetupFormScript`. This value must be an integer between `minValue` and `maxValue`, inclusive. During

	execution, the `viewValue` slot stores the current gauge setting by interpolating between `minValue` and `maxValue`.
`viewFlags`	The default setting is `vVisible + vClickable`. To make a gauge that is read-only, set the `vReadOnly` flag (and not `vClickable`). For read-only gauges, the diamond-shaped knob is not drawn on the gauge.
`viewFormat`	Optional. The default setting is `nil`.
`minValue`	Optional. This specifies the minimum gauge value; the number set as the `viewValue` when the gauge reads fully to the left side. The default is zero, which you can change if you wish.
`maxValue`	Optional. This specifies the maximum gauge value; the number set as the `viewValue` when the gauge reads fully to the right side. The default is 100, which you can change if you wish.
`gaugeDrawLimits`	Optional. The default is non-`nil`, which displays the gray background. If you set this slot to `nil`, the gray background is not displayed.

ViewChangedScript

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

This method is called whenever the value of the gauge view changes. If you check the value of the `viewValue` slot in this method, you can dynamically track the changes the user is making to the gauge indicator. This method is called repeatedly as the gauge knob is dragged. The return value of this method is ignored.

slot	The name of the slot that changed.
view	The name of the view.

ViewFinalChangeScript

view`:ViewFinalChangeScript(`*valueBefore*, *valueAfter*`)`

This method is called once each time the gauge is moved, after the pen is lifted from the knob. If the gauge was moved but reset to its original value

before the pen was lifted, this method is not called. The return value of this method is ignored.

valueBefore The initial `viewValue` of the gauge before it was changed.

valueAfter The final `viewValue` of the gauge after it was changed.

The following example is a view definition of the `clGaugeView` class:

```
soundGauge := {...
   viewClass: clGaugeView,
   viewBounds: {left:80, top:20, right:180, bottom:28},
   viewFlags: vVisible+vClickable,
   gaugeDrawLimits: true,
   minValue: 0,
   maxValue: 11
   viewSetupFormScript: func()
         self.viewvalue := userConfiguration.soundVolume,
   viewChangedScript: func(slot, context)
      begin
         userConfiguration.soundVolume := viewValue;
         :SysBeep(); //play it so they can hear new level
      end,
...}
```

Dates and Time

The protos in this section provide various ways for the user to specify dates and time. The protos include the following:

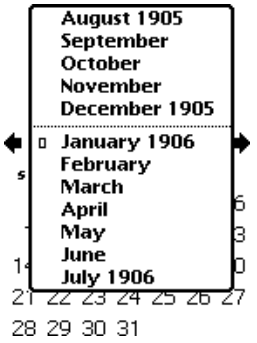
- `protoDatePicker` facilitates the selection of a date.
- `protoDigitalClock` displays a digital clock that can be used to set the time.

- `protoSetClock` creates an analog clock that can be used to set the time.
- `protoNewSetClock` displays an analog clock that can be used to set the time.
- `protoAMPMCluster` includes a.m. and p.m. radio buttons in a view.

protoDatePicker

This proto facilitates the selection of a date. It is assumed that the desired date is relatively close to the current date, since it is difficult to quickly change the year. The following graphic shows a date picker:

Tapping either of the arrows scrolls to the prior / next month; tapping a day selects the day; tapping the spaces before the first or after the last day of a month selects the appropriate day in the prior / next month; and tapping the month / year banner at the top displays a pop-up menu to change the month. The pop-up menu looks like this:



The following slots and methods are used internally:

`monthChangedScript`, `SetTitle`, `viewSetupDoneScript`

These are listed so that you don't inadvertently override them.

Slot description

`selectedDates`	An array containing one element, an integer representing the selected date (as returned by the `Time` function). You can supply an initial date to display here as well; if you don't, the current date is used. To change the selected date programmatically, supply the new date in this slot and then call the `Refresh` method.

DateChanged

picker`:DateChanged(`*array*`)`

This method is called when a date is selected, to give you a chance to take some action. The return value is ignored.

array	An array containing a single element, the selected date.

Refresh

picker`:Refresh()`

To change the selected date programmatically, supply a new date in the `selectedDates` slot and then call this method to update the view.

protoDigitalClock

This proto is used to display a digital clock that can be used to set the time. The user can change the time by tapping each digit. A tap on the upper part of the digit increments it to the next number, and a tap on the lower part of the digit decrements it. The following digital clock graphic is an example.

Note that the `digitsContainer` method is used internally (in the `protoDigit` base proto) and should not be overridden.

Slot descriptions

`viewFlags`	For future compatibility, you must set the `vClickable` flag. (Note however, that clicks are taken and processed by the children of this proto.)
`viewBounds`	The clock size is fixed at 119 x 28 pixels.
`viewJustify`	The default setting is `vjParentLeftH + vjParentTopV`.
`increment`	The number of minutes to change on each tap (default=1).
`time`	Required. The time to which the clock should be set, expressed in the number of minutes elapsed since midnight, January 1, 1904. When the time is changed, this slot is updated with the currently set time. Note that a `time` slot must be set, either here or somewhere above this proto in the inheritance hierarchy.
`wrapping`	Set to non-`nil` (the default value) to wrap around day boundaries.
`midnite`	Set to non-`nil` if the value 0 should indicate midnight tomorrow (in other words, the end of the current day). The default value is `nil`, meaning that 0 indicates midnight today (the beginning of the current day).

Refresh

picker`:Refresh()`

You can call this method to update the appearance of the clock when the system time changes by some external event. For example, if there are two clocks present, the user changing the time in one should be followed by sending the `Refresh` message to the second.

TimeChanged

clock:TimeChanged()

This method is called when the time is changed, to give you a chance to take some action. The return value is ignored.

protoSetClock

This proto is used to create an analog clock with which the user can set a time. The user sets the hour by tapping the inner circle where the hour hand should be positioned and by tapping the outer circle where the minute hand should be positioned. The following graphic is an example:

The following methods are defined internally: ViewDrawScript, ViewStrokeScript, Diff, Distance, DrawHand, DrawHilite, and FastEnoughAtan. If you need to use one of these methods, be sure to call the inherited method also (for example, inherited:?ViewDrawScript()), otherwise the proto may not work as expected.

The protoSetClock is based on a view of the clPictureView class.

Slot descriptions

viewBounds	The clock size is fixed at 64 x 64 pixels.
viewFlags	The default setting is vVisible + vClickable + vStrokesAllowed.
viewFormat	Optional. The default setting is vfNone.
hours	Initially set to nil. This slot is updated with the new hour when the user sets the hour hand.
minutes	Initially set to nil. This slot is updated with the new minute time when the user sets the minute hand.

TimeChanged

clock:TimeChanged()

This message is sent to the view when the user changes the time on the clock. It takes no parameters. If nothing else, you should at least include the following code so that the clock draws properly: self:Dirty().

The following example is a template using protoSetClock:

```
clock := {...
 _proto: protosetclock,
hours: nil, // updated when a new time is set
minutes: nil, // updated when a new time is set
TimeChanged: func()
   begin
   // do this so the old hands are erased...
   self:Dirty();
   // this is just for fun, put your own stuff here...
   print("H:" && hours && "M:" && minutes);
   end,
ViewSetupFormScript: func() // show the current time
   begin
   local t:=Time();
   self.hours:=(t DIV 60) MOD 24;
   self.minutes:=t MOD 60;
   end,
...};
```

protoNewSetClock

This proto is used to display an analog clock that can be used to set the time. It is based on a view of the clView class.

There are four ways to change the time with this proto:

- Either hand can be dragged around to the correct position.

- Tapping on the rim of the clock changes the minutes—unless it is within two degrees of the location pointed to by the hour hand, in which case the tap is interpreted as an attempt to drag the hour hand.
- Tapping on the inner circle of dots sets the hours—unless it is within two degrees of the minute hand, in which case it is interpreted as an attempt to drag the minute hand.
- A line can be drawn from the center of the clock face to either the border (to set the minutes) or the inner dial (to set the hours).

The following graphic shows an analog clock with four setting options.

The following slots and methods are used internally. They are listed so that you don't inadvertently override them.:

`ViewSetupFormScript`, `ViewSetupChildrenScript`, `ViewSetupDoneScript`, `ViewDrawScript`, `ViewClickScript`, `tickSound`, `tockSound`, `cuckooSound`, `tickTock`, `hours`, `minutes`, `icon`, `fIconAsShape`, `DrawHand`, `diff`, `FastEnoughAtaN`, `atanTable`, `sinTable`, `distance`.

Slot descriptions

`viewBounds`	Set to the size and location where you want the clock to appear. By default, the bounds are `{left: 30, top: 30, right: 146, bottom: 146}`. The height and width should be equal and a multiple of 29 to make the clock face appear its best.
`viewJustify`	The default setting is `vjParentLeftH + vjParentTopV`.
`time`	Optional. The time to which the clock should be set, expressed in the number of minutes elapsed since

midnight, January 1, 1904. If you don't include this slot, the clock is set to the current time. When the time is changed, this slot is updated with the currently set time.

`annotations` Optional. An array of four strings to be used as minute annotations around the clock face, beginning with the number at the top of the clock and proceeding clockwise. For example, the strings ["N", "E", "S", "W"] would decorate the clock like a compass. If you don't specify this slot, the following annotations are used: ["12", "3", "6", "9"].

`supressAnnotations`
Optional. If this slot exists (with any value), the four minute annotations around the clock face are not drawn.

`exactHour` Optional. A Boolean. If non-`nil`, the hour hand will cling exactly to the hour markers. If `nil`, the hour hand will adjust between the minutes appropriately, according to the minutes. By default, this is `nil`. (In general, this should probably be left set with the default value.)

Refresh

clock`:Refresh()`

You can call this method to update the appearance of the clock when the system time is changed by some external event. For example, if there are two clocks present and the user changes the time in one clock, the `Refresh` message should be sent to the second.

TimeChanged

clock`:TimeChanged()`

This method is called when the time is changed, to give you a chance to take some action. The return value is ignored.

protoAMPMCluster

This proto is used to include a.m. and p.m. radio buttons in a view, as shown in the following graphic.

The `bounds` must be 70 pixels wide and 15 pixels high. To use this proto, define a parent (a `clView`, for instance), declare a `protoDigitalClock` to the parent as `'setter`, and add a `protoAMPMCluster` to the same parent.

The `protoAMPMCluster` uses `protoRadioCluster` as its proto; `protoRadioCluster` is based on a view of the `clView` class.

`time`	Required. This slot must be set, either here or somewhere above this proto in the inheritance hierarchy.

The following example is a template using `protoAMPMCluster`.

```
picker := RDefChild(myTimePopup, 'setter, RDefTemplate( {
   _proto: protoDigitalClock,
   // ...
   }));

RDefChild(myTimePopup, 'ampmButtons, {
   _proto: protoAMPMCluster,
   viewBounds: SetBounds(0, 2, 70, 20),
   viewJustify: vjCenterH + vjSiblingCenterH+vjSiblingBottomV,

   });
```

Miscellaneous Protos

This section describes miscellaneous protos. The protos include the following:

- `protoBorder` fills a view with black.
- `protoDivider` creates a divider bar.
- `protoGlance` creates a text view that closes itself automatically after it has been shown for a brief period.
- `protoDrawer` creates a view that acts like the Extras Drawer.
- `protoDragger` creates a view that the user can move around the screen by dragging it with the pen.
- `protoDragNGo` includes a close box in the lower-right corner of the view.
- `protoFloater` creates a draggable view that floats above all other nonfloating sibling views within an application.
- `protoFloatNGo` includes a close box in the lower-right corner of the floating view.
- `protoStaticText` used for static text.
- `protoStatus` used to create a status bar at the bottom of a view.
- `protoStatusBar` creates a status bar at the bottom of a view.
- `protoTitle` creates a title centered in a black rounded rectangle at the top of a view.

protoBorder

This is simply a view filled with black. This view can be used as a border, or as a line, or anywhere a black rectangle is needed. The following graphic is an example of a border.

The `protoBorder` is based on a view of the `clView` class.

Slot descriptions

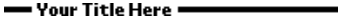
`viewBounds`	Set to the size and location where you want the border to appear.
`viewFlags`	The default setting is `vVisible`.
`viewFormat`	Optional. The default setting is `vfFillBlack`.

The following example is a template using `protoBorder`:

```
theBorder := {...
_proto: protoBorder,
viewBounds: SetBounds( 0, 0, 0, 2 ), // 2-pixel wide line
viewJustify: vjParentFullH,
...}
```

protoDivider

This proto is used to create a divider bar that extends the whole width of its parent view. The divider bar consists of a text string near the left end of a thick line, as shown in the following figure.



The `ViewSetupChildrenScript` method is defined internally. If you need to use this method, be sure to call the inherited method also (for example, `inherited:?ViewSetupChildrenScript()`), otherwise the proto may not work as expected.

The `protoDivider` is based on a view of the `clView` class. It has the following two child views declared in itself:

- `divider`. This child view uses the `protoBorder` proto. It is used for the solid black line.
- `dividerText`. This child view is based on a view of the `clTextView` class (simply a `clParagraphView` that is read-only and it supports no tabs or multistyled text). It is used for the text label on the divider bar.

Slot descriptions

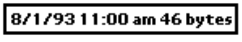
`viewBounds`	Set to the size and location where you want the divider bar to appear. By default, the divider extends the whole width of its parent view (see `viewJustify`).
`viewFlags`	The default setting is `vVisible + vReadOnly`. In most cases you won't need to change this.
`viewFont`	Optional. The default font is `ROM_fontSystem9Bold`.
`viewJustify`	Optional. The default setting is `vjParentFullH`.
`viewFormat`	Optional. The default setting is `vfNone`. In most cases you won't need to change this.
`title`	A string that is the text on the divider bar.
`titleHeight`	Optional. The height of the divider view defaults to the height of the font used. For a taller divider view, set this slot to a greater value.

The following example is a template using `protoDivider`:

```
protoCoverBorder := {...
_proto: protoDivider,
viewFont: ROM_fontSystem18Bold,
title: "COVER SHEET",
}
```

protoGlance

This proto is used to create a text view that closes itself automatically after it has been shown for a brief period. Also, if the user taps the view, it will close immediately. The following graphic shows a text view open at a glance.



The following methods are defined internally: `ViewSetupDoneScript`, `ViewClickScript`, and `ViewIdleScript`. If you need to use one of these methods, be sure to call the inherited method also (for example,

`inherited:?ViewClickScript()`), otherwise the proto may not work as expected.

The `protoGlance` is based on a view of the `clTextView` class. This kind of view is simply a `clParagraphView` that is read-only and it supports no tabs or multi-styled text.

Typically a `protoGlance` view is hidden until the user performs an action such as tapping a button. After the button is tapped, the `Open` message is sent to the `protoGlance` view to cause it to show itself.

Slot descriptions

`viewBounds`	Set to the size and location where you want the view to appear.
`viewJustify`	Optional. The default setting is `vjCenterV + vjLeftH`.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfPen(2) + vfFrameBlack + vfInset(1)`.
`viewFont`	Optional. The default font is `ROM_fontSystem9Bold`.
`viewEffect`	Optional. The default effect is `fxRight + fxRevealLine`.
`viewIdleFrequency`	Optional. This sets the length of time that the view is to remain open, in milliseconds. The default is 3000 milliseconds (three seconds). Specify an integer greater than zero.
`text`	A string that is the text to appear in the view.

The following example is a template using `protoGlance` (and the button that opens the glance view):

```
myGlance := {...
_proto: protoglance,
text: "Just a glance...",
viewIdleFrequency: 5000,
viewfont: ROM_fontSystem9Bold,
viewJustify: vjCenterV+vjCenterH,
```

```
...};

showGlance := {... // Button that opens the glance view
_proto: prototextbutton,
text: "Show it",
ButtonClickScript: func()
   myGlance:Open(),
   };
```

protoDrawer

This proto is used to create a view that acts like the Extras Drawer. It slides up from the bottom of the view when opened, and is accompanied by a drawer opening sound when it opens and a drawer closing sound when it closes. The proto has no content defined—you must add child views to it.

The `protoDrawer` is based on a view of the `clView` class.

Slot descriptions

`viewBounds`	Set to the size and location where you want the view to appear.
`viewFlags`	The default setting is `vFloating + vApplication`.
`viewFormat`	Optional. The default setting is `vfPen(2) + vfFrameBlack`.
`viewEffect`	Optional. The default setting is `fxDrawerEffect`.
`showSound`	Optional. The default setting is `ROM_draweropen`.
`hideSound`	Optional. The default setting is `ROM_drawerclose`.

The following example is a template using `protoDrawer`:

```
myDrawer := {...
_proto: protoDrawer}  // nothing else needed for drawer
// add children to drawer
```

protoDragger

This proto is used to create a view that the user can move around the screen by dragging it with the pen. This view has a rounded matte frame with a small control at the top-center of the frame. The view can be dragged only from this small control. The view can be dragged only within the bounds of its parent view. The following graphic shows this type of draggable view.

The proto defines no contents for the draggable view. You will need to add your own contents by adding child templates to it.

By default, `protoDragger` does not support scrolling or overview. If you want the view to support scrolling or overview (that is, to handle the scroll arrows and overview button), set the `vApplication` bit in the `viewFlags` slot, and provide the appropriate methods (`ViewScrollUpScript`, `ViewScrollDownScript`, and `ViewOverviewScript`) to handle scroll and overview messages.

The `protoDragger` is based on a view of the `clView` class. It has no child views.

Slot descriptions

`viewBounds`	Set to the size and location where you want the view to appear.
`viewFlags`	The default setting is `vClickable`. Do not change this flag, you may add others if you wish.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameDragger + vfPen(7) + vfInset(1) + vfRound(5)`.
`noScroll`	Optional. This slot holds a message that is used in an error alert if the scroll arrows are tapped and you have not provided a `ViewScrollUpScript` or `ViewScrollDownScript` method to handle the event.

	This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and thus it is receiving scroll events. The default message is, “This application does not support scrolling,” which you can change if you want.
`noOverview`	Optional. This slot holds a message that is used in an error alert if the overview button is tapped and you have not provided a `ViewOverviewScript` method to handle the event. This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and thus it is receiving overview events. The default message is, “This application does not support Overview,” which you can change if you want.
`declareSelf`	Do not change. This slot is set by default to `'base`. This identifies the view to be closed when the user taps the close box, if you provide one as a child of this view.

The following example is a template using `protoDragger`:

```
dragger:= {...
_proto: protoDragger,
viewBounds: {left: -5, top: 104, right: 168, bottom:
            162},
viewJustify: vjParentCenterH,
viewflags: vVisible+vClickable,
...};

theText := {...  // child of the draggable view
_proto: protostatictext,
text: "I'm draggable. . ",
viewBounds: {left: 40, top: 24, right: 144, bottom: 48},
viewfont: simpleFont12+tsBold,
...};
```

protoDragNGo

This proto is identical to `protoDragger`, except that it includes a close box in the lower-right corner of the view. The following graphic shows a draggable view with a close box.

The proto defines no contents for the view. You can add your own contents by adding child templates to it.

The `protoDragNGo` is based on the `protoDragger` proto, which is based on a view of the `clView` class. It is provided with one child view that is a close box based on the `protoCloseBox` proto.

Slot descriptions

`viewBounds`	Set to the size and location where you want the floater to appear.
`viewFlags`	The default setting is `vClickable`. Do not change this flag, but you can add others if you wish.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameDragger + vfPen(7) + vfInset(1) + vfRound(5)`.
`noScroll`	Optional. This slot holds a message that is used in an error alert if the scroll arrows are tapped and you have not provided a `ViewScrollUpScript` or `ViewScrollDownScript` method to handle the event. This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and thus it is receiving scroll events. The default message is, “This application does not support scrolling,” which you can change if you want.
`noOverview`	Optional. This slot holds a message that is used in an error alert if the overview button is tapped and you have not provided a `ViewOverviewScript` method to handle the event. This error occurs only if the

	vApplication flag is set for this view (it is not set by default), and thus it is receiving overview events. The default message is, “This application does not support Overview,” which you can change if you want.
declareSelf	Do not change. This slot is set by default to 'base. This identifies the view to be closed when the user taps the close box.

The following example is a template using protoDragNGo:

```
dragngoView:= {...
_proto: protoDragNGo,
viewBounds: {left: -2, top: 98, right: 158, bottom: 170},
viewJustify: vjParentCenterH,
viewflags: vVisible+vClickable,
...};

theText := {...  // child of the dragngo view
_proto: protostatictext,
text: "Drag'n Go view",
viewBounds: {left: 22, top: 30, right: 134, bottom: 54},
viewfont: simpleFont12+tsBold,
...};
```

protoFloater

This proto is used to create a draggable view that floats above all other nonfloating sibling views within an application. This proto is identical to protoDragger, except that protoFloater is horizontally centered within its parent view, it has an opening view effect, and it has the vFloating flag set in the viewFlags slot.

Note

For the base view of an application, it is recommended that you use `protoDragger` instead of `protoFloater`. The floating property interferes with some system services for applications. ◆

The proto defines no contents for the floating view. You can add your own contents to the floater by adding child templates to it.

By default, `protoFloater` does not support scrolling or overview. If you want your floater to support scrolling or overview (that is, to handle the scroll arrows and overview button), set the `vApplication` bit in the `viewFlags` slot, and provide the appropriate methods (`ViewScrollUpScript`, `ViewScrollDownScript`, and `ViewOverviewScript`) to handle scroll and overview messages.

The `protoFloater` is based on the `protoDragger`. It has no child views.

Slot descriptions

`viewBounds`	Set to the size and location where you want the floater to appear.
`viewFlags`	The default setting is `vFloating + vClickable`. Do not change these flags, but you can add others if you wish.
`viewJustify`	Optional. The default setting is `vjParentCenterH`.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameDragger + vfPen(7) + vfInset(1) + vfRound(5)`.
`viewEffect`	Optional. The default effect is `fxZoomOpenEffect`.
`noScroll`	Optional. This slot holds a message that is used in an error alert if the scroll arrows are tapped and you have not provided a `ViewScrollUpScript` or `ViewScrollDownScript` method to handle the event. This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and thus it is receiving scroll events. The default message is, “This application does not support scrolling,” which you can change if you want.

`noOverview`	Optional. This slot holds a message that is used in an error alert if the overview button is tapped and you have not provided a `ViewOverviewScript` method to handle the event. This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and thus it is receiving overview events. The default message is, “This application does not support Overview,” which you can change if you want.
`declareSelf`	Do not change. This slot is set by default to `'base`. This identifies the view to be closed when the user taps the close box, if you provide one as a child of this view.

protoFloatNGo

This proto is identical to `protoFloater`, except that it includes a close box in the lower-right corner of the floating view. This allows the user a way to close the view.

Note

For the base view of an application, it is recommended that you use `protoDragNGo` instead of `protoFloatNGo`. The floating property interferes with some system services for applications. ◆

The proto defines no contents for the floating view. You will need to add your own contents to the floater by adding child templates to it.

The `protoFloatNGo` is based on the `protoFloater` proto, which is based on the `protoDragger`. It is provided with one child view that is a close box based on the `protoCloseBox` proto.

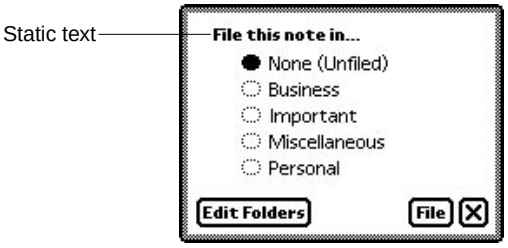
Slot descriptions

`viewBounds`	Set to the size and location where you want the floater to appear.
`viewFlags`	The default setting is `vFloating + vClickable`. Do not change these flags, you may add others if you wish.
`viewJustify`	Optional. The default setting is `vjParentCenterH`.

`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameDragger + vfPen(7) + vfInset(1) + vfRound(5)`.
`viewEffect`	Optional. The default effect is `fxZoomOpenEffect`.
`noScroll`	Optional. This slot holds a message that is used in an error alert if the scroll arrows are tapped and you have not provided a `ViewScrollUpScript` or `ViewScrollDownScript` method to handle the event. This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and thus it is receiving scroll events. The default message is, “This application does not support scrolling,” which you can change if you want.
`noOverview`	Optional. This slot holds a message that is used in an error alert if the overview button is tapped and you have not provided a `ViewOverviewScript` method to handle the event. This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and thus it is receiving overview events. The default message is, “This application does not support Overview,” which you can change if you want.
`declareSelf`	Do not change. This slot is set by default to `'base`. This identifies the view to be closed when the user taps the close box.

protoStaticText

This proto is used for static text. It defines a one-line paragraph view that is read-only and left-justified. The following graphic shows a static text view:



The `protoStaticText` is based on a view of the `clParagraphView` class.

Slot descriptions

`viewBounds`	Set to the location where you want the text to appear.
`viewFlags`	The default setting is `vVisible + vReadOnly`. In most cases you won't need to change this.
`text`	A string that is the text you want to display.
`viewFont`	Optional. The default font is `ROM_fontSystem9Bold`. This slot is ignored if the `styles` slot is present.
`viewJustify`	Optional. The default setting is `vjLeftH + oneLineOnly`.
`viewFormat`	Optional. The default is `vfNone`.
`viewTransferMode`	Optional. The default transfer mode is `modeOr`.
`tabs`	Optional. An array of as many as eight tab-stop positions, in pixels. For example: [10, 20, 30, 40].
`styles`	Optional. An array of alternating run lengths and font information, if multiple font styles are used for the text. The first element is the run length (in characters) of the first style run, and the second element is the font style of that run. The third element is the run length of the second style run, and so on. All of the run lengths must

add up to the total text length. If the text is all in a single font, the font in the `viewFont` slot specifies the font style, and the `styles` slot is not needed. For information on how to specify a font in the `styles` array, see the section "Specifying a Font" in Chapter 8, "Text and Ink Input and Display."

The following example is a template using `protoStaticText`:

```
heading := {...
viewbounds: RelBounds( 30, 15, 170, 50),
_proto: protoStaticText,
viewJustify: vjCenterH+vjTopV,
viewFont: ROM_fontSystem10,
text: "Pick your favorite color:",
...}
```

protoStatus

This proto is used to create a status bar at the bottom of a view. The status bar includes a large close button at the right side and an analog clock at the left side. If the user taps the analog clock, a digital clock is displayed for three seconds. The following graphic shows the status bar with an analog clock and close box.

Note

If you prefer a status bar without the analog clock, see the description of the `newtStatusBarNoClose` and `newtStatusBar` protos on pages 4-55 and 4-56 in Chapter 4, "NewtApp Applications."

The `viewJustify` flags for this view are set so that the status bar always appears at the bottom of its parent view and it always occupies the full width of the parent view. Instantiators are not required to set any slots. However,

the application base view in which the protoStatus view is included must include this slot:

```
declareSelf: 'base
```

This identifies the view that should be closed when the close box in the status bar is tapped.

The protoStatus is based on another proto called protoStatusBar, which is based on a view of the clView class. The protoStatusBar contains the following two child views:

- A small round analog clock that appears at the side left of the view. The clock is based on a view of the clView class.
- A digital clock display that slides out from the analog clock when the user taps the analog clock. This view is hidden automatically after three seconds. This view is based on the protoGlance proto.

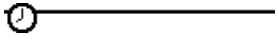
The protoStatus itself has one child view, the close button, which appears at the right side of the view. The close button is based on the protoLargeCloseBox proto.

The following example is a template using protoStatus:

```
theStatus := {_proto: protostatus} // nothing else needed

// base app must include this slot:
declareSelf: 'base
```

protoStatusBar

This proto is used to create a status bar at the bottom of a view. It is identical to protoStatus, except it does not include a close button, as shown in the following graphic.



Note

If you prefer a status bar without the analog clock, see the description of the `newtStatusBarNoClose` and `newtStatusBar` protos on pages 4-55 and 4-56 in Chapter 4, "NewtApp Applications."

The `viewJustify` flags for this view are set so that the status bar always appears at the bottom of its parent view and it always occupies the full width of the parent view. Instantiators are not required to set any slots.

The `protoStatusBar` is based on a view of the `clView` class. The `protoStatusBar` contains the following two child views:

- A small round analog clock that appears at the side left of the view. The clock is based on a view of the `clView` class.
- A digital clock display that slides out from the analog clock when the user taps the analog clock. This view is based on the `protoGlance` proto.

protoTitle

This proto is used to create a title centered in a black rounded rectangle at the top of a view. The following graphic shows an example of a title with an icon placed to the left of it with the `iconTitle` slot.:

The `ViewSetupFormScript` method is defined internally. If you need to use this method, be sure to call the inherited method also (for example, `inherited:?ViewSetupFormScript()`), otherwise the proto may not work as expected.

The `protoTitle` is based on a view of the `clTextView` class. This kind of view is simply a `clParagraphView` that is read-only and it supports no tabs or multistyled text.

Slot descriptions

`viewJustify`	Optional. The default setting is `vjParentCenterH + vjParentTopV + vjCenterV + vjCenterH`.
`viewFormat`	Optional. The default setting is `vfFillBlack + vfRound(3)`.
`viewFont`	Optional. The default font is `ROM_fontSystem10Bold`.
`title`	A string that is the title.
`titleIcon`	Optional. A bitmap frame (like the frame returned from `GetPictAsBits`).
`titleHeight`	Optional. The height of the title view (black rectangle) defaults to the height of the font used. If you want a taller title view, set this slot to a greater value.
`viewTransferMode`	Optional. The default transfer mode is `modeBic`.

The following example is a template using `protoTitle`:

```
myTitle := {...
_proto: protoTitle,
titleHeight: 18,
title: "Preferences"
...}
```

Summary

Protos

Scrollers

protoLeftRightScroller

```
aProtoLeftRightScroller := {
_proto: protoLeftRightScroller,
scrollView: viewTemplate,
scrollRect: view, // extent of scrollable area
dataRect: view, // extent of data in the view
viewRect: view, // extent of visible area
xPos: integer, // initial horizontal coordinate in
               scrollRect
yPos: integer, // initial vertical coordinate in scrollRect
scrollAmounts: array, // line, page, double-click values
pageThreshhold: integer, // lines before page scrolling
ViewScroll2DScript: function, // called when arrows tapped
ViewScrollDoneScript: function, // called when scroll done
SetArrow: function, // set scroll direction
GetArrow: function, // returns scroll direction
...
}
```

protoUpDownScroller

```
aProtoUpDownScroller := {
_proto: protoUpDownScroller,
```

```
scrollView: viewTemplate,
scrollRect: view, // extent of scrollable area
dataRect: view, // extent of data in the view
viewRect: view, // extent of visible area
xPos: integer, // initial horizontal coordinate in
               scrollRect
yPos: integer, // initial vertical coordinate in scrollRect
scrollAmounts: array, // line, page, double-click values
pageThreshhold: integer, // lines before page scrolling
ViewScroll2DScript: function, // called when arrows tapped
ViewScrollDoneScript: function, // called when scroll done
SetArrow: function, // set scroll direction
GetArrow: function, // returns scroll direction
...
}
```

protoHorizontal2DScroller

```
aProtoHorizontal2DScroller := {
_proto: protoHorizontal2DScroller,
scrollView: viewTemplate,
scrollRect: view, // extent of scrollable area
dataRect: view, // extent of data in the view
viewRect: view, // extent of visible area
xPos: integer, // initial horizontal coordinate in
               scrollRect
yPos: integer, // initial vertical coordinate in scrollRect
scrollAmounts: array, // line, page, double-click values
pageThreshhold: integer, // lines before page scrolling
ViewScroll2DScript: function, // called when arrows tapped
ViewScrollDoneScript: function, // called when scroll done
SetArrow: function, // set scroll direction
GetArrow: function, // returns scroll direction
```

```
...
}
```

Buttons and Boxes

protoTextButton

```
aProtoTextButton := {
_proto: protoTextButton,
viewBounds : boundsFrame,
viewFlags : constant,
text: string, // text inside the button
viewFont : fontFlags,
viewFormat : formatFlags,
viewJustify: justificationFlags,
viewTransferMode : constant,
ButtonClickScript: function, // when button is tapped
ButtonPressedScript: function,// while button is pressed
...
}
```

protoPictureButton

```
aProtoTextButton := {
_proto: protoPictureButton,
viewBounds : boundsFrame,
viewFlags : constant,
icon: bitmap, // bitmap to be used as button
viewFormat : formatFlags,
viewJustify: justificationFlags,
ButtonClickScript: function, // when button is tapped
ButtonPressedScript: function,// while button is pressed
...
}
```

protoInfoButton

```
aProtoInfoButton := {
_proto: protoInfoButton,
viewFlags : constant,
viewBounds : boundsFrame,
viewJustify: justificationFlags,
DoInfoAbout: function, // About item selected
DoInfoHelp: function, // Help item selected
DoInfoPrefs: function, // Prefs item selected
GetInfoAuxItems: function, // provides addt'l items
DoInfoAux: function,// addt'l items selected
...
}
```

protoOrientation

```
aProtoOrientation := {
_proto: protoOrientation,
viewFlags : constant,
viewBounds : boundsFrame,
viewJustify: justificationFlags,
...
}
```

protoRadioCluster

```
aProtoRadioCluster := {
_proto: protoRadioCluster,
viewBounds : boundsFrame,
clusterValue: integer, // value of selected button
declareSelf: symbol, // 'cluster – do not change
InitClusterValue: function, // initialize cluster
ViewSetupFormScript: function, // set initial button
```

```
ClusterChanged: function, // called when value changed
SetClusterValue: function, // change selected button
...
}
```

protoRadioButton

```
aProtoRadioButton := {
_proto: protoRadioButton,
viewBounds : boundsFrame,
viewFormat : formatFlags,
text: string, // radio button text label
buttonValue: integer, // identifies button
viewValue: integer, // current value of radio button
...
}
```

protoPictRadioButton

```
aProtoPictRadioButton := {
_proto: protoPictRadioButton,
viewBounds : boundsFrame,
viewFormat : formatFlags,
viewJustify: justificationFlags,
icon: bitmap, // bitmap for picture button
buttonValue: integer, // identifies button
viewValue: integer, // current value of radio button
declareSelf: symbol, // 'base – do not change
ViewDrawScript: function, // to highlight button
...
}
```

protoCloseBox

```
aProtoCloseBox := {
_proto: protoCloseBox,
viewFlags : constant,
viewBounds : boundsFrame,
viewJustify: justificationFlags,
viewFormat : formatFlags,
declareSelf: symbol, // 'base – do not change
ButtonClickScript: function, // called before closing
...
}
```

protoLargeCloseBox

```
aProtoLargeCloseBox := {
_proto: protoLargeCloseBox,
viewFlags : constant,
viewBounds : boundsFrame,
viewJustify: justificationFlags,
viewFormat : formatFlags,
declareSelf: symbol, // 'base – do not change
ButtonClickScript: function, // called before closing
...
}
```

protoCheckbox

```
aProtoCheckbox := {
_proto: protoCheckbox,
viewBounds : boundsFrame,
viewFormat : formatFlags,
viewFont : fontFlags, // font for text label
text: string, // the checkbox label
```

```
buttonValue: integer, // value when box is checked
viewValue: integer, // current value of box
ValueChanged: function, // checkbox value changed
ToggleCheck: function, // toggles checkbox state
...
}
```

protoRCheckbox

```
aProtoRCheckbox := {
_proto: protoRCheckbox,
viewBounds: boundsFrame,
viewFormat: formatFlags,
viewFont: fontFlags, // font for text label
text: string, // the checkbox label
indent: integer, // pixels to indent box
buttonValue: integer, // value when box is checked
viewValue: integer, // current value of box
ValueChanged: function, // checkbox value changed
ToggleCheck: function, // toggles checkbox state
...
}
```

Tabs

protoAZTabs

```
aProtoAZTabs := {
_proto: protoAZTabs,
PickLetterScript: function, // tab is tapped
...
}
```

protoAZVertTabs

```
aProtoAZVertTabs := {
_proto: protoAZVertTabs,
PickLetterScript: function, // tab is tapped
...
}
```

Status Indicators

protoSlider

```
aProtoSlider := {
_proto: protoSlider,
viewBounds: boundsFrame,
viewValue: integer, // gauge value
minValue: integer, // minimum gauge value
maxValue: integer, // maximum gauge value
ViewSetupFormScript: function, // set initial gauge value
ChangedSlider: function, // slider moved
TrackSlider: function, // viewValue changed
...
}
```

protoGauge

```
aProtoGauge := {
_proto: protoGauge,
viewBounds: boundsFrame,
viewValue: integer, // gauge value
minValue: integer, // minimum gauge value
maxValue: integer, // maximum gauge value
gaugeDrawLimits: Boolean, // non-nil for gray background
ViewSetupFormScript: function, // set initial gauge value
```

```
...
}
```

protoLabeledBatteryGauge

```
aProtoLabeledBatteryGauge:= {
_proto: protoLabeledBatteryGauge,
viewBounds: boundsFrame,
...
}
```

clGaugeView

```
aClGaugeView:= {
viewBounds: boundsFrame,
viewClass: clGaugeView,
viewValue: integer, // the value of the gauge
viewFlags: constant,
viewFormat: formatFlags,
minValue: integer, // minimum value of the gauge
maxValue: integer, // maximum value of the gauge
gaugeDrawLimits: Boolean, // non-nil for gray background
ViewChangedScript: function, // gauge dragged
ViewFinalChangeScript: function, // gauge changed
...
}
```

Dates and Time

protoDatePicker

```
aProtoDatePicker := {
_proto: protoDatePicker,
selectedDates: array, // selected date
```

```
DateChanged: function, // called when date is selected
Refresh: function, // update view with new selectedDates
...
}
```

protoDigitalClock

```
aProtoDigitalClock := {
_proto: protoDigitalClock,
viewFlags : constant,
viewBounds : boundsFrame,
viewJustify: justificationFlags,
increment: integer, //minutes to change on tap
time: integer, // initial or current time
wrapping: Boolean, // non-nil to wrap around day boundaries
midnite: Boolean, // non-nil if 0 means midnight tomorrow
Refresh: function, // update clock
TimeChanged: function, // called when time is changed
...
}
```

protoSetClock

```
aProtoSetClock := {
_proto: protoSetClock,
viewBounds: boundsFrame,
viewFlags: constant,
viewFormat: formatFlags,
hours: integer, // value set by hour hand
minutes: integer, // value set by minute hand
TimeChanged: function, // called when time is changed
...
}
```

protoNewSetClock

```
aProtoNewSetClock := {
_proto: protoNewSetClock,
viewBounds: boundsFrame,
viewJustify: justificationFlags,
time: integer, // initial or current time
annotations: array, // four strings to annotate clock face
supressAnnotations: Boolean, // if slot exists, suppress
exactHour: Boolean, //adjust hour markers
Refresh: function, // update clock
TimeChanged: function, // called when time is changed
...
}
```

protoAMPMCluster

```
aProtoAMPMCluster := {
_proto: protoAMPMCluster,
viewBounds: boundsFrame,
viewJustify: justificationFlags,
time: integer, // specify time--required
...
}
```

Miscellaneous Protos

protoBorder

```
aProtoBorder := {
_proto: protoBorder,
viewBounds: boundsFrame,
viewFlags: constant,
viewFormat: formatFlags,
```

```
...
}
```

protoDivider

```
aProtoDivider:= {
_proto: protoDivider,
viewBounds: boundsFrame,
viewFlags: constant,
viewFont: fontFlags, // font for text
viewJustify: justificationFlags,
viewFormat: formatFlags,
title: string, // text on divider bar
titleHeight: string, // height of divider
...
}
```

protoGlance

```
aProtoGlance := {
_proto: protoGlance,
viewBounds: boundsFrame,
viewJustify: justificationFlags,
viewFormat: formatFlags,
viewFont: fontFlags, // font for text
viewEffect: effectFlags,
viewIdleFrequency: integer, // time view to remain open
text: string, //text to appear in view
...
}
```

protoDrawer

```
aProtoDrawer := {
_proto: protoDrawer,
```

```
viewFlags : constant,
viewBounds : boundsFrame,
viewFormat : formatFlags,
viewEffect : effectFlags,
showSound: constant, // sound when drawer opens
hideSound: constant, // sound when drawer closes
...
}
```

protoDragger

```
aProtoDragger := {
_proto: protoDragger,
viewBounds : boundsFrame,
viewFlags : constant,
viewFormat : formatFlags,
noScroll: string, // message to display if no scrolling
noOverview: string, // message to display if no overview
declareSelf: symbol, // 'base — do not change
...
}
```

protoDragNGo

```
aProtoDragNGo := {
_proto: protoDragNGo,
viewBounds : boundsFrame,
viewFlags : constant,
viewJustify : justificationFlags,
viewFormat : formatFlags,
noScroll: string, // message to display if no scrolling
noOverview: string, // message to display if no overview
declareSelf: symbol, // 'base — do not change
```

```
...
}
```

protoFloater

```
aProtoFloater := {
_proto: protoFloater,
viewBounds : boundsFrame,
viewFlags : constant,
viewJustify : justificationFlags,
viewFormat : formatFlags,
viewEffect : effectFlags,
noScroll: string, // message to display if no scrolling
noOverview: string, // message to display if no overview
declareSelf: symbol, // 'base – do not change
...
}
```

protoFloatNGo

```
aProtoFloatNGo := {
_proto: protoFloatNGo,
viewFlags : constant,
viewBounds : boundsFrame,
viewJustify : justificationFlags,
viewFormat : formatFlags,
viewEffect : effectFlags,
noScroll: string, // message to display if no scrolling
noOverview: string, // message to display if no overview
declareSelf: symbol, // 'base – do not change
...
}
```

protoStaticText

```
aProtoStaticText:= {
_proto: protoStaticText,
viewBounds: boundsFrame,
viewFlags: constant,
text: string, //text to display
viewFont: fontFlags,
viewJustify: justificationFlags,
viewFormat: formatFlags,
viewTransferMode: constant,
tabs: array, // up to eight tab-stop positions
styles: array, // font information
...
}
```

protoStatus

```
aProtoStatus := {
_proto: protoStatus,
declareSelf: symbol, // 'base – do not change
...
}
```

protoStatusBar

```
aProtoStatusBar := {
_proto: protoStatusBar,
declareSelf: symbol, // 'base – do not change
...
}
```

protoTitle

```
aProtoTitle := {
_proto: protoTitle,
viewJustify : justificationFlags,
viewFormat : formatFlags,
viewFont : fontFlags,
title: string, // text of title
titleHeight: integer, // height of title
viewTransferMode: constant,
...
}
```

CHAPTER 8

Text and Ink Input and Display

This chapter describes how the Newton system handles text and presents interfaces that you can use to work with text in NewtonScript applications.

The chapter describes the components of Newton text-handling:

- handwritten text input
- keyboard text input
- views for text display
- fonts for text display

The first section of this chapter, “About Text,” describes the basic terms and concepts that you need to understand text processing on the Newton.

The second section, “Using Text,” describes how to use the various input and display components to handle text in your applications.

The third section, “Text Reference,” provides comprehensive reference information about the text-related constants, data structures, views, methods, and functions.

About Text

This section describes the basic concepts, terms, and processes that you need to understand to work with text in your applications.

About Text and Ink

The Newton allows you to process two forms of text input: **ink** and **recognized text**. This section describes these forms of text input.

When the user writes a line of text on the Newton screen, the Newton system software performs a series of operations, as follows:

- The raw data for the input is captured as ink, which is also known as **sketch ink** or **raw ink.**
- The raw ink is stored as a sequence of **strokes** or stroke data.
- If the view in which the ink was drawn is configured for **ink text**, the recognition system groups the stroke data into a series of **ink words**, based on the timing and spacing of the user's handwriting. A user can insert, delete, and move ink words in the same way as recognized text. Ink words can be scaled to various sizes for display and printing. Ink words can also be recognized at a later time, using the process known as **deferred recognition**.
- If the view in which the ink was drawn supports or is configured for text recognition, the ink words are processed by the recognition system into recognized text and displayed in a typeface.

The data describing the handwriting strokes of the ink word are stored as compressed data in a binary object. This **stroke data** can be accessed programmatically, using the stroke bundle methods that are described in Chapter 9, "Stroke Bundles."

The recognition system and deferred recognition are described in Chapter 10, "Recognition."

Note
To provide maximum user flexibility for your applications, you are encouraged to allow ink in all of your views. ◆

Written Input Formats

Ink words can be intermixed with recognized text. These strings, known as **rich strings**, can be used anywhere that you might expect a standard string. Each ink word is encoded as a single character in a rich string, as described in the section "Rich Strings" beginning on page 8-31.

You should use the **rich string format** to store data in a soup, because of its compact representation. You can safely use rich strings with all functions, including the string functions, which are documented in Chapter 20, "Utility Functions".

You use another data format to pair text strings with style data for viewing in text views. This format is described in the section "Text and Styles" beginning on page 8-33.

Caret Insertion Writing Mode

Caret insertion writing mode is a text input mode that the user can enable or disable. When caret insertion mode is disabled, handwritten text is inserted into the view at the location where it was written. When caret insertion writing mode is enabled, handwritten text is inserted at the location indicated by the insertion caret, regardless of where on the screen it was drawn. Caret insertion writing mode is automatically used for keyboard text entry.

To enable or disable caret insertion writing mode, the user selects or deselects the "Insert new words at caret" option from the Text Editing Settings slip. This slip can be displayed by tapping the Options button in the Recognition Preferences slip.

Your applications do not normally need to be aware of whether caret insertion writing mode is enabled or disabled. However, there are a few caret insertion writing mode routines that you can use to implement your own

version of this mode. They are described in the section “Caret Insertion Writing Mode Functions and Methods” beginning on page 8-104.

Fonts for Text and Ink Display

The Newton system software allows you to specify the font characteristics for displaying text and ink in a paragraph view on the screen. The font information is stored in a font specification structure known as a **font spec**. The font specification for built-in ROM fonts can also be represented in a frame as a packed integer. Both of these representations are described in the section “Using Fonts for Text and Ink Display” beginning on page 8-25.

The system provides a number of functions that you can use to access and modify font attributes. These functions are described in the section “Text and Styles” beginning on page 8-33.

About Text Views and Protos

There are a number of views and protos that you can use for displaying text and for receiving text input. For basic information about views, see Chapter 3, “Views.”

The views and protos that you use for text are listed in Table 8-1.

Table 8-1 Views and protos for text input and display

View or Proto	Description
edit view	Use the `clEditView` class for basic text input and display. Objects of this class can display and / or accept text and graphic data. The `clEditView` automatically creates child `clParagraph` views for text input and display and `clPolygon` views for graphic input and display. For more information about this class, see the section "General Input Views" beginning on page 8-9.
paragraph views	Use the `clParagraphView` class to display text or to accept text input. For more information about this class, see the section "Paragraph Views" beginning on page 8-12.

Table 8-1 Views and protos for text input and display (continued)

View or Proto	Description
lightweight paragraph views	If your paragraph view template meets certain criteria, the Newton system automatically creates a lightweight paragraph view for you. Lightweight paragraph views, which are read-only and use only one font, although they can contain ink. These views require significantly less memory than do standard paragraph views. For more information about lightweight paragraph views, see the section "Lightweight Paragraph Views" beginning on page 8-13.
input line protos	You can use one of the input line protos to allow the user to enter a single line of text, as described in the section "Using Input Line Protos" beginning on page 8-14.
structured list views	If you are creating a NewtApp application, you can use `protoListView` to provide the user with a structured input list such as the one used by the built-in Notes application. These input lists contain ink text or recognized text and feature a small circle topic marker and optional checkbox to the left of the text. Structured input list protos are described in the section "Structured List Views" beginning on page 8-18.

About Keyboard Text Input

Your application can provide keyboards and keypads for user text input by creating an object from one of the keyboard view classes or protos:

- You can use the `clKeyboardView` class to provide a keyboard-like array of buttons that the user can tap with the pen to perform an action. This class is described in the section "Keyboard Views" beginning on page 8-35.
- You can also use one of the keyboard protos to create keyboard views in your applications. These protos include the `protoKeyboard`, which creates a keyboard view that floats above all other views. The keyboard protos are described in the section "Keyboard Views" beginning on page 8-35.

The Keyboard Registry

You need to register any custom keyboards or keypads that you create with the Newton system's keyboard registry. Caret insertion writing mode is used whenever the user enters text from a keyboard or keypad. When a registered keyboard or keypad is opened, the system knows to display the insertion caret at the proper location.

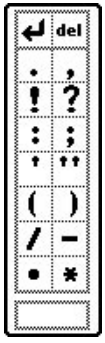
The Newton system also allows you to customize the behavior of the insertion caret and key presses by calling your application-defined methods whenever an action occurs in a registered keyboard or keypad.

For more information about the keyboard registry, see the section "Using the Keyboard Registry" beginning on page 8-46. For more information about the application-defined methods for keyboards and keypads, see the section "Intercepting Keyboard Actions" beginning on page 8-50.

The Punctuation Pop-up

The user can tap the insertion caret to display a Punctuation Pop-up menu. This menu, shown in Figure 8-1, provides an easy way to add punctuation when the user is writing with the stylus.

Figure 8-1 The Punctuation Pop-up menu



Choosing any of the items on the Punctuation Pop-up menu inserts the appropriate character into the text, at the insertion caret. The bent arrow, at the top left, is a carriage return, and the blank box at the bottom indicates a space.

You can override this menu with your own caret pop-up menu, as described in the section “The Caret Pop-up Menu” beginning on page 8-49.

Compatibility

One of the significant advances in software functionality in the Newton 2.0 system is the ability to process ink in most views, which includes deferred recognition and the ability to mix text and ink together in rich string. This expands the behavior provided by Newton 1.x machines, which generally process written input immediately for recognition and then display the resulting word in a typeface.

These additional capabilities generated the need to extend the Recognition menu. The Newton 2.0 Recognition menu adds more input options and replaces the toggling Recognizer button of the Newton 1.x status bar.

The Newton 2.0 system also behaves slightly differently when merging text into paragraph views. When caret insertion writing mode is disabled, paragraphs no longer automatically insert carriage returns or tabs. This is true regardless of whether text or ink words are being entered by the user.

Any ink written on a 1.x machine can be dragged into a Newton System 2.0 paragraph and automatically converted into an ink word.

Notes, text, or ink moved from a Newton 1.x to a Newton with the 2.0 system operates correctly without any intervention. However, the reverse is not true: you cannot insert a a card with 2.0 or later data into a 1.x machine.

The expando protos have become obsolete. These are `protoExpandoShell`, `protoDateExpando`, `protoPhoneExpando`, and `protoTextExpando`. These protos are still supported for 1.x application compatibility, but should not be used in new applications.

Using Text

This section describes how to use various features of text input and display on the Newton and provides examples of some of these features.

Using Views and Protos for Text Input and Display

This section describes the different views and protos that you can use in your applications for text input and display.

General Input Views

The `clEditView` view class is used for a view that can display and / or accept text and graphic data. Views of the `clEditView` class contain no data directly; instead, they have child views that contain the individual data items. Text items are contained in child views of the class `clParagraphView` and graphics are contained in child views of the class `clPolygonView`.

A view of the `clEditView` class includes these features:

- Automatic creation of `clParagraphView` or `clPolygonView` children as the user writes or draws in the view. These child views hold the data the user writes.
- Text and shape recognition, selection, and gestures such as scrubbing, copying to clipboard, pasting from clipboard, duplicating, and others, as controlled by the setting of the `viewFlags` slot. The initial recognition is handled by the `clEditView`. A child `clParagraphView` or `clPolygonView` is created and that child view handles subsequent editing of the data.
- Drag and drop handling. A child view can be dragged (moved or copied) out of the `clEditView` and dropped into another `clEditView` or `clView`, whose child it then becomes.

- Clipboard support. A `clParagraphView` or `clPolygonView` child view can be dragged (moved or copied) to the clipboard, from which it can be pasted into another `clEditView` or `clView`, whose child it becomes.
- Automatic resizing of `clParagraphView` child views to accommodate added input. This feature is controlled by the `vCalculateBounds` flag in the `viewFlags` slot of those child views.
- When caret insertion writing mode is enabled, automatic addition of new words to existing paragraphs when they are written nearby. That is, words gravitate to become part of nearby paragraph views, unless they are written well away from any paragraph.

Views of the class `clEditView` are intended for user input of text, shape, and ink data. Consequently, views of this class expect that any child views have been defined and created at run time, not predefined by templates created in NTK.

If you need to include predefined child views in a `clEditView`, use the `viewSetupChildrenScript` method of the `clEditView` to define the child views and set up the `stepChildren` array. You might need to do this, for example, if you store the data for child views in a soup, and you need to retrieve the data and rebuild the child views at run time.

The default font used for a `clParagraphView` created by a `clEditView` is the font set by the user on the Styles palette in the system.

The default pen width for a `clPolygonView` created by a `clEditView` is the width set by the user on the Styles palette.

The slots of `clEditView` are described in the section “General Input View (clEditView)” beginning on page 8-60.

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

```
editor := {...
   viewClass: clEditView,
   viewBounds: {left:0, top:0, right:200, bottom:200},
   viewFlags: vVisible+vAnythingAllowed,
   viewFormat: vfFillWhite+vfFrameBlack+vfPen(1)+
               vfLinesLtGray,
```

```
   viewLineSpacing: 20,

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

System Messages in Views That Are Created Automatically

When a child view is automatically created by a `clEditView`, the `vNoScripts` flag is set in the `viewFlags` slot of the child view. This flag prevents system messages from being sent to a view.

This behavior is normally desirable for these automatically created views. That is because these views have no system message-handling methods and the system can save time by not sending the messages to them.

If you want to use one of these views in a manner that requires it to receive system messages, you need to remove the `vNoScripts` flag from the `viewFlags` slot of the view.

Creating the Lined Paper Effect in a Text View

A view of the `clEditView` class can appear simply as a blank area in which the user can write information. However, this type of view usually contains a series of horizontal dotted lines, like lined writing paper. The lines indicate to the user that the view accepts input. To create the lined paper effect, you must set the following slots appropriately:

`viewFormat` Must include one of the `vfLines`... flags. This activates the line display.

`viewLineSpacing`
This sets the spacing between the lines, in pixels.

`viewLinePattern`
Optional. Sets a custom pattern that is used to draw the lines in the view. In the `viewFormat` slot editor in NTK, you must also set the Lines item to Custom to signal that you are using a custom pattern. (This sets the `vfCustom<<vfLinesShift` flag in the `viewFormat` slot.)

Patterns are binary data structures, which are described in the next section.

Defining a Line Pattern

You can define a custom line pattern for drawing the horizontal lines in a paragraph view. A line pattern is simply an eight-byte binary data structure with the class `'pattern`.

To create a binary pattern data structure on the fly, use the following NewtonScript trick:

```
myPattern := SetClass( Clone("\uAAAAAAAAAAAAAAAA"),
                                                'pattern );
```

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

Paragraph Views

The `clParagraphView` class is used to display text or to accept text input. It includes these features:

- Text recognition, correction, and editing, such as scrubbing, selection, copying to clipboard, pasting of text from clipboard, duplicating, and other gestures, as controlled by the setting of the `viewFlags` slot.
- Automatic word-wrapping.
- Clipping of text that won't fit in the view. (An ellipsis is shown to indicate text beyond what is visible.)
- Use of ink and different text fonts (styles) within the same paragraph.
- Tab-stop alignment of text.
- Automatic resizing to accommodate added text (when this view is enclosed in a `clEditView`). This feature is controlled by the `vCalculateBounds` flag in the `viewFlags` slot.

- When caret insertion writing mode is enabled, automatic addition of new words written near the view (when this view is enclosed in a `clEditView`). That is, words gravitate to become part of nearby paragraphs, unless they are written away from any other paragraph.

The slots of `clParagraphView` are described in the section "Paragraph View (clParagraphView)" beginning on page 8-63.

Note that you don't need to create paragraph views yourself if you are accepting user input inside a `clEditView`. You simply provide a `clEditView` and when the user writes in it, the view automatically creates paragraph views to hold text.

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

```
dateSample := {...
   viewClass: clParagraphView,
   viewBounds: {left:50, top:50, right:200, bottom:70},
   viewFlags: vVisible+vReadOnly,
   viewFormat: vfFillWhite,
   viewJustify: oneLineOnly,
   text: "January 24, 1994",

   // 8 chars of one font, 3 chars of another, 5 chars
   // of another
   styles: [8, 18434, 3, 12290, 5, 1060865],
...}
```

Paragraph views are normally lined to convey to the user that the view accepts text input. To add the lined paper effect to your paragraph views, see the section "Creating the Lined Paper Effect in a Text View" beginning on page 8-11.

Lightweight Paragraph Views

When you create a template using the `clParagraphView` class, and that template is instantiated into a view at run time, the system may create a

specialized kind of paragraph view object, called a lightweight paragraph view. Lightweight paragraph views have the advantage of requiring much less memory than do standard paragraph views.

The system automatically creates a lightweight paragraph view instead of a standard paragraph view if your template meets the following conditions:

- The view must be read-only, which means that its `viewFlags` slot must contain the `vReadOnly` flag.
- The view must not include any tabs, which means that the template must not contain the `tabs` slot.
- The view must not include multiple font styles, which means that the template must not contain the `styles` slot; however, the view can contain a rich string in its `text` slot.
- The `viewFlags` slot of the view must not contain the following flags: `vGesturesAllowed`, `vCalculateBounds`.

Note

Lightweight paragraph views can contain ink. ◆

All paragraph views look the same after they are instantiated; that is, there is no way to tell whether a particular paragraph view is a standard or a lightweight view.

Once a lightweight paragraph view has been instantiated, you cannot dynamically change the view flags to make it an editable view, or add multistyled text by providing a `styles` slot, since this type of view object doesn't support these features. If you need this functionality for an existing lightweight paragraph view, you'll have to copy the text out of it into an editable paragraph view.

Using Input Line Protos

You can use the input line protos to provide the user with single lines in which to enter data. There are four input line protos that you can use:

- `protoInputLine` is a one-line input field that defines a simple paragraph view in which the text input is left-justified.

- `protoRichInputLine` is the text and ink equivalent of `protoInputLine`.
- `protoLabelInputLine` is a one-line input field that includes a text label and can optionally include a pop-up menu known as a **picker.**
- `protoRichLabelInputLine` is the text and ink equivalent of `protoLabelInputLine`.

protoInputLine

This proto defines a view that accepts any kind of text input and is left-justified. Here is an example of a what a `protoInputLine` looks like on the Newton screen:

The `protoInputLine` is based on a view of the `clParagraph` class. It is provided with no child views.

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

```
myInput := {...
_proto: protoInputLine,
viewJustify: vjParentRightH+vjParentBottomV,
viewLineSpacing: 24,
viewBounds: SetBounds( -55, -33, -3, -3),
...}
```

The slots of the `protoInputLine` are described in the section "protoInputLine" beginning on page 8-66.

protoRichInputLine

This proto works exactly like `protoInputLine`. The only difference is that `protoRichInputLine` allows mixed ink and text input, as determined by the current user recognition preferences.

The slots of `protoRichInputLine` are described in the section "protoRichInputLine" beginning on page 8-67.

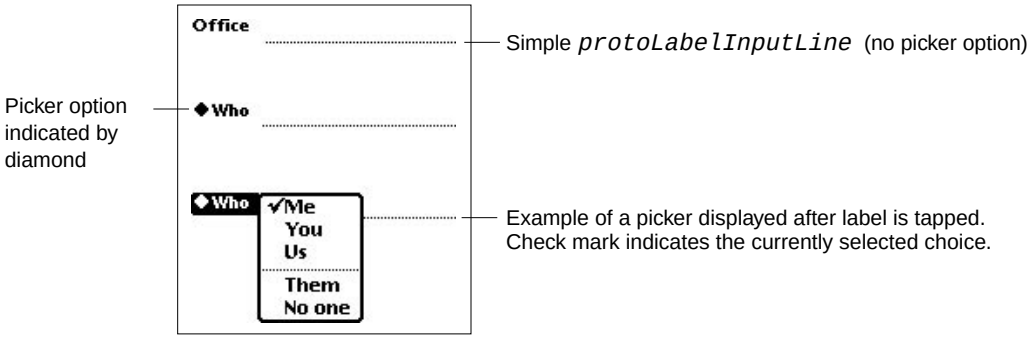
protoLabelInputLine

This proto defines a view that features a label, accepts any kind of input, and is left-justified. The `protoLabelInputLine` can optionally include a picker.

When the `protoLabelInputLine` does include a picker, the user selects a choice from the picker, that choice is entered as the text in the input line, and the choice is marked with a check mark in the picker.

Figure 8-2 shows an example of a `protoLabelInputLine` with and without the picker option:

Figure 8-2 An example of a `protoLabelInputLine`



The `protoLabelInputLine` is based on a view of the `clParagraph` class. It is provided with two child views:

- The `labelLine` child view uses the `protoStaticText` proto. It is used for the static text label and it activates the picker if the proto includes one.
- The `entryLine` child view uses the `protoInputLine` proto. It is used for the input field into which the user writes text. The text value entered into this field is stored into the `text` slot of this view.

You can have your label input line protos remember a list of recent items. To do so, all you need to do is to assign a symbol to the `'memory` slot of your

prototype. This symbol must incorporate your developer signature. The system will automatically maintain the list of recent items for your input line. To access the list, you need to use the same symbol with the `AddMemoryItem`, `AddMemoryItemUnique`, `GetMemoryItems`, and `GetMemorySlot` functions, which are described in Chapter 20, "Utility Functions."

IMPORTANT

You can programmatically access the value of the `text` slot for the `protoLabelInputLine` with the expression `entryLine.text`. If you update the text slot programmatically, you need to call the `SetValue` function to ensure that the view is updated. Here is an example:

```
   SetValue(entryLine, 'text, "new text")]
```

▲

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

```
labelLine := {...
_proto: protoLabelInputLine,
viewBounds: {top: 90, left: 42, right: 194, bottom: 114},
label: "Who",
labelCommands: ["Me", "You", "Us", 'pickseparator,
                "Them", "No one"],
curLabelCommand: 0,
...}
```

The slots of the `protoLabelInputLine` are described in the section "protoLabelInputLine" beginning on page 8-67.

protoRichLabelInputLine

This proto works exactly like `protoLabelInputLine`. The only difference is that `protoRichLabelInputLine` allows mixed ink and text input, as determined by the current user recognition preferences.

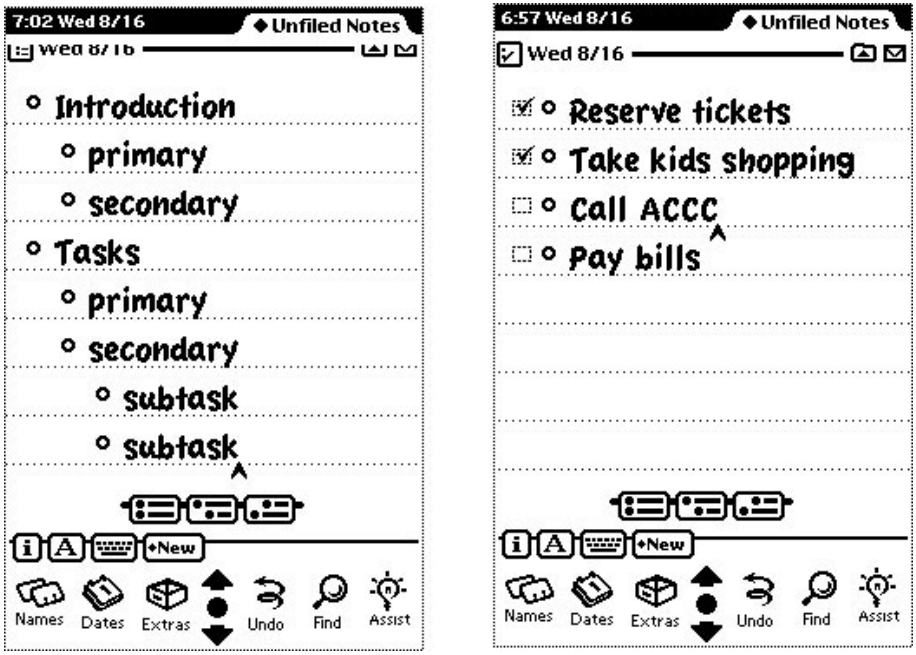
The slots of the `protoRichLabelInputLine` are described in the section “protoRichLabelInputLine” beginning on page 8-71.

Structured List Views

You can use `protoListView` to create structured list views for your applications. This proto consists of a sequence of paragraphs called topics, each of which contains ink words or text. To the left of each topic is a small circle topic marker and an optional checkbox.

These topics can also be indented to indicate a level of hierarchy. The built-in Notes application uses `protoListView` to implement its Outline and Checklist stationery. Figure 8-3 shows these as an example of a structured list view.

Figure 8-3 An Outline and Checklist protoListView



Note
Another version of `protoListView` can be seen in the built-in Calendar's To Do List. In this structured list, the topics have checkboxes and priority markers instead of topic markers. ◆

When a user collapses or expands text in a structured list view, the system automatically creates and destroys paragraph views for the topics that have been exposed or hidden.

Users of structured list views can perform the following actions:

- Remove a topic and all of its subtopics by scrubbing over its marker.
- Create a new topic in the middle of the list.
- Expand the outline automatically to show text that has been found in a collapsed part of the list.

Here is an example of a structured list view:

```
DefineImport:=func(name, realName, major, minor, entries)
   begin
   if not vars.UnitImportTable exists then
      vars.UnitImportTable := [];
   local tableNum := Length(UnitImportTable) + 2;
   AddArraySlot(UnitImportTable,
      {name:realName, major:major, minor:minor});
   local constFrame := {};
   foreach entryNum, entry in entries do
      begin
      local importRef := (tableNum <<12) + entryNum;
      constFrame.(entry) :=   // generate magic pointer
                  call Compile("@" & importRef) with ();
      end;
   DefConst(name, constFrame);
   end;

call DefineImport with (
```

```
        'ListViewUnit,          //name we want to use
        'ListView,              // the real unit name
        1,                      // major version
        0,                      // minor version
        '[protoListView, protoNewAuntButton,
          protoNewTopicButton, protoNewDaughterButton]
        );

        // slots in partFrame will be sucked into the
        //  real part frame
partFrame := {};
PartFrame._ImportTable := UnitImportTable;
```

The `protoListView` contains a slot named `topics`, which is an array of topic frames. Each topic frame specifies information about a single topic, including the text for the topic and the indentation level. Here is an example of a topic frame:

```
{  level:1,
   mtgDone:TRUE,
   mtgPriority:3,
   hideCount:0,
   text: ""
   styles: []
   viewBounds: {}
}
```

The slots of the topic frame are used as follows:

`level`	Optional. The indentation level. The default value, `1`, specifies the left margin.
`mtgDone`	Optional. Use `true` to indicate that the topic has a check, and `nil` to indicate that it does not have a check.
`mtgPriority`	Optional. The priority of this topic, as an integer value between `0` and `4`. This value corresponds to the value in the priorityItems slot, which is described in the section

	"Structured List Proto (protoListView)" beginning on page 8-71.
`hideCount`	Optional. The number of collapsed ancestors. The default value, `0`, specifies that the topic is visible.
`text`	The text for the topic.
`styles`	The text styles.
`viewBounds`	Do not change this slot.

You should use the method `MakeTopicFrame` to create a topic frame. The `MakeTopicFrame` method, which is documented on page 8-77, creates frames that have shared memory maps.

The slots of `protoListView` are described in the section "Structured List Proto (protoListView)" beginning on page 8-71.

If you are using `protoListView` in a NewtApp application, the data from the topics frame is automatically loaded from the entry target in `viewSetupFormScript`. If you are using `protoListView` from outside of a NewtApp application, you must load the topics yourself.

Displaying Text and Ink

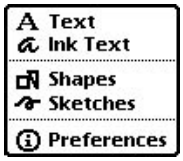
In addition to knowing about the views and protos that you can use for displaying text and ink, you need to understand how text and ink are displayed. This involves the use of fonts, text styles, and rich strings. This section describes these objects and how you can use them in your applications to control the display of text and ink.

Text and Ink in Views

When the user draws with the pen on the Newton screen, the data for the pen input is captured as ink, which is also known as sketch ink or raw ink.

What happens with the raw ink depends upon the configuration of the view in which the input action was performed and upon the choices that the user made in the Recognition menu. The view configuration is defined by the view flags and the (optional) recognition configuration (`recConfig`) frame of the view. The Recognition menu is shown in Figure 8-4.

Figure 8-4 The Recognition menu



When the `viewFlag` input flags and the `recConfig` frame of the view are set to accept both text and ink, the Recognition menu choices control what kind of data is inserted into the paragraph view. Note that you can limit the choices that are available in the Recognition menu of your application, though this is rarely necessary or advisable.

The Recognition menu, recognition view flags, and the recognition configuration frame are described in Chapter 10, “Recognition.”

Mixing Text and Ink in Views

Some views require textual input and cannot accept ink words. The global recognition controls are not used by these text-only views, in which writing is always recognized and inserted as text. If the user drops an ink word into a text-only field, the ink word is automatically recognized before control is returned to the user.

Edit views can handle both ink words and sketch ink. If an edit view receives an ink word, the edit view either merges that word into an existing paragraph view or creates a new view for the ink word. If an edit view receives sketch ink, the edit view creates a polygon view for the ink drawing.

You can also create fields that only accept ink words. However, if the user drops recognized text into such a field, the recognized text remains recognized text.

You can set a paragraph view to accept either text or ink input with the following settings:

```
viewClass: clParagraphView,
   viewFlags:  vVisible + vClipping + vClickable +
               vGesturesAllowed + vCharsAllowed +
               vNumbersAllowed,
   recConfig: rcInkOrText
```

Note
The view flags are described in Chapter 3, "Views." The recognition view flags are described in Chapter 10, "Recognition." ◆

Although raw ink is intended mostly for drawing, the user can still write with raw ink by choosing "Sketches" from the Recognition menu. The recognizer automatically segments raw ink into ink words. The raw ink can subsequently be recognized, using deferrred recognition. Unlike ink text, raw ink is not moved after it is written.

When raw ink from a 1.x system is dragged into a paragraph view, each piece of ink is automatically converted into an ink word. This conversion is not reversible.

Note
You can use one of two representations for text and ink that are mixed together. The first and more common representation is as a rich string, as described in the section "Rich Strings" beginning on page 8-31. The second representation, used in paragraph views, is as a text string with a corresponding series of matching style runs. This representation, which is used for editing operations in paragraph views, is described in the section "Text and Styles" beginning on page 8-33. ◆

Ink Word Scaling and Styling

Ink words are drawn using the pen thickness that the user specifies in the Styles menu. After the ink words are drawn, they are scaled by the system software. The scaling value is specified in the Text Editing Settings menu,

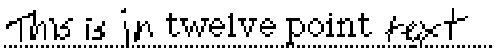
which the user can access by choosing Preferences from the Recognition menu.

The standard values for scaling ink words are fifty percent, seventy-five percent, and one hundred percent. After the system performs scaling, it assigns a font style and size to the ink word. The initial style is plain. The initial size is proportional to the x-height of the ink word, as estimated by the recognizer. This initial size is defined so that an ink word of a certain size will be roughly the same size as a text word displayed in a font of that size.

You can modify the size at which ink words are displayed in two ways: you can change the scaling percentage or you can change the font size. For example, suppose that you draw an ink word and the system calculates its font size, as written, at 36 point. If your ink text scaling is set to 50 percent, the ink word will be displayed at half of the written size, which makes its font size 18 point. If you you subsequently change the scaling of that ink word to 100 percent, its font size will change to 36 point.

Ink words drawn at a specific point size are roughly the same size as text words drawn in a typical font of the same point size. For example, an ink word of size 12 is drawn at roughly the same size as a text word in a typical 12-point font, as shown in Figure 8-5. The ink words in Figure 8-5 were first scaled to fifty percent of their written size.

Figure 8-5 Resized and recognized ink



If the user applies deferred recognition to the ink words, the recognized text is displayed in the current font family, size, and style, as specified in the Styles menu.

Note
There is a maximum ink word size. Ink words are scaled to the smaller of what would be produced by the selected scaling percentage or the maximum size. ◆

Constraining Font Style in Views

You can override the use of styles in a paragraph view so that all of the text in the paragraph is displayed with a certain font specification. To do this, use the `viewFont` slot of the paragraph view along with two of the text view flags.

If you include `vFixedTextStyle` in the view flags for a paragraph view, all recognized text in the view is displayed using the font family, point size, and character style specified for `viewFont`. This is the normal behavior for input fields.

If you include `vFixedInkTextStyle` in the view flags for a paragraph view, all ink words in the view are displayed using the point size and character style specified for `viewFont`. Note that the font family does not affect the display of ink words.

Note
Using the `vFixedTextStyle` or `vFixedInkTextStyle` flags does not modify the `'styles` slot of the view. However, if you use either of these flags, the system does not allow the user to change the text style for your paragraph view. ◆

The text view flags are described on page 8-51.

Using Fonts for Text and Ink Display

Whenever recognized text is drawn on the Newton screen, the system software examines the font specification associated with the text to determine how to draw the text. The font specification includes the font family name, the font style, and the point size for the text. You can specify a font with a font frame or with a packed integer; both of these formats are described in this section.

The constants that you can use in font specifications are shown in the section "Font Constants" beginning on page 8-52.

The Font Frame

A font frame has the following format:

`{family:` *familyName*`, face:` *faceNumber*`, size:` *pointSize*`}`

For *familyName*, you can specify a symbol corresponding to one of the available built-in fonts, which are shown in Table 8-2.

Table 8-2 Font family symbols

Symbol	Font Family
`'espy`	Espy (system) font
`'geneva`	Geneva font
`'newYork`	New York font
`'handwriting`	Casual (handwriting) font

For *faceNumber*, you can specify a combination of the values shown in Table 8-3:

Table 8-3 Font style (face) values

0	Normal font
1	Bold font
2	Italic font
4	Underline font

Table 8-3 Font style (face) values

8	Outline font
128	Superscript font
256	Subscript font

For *pointSize*, use an integer that specifies the point size value.

The Packed Integer Font Specification

You can specify a font in one 30-bit integer. A packed integer font specification uses the lower 10 bits for the font family, the middle 10 bits for the font size, and the upper 10 bits for the font style. Since only the ROM fonts have predefined font family number constants, you can only specify ROM fonts in a packed value.

Using the Built-in Fonts

The system provides several constants you can use to specify one of the built-in fonts. These constants are listed in Table 8-4. The fonts shown in the table can be specified by the constant (usable at compile time only), by their font frame, or by an integer value that packs all of the font information into an integer (sometimes this is what you see at run time if you examine a `viewFont` slot in the NTK Inspector).

Table 8-4 Built-in font constants

Constant	Font Frame	Integer Value
`ROM_fontsystem9`	`{family:'espy, face:0, size:9}`	9216
`ROM_fontsystem9bold`	`{family:'espy, face:1, size:9}`	1057792
`ROM_fontsystem9underline`	`{family:'espy, face:4, size:9}`	4203520

Table 8-4 Built-in font constants (continued)

Constant	Font Frame	Integer Value
ROM_fontsystem10	{family:'espy, face:0, size:10}	10240
ROM_fontsystem10bold	{family:'espy, face:1, size:10}	1058816
ROM_fontsystem10underline	{family:'espy, face:4, size:10}	4204544
ROM_fontsystem12	{family:'espy, face:0, size:12}	12288
ROM_fontsystem12bold	{family:'espy, face:1, size:12}	1060864
ROM_fontsystem12underline	{family:'espy, face:4, size:12}	4206592
ROM_fontsystem14	{family:'espy, face:0, size:14}	14336
ROM_fontsystem14bold	{family:'espy, face:1, size:14}	1062912
ROM_fontsystem14underline	{family:'espy, face:4, size:14}	4208640
ROM_fontsystem18	{family:'espy, face:0, size:18}	18432
ROM_fontsystem18bold	{family:'espy, face:1, size:18}	1067008
ROM_fontsystem18underline	{family:'espy, face:4, size:18}	4212736
simpleFont9	{family:'geneva, face:0, size:9}	9218
simpleFont10	{family:'geneva, face:0, size:10}	10242

Table 8-4 Built-in font constants (continued)

Constant	Font Frame	Integer Value
`simpleFont12`	`{family:'geneva, face:0, size:12}`	12290
`simpleFont18`	`{family:'geneva, face:0, size:18}`	18434
`fancyFont9` or `userFont9`	`{family:'newYork, face:0, size:9}`	9217
`fancyFont10` or `userFont10`	`{family:'newYork, face:0, size:10}`	10241
`fancyFont12` or `userFont12`	`{family:'newYork, face:0, size:12}`	12289
`fancyFont18` or `userFont18`	`{family:'newYork, face:0, size:18}`	18433
`editFont10`	`{family:'handwriting, face:0, size:10}`	10243
`editFont12`	`{family:'handwriting, face:0, size:12}`	12291
`editFont18`	`{family:'handwriting, face:0, size:18}`	18435

The integers in Table 8-4 are derived by packing font family, face, and size information into a single integer value. Each NewtonScript integer is 30 bits in length. In packed font specifications, the lower 10 bits are used to hold the font family, the middle 10 bits are used to hold the font size, and the upper 10 bits are used to hold the font style.

These three parts added together specify a single font in one integer value. You can use the constants listed in Table 8-5 at compile time to specify all of the needed information. To use them, add one constant from each category together to yield a complete font specification. At run time, of course, you'll need to use the integer values.

Table 8-5 Font packing constants

Constant	Value	Description
Font Family		
(none defined)	0	Identifies the System font (Espy)
`tsFancy`	1	Identifies the New York font
`tsSimple`	2	Identifies the Geneva font
`tsHWFont`	3	Identifies the Casual (Handwriting) font
Font Size		
`tsSize(`*pointSize*`)`	*pointSize* << 10	Specify the point size of the font in *pointSize*
Font Face		
`tsPlain`	0	Normal font
`tsBold`	1048576	Bold font
`tsItalic`	2097152	Italic font
`tsUnderline`	4194304	Underlined normal font
`tsOutline`	8388608	Outline font
`tsSuperScript`	134217728	Superscript font
`tsSubScript`	268435456	Subscript font

You can use the `MakeCompactFont` function to create a packed integer value from a specification of the font family, font size, and font face. You can only specify ROM fonts with the packed integer format. Here is an example:

```
fontValue := MakeCompactFont('tsSimple, 12, tsItalic)
```

The `MakeCompactFont` function is described on page 8-88.

Rich Strings

You can use rich strings to store text strings and ink into a single string. If you application supports user-input text or ink, you can use rich strings to represent all user data. You can convert between the text and styles pairs in paragraph views and rich strings. Text and styles pair are described in section "Text and Styles" beginning on page 8-33.

Rich strings are especially useful for storing text with embedded ink in a soup. You can use the rich string functions, described in "Rich String Functions" beginning on page 8-33, to work with rich strings.

The system software automatically handles rich strings properly, including their use in performing the following operations:

- screen display
- sorting and indexing
- concatenation with standard functions such as `StrConcat` and `ParamStr`, which are described in Chapter 20, "Utility Functions."
- measuring

Important Rich String Considerations

Although the Newton system software allows you to use rich strings anywhere that plain strings are used, there are certain considerations of which you must be aware when using rich strings. These include:

- Do not use functions that are not rich-string-aware. These include the `Length`, `SetLength`, `BinaryMunger`, and `StuffXXX` functions.
- Use the `StrLen` function to find the length of a string.
- Use the StrMunger function to perform operations that modify the length of a string, such as appending characters to or deleting characters from a string.
- You cannot assume that a rich string contains a string terminator character. And the string terminator character is definitely not the last character in a rich string.

- You cannot truncate a rich string by inserting a string terminator character into the string.
- You need to exercise caution when assigning characters into a rich string, due to the presence of ink placeholder characters.
- Do not use undocumented string functions, which are not guaranteed to work with rich strings.

Using the Rich String Storage Format

Ink data is embedded in rich strings by inserting a placeholder character in the string for each ink word. Data for each ink word is stored following the string terminator character.

Each ink word is represented in the text portion of the rich string by a special character: either `kInkChar` (`0xF700`) or `kParaInkChar` (`0xF701`). These are reserved Unicode character values.

The ink data for all ink words in the string follows the string terminator character. The final 32 bits in a rich string encode information about the rich string.

Automatic Conversion of Rich Strings

Text is automatically converted from the rich string format to a text/styles pair whenever a paragraph is opened and whenever the `SetValue` function is called.

When a paragraph view is opened, the `'text` slot is first examined to determine whether or not the text contains any embedded ink. If so, new versions of the view's `'text` and `'styles` slots are generated and placed in the context frame of the view.

When `SetValue` is called with a string parameter that is a rich string, it is automatically decoded into a text and style pair. The result is stored in the context frame of the paragraph view.

Rich String Functions

You can use the rich string functions to convert and work with rich strings. Each of these functions, which are shown in Table 8-6, is described in the section “Rich String Functions and Methods” beginning on page 8-91.

Table 8-6 Rich string functions

Function or Method Name	Description
`MakeRichString`	Converts the data from two slots into a rich string. `MakeRichString` uses the text from the '`text` slot of the view and the `styles` array from the '`styles` slot of the view.
`DecodeRichString`	Converts a rich string into a frame containing a `'text` slot and a `'styles` slot. These slots can be placed in a paragraph view for editing or viewing.
`ExtractRangeAsRichString`	Returns a rich string for a range of text from a paragraph view.
`IsRichString`	Determines if a string is a rich string (i.e. contains ink).
`view:GetRichString`	Returns the text from a paragraph view as a rich string or plain string, depending on whether the paragraph view contains any ink.
`StripInk`	Strips any ink from a rich string. Either removes the ink words or replaces each with a specified replacement character or string.

Text and Styles

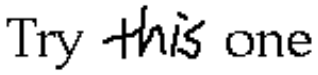
Within a paragraph view, text is represented in two slots: the `'text` slot and the `'styles` slot. The `'text` slot contains the sequence of text characters in

the paragraph, including an instance of the `kParaInkChar` placeholder character (`0xF701`) for each ink word.

The `'styles` slot specifies how each **text run** is displayed in the paragraph. A text run is a sequence of characters that are all displayed with the same font specification. The `'styles` slot consists of an array of alternating length and style information: one length value and one style specification for each text run. For ink words, the length value is always `1`, and the style specification is a binary object that contains the ink data.

For example, consider the paragraph text shown in Figure 8-6.

Figure 8-6 A paragraph view containing an ink word and text



In the paragraph view shown in Figure 8-6, the `'text` slot contains the following sequence of Unicode characters:

```
'T' 'r' 'y' ' ' 0xF701 'o' 'n' 'e'
```

The `'styles` slot for this paragraph consists of the following array:

```
styles: [4, 12289,   1, <inkData, length 42>, 4, 12289]
```

The first pair of values in the array, (`4, 12289`), covers the word “Try” and the space that follows it. The length value, `4`, specifies that the text run consists of four characters.The packed integer font specification value `12289` specifies plain, 12-point, New York.

The second pair of values in the array, (`1, inkData`), covers the ink word. The length value is `1`, which is always the case for ink words. The value `inkData` is a binary object that contains the compressed data for the handwritten “this” that is part of the text in the paragraph view. The data is automatically extracted from the tablet data as part of a preliminary recognition process that precedes word recognition.

The third and final pair of values in the `'styles` slot array, `(4, 12289)`, covers the word “one” and the space that precedes it. This text run is 4 characters long and is displayed 12 points high in the plain version of the New York font family.

Note
The packed integer font specification values are shown in Table 8-5 on page 8-30. ◆

Using Keyboards

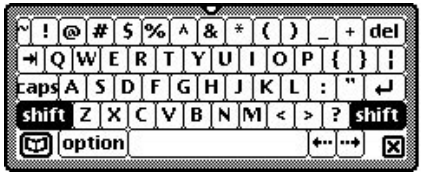
You can provide the user with on-screen keyboard input in your applications using the built-in keyboard views. You can also define new keyboard views and register them with the system, which makes these keyboards available for caret input.

Keyboard Views

There are four different keyboards built into the system root view. Each of the built-in keyboards can be accessed as a child of the root with a symbol.

To use the full alphanumeric keyboard, which is shown in Figure 8-7, use the symbol `'alphaKeyboard`.

Figure 8-7 The built-in alphanumeric keyboard



To use the numeric keyboard, which is shown in Figure 8-8, use the symbol `'numericKeyboard`.

Figure 8-8 The built-in numeric keyboard

To use the phone keyboard, which is shown in Figure 8-9, use the symbol `'phoneKeyboard`.

Figure 8-9 The built-in phone keyboard

To use the time and date keyboard, which is shown in Figure 8-10, use the symbol `'dateKeyboard`.

Figure 8-10 The built-in time and date keyboard

An on-screen keyboard can be opened by the user with a double tap on an input field. The kind of keyboard displayed is determined by what type of input is recognized by the field. For example, a field in which only numbers are recognized would use the numeric keyboard. The user can also open a keyboard from the corrector pop-up list, which appears when you correct a recognized word.

If you want to open one of these keyboards programmatically, use code like this to send it the `Open` message:

```
getroot().alphaKeyboard:open()
```

The keystrokes entered by the user are sent to the current key receiver view. There can be only one key receiver at a time, and only views of the classes `clParagraphView` and `clEditView` can be key receiver views. When a keyboard is open, acaret is shown in the key receiver view at the location where characters will be inserted.

The keyboard views are based on `clKeyboardView`, which is described in the section “Keyboard View (clKeyboardView)” beginning on page 8-95.

Using Keyboard Protos

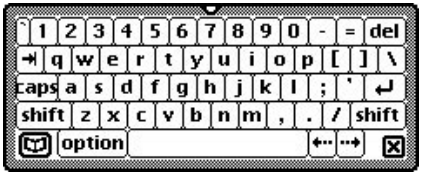
You can use the keyboard protos to provide users of your applications with on-screen keyboards with which to enter text. There are four keyboard protos available:

- `protoKeyboard` provides a standard keyboard view that floats above all other views.
- `protoKeypad` allows you to define a customized floating keyboard.
- `protoKeyboardButton` includes a keyboard button in a view.
- `protoSmallKeyboardButton` includes a small keyboard button in a view.

protoKeyboard

This proto is used to create a keyboard view that floats above all other views. It is centered within its parent view and appears in a location that won't obscure the key-receiving view (that is, the view to which the keystrokes from the keyboard are to be sent). The user can drag the keyboard view by its drag-dot to a different location, if desired. Figure 8-11 shows an example of what a `protoKeyboard` looks like on the screen.

Figure 8-11 An example of a `protoKeyboard`



This proto enables the caret (if it is not already visible) in the key-receiving view while the keyboard is displayed. Characters corresponding to tapped keys are inserted in the key-receiving view at the insertion bar location. The caret is disabled when the keyboard view is closed.

This proto is used in conjunction with `protoKeypad` to implement a floating keyboard. This proto defines the parent view, and `protoKeypad` is a child view that defines the key characteristics.

Here is an example of a template using `protoKeyboard`.

protoKeypad

This proto is used to define key characteristics for a keyboard view (`clKeyboardView` class). It also contains functionality that automatically registers an open keyboard view with the system. If you want to get this behavior in your custom keyboard, use `protoKeypad`.

You use this proto along with `protoKeyboard` to implement a floating keyboard. The view using the `protoKeypad` proto should be a child of the view using the `protoKeyboard` proto.

protoKeyboardButton

This proto is used to include the keyboard button in a view. This is the same keyboard button that is shown on the status bar in the notepad. Tapping the button causes the on-screen keyboard to appear. If the keyboard is already displayed, a picker listing available keyboard types is displayed. The user can tap one to open that keyboard.

Figure 8-12 shows an example of the keyboard button:

Figure 8-12 The keyboard button

protoSmallKeyboardButton

This proto is used to include a small keyboard button in a view. Tapping the button causes the on-screen keyboard to appear. If the keyboard is already displayed, a picker listing available keyboard types is displayed. The user can tap one to open that keyboard.

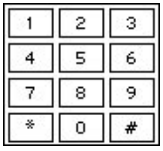
Figure 8-13 shows an example of the small keyboard button:

Figure 8-13 The small keyboard button

Defining Keys in a Keyboard View

When you define a keyboard view, you need to specify the appearance and behavior of each key in the keyboard. This section presents the definition of an example keyboard view, which is shown in Figure 8-14.

Figure 8-14 A generic keyboard view



Here is the view definition of the keyboard shown in Figure 8-14. The values in the row arrays are explained in the remainder of this section.

```
row0 := [ keyVUnit, keyVUnit,
 "1",1, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite,
 "2",2, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite
 "3",3, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite ];

row1 := [ keyVUnit, keyVUnit,
 "4",4, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite,
 "5",5, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite,
 "6",6, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite ];
```

```
row2 := [ keyVUnit, keyVUnit,
 "7",7, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite,
 "8",8, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite,
 "9",9, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite ];
row3 := [ keyVUnit, keyVUnit,
 "*",$*, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite,
 "0",0, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite,
 "#",$#, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite ];

keypad := {...
   viewClass: clKeyboardView,
   viewBounds: {left:65, top:65, right:153, bottom:145},
   viewFlags: vVisible+vClickable+vFloating,
   viewFormat: vfFrameBlack+vfFillWhite+vfPen(1),
   keyDefinitions: [ row0, row1, row2, row3 ],  // defined above
   keyPressScript: func (key)
      begin
      Print("You pressed " & key);
      end,
...}
```

The Key Definitions Array

Each keyboard view contains a key definitions array, which determines the layout of the individual keys in the keyboard. The key definitions array is an array of rows. Each row is an array of values that looks like this:

```
row0 := [ rowHeight, rowMaxKeyHeight,
  key0Legend, key0result, key0Descriptor,
  key1Legend, key1result, key1Descriptor,
  key2Legend, key2result, key2Descriptor,
  ...
  ]
```

The first two elements describe the height to allot for the row (*rowHeight*) and the height of the tallest key in the row (*rowMaxKeyHeight*), in key units . These two measurements are often the same, but they may differ. Key units are described in the section "Key Dimensions" beginning on page 8-45.

Next in the row array is a series of three elements for each key in the row:

- *keyLegend*
- *keyResult*
- *keyDescriptor*

These values are described in the following sections.

The Key Legend

The key legend specifies what appears on the keycap. It can be one of the following types of data:

- `nil`, in which case the key result is used as the legend.
- A string, which is displayed centered in the keycap.
- A character constant, which is displayed centered in the keycap.
- A bitmap object, which is displayed centered in the keycap.
- An integer. The number is displayed centered in the keycap and is used directly as the key result, unless the `keyResultsAreKeycodes` slot is set to `true`, as described in the next section.
- A method. The method is evaluated and its result is treated as if it had been specified as the legend.
- An array. An element of the array is selected and treated as one of the above data types. The index of the array element is determined by the value of the `keyArrayIndex` slot (which can be changed dynamically). Note that arrays of arrays are not allowed here, but an array can include any combination of other data types.

The Key Result

The key result is the value that is returned when the key is pressed. This value is passed as a parameter to the `keyPressScript` method. If this

method doesn't exist, the result is converted (if possible) into a sequence of characters that are posted as key events to the key receiver view.

The key result element can be one of the following types of data:

- A string, character constant, or bitmap object, which is simply returned.
- An integer, which is returned. Alternately, if the `keyResultsAreKeycodes` slot is set to `true`, the integer is treated as a key code. In this case, the character corresponding to the specified key code is returned. If you are using keycodes, make sure that you register your keyboard by including the `kKbdUsesKeycodes` view flag.

  See Figure 8-15 on page 8-44 for the numeric key codes returned by each of the keys on a keyboard.
- A method. The method is evaluated and its result is treated as if it had been specified as the result.
- An array. An element of the array is selected and treated as one of the above data types. The index of the array element is determined by the value of the `keyArrayIndex` slot (which can be changed dynamically). Note that arrays of arrays are not allowed, but an array can include any combination of other data types.

Figure 8-15 Keyboard codes

The Key Descriptor

The appearance of each key in a keyboard is determined by its key descriptor. The key descriptor is a 30-bit value that determines the key size, framing, and other characteristics. The descriptor is specified by combining any of the constants shown in Table 8-7.

Table 8-7 Key descriptor constants

`keySpacer`	Nothing is drawn in this space; it is a spacer, not a key.
`keyAutoHilite`	Highlight this key when it is pressed.
`keyInsetUnit`	Inset this key's frame a certain number of pixels within its space. Multiply this constant by the number of pixels you want to inset, from 0–7 (for example, `keyInsetUnit*3`).

Table 8-7 Key descriptor constants (continued)

`keyFramed`	Specify the thickness of the frame around the key. Multiply this constant by the number of pixels that you want to use for the frame thickness, from 0-3.
`keyRoundingUnit`	Specify the roundedness of the frame corners. Multiply this constant by the number of pixels that you want to use for the corner radius, from 0-15, zero being square.
`keyLeftOpen`	No frame line is drawn along the left side of this key.
`keyTopOpen`	No frame line is drawn along the top side of this key.
`keyRightOpen`	No frame line is drawn along the right side of this key.
`keyBottomOpen`	No frame line is drawn along the bottom side of this key.

`keyHUnit`
`keyHHalf`
`keyHQuarter`
`keyHEighth`
	A combination of these four constants specifies the horizontal dimension of the key in units. For details, see the next section.

`keyVUnit`
`keyVHalf`
`keyVQuarter`
`keyVEighth`
	A combination of these four constants specifies the vertical dimension of the key in units. For details, see the next section.

Key Dimensions

The width and height of keys are specified in units, not pixels. A key unit is not a fixed size, but is used to specify the size of a key relative to other keys. The width of a unit depends on the total width of all keys in the view and on the width of the view itself. Key widths and heights can be specified in whole units, half units, quarter units, and eighth units.

When it is displayed, the whole keyboard is scaled to fit entirely within whatever size view bounds you specify for it.

To fit the whole keyboard within the width of a view, the total unit widths are summed for each row, and the scaling is determined based on the widest row. This row is scaled to fit within the view width, giving an equal pixel width to each whole key unit. A similar process is used to scale keys vertically to fit within the height of a view.

Fractional key units (half, quarter, eighth), when scaled, must be rounded to an integer number of pixels, and thus may not be exactly the indicated fraction of a whole key unit. For example, if the keys are scaled to fit in the view bounds, a whole key unit ends up to be 13 pixels wide. This means that a key specified to have a width of $1\frac{3}{8}$ units (`keyHUnit+keyHEighth*3`) is rounded to 13 + 5, or 18 pixels, which is not exactly $1\frac{3}{8}$*13.

Key dimensions are specified by summing a combination of horizontal and vertical key unit constants within the `keyDescriptor`. For example, to specify a key that is $2\frac{3}{4}$ units wide by $1\frac{1}{2}$ units high, specify these constants for `keyDescriptor`:

```
keyHUnit*2 + keyHQuarter*3 + keyVUnit + keyVHalf
```

Using the Keyboard Registry

If your application includes its own keyboard, you need to register it with the system keyboard registry. This makes it possible for the system to call any keyboard-related functions that you have defined and to handle the insertion caret properly.

You use the `RegisterOpenKeyboard` method of a view to register a keyboard for use with that view.

You use the `UnregisterOpenKeyboard` method of a view to remove the keyboard view from the registry. If the insertion caret is visible, calling this method hides it.

Note
The system automatically unregisters the keyboard when the registered view is hidden or closed. You do not need to call the `UnregisterOpenKeyboard` method in these cases. ◆

You can use the `OpenKeypadFor` function to open a context-sensitive keyboard for a view. This function first attempts to open the keyboard defined in the view's `_keyboard` slot. If the view does not define a keyboard in that slot, `OpenKeypadFor` determines if the view allows only a single type of input, such as date, time, phone number, or numbers. If so, `OpenKeypadFor` opens the appropriate built-in keyboard for that input type. If none of these other conditions are met, `OpenKeypadFor` opens the `alphaKeyboard` keyboard for the view.

Theses methods and functions, as well as several others that you can use with the keyboard registry in your applications, are described in the section "Keyboard Registry Functions and Methods" beginning on page 8-102.

Defining Tabbing Orders

You can define the tabbing order for an input view with the `_tabChildren` slot, which contains an array of view paths.

Each view path must specify the actual view that accepts the input. An example of a suitable path is shown here:

```
'myInputLine, 'myLabelInputLine.entryLine
```

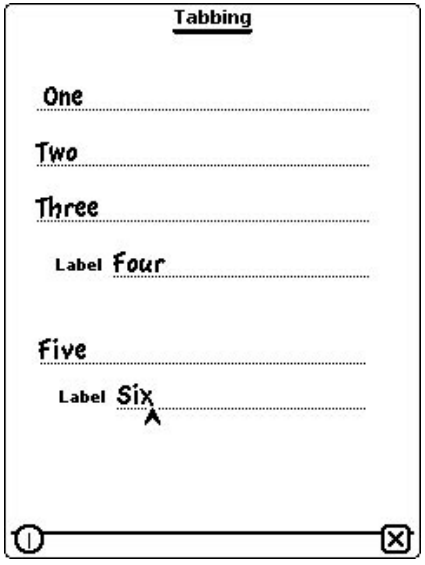
When the user tabs through this list, it loops from end to beginning and, with reverse-tabbing, from beginning to end.

You can use the `_tabParent` slot to inform the system that you want tabbing in a view restricted to that view. Each view in which `_tabParent` is non-`nil` defines a tabbing context. theThis makes it possible for you to have several views on the screen at once with independent tabbing within each view. In this case, the user must tap in another view to access the tabbing order in that view.

For example, in Figure 8-16, there are two independent tabbing orders. The first consists of the input lines that contain the text "One", "Two", "Three",

and “Four”. The second tabbing order consists of the input lines that contain the text “Five” and “Six”.

Figure 8-16 Independent tabbing orders within a parent view



The user taps in any of the top four slots; thereafter, pressing the tab key on a keypad or external keyboard will move among the four slots in that tabbing order. If the user taps in one of the bottom two slots, the tab key will then jump between those two slots.

The slots `_tabParent` and `_tabChildren` can coexist in a view, but the `_tabChildren` slot takes precedence in specifying the next key view.

The Caret Pop-up Menu

Normally, when the user taps on the insertion caret, the system-provided Punctuation Pop-up Menu opens. However, you can override this with a pop-up menu of your own creation.

When the user taps on the insertion caret, the system starts searching for a slot named `_caretPop-up`. The search begins in the view owning the caret, and follows both the proto and parent inheritance paths. The default punctuation pop-up is stored in the root view.

The `_caretPop-up` slot must hold a frame containing two slots. The first slot, `pop`, defines a list of pop-up items suitable for passing to `DoPop-up`. The second slot must contain a `pickActionScript`. If not, control passes to the default Caret Pop-up menu, which has its own version of the `pickActionScript`. This routine then inserts a string, corresponding to the selected character at the caret, by using the function `PostKeyString`.

Handling Input Events

You sometimes need to respond to input events that occur in your text views. This section describes how to test for a selection hit and how to respond to keystrokes and insertion events

Testing for a Selection Hit

After the user taps the screen, you can determine if the point “hits” a specific character or word in a paragraph view.

The `view:PointToCharOffset` method returns the offset within the paragraph that is closest to the point (*x*, *y*). This method is described on page 8-112.

The `view:PointToWord` method returns a frame that indicates the position of the word within the paragraph that is closest to the point (*x*, *y*). This method is described on page 8-112.

Note
Both of these functions return `nil` if the view is not a paragraph view. And the point that you are testing must correspond to a visible position within the paragraph view; you cannot hit-test on off-screen portions of a view. ◆

Intercepting Keyboard Actions

You can implement scripts that are called whenever the user presses a key. There are two keyboard-related scripts associated with text views:

- the `viewKeyDownScript` is called when a key is pressed
- the `viewKeyUpScript` is called when a key is released

Both of these scripts are applicable to software and external (hardware) keyboards.

IMPORTANT
If you want to implement scripts that are called for each keystroke, you must include the `vSingleKeystrokes` text flag in your view. This flag can cause performance degradation with external keyboards. ▲

Both scripts receive the character that was pressed on the keyboard and a packed integer flags word. The flags word encodes which modifier keys were in effect for the key, the unmodified key value, and the keycode. The layout of the flags word is shown in Table 8-9 on page 8-110. The modifier key constants are shown in the section "Keyboard Modifier Keys" beginning on page 8-59.

Text Reference

This section provides reference information for all of the constants, data structures, methods, and functions that you can use in your applications to work with text.

Text Constants and Data Structures

This section describes the constants and data structures that you can use in your applications to work with text.

Text Flags

The text flags listed below are view flags that you can use to specify information about text in views.

Constant	Value
vWidthIsParentWidth	(1 << 0)
vNoSpaces	(1 << 1)
vWidthGrowsWithText	(1 << 2)
vFixedTextStyle	(1 << 3)
vFixedInkTextSTyle	(1 << 4)
vExpectingNumbers	(1 << 9)
vSingleKeystrokes	(1 << 10)

Constant descriptions

vWidthIsParentWidth
: The view's width is the same as that of its parent view.

vNoSpaces
: Do not insert spaces between words.

vWidthGrowsWithText
: Causes the right horizontal boundary of the view to extend only as far as the widest line of text in the

paragraph. This flag can only be used for paragraph views that are children of an edit view.

`vFixedTextStyle`
: The font family, point size, and style of the `viewFont` are applied to all recognized text in the paragraph.

`vFixedInkTextSTyle`
: The font point size and style of the `viewFont` are applied to all ink words in the paragraph.

`vExpectingNumbers`
: Causes ink words to be scaled based on the assumption that they represent numbers rather than lowercase letters. Use for numeric fields in which the `vNumbersAllowed` flag is not set.

`vSingleKeystrokes`
: Required for views for which you have implemented the `viewKeyUpScript` or `viewKeyDownScript` methods. Note that this flag slows down the processing of external keyboard input.

Font Constants

This sections describes the constants that you can use to specify font information in your applications.

Built-in Fonts

The built-in font constants allow you to use a single integer value to specify one of the fonts built into the Newton system, including the font family, font face, and font size.

Constant	Value
`ROM_fontsystem9`	`9216`
`ROM_fontsystem9bold`	`1057792`
`ROM_fontsystem9underline`	`4203520`
`ROM_fontsystem10`	`10240`
`ROM_fontsystem10bold`	`1058816`

Constant	Value
ROM_fontsystem10underline	4204544
ROM_fontsystem12	12288
ROM_fontsystem12bold	1060864
ROM_fontsystem12underline	4206592
ROM_fontsystem14	14336
ROM_fontsystem14bold	1062912
ROM_fontsystem14underline	4208640
ROM_fontsystem18	18432
ROM_fontsystem18bold	1067008
ROM_fontsystem18underline	4212736
simpleFont9	9218
simpleFont10	10242
simpleFont12	12290
simpleFont18	18434
fancyFont9 or userFont9	9217
fancyFont10 or userFont10	10241
fancyFont12 or userFont12	12289
fancyFont18 or userFont18	18433
editFont10	10243
editFont12	12291
editFont18	18435

Constant descriptions

`ROM_fontsystem9`
: 9-point, plain face, Espy font

`ROM_fontsystem9bold`
: 9-point, boldface, Espy font

`ROM_fontsystem9underline`
: 9-point, underline face, Espy font

`ROM_fontsystem10`
: 10-point,t plain face, Espy fon

`ROM_fontsystem10bold`
: 10-point, boldface, Espy font

`ROM_fontsystem10underline`
: 10-point, underline face, Espy font

`ROM_fontsystem12`
: 12-point, plain face, Espy font

`ROM_fontsystem12bold`
: 12-point, boldface, Espy font

`ROM_fontsystem12underline`
: 12-point, underline face, Espy font

`ROM_fontsystem14`
: 14-point, plain face, Espy font

`ROM_fontsystem14bold`
: 14-point, boldface, Espy font

`ROM_fontsystem14underline`
: 14-point, underline face, Espy font

`ROM_fontsystem18`
: 18-point, plain face, Espy font

`ROM_fontsystem18bold`
: 18-point, boldface, Espy font

`ROM_fontsystem18underline`
: 18-point, underline face, Espy font

`simpleFont9` 9-point, plain face, Geneva font

`simpleFont10` 10-point, plain face, Geneva font

`simpleFont12` 12-point, plain face, Geneva font

`simpleFont18` 18-point, plain face, Geneva font

`fancyFont9 (userFont9)`
9-point, plain face, New York font

`fancyFont10 (userFont10)`
10-point, plain face, New York font

`fancyFont12 (userFont12)`
12-point, plain face, New York font

`fancyFont18 (userFont18)`
18-point, plain face, New York font

`editFont10` 10-point, plain face, handwriting font

`editFont12` 12-point, plain face, handwriting font

`editFont18` 18-point, plain face, handwriting font

Font Family Constants

You can use the font family constants to specify the family ID in a font specification.

Constant descriptions

`(none)` The Espy (system) font

`tsFancy` The New York font

`tsSimple` The Geneva font

`tsHWFont` The Casual (handwriting) font

Font Face Constants

You use the font face constants to specify the font face in a font specification.

Constant	Value
`tsPlain`	`0`
`tsBold`	`1048576`
`tsItalic`	`2097152`
`tsUnderline`	`4194304`

Constant	Value
tsOutline	8388608
tsSuperScript	134217728
tsSubScript	268435456

Constant descriptions

tsPlain	Plain font face
tsBold	Bold font face
tsItalic	Italic font face
tsUnderline	Underlined font face
tsOutline	Outlined font face
tsSuperScript	Superscripted font face
tsSubScript	Subscripted font face

Keyboard Constants

This section describes the constants that you can use with keyboard views.

Keyboard Registration Constants

When you register a keyboard, you can specify these flags to define how the keyboard is used.

Constant	Value
kKbdUsesKeyCodes	1
kKbdTracksCaret	2
kKbdforInput	4

Constant descriptions

kKbdUsesKeyCodes

The keyboard is key code–based, which means that the system has to redraw the view whenever the Shift, Option, or another modifier key is pressed on this or

any other key code–based view. This is because a single key map is used for all keyboard views.

`kKbdTracksCaret`
: The `viewCaretChangedScript` method of the keyboard view gets called whenever the caret changes position.

`kKbdforInput`
: The insertion caret is activated when this keyboard opens, if the caret was not already active. Use this when your keyboard provides input capabilities.

Key Descriptor Constants

The key descriptor constants specify the appearance of each key in a keyboard.

Constant	Value
`keySpacer`	`(1 << 29)`
`keyAutoHilite`	`(1 << 28)`
`keyInsetUnit`	`(1 << 25)`
`keyFramed`	`(1 << 23)`
`keyRoundingUnit`	`(1 << 20)`
`keyLeftOpen`	`(1 << 19)`
`keyTopOpen`	`(1 << 18)`
`keyRightOpen`	`(1 << 17)`
`keyBottomOpen`	`(1 << 16)`
`keyHUnit`	`(1 << 11)`
`keyHHalf`	`(1 << 10)`
`keyHQuarter`	`(1 << 9)`
`keyHEighth`	`(1 << 8)`
`keyVUnit`	`(1 << 3)`
`keyVHalf`	`(1 << 2)`
`keyVQuarter`	`(1 << 1)`
`keyVEighth`	`(1 << 0)`

Constant descriptions

`keySpacer`	Nothing is drawn in this space; it is a spacer, not a key.
`keyAutoHilite`	Highlight this key when it is pressed.
`keyInsetUnit`	Inset this key's frame a certain number of pixels within its space. Multiply this constant by the number of pixels you want to inset, from 0–7.
`keyFramed`	The thickness of the frame around the key. Multiply this constant by the number of pixels that you want to use for the frame thickness, a value in the range 0-3.
`keyRoundingUnit`	The roundedness of the frame corners. Multiply this constant by the number of pixels that you want to use for the corner radius, from 0-15, zero being square.
`keyLeftOpen`	No frame line is drawn along the left side of this key.
`keyTopOpen`	No frame line is drawn along the top side of this key.
`keyRightOpen`	No frame line is drawn along the right side of this key.
`keyBottomOpen`	No frame line is drawn along the bottom side of this key.
`keyHUnit`	Used in a key dimensions formula to specify horizontal units.
`keyHHalf`	Defines a number of half-units.
`keyHQuarter`	Defines a number of quarter-units.
`keyHEighth`	Defines a number of eighth-units.
`keyVUnit`	Used in a key dimensions formula to specify vertical units.
`keyVHalf`	Defines a number of half-units.
`keyVQuarter`	Defines a number of quarter-units.
`keyVEighth`	Defines a number of eighth-units.

Note
See the section "Key Dimensions" beginning on page 8-45 for more information about the `keyHUnit`, `keyHHalf`, `keyHQuarter`, `keyHEighth`, `keyVUnit`, `keyVHalf`, `keyVQuarter`, and `keyVEighth` constants. ◆

Keyboard Modifier Keys

You use the keyboard modifier key constants to determine which modifier keys were pressed or in effect when a character is delivered from a keyboard. See the section "Intercepting Keyboard Actions" beginning on page 8-50 for more details.

Constant	Value
`kIsSoftKeyboard`	`(1 << 24)`
`kCommandModifier`	`(1 << 25)`
`kShiftModifier`	`(1 << 26)`
`kCapsLockModifier`	`(1 << 27)`
`kOptionsModifier`	`(1 << 28)`
`kControlModifier`	`(1 << 29)`

Constant descriptions

`kIsSoftKeyboard`
If `true`, the character was entered on a "soft" keyboard; if not, the character was entered on an external keyboard.

`kCommandModifier`
If `true`, the Command key was in effect.

`kShiftModifier` If `true`, the Shift key was in effect.

`kCapsLockModifier`
If `true`, the Caps Lock key was in effect.

`kOptionsModifier`
If `true`, the Option key was in effect.

`kControlModifier`
If `true`, the Control key was in effect.

Line Patterns

A line pattern, which you use for customizing the display of the ruling lines in an edit or paragraph view, is simply an 8-byte binary data structure with the class `'pattern`.

The bit pattern of the bytes defines which pixels are on in the line. A typical line pattern is defined as shown here:

```
myPattern := SetClass( Clone("\uAAAAAAAAAAAAAAAA"),
                       'pattern );
```

This code clones a string, which is already a binary object, and changes its class to `'pattern`. The string is specified with hex character codes whose binary representation is used to create the pattern. Each 2-digit hex code creates one byte of the pattern.

When the line is drawn, the first bit of the pattern is aligned with the first pixel of the line. The pattern is repeated as necessary.

The Rich String Format

The rich string format allows ink data to be embedded in a text string. The location of each ink word in the string is indicated by a placeholder character (`0xF700` or `0xF701`), and the data for each ink word is stored after the string terminator character at the end of the string. The final 32 bits in a rich string also have special meaning.

Text Views and Protos

This section describes the views and protos that you can use to display text and receive text input.

General Input View (clEditView)

The `clEditView` class is used to accept text input. The `clEditView` class contains no data. When it receives input it creates child views—a

clParagraphView to hold text or ink text and a clPolygonView to hold graphics or raw ink.

For a list of the features provided by clEditView, see the section "General Input Views" beginning on page 8-9. The same section provides an example of a template that defines a view of the clEditView class.

Slot descriptions

viewBounds	Set to the size and location where you want the view to appear.
viewFlags	The default setting is vVisible. You will most likely want to set additional flags to control the recognition behavior of the view.
viewFormat	Optional. The default setting is: vfFillWhite + vfFrameBlack + vfPen(1)

A view of this class can appear as a blank space. Normally, you want the view to contain a series of horizontal dotted lines, like lined writing paper to that the view accepts input. The section "Creating the Lined Paper Effect in a Text View" on page 8-11 describes how to create this effect.

Child views that are automatically created by a clEditView have the vNoScripts flag set in their viewFlags slot, as described in the section "System Messages in Views That Are Created Automatically" on page 8-11.

Functions and Methods for Edit Views

This section describes the messages that are sent to edit views. You can define methods for these messages.

EditAddWordScript

view:`EditAddWordScript(`*form, bounds*`)`

This message is sent to an edit view when a new paragraph is about to be added to the edit view.

form	The paragraph template that is about to be added to the edit view.
bounds	The bounds of the written ink or typewritten character that has caused the new paragraph to be added.

You can use this script to modify the paragraph that is about to be added to the edit view. Your method must return the template to be added.

If you do not provide this method, or if you return *form* unchanged, the default action is taken: the system adds the paragraph view to the edit view in the usual manner at the usual location.

NotesText

`NotesText(`*childArray*`)`

This function returns a string that represents all of the text in an edit view.

childArray	An array of child views of an edit view. This is the same array that is returned by called the `editView:ChildViewFrames()` method.

The `NotesText` function creates a string in which distinct paragraphs are separated by carriage return characters. The `NotesText` function uses the location of each child view within the edit view to determine the order in which the strings are output.

If any of the child views contains ink, `NotesText` returns a rich string. If none of the views contains ink, `NotesText` returns a plain string.

You can use the `NotesText` function to export edit view text to a non-Newton computer or e-mail system.

Paragraph View (clParagraphView)

The `clParagraphView` class is used to display text or to accept text input. For a list of the features provided by `clParagraphView`, see the section "Paragraph Views" beginning on page 8-12. The same section provides an example of a template that defines a view of the `clParagraphView` class.

Slot descriptions

`viewBounds`	Set to the size and location where you want the view to appear.
`text`	A string that is the text currently contained in the view.
`viewFont`	Required, unless the `styles` slot is specified. The `viewFont` slot sets the font used to display text in the view. Note that if the view template itself does not contain this slot, it is inherited through proto inheritance only, not parent inheritance. See the section "Using Fonts for Text and Ink Display" beginning on page 8-25, for a detailed description of how to specify a font. If the text in the view has multiple fonts, then the `styles` slot is used to specify the font, instead of the `viewFont` slot.
`viewFlags`	The default setting is `vVisible`. You will most likely want to set additional flags to control the recognition behavior of the view. See the discussion of recognition flags in Chapter 10, "Recognition."
`viewFormat`	Optional. The default setting is `vfFillWhite+vfFrameBlack+vfPen(1)+vfLinesGray`.
`viewJustify`	Optional. The default setting is `vjLeftH+vjTopV+vjParentLeftH+vjParentTopV+noLineLimits`. Note that this view class does not support vertical justification of the view text for multiline text views. So the vertical justification flags (`vjCenterV`, `vjBottomV`, and `vjFullV`) apply only if the `oneLineOnly` flag is also set.
`tabs`	Optional. An array of up to eight tab-stop positions, in pixels. For example: [10, 20, 30, 40]. These positions are pixel values, relative to the left boundary of the view.

`styles` Optional. An array of alternating run lengths and font information, if multiple font styles are used. The first element is the run length (in characters) of the first style run, and the second element is the font style of the first run. The third element is the run length of the second style run, and so on. All of the run lengths must add up to the total text length. If the text is all in a single font, the font in the `viewFont` slot specifies the font style, and the `styles` slot is not needed. For information on how to specify a font in the `styles` array, see the section "Text and Styles" beginning on page 8-33.

`textFlags` Optional. Can contain one or more of the following bit flags (which can be added together) to control special features:

`vNoSpaces` Prevents spaces from being included in text written in the paragraph view. This is useful, for example, in numeric fields where you want the user to be able to enter a number without having spaces recognized between digits that may have been written spaced apart or added later.

`vWidthIsParentWidth` Causes the right boundary of the view to extend to the right boundary of its parent view. (Only applies when the view is enclosed in a `clEditView`.) Note that once the user resizes the view, however, this feature is disabled for that view. The assumption is that if the user resized the view, he or she wants it to stay a particular size and not to change.

`vWidthGrowsWithText` You should only use this flag if the paragraph view is a child view of an edit view. Causes the right horizontal boundary of the view to extend only as far as the widest line of text in that paragraph. This makes it possible to

create side by side paragraphs. When caret insertion mode is disabled, words written far enough to the right of a paragraph create a new paragraph. Similarly, when caret insertion mode is enabled, tapping far enough to the right of an existing paragraph positions the insertion caret outside of the old paragraph, and the next text entered creates a new paragraph at that position.

`copyProtection` Specifies restrictions on copying the view by dragging it into another view or by using the clipboard. This slot applies only to views of the class `clParagraphView`. If this slot is not present, there are no copy restrictions. In this slot you can specify one or more copy protection attributes, which are represented by constants defined as bit flags. The copy protection attributes are listed and described in Table 8-8.

Table 8-8 CopyProtection constants

Constant	Value	Description
`cpNoCopies`	1	The view cannot be copied.
`cpReadOnlyCopies`	2	The view can be copied, but the copy cannot be modified.
`cpOriginalOnlyCopies`	4	The original view can be copied, but copies of it cannot be copied. When a copy is made, the copy's `copyProtection` slot is changed to 1 (`cpNoCopies`) to prevent further copying.
`cpNewtonOnlyCopies`	8	The view can be copied, but on one Newton device only. Copies cannot be exported to a different Newton device.

Input Line Protos

An input line is just that, a single line in which the user can enter data. Protos are provided for input lines with and without an identifying label, and for regular and rich-text input. The use of input line protos is described in section “Using Input Line Protos” beginning on page 8-14.

protoInputLine

This proto is used for a one-line input field that is indicated by a dotted line to write on. It defines a simple paragraph view that accepts any kind of text input and is left-justified, as described in the section “protoInputLine” beginning on page 8-15. The same section provides an example of a template using `protoInputLine`.

Slot descriptions

`viewBounds` Set to the location where you want the input field to appear.

`viewFlags` Set particular view flags to limit recognition, if desired. The default setting is `vVisible + vClickable + vGesturesAllowed + vCharsAllowed + vNumbersAllowed`. For more information about the recognition view flags, see Chapter 10, “Recognition.”

`text` Optional. Set to a string that is the initial text, if any, to be shown in the input field. The default is no text. During run time, this slot holds the current text that exists in the input field.

`viewFont` Optional. This sets the font for text the user writes in the input field. The default is `editFont12`.

`viewJustify` Optional. The default setting is `vjLeftH + oneLineOnly`.

`viewFormat` Optional. The default setting is `vfLinesGray`.

`viewTransferMode` Optional. The default mode is `modeOr`.

`viewLineSpacing` Optional. The line spacing is the height of the input line

in pixels and it defaults to the setting of the parent view, or to 20, if there is no parent setting.

`viewLinePattern`
: Optional. Sets a custom pattern that is used to draw the line in the view. A pattern is simply an 8-byte binary data structure with the class `'pattern`. For information about specifying a line pattern, see section "Defining a Line Pattern" beginning on page 8-12.

`viewChangedScript`
: Optional. This method is called whenever the value of the input field is changed.

`memory`
: Used to reference a list of the last n items chosen. The value of this slot is a symbol that names the list. The symbol must incorporate your developer signature, as described in the section "protoLabelInputLine" beginning on page 8-16.

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

protoRichInputLine

This proto is the text and ink equivalent of the `protoInputLine`. All of the slot descriptions and discussion are exactly the same as for `protoInputLine`.

protoLabelInputLine

This proto is used for a one-line input field that includes a text label and can optionally feature a pop-up menu. The section "protoLabelInputLine" beginning on page 8-16 describes how to use this proto and provides an example of a template that uses it.

Slot descriptions

`viewBounds` Set to the location where you want the view to appear. Note that the view should have a height equal to or greater than the value set for `viewLineSpacing`.

`entryFlags` Set particular flags to limit recognition, if desired. The setting you specify in this slot is used for the `viewFlags` slot of the input field. The default setting is `vVisible + vClickable + vGesturesAllowed + vCharsAllowed + vNumbersAllowed`. For more information about the recognition view flags, see Chapter 10, "Recognition."

`label` Set to a string that is the label text.

`labelFont` Optional. Sets the font used for the label. The default is `ROM_fontSystem9Bold`.

`labelCommands` Optional. If this slot is supplied, the picker feature is activated and the label is shown with a diamond to its left to indicate that it is a picker.

Specify an array of strings that should appear in a picker when the user taps the label. To include a thin, gray separator line, specify the symbol `'pickSeparator`. For a thicker black line, specify the symbol `'pickSolidSeparator`.

The currently selected item in the list, if there is one, is marked with a check mark to its left.

`curLabelCommand`

Optional. If the `labelCommands` slot is supplied, this slot specifies which item in that array should be initially marked with a check mark. Specify an integer, which is used as an index into the `labelCommands` array. If you omit this slot, no item is initially marked with a check mark.

`indent` Optional. Set to the distance from the left edge of the view where the dotted input line should begin. The default is 4 pixels to the right of the label text. This slot is useful if you are specifying several labeled input fields in a column, and you want all the dotted input

lines to line up beneath one another. If you specify this slot, be sure to leave enough room for the label text.

`viewLineSpacing`

Optional. The line spacing is the height of the input line in pixels and it defaults to the setting of the parent view, or to 20, if there is no parent setting.

`viewLinePattern`

Optional. Sets a custom pattern that is used to draw the line in the view. A pattern is simply an 8-byte binary data structure with the class `'pattern`. For information about specifying a line pattern, see section "Defining a Line Pattern" beginning on page 8-12.

`textSetup` Optional. This method is called when the view is instantiated to set an initial value in the input field. This method is passed no parameters and should return a string, which is set as the initial value in the input field. If you don't supply this method, the input field is initially empty.

`updateText` Optional. You can call this method to programmatically change the value of the text in the input field. This action is reversible by the user with the Undo button. This method takes one parameter, a string that is the new value of the input field. Note that you don't normally need to call this method; the input field is updated automatically when the user writes in it.

`textChanged` Optional. This method is called whenever the value of the input field is changed. It is passed no parameters. If you don't supply this method, no default action occurs.

`setLabelText` Optional. You can call this method to dynamically change the label text after the view has already been opened. This method takes one parameter, a string that is the new label text.

`setLabelCommands`

Optional. You can call this method to dynamically set the `labelCommands` array. This method takes one

parameter, an array of strings that should appear in the picker.

`labelClick` Optional. This method is called when the user taps the label. It is passed one parameter, the stroke unit that was passed to the `viewClickScript` method of the label. This message notifies the view; then you have a chance to handle the event when the label is tapped. If you don't supply this method or if you choose not to handle the event, the default action is to display the picker, get the user's choice, enter the chosen text into the input line, and dirty the input line to cause a redraw.

This function must return either `true` or `nil`. If it returns `true`, the default action is not finished; the assumption is that you have handled the event yourself. If it returns `nil`, the default action is still performed after this method returns.

`labelActionScript`

Optional. This method is called when an item is chosen from the picker. It is passed one parameter, which is the index of the item selected from the `labelCommands` array. This message notifies the view; then you have a chance to handle the event when an item is chosen from the picker. If you don't supply this method or if you choose not to handle the event, the default action is to set the text in the input line to the string that was chosen from the picker.

This function must return either `true` or `nil`. If it returns `true`, the default action is not finished; the assumption is that you have handled the event yourself. If it returns `nil`, the default action is still performed after this method returns.

Note that inking is automatically turned off when the label is tapped.

The `protoLabelInputLine` is based on a view of the `clView` class, and includes two child views: `labelLine` and `entryLine`. These views are described in the section "protoLabelInputLine" beginning on page 8-16.

protoRichLabelInputLine

This proto is the text and ink equivalent of the `protoLabelInputLine`. All of the slot descriptions and discussion are exactly the same as for `protoLabelInputLine`.

Structured List Proto (protoListView)

You can use the structured list proto to create structured list views for your applications. The section "Structured List Views" beginning on page 8-18 provides a description of using `protoListView` and an example of a template that contains it. The slots of this proto are described here.

Slot descriptions

`ViewLineSpacing` This sets the spacing between the lines, in pixels. The default is 28.

`checkBitmaps` A two -element array specifying bitmaps to use for checkboxes. The default is:
`[checkOffBitmap, checkOnBitmap]`
If you specify your own bitmaps, they must be the same size as the checkboxes provided in the system.

`leftMarkGap` Specifies the space, in pixels, to the left of the topic marker. The default value is 0. You could add to this to leave space in which to draw your own gadgets.

`listViewFlags` Specifies what interface gadgets are displayed. The default is 0. You can use the following values:

`lvNoMarkers:=1`

`lvShowChecks:=2`

`lvShowTopPriority:=4`

`lvShowSubPriority:=8`

`lvShowPriority:=12`

`lvAdjustTopicBounds:=16`

`markers` This is the bitmap used to draw topic markers, with the three states adjacent. The default is: `topicMarkers`. If

	you specify your own bitmap, each element must be the same size as the default.
`maxLevel`	The highest number of subtopics allowed. Default is 8 (to accommodate small screens). Set `MaxLevel` to 1 if you don't want to allow subtopics.
`priorityItems`	A four-element array specifying the bitmaps used to draw priority markers in the To Do List application. Each element corresponds to the meeting priority value of a topic. You can specify as many elements as you want (more than four is fine). The default array is: `[ABitmap, BBitmap, CBitmap, DBitmap]` If you specify your own bitmaps, they must be the same size as the built-in priority markers for the To Do list.
`rightIndent`	Text extends this close to the edge of the view. The default value is zero. Set this slot to a negative value if you want to draw something in the right margin.
`rightMarkGap`	Space to right of topic marker. Default is 0. You could add to this if you want to draw your own gadgets.
`topicFont`	This is the font used to create new topics. Default is `nil`, in which case the system default is used.

The following slots are set by `protoListView`. They are included for your information.

`curTopic`	The index of current active topic, or `nil` if there is none.
`firstTopic`	The index of first topic displayed. You can scroll by first changing the `firstTopic` slot and then calling `RedoChildren()`.
`minimalChildren`	Set to `true` if your list view is not in a paper roll.

ActiveTopic

list:`ActiveTopic`(*anIndex*)

Sets the `curTopic` slot and makes the topic specified by the parameter *anIndex* the active view.

anIndex — An integer index referring to the position of a topic in a list.

This method always returns `nil`.

AddEmptyTopic

list:`AddEmptyTopic`(*anIndex*, *aLevel*)

Adds a new topic frame, containing no data, to the array of topics.

anIndex — An integer index referring to the position of a topic in a list. The new topic is added after the item to which you refer.

aLevel — An integer signifying the indent level of the topics. The maximum number of levels is 8 (set in the slot `MaxLevel`) and the top level is 1.

This method always returns `nil`.

CalcLevel

list:`CalcLevel`(*anX*)

Converts the coordinate *anX* to an integer representing an outline level.

anX — The left coordinate of a `clEditView`.

This method returns an integer.

ClickBounds

list:`ClickBounds(`*anIndex*`)`

Returns a hit target rectangle for a topic.

anIndex An integer index referring to the position of a topic in a list.

CollapseTopic

list:`CollapseTopic(`*anIndex*`)`

Collapses a topic by hiding its subtopics.

anIndex An integer index referring to the position of a topic in a list.

This method always returns `nil`.

DeleteTopics

list:`DeleteTopics(`*aList*`)`

Removes the topics in the parameter *aList*, which is an array of indexes. The caller should manage any Undo functionality and must also call `RedoChildren`, since this method does not change views.

aList An array of one or more indexes to be removed.

This method always returns `nil`.

DirtyBelow

list:`DirtyBelow(`*anIndex*`)`

Forces redraw of topics from the topic specified by the parameter *anIndex* to the end of the list.

anIndex An integer index referring to the position of a topic in a list.

This method always returns `nil`.

EnsureVisibleTopic

list:`EnsureVisibleTopic(`*anIndex*`)`

Scrolls the view to make the topic specified in the parameter *anIndex* visible.

anIndex	An integer index referring to the position of a topic in a list.

This method always returns `nil`.

ExpandTopic

list:`ExpandTopic(`*anIndex*`)`

Expands a topic by revealing its subtopics.

anIndex	An integer index referring to the position of a topic in a list.

This method always returns `nil`.

GetTopicCount

list:`GetTopicCount()`

Returns the number of topics in a list.

HandleCheck

list:`HandleCheck(`*anIndex*, *aTopic*`)`

Handles taps on the checkbox. You ordinarily don't override this method; however, you can if you want to handle a tap on a checkbox.

anIndex	An integer index referring to the position of a topic in a list.
aTopic	A topic frame.

This method always returns `nil`.

HandlePriority

list:`HandlePriority(`*anIndex*, *aTopic*`)`

Handles taps on a priority marker such as those in the To Do List application. You ordinarily don't override this method; however, you can if you want to handle a tap on a priority marker.

anIndex An integer index referring to the position of a topic in a list.

aTopic A topic frame.

This method always returns `nil`.

HandleScrub

list:`HandleScrub(`*aBounds*`)`

Override this if you want to handle a scrub. The parameter *aBounds* should contain the local coordinates of the scrub.

aBounds The local coordinates of the scrub.

Returns `true` if you've deleted any topics.

HasKids

list:`HasKids(`*anIndex*`)`

Returns `true` if the topic has subtopics.

anIndex An integer index referring to the position of a topic in a list.

IsCollapsed

list:`IsCollapsed(`*anIndex*`)`

Returns `true` if the topic is collapsed.

anIndex An integer index referring to the position of a topic in a list.

IsEmpty

list`:IsEmpty(`*topicIndex*`)`

Returns `true` if the topic, specified by the *topicIndex* parameter, contains no data.

topicIndex An integer index referring to the position of a topic in a list.

IsReadOnly

list`:IsReadOnly(`*topicIndex*`)`

Returns `true` if the topic, specified by the *topicIndex* parameter, is set for read-only access to text.

topicIndex An integer index referring to the position of a topic in a list.

ListBottom

list`:ListBottom()`

Returns the lowest coordinate in the list, depending on the current state of expansion.

MakeTopicFrame

list`:MakeTopicFrame(`*aLevel*`)`

Returns a new frame with the default settings taken from `listViewFlags`. Use this function to make topic frames that have shared memory maps.

aLevel An integer signifying the indent level of the topics. The maximum number of levels is 8. The top is level is level 1.

MarkerBounds

list:`MarkerBounds(`*aTopic*`)`

Returns a bounds frame for the topic marker.

aTopic A topic frame.

Mother

list:`Mother(`*anIndex*`)`

Returns the index of the parent of a subtopic or `nil` if the index parameter refers to a top-level (`level=1`) topic.

anIndex An integer index referring to the position of a topic in a list.

NewRelative

list:`NewRelative(`*aRelation*`)`

Creates a new topic frame relative to the frame stored in `curTopic`. If `curTopic` is `nil`, this method creates a new topic relative to the last topic in the list. The parameter *aRelation* must be either `'aunt`, `'sister`, or `'daughter`.

aRelation A symbol of the value `'aunt`, `'sister`, or `'daughter`.

NewTopic

list:`NewTopic(`*aLevel, aText*`)`

Creates an empty topic at the specified level containing the specified text. This is normally called from the floating buttons New Item and New Subitem.

aLevel An integer signifying the indent level of the topics. The maximum number of levels is 8 (default value of the slot `MaxLevels`) and the top level is 1.

aText The string stored in the `text` slot of the topic frame.

OlderSister

list:OlderSister(*anIndex*)

Returns the index of the previous topic at the same level with the same parent, or nil if there is none.

anIndex An integer index referring to the position of a topic in a list.

OldestSister

list:OldestSister(*anIndex*)

Returns the index of the oldest topic at the same level with the same parent. If there is none, it returns the value of the parameter, anIndex.

anIndex An integer index referring to the position of a topic in a list.

PickActionScript

list:PickActionScript(*anAction*)

This method is called when the user has chosen a priority item from the Priority Pop-up menu. Override PickActionScript to define new behavior for your Priority menu items. You may also want to call the inherited method since it handles an undoable change in priority.

anAction An integer in the range 0…3 that indicates the item to be chosen from the pop-up menu. These correspond to the number of bitmaps specified in the PriorityItems slot.

ResolveClick

list:ResolveClick(*anX*, *aY*)

Returns a frame indicating where and in what topic a tap occurred.

anX, *aY* The coordinates of the tap.

The returned frame includes a topic Index (or `nil` if the tap is outside a topic), as well as an indication of whether or not the click was on the marker.

RevealTopic

list`:RevealTopic(`*anIndex*`)`

Ensures that a topic is visible by expanding the enclosing views.

anIndex	An integer index referring to the position of a topic in a list.

SetDone

list`:SetDone(`*anIndex*, *aTopic*, *aDone*, *aPropogateUp*, *aPropogateDown*`)`

Marks the done state of a topic and propagates that change to parents and children. Normally, `aPropagateUp` and `aPropagateDown` should be set to the constant values `kPropagateUp` and `kPropagateDown`, respectively, so the check state remains consistent. The values of the constants, `kPropagateUp` and `kPropagateDown`, are `true`.

anIndex	An integer index referring to the position of a topic in a list.
aTopic	A topic frame.
aDone	Sets the `mtgDone` slot of the topic frame.
aPropogateUp	A `true` or `nil` value, usually called with the constant `kPropagateUp`.
aPropogateDown	A `true` or `nil` value, usually called with the constant `kPropagateDown`. However, you can call it with the value `kNoPropagateDown`.

SetPriority

`SetPriority(`*anIndex*, *aPriority*, *anUndo*`)`

Called to set or undo the setting of the priority of a topic. Depending on what that priority number is, it also reorganizes the order of the topics in the list.

anIndex An integer index referring to the position of a topic in a list.

aPriority An integer in the range 0…3 that indicates the item to be chosen from the pop-up menu. These correspond to the number of bitmaps specified in the `PriorityItems` slot.

anUndo This value should be `nil` to set a topic's priority, or `true` if this is part of an Undo operation, as shown in the following code line.

```
AddUndoAction('SetPriority, [theIndex, oldPriority, true]);
```

SetReadOnly

list`:SetReadOnly(`*anIndex*, *aStatus*`)`

Sets the state of the indicated topic to read-only or read-write.

anIndex An integer index referring to the position of a topic in a list.

aStatus The read-only status for the topic. Use `true` to indicate read-only and `nil` to indicate read-write.

ToggleTopic

list`:ToggleTopic(`*anIndex*`)`

Either expands or collapses a topic.

anIndex An integer index referring to the position of a topic in a list.

Always returns `nil`.

TopicBottom

list:`TopicBottom(`*anIndex*`)`

Returns either the lowest coordinate of its topic marker or of its paragraph view.

anIndex An integer index referring to the position of a topic in a list.

TopicHeight

list:`TopicHeight(`*anIndex*`)`

Returns the height, in pixels, of the `clView` of a topic.

anIndex An integer index referring to the position of a topic in a list.

YoungerSister

`YoungerSister(`*anIndex*`)`

Returns the index of the next topic at the same level. If there are no younger siblings, this method returns `nil`.

anIndex An integer index referring to the position of a topic in a list.

Text and Ink Display Functions and Methods

This section describes the functions and methods that you can use in your applications to display text and ink in views. For more information, see the section "Text and Ink in Views" beginning on page 8-21.

Functions and Methods for Measuring Text Views

This section describes the functions that you can use to measure or predict the bounds of a text view.

There are two measurement functions: `TextBounds` and `TotalTextBounds`. The `TextBounds` function is more efficient, but is accurate only in limited circumstances. You can use the `TextBounds` function if the view meets the following conditions:

- it contains no tabs
- it uses a single font
- it uses fixed line spacing

If your view does not meet these conditions, use the `TotalTextBounds` function for measuring the bounds of the view.

TextBounds

`TextBounds`(*rStr, fontFrame, viewBounds*)

Computes the bounds of a text string within a view.

rStr — A string or rich string that does not contain any tabs or line breaks.

fontFrame — Either a standard font specification, or a frame that contains the following two slots:

`font`	A font specification.
`justification`	Optional. The text justification, which must be one of: `'left`, `'center`, `'right`. The default value is `'left`.

viewBounds — A bounds frame in which either the `right` or `bottom` slot has a value of `0`.

The `TextBounds` function computes the bounds frame for a text string that is drawn using the supplied font specification. The `TextBounds` function modifies the slots in `viewBounds` to specify the bounds for *rStr*.

If the `right` value of the original bounds frame is 0, `TextBounds` computes how wide the bounds box needs to be for the text to fit into a specified height value, and stores that value into the `right` slot.

If the `bottom` value of the original bounds frame is `0`, `TextBounds` computes how tall the bounds box needs to be for the text to fit into a specified width value, and stores that value into the `bottom` slot.

If both the `right` and `bottom` values of the original bounds frame are `0`, the `width` and `height` slots are modified based on the explicit line breaks in *rStr*.

TotalTextBounds

`TotalTextBounds(`*paraSpec*`,` *editSpec*`)`

Predicts the bounds of a complex paragraph view, based on the text in the view.

paraSpec	A paragraph view template that must contain the following slots: `text`, `viewFont`, and `viewBounds`. The `bottom` slot in `viewBounds` should have a value of `0`.
editSpec	A template for the edit view in which the paragraph is to be enclosed. This can be `nil`. You should include this parameter if you are going to create a paragraph view as the child of a `clEditView`, since the properties of the edit view affect the computation.

The `TotalTextBounds` function returns a bounds frame for a `clParagraphView` that encloses the specified text. The returned bounds frame contains the same `left`, `right`, and `top` values as the `viewBounds` slot of *paraSpec*. The `bottom` slot of the returned bounds frame is filled in with the appropriate height value for the paragraph view.

The text slot of the paragraph view can contain plain strings or rich strings.

Functions and Methods for Determining View Ink Types

This section describes the functions and methods that you can use to determine whether a view accepts raw ink or ink words as input.

AddInk

`AddInk(`*edit*`, `*poly*`)`

Adds ink to an edit view.

edit An edit view object.

poly A polygon frame that can be expanded into a `clPolygonView` object. This frame contains two slots:

`ink`	The ink data.
`viewBounds`	The bounds box for the ink.

The `AddInk` function adds ink to an edit view. The ink is stored within the edit view as a polygon view.

ViewAllowsInk

`ViewAllowsInk(`*view*`)`

Determines if *view* accepts raw ink as input.

view A view object.

The `ViewAllowsInk` function returns a non-`nil` value if view accepts raw ink as input. This function uses the view's recognition configuration and view flags to determine the return value.

Note

The value returned by the ViewAllowsInk function is not necessarily the same as the state of the Recognition menu. This is because a view that does not receive ink due to the Recognition menu setting can allow ink. ◆

ViewAllowsInkWords

`ViewAllowsInkWords(`*view*`)`

Determines if *view* accepts raw ink as input.

view A view object.

The `ViewAllowsInkWords` function returns a non-`nil` value if view accepts ink words as input. This function uses the view's recognition configuration and view flags to determine the return value.

Font Attribute Functions and Methods

You can use the font attribute functions and methods to store or retrieve the settings stored in a font specification. For more information about using fonts in your text views, see "Using Fonts for Text and Ink Display" beginning on page 8-25.

FontAscent

`FontAscent(`*fontSpec*`)`

Returns the ascent, in pixels, of the font specified by *fontSpec*. The ascent is the vertical distance from the font baseline to the font ascent line.

fontSpec A font specification. This can be an integer, frame, or binary object specification of a font.

FontDescent

`FontDescent(`*fontSpec*`)`

Returns the descent, in pixels, of the font specified by *fontSpec*. The descent is the vertical distance from the font baseline to the font descent line.

fontSpec A font specification. This can be an integer, frame, or binary object specification of a font.

FontHeight

`FontHeight(`*fontSpec*`)`

Returns the maximum height, in pixels, of the font specified by *fontSpec*. This equals the font ascent plus the descent plus the leading.

fontSpec A font specification. This can be an integer, frame, or binary object specification of a font.

FontLeading

`FontLeading(`*fontSpec*`)`

Returns the font leading, in pixels, of the font specified by *fontSpec*. This is the vertical distance from the font descent line to the ascent line of the next text line below it.

fontSpec A font specification. This can be an integer, frame, or binary object specification of a font.

GetFontFace

`GetFontFace`*(fontSpec)*

Returns the face of the font specified by *fontSpec*. The face is returned as an integer value.

fontSpec A font specification. This can be an integer, frame, or binary object specification of a font.

GetFontFamilyNum

`GetFontFamilyNum`*(fontSpec)*

Returns the family number for the font specified by *fontSpec*. Only the Espy, Geneva, Handwriting (Casual), and New York font families currently have numbers.

Returns `nil` if no number is available or if the font is an ink font.

fontSpec A font specification. This can be an integer, frame, or binary object specification of a font.

GetFontFamilySym

`GetFontFamilySym`*(fontSpec)*

Returns the symbol representing the typeface of the font specified by *fontSpec*. The returned value is one of the font family symbols, as shown in Table 8-2 on page 8-26.

Returns `nil` if the *fontSpec* is an ink font binary object.

fontSpec A font specification. This can be an integer, frame, or binary object specification of a font.

GetFontSize

`GetFontSize`*(fontSpec)*

Returns the size of the font specified by *fontSpec*.

fontSpec A font specification. This can be an integer, frame, or binary object specification of a font.

Note

You can only replace the current values in a *fontSpec* with your input specification. You cannot supplement the current values. For example, you cannot add the bold attribute to a font that already uses the underline attribute; instead, you must specify both attributes in your input specification. To combine existing values with new values, call the appropriate font attribute retrieval function (e.g. `GetFontFace`) and add in your new value(s).

MakeCompactFont

`MakeCompactFont`*(family, size, face)*

Makes a new font specification from the supplied components.

family Can be either a symbol or integer that specifies a font family.

size The point size as an integer value.

face The font face as an integer value.

Returns a font specification. If the font is a ROM font, a packed integer is returned.

Returns the new font specification as a packed integer (if it is a ROM font) or as a frame.

SetFontFace

`SetFontFace`*(fontSpec, newFace)*

Sets the face of the font specified by *fontSpec* to the face specified by *newFace* and returns the altered *fontSpec*.

fontSpec	A font specification. This can be an integer, frame, or binary object specification of a font.
newFace	An integer, which specifies a font face.

Returns the altered *fontSpec*. If the font is a ROM font, a packed integer is returned.

If you specify the *fontSpec* as a frame, the returned frame is cloned from the input parameter *fontSpec*. If you specify the *fontSpec* as a binary object, the binary object itself is modified.

SetFontFamily

`SetFontFamily`*(fontSpec, newFamily)*

Sets the family of the font specified by *fontSpec* to the family specified by *newFamily* and returns the altered *fontSpec*.

fontSpec	A font specification. This can be an integer, frame, or binary object specification of a font.
newFamily	Can be either a symbol or integer that specifies a font family.

Returns the altered *fontSpec*. If the font is a ROM font, a packed integer is returned.

If you specify the *fontSpec* as a frame, the returned frame is cloned from the input parameter *fontSpec*. If you specify the *fontSpec* as a binary object, the binary object itself is modified.

SetFontParms

`SetFontParms` *(fontSpec, whichParms)*

This function alters one or more components of a font specification. The *whichParms* parameter specifies which components of the *fontSpe*c to alter.

`SetFontParms` returns a modified version of the font specification. If the specification can be packed into an integer (if the font is a ROM font), then a packed integer is returned.

The returned value may be a modified version of the font passed in, or may be a modified clone of the original *fontSpec*. If possible, a packed integer is returned.

fontSpec A font specification. This can be an integer, frame, or binary object specification of a font.

whichParms A frame that specifies which components of the font spec to alter. The slots that can be used individually or in combination in this frame include:

`size`: An integer representing the point size of the type. Usual values include: 9,10,12,14, and 18.

`face`: An integer representing the font style attribute. The constants that you can use for font face values are shown in “Font Face Constants” on page 8-55.

`family`: A symbol or integer representing the typeface. Note that you cannot change the family of an ink font. The constants that you can use for font family numbers are shown in “Font Family Constants” on page 8-55

`scale`: Only applies to ink fonts. An integer percentage of the original written ink size. When this slot is present, the `size` slot is ignored.

`penSize`: Only applies to ink fonts. An integer between 1 and 4.

If you specify the *fontSpec* as a frame, the returned frame is cloned from the input parameter *fontSpec*. If you specify the *fontSpec* as a binary object, the binary object itself is modified.

SetFontSize

`SetFontSize`*(fontSpec, newSize)*

Sets the font size of the font specified by *fontSpec* to the size specified by *newSize* and returns the altered *fontSpec*.

fontSpec	A font specification. This can be an integer, frame, or binary object specification of a font.
newSize	The new font size, specified as an integer value.

Returns the altered *fontSpec*. If the font is a ROM font, a packed integer is returned.

If you specify the *fontSpec* as a frame, the returned frame is cloned from the input parameter *fontSpec*. If you specify the *fontSpec* as a binary object, the binary object itself is modified.

Rich String Functions and Methods

This section describes the functions and methods that you can use to operate with rich strings. For a description of rich strings and the rich string format, see the section “Rich Strings” beginning on page 8-31.

DecodeRichString

`DecodeRichString`*(richString, defaultFontSpec)*

Returns a frame containing two slots:`text` and a `styles`. These slots can be placed in a paragraph view for editing or viewing.

richString	A rich string that can contain text and ink words.
defaultFontSpec	The font specification for the text in *richString*. This is usually the same as the `viewFont` slot of the view in which the text is displayed.

Note
The `SetValue` function, which also decodes a rich string, is more efficient than the `DecodeRichString` function.

ExtractRangeAsRichString

view:`ExtractRangeAsRichString`*(offset, length)*

Returns a rich string for the range of text specified from a paragraph view. This method can only be used on paragraph views.

offset	The beginning offset of the text range, specified as an integer value.
length	The number of characters in the range of text, specified as an integer value. Each ink word in the rich string is counted as a single character.

GetRichString

view:`GetRichString()`

The `GetRichString` function returns either a rich string or a plain string that represents the text in the paragraph view to which the `GetRichString` message is sent. If the paragraph contains ink, `GetRichString` returns a rich string; if not, `GetRichString` returns a plain string.

IsRichString

`IsRichString`*(testString)*

Returns non-`nil` if the *testString* parameter is a rich string containing ink.

testString	A rich string that can contain text and ink words.

MakeRichString

`MakeRichString`*(text, styleArray)*

Encodes the data from the *text* and *styleArray* parameters into a rich string.

text	The text from the `text` slot of a paragraph view.

styleArray The array found in the `styles` slot of a view. The format of this array is described in the section "Text and Styles" beginning on page 8-33.

Returns a rich string that has the encoded information for the text and style array parameters.

StripInk

`StripInk`(*richString, replaceChar*)

The `StripInk` function modifies *richString*, replacing every ink word placeholder in the string with *replaceChar*. If *replaceChar* is `nil`, the ink words are deleted.

richString The rich string to be stripped of ink word placeholders.

replaceChar The character to insert into *richString* in place of the ink word placeholders. Use `nil` to delete all ink words from the rich string.

Returns the modified string.

WARNING
The `StripInk` function destructively modifies *richString*. ▲

Functions and Methods for Accessing Ink in Views

This section describes the functions that you can use to determine if a view has ink in it and to access the ink in a paragraph view.

GetInkAt

`GetInkAt`*(para, index)*

The `GetInkAt` function returns the next ink in the paragraph view specified by *para*.

para A paragraph view.

index The starting position of the search. If this value is `nil`, `GetInkAt` starts searching at the beginning of the

paragraph text. If this value is an integer, `GetInkAt` starts searching at the next position after *index*.

The `GetInkAt` function returns a polygon view that contains the ink.

NextInkIndex

`NextInkIndex`*(para, index)*

Finds the next piece of ink within the paragraph view specified by *para*.

para	A paragraph view.
index	The starting position of the search. If this value is `nil`, `NextInkIndex` starts searching at the beginning of the paragraph text. If this value is an integer, `NextInkIndex` starts searching at the next position after *index*.

Returns the offset of the next ink in the paragraph. If `NextInkIndex` does not find ink, it returns `nil`.

To start checking at the beginning of the text, use nil as the value of *index*. To start checking for ink at offset *i*, use *i-1* as the value of *index*. To start checking at the next location in the text, use the value returned by the previous call to `NextInkIndex`.

ParaContainsInk

`ParaContainsInk`*(para)*

The `ParaContainsInk` function tests to see if the paragraph view specified by *para* contains ink.

para	A paragraph view.

If the paragraph view contains ink, `ParaContainsInk` returns the offset within the paragraph of the first piece of ink. If the paragraph view does not contain ink, `ParaContainsInk` returns `nil`.

PolyContainsInk

`PolyContainsInk`*(poly)*

The `PolyContainsInk` function tests to see if the polygon specified by *poly* contains ink.

poly A polygon view.

Returns `true` if the polygon contains ink and `nil` if not.

Keyboards

This section describes the views, protos, and functions that you can use in your applications to work with on-screen keyboards.

Keyboard View (clKeyboardView)

The `clKeyboardView` class is used to display keyboard-like arrays of buttons that can be pressed (tapped with the pen) to perform an action. To read about how to use this class, see the section "Keyboard Views" beginning on page 8-35, which includes several examples.

Slot descriptions

`_noRepeat`	If present, indicates that keys do not repeat while held down.
`viewBounds`	Set to the size and location where you want the view to appear.
`keyDefinitions`	An array that defines the layout of the keys, as described in the section "The Key Definitions Array" beginning on page 8-41.
`viewFlags`	The default setting is `vVisible + vClickable`.
`viewFormat`	Optional. The default setting is `nil`.
`keyArrayIndex`	Optional. Determines the array element to be used for a key legend or result, allowing dynamic indexing into an

array for legends or results. See “The Key Definitions Array.”.

`keyHighlightKeys`
Optional. An array of keys that are to be highlighted on the displayed keyboard. Specify an array of `keyResult` items, as described in the section “The Key Definitions Array.”.

`keyResultsAreKeycodes`
Optional. If true, indicates that integers specified as results are to be interpreted as key codes, and the corresponding character is returned. If `nil` (the default), integers are not converted to characters.

`keyReceiverView`
Optional. The view to which key commands (as a result of key presses) should be posted if no `keyPressScript` method exists. If the `keyReceiverView` slot is not found, the view identified by the symbol `'viewFrontKey` is used. This symbol evaluates at run time to the current key receiver view.

`keySound` Optional. A reference to a sound frame. The sound is played whenever a key is pressed. The default is no sound.

`keyPressScript` Optional. This method is called whenever a key is pressed. The key result of the key pressed is passed as a parameter to this method. If this method is not supplied, the key result is converted (if possible) into a sequence of characters, which are posted as key events to the key receiver view.

An example of a view definition of the `clKeyboardView` class, including the key definitions for the view, is shown in the section “Defining Keys in a Keyboard View” beginning on page 8-40.

Keyboard Protos

This section provides reference information for the keyboard protos.

protoKeyboard

This proto is used to create a keyboard view that floats above all other views. It is centered within its parent view and appears in a location that won't obscure the key-receiving view. The section "protoKeyboard" beginning on page 8-38 describes how to use this proto and provides an example of a template that uses it.

Slot descriptions

`saveBounds`	Set to the size and location where you want the keyboard view to appear. (This is used as the `viewBounds` value for the keyboard view.) Note that the keyboard view may be displayed above or below the location you specify, if it must be moved so as not to obscure the key-receiving view. (You can "freeze" it in place by using the `freeze` slot.)
`freeze`	Optional. If set to `true`, prevents automatic movement of the keyboard view. This slot is set to `nil` by default, allowing movement of the keyboard view so as not to obscure the key-receiving view, if it would be blocked by the bounds you specified for the keyboard.

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

This proto is used in conjunction with `protoKeypad` to implement a floating keyboard. This proto defines the parent view, and `protoKeypad` is a child view that defines the key characteristics.

The `protoKeyboard` itself uses the `protoFloater` proto, which is described in Chapter 7, "Controls and Other Protos."

protoKeypad

This proto is used to define key characteristics for a keyboard view (`clKeyboardView` class). The section "protoKeyboard" beginning on

page 8-38 describes how to use this proto and provides an example of a template that uses it.

Slot descriptions

`keyDefinitions` An array that defines the layout of the keys. Refer to the `clKeyboardView` description in the section "The Key Definitions Array" beginning on page 8-41.

`viewFont` Optional. The default font is `ROM_fontSystem9Bold`.

`viewFormat` Optional. The default setting is `vfFillWhite`.

`keyArrayIndex` Optional. Set by this proto to zero.

`keyHighlightKeys` Optional. Set by this proto to `nil`.

`keyResultsAreKeycodes` Optional. Set by this proto to `true`.

`keyReceiverView` Optional. Set by this proto to `'viewFrontKey`.

`keySound` Optional. Set by this proto to `typewriter`.

`keyPressScript` Optional. This method is called whenever a key is pressed. The key result of the key pressed is passed as a parameter to this method. If this method is not supplied, the key result is converted (if possible) into a sequence of characters that are posted as key events to the key receiver view.

The `protoKeypad` is based on a view of the class `clKeyboardView`. For more information about the key slots listed above, refer to the section "Keyboard View (clKeyboardView)" beginning on page 8-95.

You use this proto along with `protoKeyboard` to implement a floating keyboard. The view using the `protoKeypad` proto should be a child of the view using the `protoKeyboard` proto.

protoKeyboardButton

This proto is used to include the keyboard button in a view. The section "protoKeypad" beginning on page 8-39 describes how to use this proto and provides an example of a template that uses it.

Slot descriptions

`viewFlags`	The default is `vVisible + vReadOnly + vClickable`.
`viewBounds`	Set to the size and location where you want the keyboard to appear.
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV`.
`current`	Required. The symbol of the default keyboard to open. This value is not actually in the button view frame, but is found by inheritance.

Note that the `viewClickScript`, `buttonClickScript`, and `pickActionScript` methods are used internally in the `protoPictureButton` and should not be overridden.

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

protoSmallKeyboardButton

This proto is used to include the small keyboard button in a view. The section “protoSmallKeyboardButton” beginning on page 8-39 describes how to use this proto and provides an example of a template that uses it.

Slot descriptions

`viewFlags`	The default is `vVisible + vReadOnly + vClickable`.
`viewBounds`	Set to the size and location where you want the keyboard to appear.
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV`.
`current`	Required. The symbol of the default keyboard to open. This value is not actually in the button view frame, but is found by inheritance.

The `protoSmallKeyboardButton` uses the `protoKeyboardButton` as its proto, and `protoKeyboardButton` uses the `protoPictureButton` as its proto.

Note that the `viewClickScript`, `buttonClickScript`, and `pickActionScript` methods are used internally in the `protoPictureButton` and should not be overridden.

Keyboard Functions and Methods

This section describes the functions and methods that you can use with keyboards in your Newton applications.

GetCaretBox

`GetCaretBox()`

Returns a bounds frame containing the global coordinates of the text insertion caret, if it is displayed. If there is a text selection in a view, the caret is positioned before the first character of the selection, though it may not be visible. If there is no text selection and the caret is not displayed, this function may still return a bounds frame giving the virtual position of the text caret. This is the last position of the caret when it was displayed, or the position where handwritten text would be inserted (usually immediately following existing text).

If there is no key-receiving view, `nil` is returned.

KeyboardInput

view:`KeyboardInput()`

Returns `true` if the view is the current key view (the view receiving keystrokes) and the keyboard is enabled (visible). Otherwise, this function returns `nil`.

This method applies only to views of the class `clEditView` and `clParagraphView`.

KeyIn

`KeyIn(`*keyCode*`, `*down*`)`

Allows you to programmatically change the state of the modifier keys (Caps Lock, Shift, and Option) on the alpha keyboard.

keyCode	The physical keycode of the key whose state you want to change. Caps Lock is 0x39, Shift is 0x38, and Option is 0x3A.
down	Specify `true` to cause the equivalent of a key press. Specify `nil` to release the key.

The key is highlighted on the alpha keyboard when it is pressed (*down* = `true`), and unhighlighted when it is released (*down* = `nil`). Note that if the keyboard is open, you must send it the `Dirty` message after changing the key state in order for the visual change to occur. This is not necessary if you use the `KeyIn` function to change the key state before opening the keyboard.

PostKeyString

`PostKeyString(`*view*`, `*keyString*`)`

Sends keystrokes to a view, as if they had been entered on a keyboard.

view	The view to which to send keystrokes.
keyString	A string containing the keystrokes to send.

This function always returns `nil`.

SetKeyView

`SetKeyView(`*view*, *offset*`)`

Sets the view that is to receive keyboard input from an on-screen keyboard and positions the caret at the specified offset in that view. Note that this

function is only guaranteed to work with a `clParagraphView`. To place the caret in an edit view, you should use `SetCaretInfo` or `PositionCaret`.

view The view to receive keyboard input. This must be a `clParagraphView`. Using `nil` for this value makes the caret disappear.

offset The text caret is displayed at this character location. An offset of zero indicates the beginning of the view, an offset of one is after the first character, and so on.

Note that you may also call this function with only `nil` as the argument, to make the caret disappear. This function always returns `nil`.

Keyboard Registry Functions and Methods

If your application includes its own keyboard, you may need to use these functions. The system needs to know when keyboards are open, both for the purposes of the insertion caret and for keyboard-related callbacks.

KeyboardConnected

`KeyboardConnected()`

Returns non-`nil` if a keyboard is connected to the Newton.

OpenKeyPadFor

`OpenKeyPadFor`*(view)*

Opens a context-sensitive keyboard for the specified view.

The `OpenKeyPadFor` function first searches the proto chain to see if the view defines a keyboard in a `_keyboard` slot. If so, it opens the keyboard specified by that slot.

If the view does not define a keyboard, `OpenKeyPadFor` checks to see if the view allows only a single type of input for which the Newton system has a corresponding keyboard: date, time, phone number, or number. If so, it opens the appropriate keyboard.

If none of these other constraints is met, `OpenKeyPadFor` opens the `alphaKeyboard`.

view A view for which a context-sensitive keypad exists. Generally this should be the view that is returned by `GetKeyView`.

RegGlobalKeyboard

`RegGlobalKeyboard(`*kbdSymbol*`,` *kbdTemplate*`)`

Installs a keyboard as the only alphanumeric keyboard. This replaces the built-in alpha keyboard view.

kbdSymbol A unique identifier symbol for the keyboard view.

kbdTemplate A view template use to create the new keyboard. This template must include the the following slot:

```
preAllocatedContext
                    'This slot must have the value
                    'alphaKeyboard.
```

Note

This function may not be defined in ROM—it may be supplied by NTK. ◆

RegisterOpenKeyboard

view`:RegisterOpenKeyboard(`*flags*`)`

Notifies the system that a keyboard view has been opened and displays the insertion caret if necessary. You should call this method in your `viewSetupDoneScript`.

flags Specifies how the keyboard is used. You can use a combination of the constants shown in the section “Keyboard Registration Constants” beginning on page 8-56.

Note
Each of the keyboard prototypes automatically calls the `RegisterOpenKeyboard` method. If your are using a keyboard prototype, you need not call this method. ◆

UnRegGlobalKeyboard

`UnRegGlobalKeyboard(`*kbdSymbol*`,` *kbdTemplate*`)`

De-installs a keyboard that was installed bythe `RegGlobalKeyboard` functions. This restoresthe built-in alpha keyboard view.

kbdSymbol A unique identifier symbol for the keyboard view.

Note
This function may not be defined in ROM—it may be supplied by NTK. ◆

UnregisterOpenKeyboard

view`:UnregisterOpenKeyboard()`

Notifies the system that a keyboard view is no longer visible, which causes the insertion caret to be hidden, if necessary.

Note
The system automatically unregisters a keyboard when it is hidden or closed. ◆

Caret Insertion Writing Mode Functions and Methods

Use these functions to determine the setting of caret insertion writing mode or to set it yourself.

GetRemoteWriting

`GetRemoteWriting()`

Returns non-`nil` if caret insertion writing mode is currently enabled.

SetRemoteWriting

`SetRemoteWriting`*(newSetting)*

Sets the caret insertion writing mode preference. If *newSetting* is `nil`, caret insertion writing mode is disabled; otherwise, caret insertion writing mode is enabled.

newSetting	Indicates the new setting (enabled or disabled) for caret insertion writing mode. If *newSetting* is `nil`, caret insertion writing mode is disabled; otherwise, it is enabled.

IMPORTANT

The caret insertion writing mode is a user preference that you should very rarely override. The `SetRemoteWriting` method is meant to be called only from preferences or applications that serve a similar purpose. ▲

Insertion Caret Functions and Methods

This section describes the functions and methods that you can use to retrieve information about or manipulate the insertion caret.

GetCaretInfo

`GetCaretInfo()`

Returns `nil` if there is no insertion caret. If there is an insertion caret, returns a frame with the following two slots:

`view` The view that owns the insertion caret. This can be either a `clParagraphView` or a `clEditView`.

`info` A frame whose contents depend on the type of view in which the caret is positioned.

If the caret is in a paragraph view, the slots are:

`class`	`'paraCaret`
`offset`	The offset in characters of the caret position or the start of the selection if there is one.
`length`	The length of the selection. The value of this slot is `0` if there is no selection.

If the caret is in an edit view and not inside any existing text, the slots are:

`class`	`'editCaret`
`x`	The x-coordinate of the caret, in local coordinates.
`y`	The y-coordinate of the caret, in local coordinates.

If the caret is in a view that is more complex than a single paragraph, the slots are:

`class`	`'hilite`

GetKeyView

`GetKeyView()`

Returns the view that owns the insertion caret.

Returns `nil` if there is no insertion caret.

Note

The insertion caret may have a defined view and offset even if it is not visible. The insertion caret is only shown when caret insertion writing mode is on, a keyboard is connected, or one or more keyboards are open on the screen. ◆

PositionCaret

view:`PositionCaret`*(x, y, playSound)*

You can use the `PositionCaret` method in an edit view to position the caret at local coordinates with the view.

x	The *x* position of the insertion caret in coordinates local to the view.
y	The *y* position of the insertion caret in coordinates local to the view.
playSound	If this value is non-`nil`, the system plays a sound when the caret is positioned.

WARNING

You can use the `PositionCaret` method only with an edit view. ▲

SetCaretInfo

`SetCaretInfo(`*view*, *info*`)`

Use the `SetCaretInfo` function to restore the position of the insertion caret in a custom view that manages the caret location itself.

view	The view in which you want to modify the insertion caret information.
info	A frame in which you have specified the insertion caret information, using the same value types as are returned in the info parameter of the `GetCaretInfo` function, asdescribed on page 8-106.

WARNING

You can use the `SetCaretInfo` function to restore the caret information for caret classes `'paraCaret` or `'editCaret`. You cannot use this function to restore caret information for caret class `'hilite`. The caret classes are described on page 8-106. ▲

Application-Defined Methods for Keyboards

This section describes the keyboard-related methods that you can define, if you wish to perform keyboard-related actions at certain times.

ViewCaretChangedScript

`ViewCaretChangedScript`*(view, offset, length)*

This message is sent to a registered keyboard view whenever the caret position or text selection has changed. Implement this method for a

registered keyboard if you need to respond in some way to a change in the caret position or text selection.

view	The view in which the caret appears.
offset	Character offset of the insertion caret within the view, beginning with zero.
length	The length of the text selection. If this value is `0`, there is no selection.

viewKeyDownScript

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

This message is sent to the key receiver view whenever the user presses down on a keyboard key. This applies to using a hardware keyboard or an on-screen keyboard.

char	The character that was entered on the keyboard. Note that if a modifier key is the only key pressed (for example, the Shift key), this value will be `0`.
flags	A 30-bit integer that specifies which modifier keys were pressed and additional information, as described in Table 8-9.

Your implementation of this method should return `nil` if you want the default action to occur. The default action is usually to insert the character into the paragraph. If you return a non-`nil` value, the default action will not occur.

The *flags* value is encoded as shown in Table 8-9:

Table 8-9 Bits in the keyboard script flags word

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

The modifier key constants are described in the section "Keyboard Modifier Keys" beginning on page 8-59.

Note
You must include the `vSingleKeyStroke` flag in the view flags for the system to send the `ViewKeyUpScript` message to a view. ◆

ViewKeyUpScript

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

This message is sent to the key receiver view whenever the user releases a keyboard key that was depressed. This applies to using a hardware keyboard or an on-screen keyboard.

char	The character that was entered on the keyboard. Note that if a modifier key is the only key pressed (for example, the Shift key), this value will be `0`.
flags	A 30-bit integer that specifies which modifier keys were pressed and additional information, as described in Table 8-9 on page 8-110.

Your implementation of this method should return `nil` if you want the default action to occur. The default action is usually to insert the character into the paragraph. If you return a non-`nil` value, the default action will not occur.

Note
You must include the `vSingleKeyStroke` flag in the view flags for the system to send the `ViewKeyUpScript` message to a view. ◆

Input Event Functions and Methods

This section describes the methods that you can use to handle and respond to input events in your applications.

Functions and Methods for Hit-Testing

This section describes the methods that you can use to gather information about the location of user input in a paragraph view.

PointToCharOffset

view:`PointToCharOffset`*(x,y)*

Performs hit-testing for the character closest to the point specified by *x* and *y* in a paragraph view. The *x* and *y* values are specified as global point coordinates.

x,y Global point coordinates.

If the point (*x*,*y*) is within the paragraph margins, `PointToCharOffset` finds the character nearest to the point and returns its offset, measured from the beginning of the paragraph. If `PointToCharOffset` cannot find a character, it returns `-1`.

Note

This method works only for visible points in a paragraph view. You cannot hit-test an off-screen or clipped point. ◆

PointToWord

view:`PointToWord`*(x,y)*

Performs hit-testing for the word closest to the point specified by *x* and *y*. The *x* and *y* values are specified as global point coordinates.

x,y Global point coordinates.

If the point (*x*,*y*) is within the paragraph margins, `PointToWord` finds the word nearest to the point and returns a frame with two slots: `start` and `end`. The `start` slot specifies the offset from the beginning of the paragraph to the start of the word. The `end` slot specifies the offset from the beginning of the paragraph to the end of the word.

If `PointToWord` cannot find a word, it returns `nil`.

Note

This method only works for visible points in a paragraph views. You cannot hit-test an off-screen or clipped point. ◆

Functions and Methods for Handling Insertions

This section describes the methods and functions that you can use to handle insertion events.

The Insert Specification Frame

Several of the methods in this section receive an input parameter that is an insert specification frame. This frame contains the following six slots:

`insertItems`	The items to be inserted. This can be a single item or an array of items. Each item must be one of the valid item forms shown in Table 8-10 on page 8-114.
`addSpace`	Optional. The value `true` adds spaced between items unless `vNoSpaces` is set.
`undoable`	Optional. If `true`, indicates that the insertion can be undone; otherwise, the insertion cannot be undone. The default value is `true`.
`insertOffset`	Optional. The number of characters to offset the insertion from the beginning of the paragraph.
`replaceChars`	Optional. Replaces this number of characters, starting at the insert offset. If there is no insert offset specified, replaces the selection (if there is one.)
`moveCaret`	Optional. If `true`, indicates that the insertion caret should be moved to the position following the insertion;

otherwise, the insertion caret is not moved. The default value is `true`.

Table 8-10 Valid items in an insert specification

Item type	Description	Example
string	Used for keyboard and plain text insertions.	`"hello"`

text and styles frame	Used for styled text.
	Note that if `styles` is not an array, it is assumed to be a single `fontSpec` that applies to all of the text.
		`{`
		`text: "hi there"`
		`styles: [len, fontSpec,`
		`len, fontSpec, ...]`
		`}`

rich string	Used for rich string insertions.	
ink binary object	Used for ink words (class `'inkWord`).	
`correctInfo` frames	Used for handwritten words.	

For more information about `correctInfo` frames, see the section “protoCorrectInfo” beginning on page 10-75

HandleInsertItems

view:`HandleInsertItems`*(insertSpec)*

Use the `HandleInsertItems` method to insert one or more items into a paragraph.

You usually implement this method for paragraph views; however, you can implement it for a `clView` that has scripts set up to handle the `InsertItems` event.

insertSpec An insert specification frame, as described in the section "The Insert Specification Frame" beginning on page 8-113.

Returns `nil`.

InsertItemsAtCaret

`InsertItemsAtCaret`*(insertSpec)*

Use the `InsertItemsAtCaret` method to insert one or more items into a paragraph at the caret position. The inserted items replace the selection, if there is one.

You usually implement this method for paragraph views; however, you can implement it for a `clView` that has scripts set up to handle the `InsertItems` event.

insertSpec An insert specification frame, as described in the section "The Insert Specification Frame" beginning on page 8-113.

Note

You should not use the following insert specification frame slots for this method: `replaceChars`, `insertOffset`, and `moveCaret`. ◆

Application-Defined Methods for Handling Insertions

This section describes the messages that are sent for handling insertions.

ViewInsertItemsScript

view:`ViewInsertItemsScript(`*insertSpec*`)`

A paragraph or edit view looks for this script before handling an insertion. This script is searched for only in the proto chain of the view.

insertSpec The insert specification frame, as described in the section "The Insert Specification Frame" beginning on page 8-113.

You can handle the insertion yourself in this script, or you can modify the insert specification frame and then allow the default handler to manage the insertion.

Returns `true` if your script has handled the insertion, and `nil` if not.

Functions and Methods for Handling Ink Words

This section describes the functions and methods that you can use to work with ink words in your applications.

GetInkWordInfo

`GetInkWordInfo`*(inkWord)*

Returns information about an ink word.

inkWord An ink word.

Returns a frame with the following slots:

`origWidth`	The width of the originally written ink word.
`origAscent`	The ascent of the originally written ink word.
`origDescent`	The descent of the originally written ink word.
`origXHeight`	The x-height of the originally written ink word.
`fontFace`	The font style of the ink word.
`scale`	The scaling percentage for the ink word.
`origPenSize`	The pen width used to display the word. This is the value defined in the Styles menu.
`origFontSize`	The font size of the originally written ink word.
`curFontSize`	The current font size of the ink word.
`curPenSize`	Unused.
`curWidth`	The current (scaled) width of the ink word.
`curHeight`	The current (scaled) height of the ink word.
`curAscent`	The current (scaled) ascent of the ink word.
`curXHeight`	The current (scaled) x-height of the ink word.
`curDescent`	The current (scaled) descent of the ink word.

HandleInkWord

view:`HandleInkWord`*(strokeBundle)*

You can implement your own version of this method to handle the placement of ink words in a view. The system searches for this method in the current view and its protos.

strokeBundle	Raw stroke data for the ink word. You need to convert this data to an ink word by calling the `StrokeBundleToInkWord` method, which is described Chapter 9, “Stroke Bundles.”

Returns `true` if your method handles the incoming ink word and `nil` if not.

If you do not handle the ink word, the edit and paragraph view default handlers are used. Note that views other than edit and paragraph views do not have default handlers for ink words.

HandleRawInk

view:`HandleRawInk`*(strokeBundle)*

You can implement your own version of this method to handle the placement of sketch ink in a view. The system searches for this method in the current view and its protos.

strokeBundle Raw data for the sketch ink, as described Chapter 9, "Stroke Bundles."

Returns `true` if your method handles the incoming sketch ink and `nil` if not.

If you do not handle the sketch ink, the edit and paragraph view default handlers are used. Note that views other than edit and paragraph views do not have default handlers for ink words.

Application-Defined Methods for Handling Ink in a View

This section describes the messages that are sent for handling ink in a view.

ViewInkWordScript

view:`ViewInkWordScript`*(strokeBundle)*

The system sends this message when an ink word has been recognized and is sent to a view. The system searches for this method in the current view and its protos.

strokeBundle Stroke data for the ink word.

Returns `true` if your method handles the incoming ink word and `nil` if not.

If you do not handle the ink word, the edit and paragraph view default handlers are used. Note that views other than edit and paragraph views do not have default handlers.

ViewRawInkScript

view:`ViewRawInkScript`*(strokeBundle)*

This message is sent when sketch ink is passed to a view. The system searches for this method in the current view and its protos.

strokeBundle Stroke data for the sketch ink.

Returns `true` if your method handles the incoming sketch ink and `nil` if not.

If you do not handle the sketch ink, the edit and paragraph view default handlers are used. Note that views other than edit and paragraph views do not have default handlers.

Summary of Text

This section summarizes the constants, data structures, functions, and methods that you can use when working with text in your applications.

Text Constants and Data Structures

This section describes the constants and data structures that you use when working with text.

Text Flags

```
vWidthIsParentWidth     (1 << 0)
vNoSpaces               (1 << 1)
vWidthGrowsWithText     (1 << 2)
vFixedTextStyle         (1 << 3)
vFixedInkTextSTyle      (1 << 4)
vExpectingNumbers       (1 << 9)
vSingleKeystrokes       (1 << 10)
```

Built-in Font Constants

ROM_fontsystem9	9216
ROM_fontsystem9bold	1057792
ROM_fontsystem9underline	4203520
ROM_fontsystem10	10240
ROM_fontsystem10bold	1058816
ROM_fontsystem10underline	4204544
ROM_fontsystem12	12288
ROM_fontsystem12bold	1060864
ROM_fontsystem12underline	4206592
ROM_fontsystem14	14336
ROM_fontsystem14bold	1062912
ROM_fontsystem14underline	4208640
ROM_fontsystem18	18432
ROM_fontsystem18bold	1067008
ROM_fontsystem18underline	4212736
simpleFont9	9218
simpleFont10	10242
simpleFont12	12290
simpleFont18	18434
fancyFont9 or userFont9	9217
fancyFont10 or userFont10	10241
fancyFont12 or userFont12	12289
fancyFont18 or userFont18	18433
editFont10	10243

```
editFont12                  12291
editFont18                  18435
```

Font Family Constants

```
(none)
tsFancy
tsSimple
tsHWFont
```

Font Face Constants

```
tsPlain         0
tsBold          1048576
tsItalic        2097152
tsUnderline     4194304
tsOutline       8388608
tsSuperScript   134217728
tsSubScript     268435456
```

Keyboard Registration Constants

```
kKbdUsesKeyCodes     1
kKbdTracksCaret      2
kKbdforInput         4
```

Key Descriptor Constants

```
keySpacer            (1 << 29)
keyAutoHilite        (1 << 28)
keyInsetUnit         (1 << 25)
keyFramed            (1 << 23)
keyRoundingUnit      (1 << 20)
keyLeftOpen          (1 << 19)
keyTopOpen           (1 << 18)
keyRightOpen         (1 << 17)
```

```
keyBottomOpen        (1 << 16)
keyHUnit             (1 << 11)
keyHHalf             (1 << 10)
keyHQuarter          (1 << 9)
keyHEighth           (1 << 8)
keyVUnit             (1 << 3)
keyVHalf             (1 << 2)
keyVQuarter          (1 << 1)
keyVEighth           (1 << 0)
```

Keyboard Modifier Keys

```
kIsSoftKeyboard      (1 << 24)
kCommandModifier     (1 << 25)
kShiftModifier       (1 << 26)
kCapsLockModifier    (1 << 27)
kOptionsModifier     (1 << 28)
kControlModifier     (1 << 29)
```

Structured List View Functions and Methods

This section summarizes the functions and methods that you can use to work with structured list views in your applications.

list:ActiveTopic(*anIndex*)
list:AddEmptyTopic(*anIndex*, *aLevel*)
list:CalcLevel(*anX*)
list:ClickBounds(*anIndex*)
list:CollapseTopic(*anIndex*)
list:DeleteTopics(*aList*)
list:DirtyBelow(*anIndex*)
list:EnsureVisibleTopic(*anIndex*)
list:ExpandTopic(*anIndex*)
list:GetTopicCount()

list:HandleCheck(*anIndex*, *aTopic*)
list:HandlePriority(*anIndex*, *aTopic*)
list:HandleScrub(*aBounds*)
list:HasKids(*anIndex*)
list:IsCollapsed(*anIndex*)
list:IsEmpty(*topicIndex*)
list:IsReadOnly(*topicIndex*)
list:ListBottom()
list:MakeTopicFrame(*aLevel*)
list:MarkerBounds(*aTopic*)
list:Mother(*anIndex*)
list:NewRelative(*aRelation*)
list:NewTopic(*aLevel*, *aText*)
list:OlderSister(*anIndex*)
list:PickActionScript(*anAction*)
list:ResolveClick(*anX*, *aY*)
list:RevealTopic(*anIndex*)
list:SetDone(*anIndex*, *aTopic*, *aDone*, *aPropogateUp*, *aPropogateDown*)
list:SetPriority(*anIndex*, *aPriority*, *anUndo*)
list:SetReadOnly(*anIndex*, *aStatus*)
list:ToggleTopic(*anIndex*)
list:TopicBottom(*anIndex*)
list:TopicHeight(*anIndex*)
list:YoungerSister(*anIndex*)

Text and Ink Display Functions and Methods

This section summarizes the functions and methods that you can use to work with text and ink in your applications.

Functions and Methods for Edit Views

view:EditAddWordScript(*form, bounds*)
NotesText(*childArray*)

Functions and Methods for Measuring Text Views

TextBounds(*rStr, fontSpec, viewBounds*)
TotalTextBounds(*paraSpec, editSpec*)

Functions and Methods for Determining View Ink Types

AddInk(*edit, poly*)
ViewAllowsInk(*view*)
ViewAllowsInkWords(*view*)

Font Attribute Functions and Methods

FontAscent(*fontSpec*)
FontDescent(*fontSpec*)
FontHeight(*fontSpec*)
FontLeading(*fontSpec*)
GetFontFace*(fontSpec)*
GetFontFamilyNum*(fontSpec)*
GetFontFamilySym*(fontSpec)*
GetFontSize*(fontSpec)*
MakeCompactFont*(family, size, face)*
SetFontFace*(fontSpec, newFace)*
SetFontFamily*(fontSpec, newFamily)*
SetFontParms *(fontSpec, whichParms)*
SetFontSize*(fontSpec, newSize)*

Rich String Functions and Methods

DecodeRichString*(richString, defaultFontSpec)*
view:ExtractRangeAsRichString*(offset, length)*
view:GetRichString()
IsRichString*(testString)*

`MakeRichString`*(text, styleArray)*
`StripInk`(*richString, replaceString*)

Functions and Methods for Accessing Ink in Views

`GetInkAt`*(para, index)*
`NextInkIndex`*(para, index)*
`ParaContainsInk`*(para)*
`PolyContainsInk`*(poly)*

Keyboard Functions and Methods

This section summarizes the functions and methods that you can use to work with keyboards in your applications.

General Keyboard Functions and Methods

`GetCaretBox()`
view`:KeyboardInput()`
`KeyIn(`*keyCode, down*`)`
`PostKeyString(`*view, keyString*`)`
`SetKeyView(`*view, offset*`)`

Keyboard Registry Functions and Methods

`KeyboardConnected()`
`OpenKeyPadFor`*(view)*
`RegGlobalKeyboard(`*kbdSymbol, kbdTemplate*`)`
view`:RegisterOpenKeyboard(`*flags*`)`
`UnRegGlobalKeyboard(`*kbdSymbol*`)`
view`:UnregisterOpenKeyboard()`

Caret Insertion Writing Mode Functions and Methods

`GetRemoteWriting()`
`SetRemoteWriting`*(newSetting)*

Insertion Caret Functions and Methods

`GetCaretInfo()`
`GetKeyView()`
view`:PositionCaret`*(x, y, playSound)*
`SetCaretInfo(`*view, info*`)`

Application-Defined Methods for Keyboards

`ViewCaretChangedScript`*(view, offset, length)*
`ViewKeyDownScript(`*keyCharacter, keystrokeFlag*`)`
`ViewKeyUpScript(`*keyCharacter, keystrokeFlag*`)`

Input Event Functions and Methods

This section summarizes the functions and methods that you can use to work with input events in your applications.

Functions and Methods for Hit-Testing

view`:PointToCharOffset`*(x,y)*
view`:PointToWord`*(x,y)*

Functions and Methods for Handling Insertions

view`:HandleInsertItems`*(insertSpec)*
`InsertItemsAtCaret`*(insertSpec)*

Application-Defined Methods for Handling Insertions

view`:ViewInsertItemsScript(`*insertSpec*`)`

Functions and Methods for Handling Ink Words

`GetInkWordInfo(`*inkWord*`)`
view`:HandleInkWord`*(strokeBundle)*
view`:HandleRawInk`*(strokeBundle)*

Application-Defined Methods for Handling Ink in a View

view:`ViewInkWordScript`*(strokeBundle)*
view:`ViewRawInkScript`*(strokeBundle)*

CHAPTER 9

Stroke Bundles

This chapter describes stroke bundles, which are used by the Newton recognition system to represent the ink drawn by the user on the screen. You can access the information in stroke bundles if you want to perform your own recognition, or if you want to examine or modify the information before it is recognized.

About Stroke Bundles

A **stroke bundle** is a representation of the ink used to draw text or shape on the Newton screen. Each stroke bundle is a decompressed version of ink that can contain multiple strokes. Each stroke in the bundle is a binary object that contains tablet-resolution data.

The Newton system software provides accessor functions that allow you to retrieve the table points from each stroke. You can access these points in one of two resolutions: **screen resolution** or **tablet resolution**. In screen resolution, each coordinate value is rounded to the nearest screen pixel. In tablet resolution, each coordinate has an additional three bits of data.

Using Stroke Bundles

You can use the stroke bundle functions to work with ink data in your applications. For example, you might want to perform some processing of ink words before they are passed to the recognition system.

You can use one of several functions that are documented in Chapter 8, "Text and Ink Input and Display" to access the ink in a view. These include the `ParaContainsInk`, `PolyContainsInk`, and `GetInkAt` functions.

The system software provides a number of functions for working with stroke bundles. These functions, which are documented in the next section, "Stroke Bundle Reference" beginning on page 9-3, allow you to extract information from a stroke bundle and to convert the information in stroke bundles into other forms.

An Example of Using Stroke Bundles

This section shows an example of working with stroke bundles before they are passed to the recognition system. One way to do this, as shown in the code below, is to implement the `viewInkWordScript` method of an input view. The `viewInkWordScript` method is described in Chapter 8, "Text and Ink Input and Display."

```
GetKeyView().viewInkWordScript := func(strokeBundle)
begin
   // convert the stroke bundle into an ink word
local inkPoly := CompressStrokes(strokeBundle);
local inkWord := inkPoly.ink;
local textSlot := "\uF701";
local stylesSlot := [1, inkWord];
local root := GetRoot();
   // create a rich string with the ink word in it
```

```
local appendString := MakeRichString(textSlot,
                                     stylesSlot);
   // append the rich string to myRichString
if root.myRichString then
root.myRichString := root.myRichString && appendString;
else
root.myRichString := appendString;
   // return nil so default handling still happens
nil;
end;
```

This implementation converts the stroke bundle into an ink word, creates a rich string that includes the ink word, and appends that rich string to a rich string that is stored in the root (`myRichString`). The method then returns nil, which allows the built-in handling of the stroke bundle to occur.

Stroke Bundle Reference

This section describes the constants, data structures, functions, and methods that you use to work with stroke bundles in your applications.

Stroke Bundle Constants

This section describes the constants that you use to work with stroke bundles in your applications.

Data Resolution and Filtering Constants

Several of the stroke bundle functions use a format specification to determine the resolution of point data. Some of the function also use this format

specification to determine whether or not to copy duplicate point values. The format specification values are shown in Table 9-1.

Table 9-1 Stroke bundle data format specifications

Value	Description
`0`	Data in screen resolution. Filter out duplicate points.
`1`	Data in screen resolution. Duplicate points are allowed.
`2`	Data in tablet resolution. Filter out duplicate points.
`3`	Data in tablet resolution. Duplicate points are allowed.

NOTE

Points are stored in a compressed format that is based on screen resolution.

Filtering of duplicate points is irrelevant for several of the stroke bundle functions. These functions use screen resolution if you supply a filter value of `0` or `1`, and use tablet resolution if you supply a filter value of `2` or `3`. For example, the `GetStrokePoint` function, which is described on page 9-8, retrieves a specific point from a stroke bundle, and needs to know only the resolution in which to return that point.

Stroke Bundle Data Structures

This section describes the data structures that you use to work with stroke bundles in your applications.

The Stroke Bundle Frame

The stroke bundle frame describes the point data from an ink stroke drawn on the Newton tablet. This frame contains two slots:

`bounds`	The bounding rectangle for the ink strokes in the bundle.
`strokes`	An array with one element for each stroke in the bundle. Each element is a binary object containing tablet resolution data.

Stroke, Word, and Gesture Units

The Newton recognition system uses stroke units to describe information about pen input. You cannot look at a stroke unit directly, but some of the stroke bundle and recognition functions use this object type as an input parameter. Stroke units are the object type that is passed into the `viewStrokeScript` method of a view.

The Newton recognition system also uses other units, including word units, which are passed to a view in the `viewWordScript`, and gesture units, which are passed to a view in the `viewGestureScript` method.

Stroke, word, and gesture units, as well as the application-defined view methods that use them, are described in Chapter 10, “Recognition.”

Point Arrays

Several of the stroke bundle functions use or return point arrays. This is a single array of coordinate values, with alternating Y and X coordinates.

Note that the first value in each pair is the Y coordinate value, followed by the X coordinate value.

The point array structure is the same structure type that is returned by the `GetPointsArray` function, which is described in Chapter 10, “Recognition.”

Stroke Bundle Functions and Methods

This section describes the functions and methods that you can use in your applications to work with stroke bundles.

CompressStrokes

`CompressStrokes`*(strokeBundle)*

Compresses the *strokeBundle* and returns a polygon view.

strokeBundle A stroke bundle frame, as described in the section "The Stroke Bundle Frame" on page 9-5.

CountPoints

`CountPoints`*(stroke)*

Returns the number of points in the specified *stroke*.

stroke A binary object representing an ink stroke.

The `CountPoints` function counts the number of points in stroke and returns the count as an integer value.

CountStrokes

`CountStrokes`*(strokeBundle)*

Returns the number of strokes in the stroke bundle.

strokeBundle A stroke bundle frame, as described in the section "The Stroke Bundle Frame" on page 9-5.

ExpandInk

`ExpandInk`*(poly, format)*

Decompresses the ink in a polygon view and returns it as a stroke bundle.

poly A `clPolygonView`, which is stored as a child of a `clEditView` and has an `ink` slot. You can test this by calling the `PolyContainsInk` function, which is

described in Chapter 8, “Text and Ink Input and Display.”.

format The data resolution and filtering value. Use one of the values shown in Table 9-1 on page 9-4.

The stroke bundle returned by `ExpandInk` uses the same coordinate system as does the polygon view and has the same bounds as the polygon view. Every point within the returned stroke bundle falls within those bounds.

If you expand ink at tablet resolution, the returned stroke bundle contains points that are at the highest resolution that can be derived from the compressed ink. If you expand ink at screen resolution, the points in the stroke bundle are spaced at a resolution approximately equal to screen resolution. The former expansion is suitable for recognition; the latter for display.

ExpandUnit

`ExpandUnit`*(unit)*

Creates a stroke bundle from a information in *unit* and returns the stroke bundle, which uses global coordinate values.

unit An object that describes pen input information, as described in the section “Stroke, Word, and Gesture Units” on page 9-5. This is the object passed to one of the following application-defined view methods: the `viewStrokeScript` method (stroke units), the `viewWordScript` method (word units), or the `viewGestureScript` method (gesture units).

Note that if you would want a reference to the stroke bundle that is cached in a word unit, you should use the `GetCorrectionWordInfo` function, which returns a frame that contains the stroke bundle in a slot named `strokes`. This function is documented in Chapter 10, “Recognition.”

GetStroke

`GetStroke`*(strokeBundle, index)*

Returns the stroke binary object specified by *index* from the strokes array in the *strokeBundle*.

strokeBundle	A stroke bundle frame, as described in the section "The Stroke Bundle Frame" on page 9-5.
index	An integer specifying a stroke in the stroke bundle array.

GetStrokeBounds

`GetStrokeBounds`*(stroke)*

Calculates the bounding rectangle for the specified *stroke*, and returns it as a frame.

stroke	A binary object representing an ink stroke.

GetStrokePoint

`GetStrokePoint`*(stroke, index, point, format)*

Copies the data from a specified point in a stroke into a new point.

stroke	A binary object representing an ink stroke.
index	An integer specifying the point in the stroke to copy.
point	A frame containing slots named `x` and `y`.
format	The data resolution and filtering value. Use one of the values shown in Table 9-1 on page 9-4. Note that the duplication filter is ignored by this function.

The `GetStrokePoint` function copies the data for the point in *stroke* specified by *index*. The data is copied into the *point* frame, using the resolution specified by *format*.

GetStrokePointsArray

`GetStrokePointsArray`*(stroke, format)*

Copies the data for all the points in *stroke* into an array. The points are filtered and scaled according to the value of the *format* parameter.

stroke A binary object representing an ink stroke.

format The data resolution and filtering value. Use one of the values shown in Table 9-1 on page 9-4.

The `GetStrokePointsArray` function returns a point array, as described in the section “Point Arrays” on page 9-5.

InkConvert

`InkConvert`*(ink, outputFormat)*

Converts ink data from the Newton System 1.x format to the Newton System 2.x ink data format and vice-versa.

ink A binary object that contains the ink to be converted.

outputFormat A symbol that defines the conversion type. Use one of the following values:

`'ink`	The *ink* is converted to 1.x compatible ink.
`'ink2`	The *ink* is converted to 2.0 sketching ink.
`'inkword`	The *ink* is converted to 2.0 ink text.

The `InkConvert` function converts the input *ink* to a different format and returns the converted ink as a binary object.

If *ink* is not a valid ink object, the `InkConvert` function returns `nil`.

MakeStrokeBundle

`MakeStrokeBundle`*(strokes, format)*

Creates a stroke bundle from an array of points.

strokes	An array of point arrays. The structure of each point array is described in the section "Point Arrays" on page 9-5. Each point array represents the coordinate data for a single stroke in the stroke bundle.
format	The data resolution and filtering value for the point values in the *strokes* array. Use one of the values shown in Table 9-1 on page 9-4.

The `MakeStrokeBundle` function uses the coordinate data in *strokes* to create a stroke bundle and returns that bundle. The input data is assumed to be in the resolution specified by *format*.

You can use the `MakeStrokeBundle` function to synthesize ink words for recognition; however, the quality of recognition is uncertain for such data. The recognizer generally requires high-quality, tablet-resolution data to work accurately.

MergeInk

`MergeInk`*(poly1, poly2)*

Decompresses the ink words in two polygons and recompresses them as a union of the original two.

poly1, poly2	A `clPolygonView`, which is stored as a child of a `clEditView` and has an `ink` slot. Test this by using the method `PolyContainsInk`, which is described in Chapter 8, "Text and Ink Input and Display.".

The `MergeInk` function assumes that both of its input parameters, *poly1* and *poly2*, are polygon views that contain ink words and that these views are horizontally adjacent with no intervening space.

The `MergeInk` function decompresses the ink in *poly1* and *poly2* and then recompresses the union of those two. The resulting polygon is returned as the function value.

If a memory error occurs, `MergeInk` returns `nil`.

PointsArrayToStroke

`PointsArrayToStroke`*(pointsArray, format)*

Creates a stroke from a point array.

pointsArray	A point array, as described in the section "Point Arrays" on page 9-5.
format	The data resolution and filtering value for the point values in the *strokes* array. Use one of the values shown in Table 9-1 on page 9-4.

The `PointsArrayToStroke` function creates a stroke from the coordinate data in pointsArray and returns the stroke object. The resolution of the input points is specified by *format*.

Note that the `PointsArrayToStroke` function is the inverse of the `GetStrokePointsArray` function, which is described on page 9-9.

SplitInkAt

`SplitInkAt`*(poly, x, slop)*

Examines a polygon with ink for a word break and splits the polygon at that word break. The result is an array of two polygons, each of which contains an ink word.

poly	A `clPolygonView`, which is stored as a child of a `clEditView` and has an `ink` slot. Test this by using the method `PolyContainsInk`, which is described in Chapter 8, "Text and Ink Input and Display."
x	An integer specifying the horizontal position near which to look for a word break.
slop	An integer specifying how far in either direction (from *x*) to search for a word break. The recommended value for *slop* is somewhere between `xHeight` and `xHeight/2` for the word.

If `SplitInkAt` finds a break, it returns an array whose first element is a polygon containing the first word, and whose second element is the polygon

containing the second word. If `SplitInkAt` cannot find a reasonable break, it returns `nil`.

NOTE
The `SplitInkAt` function never finds a word break in the middle of a stroke. ◆

StrokeBundleToInkWord

`StrokeBundleToInkWord`*(strokeBundle)*

Converts the stroke bundle to an ink word. You can pass the resulting ink word object to the HandleInsertItems function, which is described in Chapter 8, "Text and Ink Input and Display."

strokeBundle A stroke bundle frame, as described in the section "The Stroke Bundle Frame" on page 9-5.

Summary of Stroke Bundles

Stroke Bundle Functions and Methods

`CompressStrokes`*(strokeBundle)*
`CountPoints`*(stroke)*
`CountStrokes`*(strokeBundle)*
`ExpandInk`*(poly, format)*
`ExpandUnit`*(unit)*
`GetStroke`*(strokeBundle, index)*
`GetStrokeBounds`*(stroke)*
`GetStrokePoint`*(stroke, index, point, format)*
`GetStrokePointsArray`*(stroke, format)*
`InkConvert`*(ink, outputFormat)*
`MakeStrokeBundle`*(strokes, format)*
`MergeInk`*(poly1, poly2)*
`PointsArrayToStroke`*(pointsArray, format)*
`SplitInkAt`*(poly, x, slop)*
`StrokeBundleToInkWord`*(strokeBundle)*

CHAPTER 10

Recognition

This chapter describes the recognition system and the concepts that underlie its use. The recognition system accepts written input from views and returns text, ink text, graphical objects or sketch ink to views.

This chapter includes information on configuring the recognition system, using deferred recognition, using custom dictionaries for recognition, and accessing correction information for misrecognized words.

If you are developing an application that needs to derive text or graphical data from written input, you should become familiar with the recognition system and the concepts discussed in this chapter.

Before reading this chapter, you should be familiar with

- message-passing between views and the use of view flags to specify the characteristics of views, as described in Chapter 3, “Views.”
- text and ink, as described in Chapter 8, “Text and Ink Input and Display.”
- stroke bundles, as described in Chapter 9, “Stroke Bundles.”

You may also find it helpful to be familiar with soups, as described in Chapter 11, “Data Storage and Retrieval.”

About the Recognition System

The Newton recognition system uses a sophisticated multiple-recognizer architecture that incorporates recognizers for text, shapes, and gestures. Most recognition events are handled automatically by the system-supplied view classes. Your application need not handle recognition events explicitly unless it needs to do something unusual. You can specify numerous aspects of each view's recognition behavior including the amount by which the user can modify this behavior.

For each view that accepts user input, you can specify which recognizers are enabled and how they are configured. For example, you might enable only the text recognizer for a view that is not intended to accept graphical input. In addition, the text recognizer can be configured to recognize only names, or names and phone numbers, and so on.

When the user writes or draws on the screen, the system first attempts to group the strokes into meaningful units based on temporal and spatial data. This grouping process is influenced by the user preference settings for handwriting style and letter styles.

When the units are made available to the recognition system, all of the recognizers enabled for the view compete equally to classify the input. Each recognizer compares the input to a system-defined model; if there is a match, the recognizer claims the strokes as its own. Once a set of strokes is claimed it is not returned to the other recognizers for additional classification.

The number of interpretations returned varies according to which recognizer actually interpreted the strokes. The gesture and shape recognizers return only one interpretation to the view. The text recognizer may return multiple interpretations.

Associated with each interpretation is a value, called the **score**, which indicates how well the input matched the system-defined model used by the recognizer that returned the interpretation. When multiple recognizers are

enabled, the system selects the best interpretations based on their scores and the application of appropriate heuristics.

The system places the best interpretations in a **unit** that is returned to the view. The view displays or uses the interpretation having the best score. In the case of text recognition, the best of these alternate interpretations are made available to the user to facilitate the correction of misrecognized words.

As you can see, the recognition system's classification of user input is essentially a process of elimination. Enabling and configuring only the recognizers and dictionaries appropriate to a particular context is the developer's primary means of optimizing recognition system performance. Configuration of the recognition system is discussed in detail later in this chapter.

The system's next action depends on whether it interpreted the input as a gesture, as a shape or as text. The next several sections describe how the system handles each of these interpretations.

Gestures

When a stroke is recognized as one of the system-defined gestures the system performs the action associated with that gesture. System-defined gestures include scrubbing items on the screen, adding spaces to previously-recognized words, selecting items on the screen and so on.

You can define a `ViewGestureScript` method for views in which you wish to take action in response to these gestures. For custom gestures, you must implement Undo capabilities yourself.

Shapes

When a stroke or group of strokes is interpreted as a shape, the recognition system returns a single interpretation that the view images on the screen; this interpretation is a "cleaned up" version of the original strokes. The shape recognizer may make the original shape more symmetrical, straighten its curves or close the shape, according to user preference settings.

Text

The system includes a cursive word recognizer and a printed word recognizer. The user can switch between them from the Handwriting Recognition Preferences slip. Only one text recognizer can be active at a time—all views on the screen share the same text recognizer.

When a stroke or group of strokes is interpreted as text, the text recognizer usually returns multiple interpretations of the word. These interpretations may be words from the system dictionaries or they may be groups of individually-recognized characters, according to the recognizer in use and how it is configured.

The cursive recognizer normally works with several system-supplied dictionaries to analyze user input. As used here, the term **dictionary** refers to a system construct against which the user's input strings are matched. The user can add new words to a system-supplied user dictionary and developers can provide custom dictionaries for the recognition system's use.

For dictionary-based recognition, you specify the set of dictionaries to be used in addition to the system-defined handwriting model. The system supplies dictionaries for common words such as names, places, dates, times, phone numbers and commonly-used words. In addition, you can create custom dictionaries to supply specialized words such as medical or legal terminology.

The system also maintains a RAM-based dictionary to which the user can add new words as they are successfully recognized. The user can easily correct misrecognized words and add new ones to the user dictionary.

The printed recognizer uses dictionaries as well, but may also return strings that are not necessarily dictionary words.

Note that the cursive recognizer can be configured to perform character-based recognition also. In views configured to support it, this behavior may be enabled by the user from the Handwriting Settings slip that is available from the Options button in the Handwriting Recognition preferences slip. The "Letter-by-letter in fields" and "Letter-by-letter in notes" checkboxes in this slip enable character-based recognition. The "Letter-by-letter in fields" checkbox enables character-based recognition in

`protoLabelInputLine` views that are configured to support it. You can also enable this behavior programmatically in other kinds of views. The "Letter-by-letter in notes" checkbox enables this behavior for the built-in notepad application and notes associated with items in the Names and Dates applications.

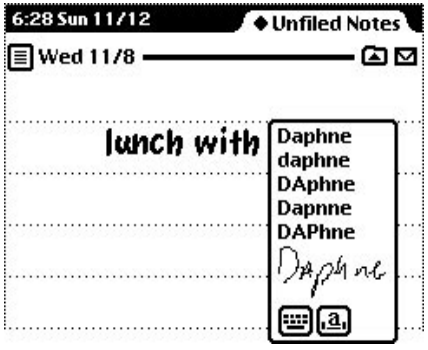
Once the text recognizer claims the strokes, it may return multiple interpretations of this input. The score associated with each interpretation indicates how well the original input matched the model against which it was compared. The view displays the interpretation having the best score and stores the alternate interpretations as correction data. The top-ranking alternate interpretations may include both dictionary-based and character-based interpretations.

Correcting Misrecognized Words

Text-input views handle the correction of misrecognized text automatically. When the user double-taps a word to correct it, a pick list displays words that the recognition system supplied as possible matches for the hand-written input. In addition to these alternate interpretations of the text, the list includes a representation of the original ink and buttons that display the soft keyboard and corrector views, as shown in Figure 10-1.

The words in this list are one example of correction information stored by the system as it recognizes words; in addition to word lists, correction information includes the original stroke data and other information described later in this section. The correction data is preserved through undo operations; it is also preserved in drag and drop operations.

Figure 10-1 Corrector pick list



The list of alternate interpretations displayed when the user double-taps recognized text includes

- the five best interpretations returned by the recognizer
- the alternate capitalization of the most highly-scored interpretation (if not already in the list)
- the expansions of words that match entries in the expansion dictionary.

For complete descriptions of the system prototypes that provide access to correction information, see the section “Recognition System Prototypes” beginning on page 10-54.

Configuring Views for Recognition

There are two ways to specify recognition characteristics for each view that accepts user input. The application developer can set the view’s view flags or supply a recognition configuration frame for the view. The recognition configuration frame, also called the `recConfig` frame, is a new feature introduced in version 2.0 of Newton system software. View flags are a feature of the original Newton object system that remains fully supported in version 2.0 of Newton system software.

View Flags

The recognizers and dictionaries enabled for a particular view are determined by the view flags that are configured for the view; hence, this set of flags defines the kinds of data that the user can enter in the view.

You can use the graphical view editor in Newton Toolkit to set view flags or you can use NewtonScript to set them programmatically. Either approach allows you to set combinations of view flags to produce a variety of behaviors. This chapter uses NewtonScript for all view flag examples. For information on using Newton Toolkit to set view flags, see *A Guide To The Newton Toolkit*.

Each view flag sets bits in a bit field to specify characteristics that the view does not inherit from its view class, such as recognition behavior. Not all of the bits in this field affect recognition; some are used to set other characteristics such as the view's placement on the screen. The bits in this field that affect the recognition system are referred to as the view's **input mask**. These bits are put in the view's `viewFlags` slot when the view is constructed at run time.

Recognition Configuration Frames

The implementation of specialized recognition behavior may require settings that view flags do not provide. The `recConfig` frame allows you to set individual bits in the input mask while still fully supporting the view flag model. You can also specify other aspects of recognition system configuration in this frame, such as the dictionaries to be used for dictionary-based recognition. The `recConfig` frame provides more flexible and precise control over the configuration of recognition behavior than view flags do, but a `recConfig` frame requires a little more effort to set up.

Using View Flags or RecConfig Frames

In most cases, view flags provide the easiest and most efficient way to configure the recognition system. It is recommended that you use view flags to configure recognition unless you need some special recognition behavior that they cannot provide. Examples of such behavior include the support of

ink text and the constraint of recognition on a character-by-character basis. More examples of special recognition behavior requiring the use of `recConfig` frames are described later in this chapter, in the section "About Recognition Configuration Frames" beginning on page 10-9.

The rest of this section discusses configuration of the recognition system only in terms of the view flag model. You need to read this section even if you are planning on using `recConfig` frames in your application. The description of `recConfig` frames later in this chapter assumes an understanding of the view flag model upon which these frames are based.

View Flags and Recognition Performance

To obtain the most accurate results from the recognition system, you must define as precisely as possible the type of input that the view is to recognize. By restricting the possible interpretations returned by the recognition system to only those that are appropriate for a particular view, you increase the system's chances of interpreting the input correctly. For example, when configuring a view for entry of phone number data, you would not specify that the recognition system return alphabetic characters to that view.

Aside from potentially introducing errors, enabling unnecessary recognizers may also slow the recognition system's performance. For best performance, you need to specify the minimum combination of view flags required to recognize the type of input you anticipate for the view. As a rule, keep in mind that "Less is more" when configuring view flags for recognition.

Recognition Failure

Recognition may fail when the handwritten input is too sloppy for the system to make a good match against its internal handwriting model, or (in the case of dictionary-based recognition only) when none of the words match a dictionary entry. In such cases, the recognition system may return sketch ink rather than text or ink text.

Ink text looks similar to sketch ink; however, ink text is scaled and placed in a paragraph view as text. Sketch ink is not placed in a paragraph but drawn in a `clPolygonView` on top of whatever else is in the view. The sketch ink

holds stroke data for use by deferred recognition at another time. For more information, see Chapter 9, “Stroke Bundles.”

When text recognition is not enabled, the system recognizes words as ink text. When ink recognition is not enabled, the system returns sketch ink. Table 10-1 summarizes the kinds of data returned by the recognition system under various circumstances.

Table 10-1 Data returned when recognition fails

recToggle setting	Returns on failure
Text	ink text
ink text	ink text (does not fail)
shapes	sketch ink
sketch ink	nothing (extremely rare)

About Recognition Configuration Frames

Views requiring special recognition behavior may need to configure the recognition system in ways that the view flag model does not support. For such views, version 2.0 of Newton system software provides the `recConfig` frame as a means of configuring the recognition system programmatically. The use of `recConfig` frames affords the software developer more flexible and precise control than view flags provide. The `recConfig` frame can be used to specify any set of recognizers and dictionaries for recognition, including combinations not supported by the view flag model.

Certain new features of system software version 2.0 require the use of `recConfig` frames. Any view that uses ink text must provide an appropriate `recConfig` frame. Views can also use `recConfig` frames to constrain character recognition in precise ways.

Although the Newton platform currently supports only its built-in recognizers, future versions of the system will permit the use of third-party

recognizer engines. The `recConfig` frame allows applications to specify recognition configurations in a uniform way that is not dependent on the use of any particular recognizer engine.

The system provides several standard `recConfig` frames which can be referenced by your view's `recConfig` slot or used as a starting point for building your own `recConfig` frames. The system-supplied `recConfig` frames are described later in this chapter.

About Dictionaries

There are two kinds of dictionaries used by the system: enumerated and lexical. An enumerated dictionary is simply a list of dictionary items (strings) that can be matched. A lexical dictionary specifies a grammar or syntax that is used to classify user input. The kind of dictionary used for a particular task is dependent upon task-specific requirements. The system-supplied dictionaries include both enumerated and lexical dictionaries.

Dictionaries can be in ROM or in RAM. Most of the system-supplied dictionaries are in ROM; however, the user dictionary resides in RAM. Note that RAM-based and ROM-based custom dictionaries can also reside on a PCMCIA card.

About the User Dictionary

Applications must never add items to the user dictionary without the user's consent. The user dictionary is intended to be solely in the user's control—adding items to it is akin to changing the user's handwriting preferences or Names file entries. It's also important to leave room for users to store their own items.

IMPORTANT

An excessively large user dictionary can slow the system when performing searches that are not related to your application. It is therefore recommended that applications do not add items to the user dictionary at all. ▲

The user dictionary browser supports a total of about 1,000 items in the RAM-based user dictionary. Note that this number may change in future Newton devices. A persistent copy of the user word list is kept on the user store in the `UserDictionary` system soup entry. This entry is loaded when you restart and saved when you close the Personal Word List slip.

The user dictionary also lets you define word expansions that happen automatically after the word has been recognized, but before it has been displayed. These definitions are stored in a separate dictionary called the expand dictionary.

About the Expand Dictionary

The expand dictionary allows the user user to specify the substitution of an entirely different string for one that has been recognized. For example, the user may specify that the string `w/` expand to `with`, or the string `appt` expand to `appointment`. The expand dictionarly is also useful for automatically correcting recurrent recognition mistakes or misspellings.

The expand dictionary is not used directly by the recognition system. Instead each word to be expanded is added to both the user dictionary and the expand dictionary. The user dictionary and any appropriate additional dictionaries are used to perform stroke recognition. Before the recognizer returns the list of recognized words to the view, it determines whether any of the items in the list are present in the expand dictionary. If so, the expanded version of the word is inserted into the list of recognized words before the original version of the word. The original version is left in the list in case the user doesn't want the expansion to take place.

The expand dictionary itself is stored in RAM and its size is limited to 256 entries. A persistent copy of the expand dictionary is stored in the system soup entry named `ExpandDictionary`. This soup entry is loaded when the system restarts and is saved when the Personal Word List slip is closed.

About the AutoAdd Dictionary

The autoadd mechanism adds new words to the user dictionary automatically as they are recognized. This feature is enabled when the

cursive recognizer is active, but not when the printed recognizer is active. Note that the printed recognizer uses the user dictionary to some extent, so you can still improve the recognition of problematic non-dictionary words by adding them to the user dictionary, but you'll have to do so manually by invoking the user dictionary's `AddWord` method (see page 10-77.)

The autoadd dictionary is a list of words that have been added to the user dictionary automatically. The user can display this list to decide whether any of the recently added words should actually stay in the user dictionary.

If the autoadd dictionary is not empty, the Recently Added Words slip is displayed when the user opens the Personal Word List. This slip prompts the user to step through the words in the autoadd dictionary, one by one, to decide whether each should be left in the user dictionary. To encourage the user to make decisions about all of the words in the list, selection is not allowed in this slip.

Note that although words are actually added to both the user dictionary and the autoadd dictionary automatically, the slip asks the user whether to add each word to the Personal Word List. In reality, this slip actually removes words from these dictionaries according to user input.

Note

The automatic addition of words to the user dictionary and the AutoAdd dictionary is disabled when the Printed recognizer is enabled, even though in the user preferences slip it appears to be enabled. ◆

The size of the autoadd dictionary is limited to 100 words. A persistent copy of the autoadd dictionary is stored in the user store in the system soup entry named `AutoAdd`. This entry is loaded when the system restarts and saved when the user opens or edits the Recently Added Words slip.

About Recognition Areas and Dictionary Lists

The recognition system creates a data structure called a **recognition area** which describes the recognition characteristics for one or more views (similar views may share an area). The use of recognition areas allows the system to

switch easily between views having different recognition behaviors without unnecessary repetition of the setup work for recognition in a particular view.

When the user writes, draws or gestures in a view, the system passes the view's `recConfig` frame to the recognition area associated with that view. If the view has no `recConfig` frame, the recognition system builds one, taking into account the view flags enabled for that view and user preferences as appropriate. The recognition area uses the `recConfig` frame to determine the precise settings for all the recognizers needed for that view.

When the system builds the recognition area associated with the view, it also stores a list of the dictionaries to be used for recognition of input to that view. Note that a single dictionary constant can represent multiple dictionaries; for example, when the `kCommonDictionary` constant is specified, the system may actually add several dictionaries to the list.

■

About User Preferences for Recognition

The user can specify several preferences that affect the overall configuration of the recognition system. This information is provided for reference purposes only; generally, you should not interfere with the user's settings for recognition preferences.

This section describes only those user preferences for which a developer interface to the recognition system has been provided. It does not provide a comprehensive summary of the user interface to recognition, which may vary on different Newton devices. For a description of the user interface for a particular Newton device, see the user manual for that device.

The user preference settings described here may be affecterd by the setting of a `protoRecToggle` view associated with the view performing recognition. For more information about the `protoRecToggle` view, see its description on page 10-67.

The user preference settings described here may also be overridden by a `recConfig` frame associated with the view performing recognition. For

more information about the `recConfig` frame, see its description on page 10-54.

In views that set the `vAnythingAllowed` flag, the actual run-time configuration of the recognition system is derived from a combination of the recognition preferences specified by the user and the current setting of an associated `protoRecToggle` view.

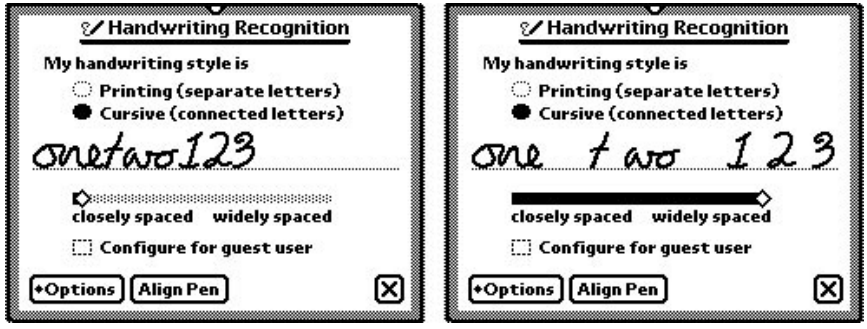
For more information about view flags, including the `vAnythingAllowed` flag, see the section "The viewFlags Slot" beginning on page 10-20.

Handwriting Recognition

The Handwriting Recognition preference panel shown in Figure 10-2 specifies the overall characteristics of the user's handwriting. For example, the user can specify here whether a printed or cursive style of lettering is used. This system-wide setting enables either the printed or cursive recognizer by setting the value of the `letterSetSelection` slot in the global `userConfiguration` frame. Do not override this setting.

The user can also specify the amount of blank space the recognizer may find between words; this setting influences the recognition system's initial grouping of stroke data. The value returned by the slider control in this slip is kept in the `letterSpaceCursiveOption` slot in the global `userConfiguration` frame and may be overridden by the view.

Figure 10-2 Handwriting Recognition preferences



The “Configure for guest user” item takes the following actions:

saves all current recognition system settings

saves owner’s recognition system learning data

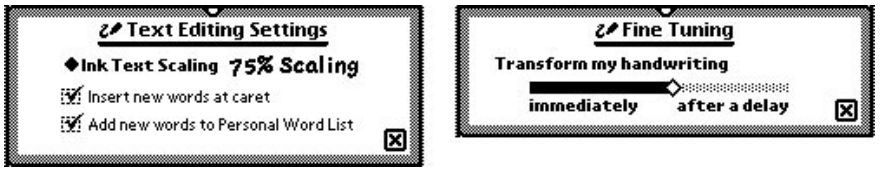
temporarily resets all recognition system preferences to default values

learns guest user’s writing style as misrecognized words are corrected

The Options button displays a popup menu from which you can access preferences for the various recognizers. The items included in this menu vary according to whether the printed or cursive recognizer is enabled.

Figure 10-3 shows the Text Editing Settings and Fine Tuning preferences panels, which are displayed for both the printed and cursive recognizers. Of the adjustments available from the Text Editing Settings slip, the “Add new words to Personal Word List” checkbox is of interest to users of the recognition system. Only the cursive recognizer adds new words to the RAM-based user dictionary automatically; however, words can always be added manually regardless of which recognizer is enabled. To display or edit the personal word list, the user taps the book icon on the soft keyboard.

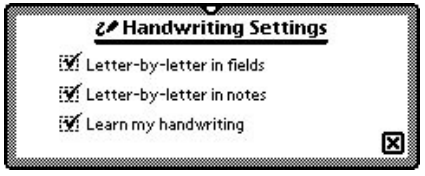
Figure 10-3 Text Editing Settings and Fine Tuning preferences



The Fine Tuning slip provides a “Transform my handwriting” slider that allows the user to fine-tune the system’s use of temporal cues to determine when a group of strokes is complete. The slider control in this slip sets the value of the `timeoutCursiveOption` slot in the global `userConfiguration` frame.

When the cursive recognizer is enabled, the Options button in the Handwriting Recognition preferences slip provides access to the Handwriting Settings slip shown in Figure 10-4.

Figure 10-4 Handwriting Settings slip



When the “Learn my handwriting” box is checked, the system sets the value of the `learningEnabledOption` slot in the global `userConfiguration` frame to `nil`.

About Deferred Recognition

Deferred recognition is the ability to convert strokes to text at some time other than when the strokes are first entered on the tablet. Deferred recognition was introduced in Newton system software version 1.3, which was first made available on the MessagePad 100 and MessagePad 110.

Views that are to perform deferred recognition must be capable of capturing ink text or ink; for example, a view that uses the `ROM_InkOrText` recConfig frame and a `protoRecToggle` view to configure recognition is capable of performing deferred recognition.

This section describes the user interface to deferred recognition and follows that with a programmer's overview of this system service.

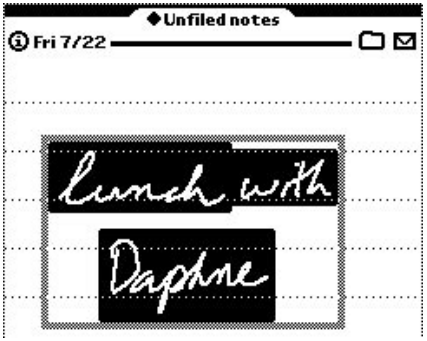
User Interface to Deferred Recognition

The user can enter unrecognized ink by turning on ink text or sketch ink. In this mode, the user's notes appear as ink.

Deferred recognition uses a combination of the current user-specified recognition preference settings as well as those the application developer has specified for the view in which recognition takes place.

To convert ink to text, the user double-taps the ink word; multiple words can be recognized by selecting them beforehand and then double-tapping any word in the selection. The recognition system responds by inverting the ink word or selection, as shown in Figure 10-5, and returning the recognized text.

Figure 10-5 User interface to deferred recognition



Programmer's Overview of Deferred Recognition

Deferred recognition is available in views based on the `clEditView` class or `clParagraphView` views that support ink text. This feature works with any amount of input, from a single letter to a full page of text.

The user must double-tap "ink" children of the edit view to initiate deferred recognition. The recognized text is added to the edit view as if the user had just written it. That is, a new `clParagraphView` child is added, or the recognized text is appended to a nearby `clParagraphView`. After the recognized text has been added, the original view containing the sketch ink or the ink text is removed as if the user had scrubbed out the ink.

The recognition also invokes the `viewAddChildScript` and `viewDropChildScript` methods of the recognized text and unrecognized ink views. Words added to nearby paragraphs invoke `viewChangedScript` methods for those paragraphs, updating the `'text` slot; for some paragraph views, the `'viewBounds` slot is updated as well.

Compatibility Information

In addition to the cursive recognizer available in previous systems, version 2.0 of system software adds a recognizer optimized for printed characters. The particular recognizer in use is a system-wide preference set by the user.

System software prior to version 2.0 does not necessarily call the view's `ViewClickScript` and `ViewStrokeScript` methods for every pen tap. If recognizers that group strokes are enabled, these methods are called only once for each group. System software version 2.0 always calls the view's `ViewStrokeScript` method once for each stroke when the vStrokesAllowed flag is set. System software version 2.0 always calls the view's `ViewClickScript` method regardless of the setting of the `vStrokesAllowed` flag.

The `protoLetterByLetter` prototype, which appears at the lower-left corner of the screen in the Notepad application on the MessagePad 100 and MessagePad 110, is obsolete. It has been replaced by the `protoRecToggle` prototype described in this chapter.

Prior to version 2.0 of Newton system software, correction information was not accessible from NewtonScript. Version 2.0 of Newton system software makes this information available as frame data.

Using the Recognition System

This section describes how to enable recognizers for views and how to use the system-supplied prototypes related to recognition. This section presumes understanding of the material in previous sections.

This chapter discusses only those view flags that interact with the recognition system. For information on other kinds of view flags, see the section "Defining View Characteristics" in the Views chapter of the first edition of the *Newton Programmer's Guide*.

Configuring the Recognition System

You can take the following approaches to configuring the recognition system:

- set view flags only. This approach works well for most applications.
- set view flags and allow the user to configure recognition from a `protoRecToggle` view that you provide.
- set view flags and supply a recognition configuration frame based on `ROM_InkOrText`.This approach is the easiest way to support ink text. You can use a `protoRecToggle` view here as well.
- supply a recognition configuration frame of some other kind. This approach offers you the most control and flexibility, but also requires the most work to implement. Support for `protoRecToggle` views depends on the particular implementation of your `recConfig` frame.
- use the `recogSettingsChanged` message sent by the `protoRecToggle` view to reconfigure the recognition system at run time.

The viewFlags Slot

Except where noted, all of the flags described in this section are set in the view's `viewFlags` slot.

The value of the `viewFlags` slot is actually a bit field and the view flags themselves are bit masks that can be combined to provide a particular set of attributes for the view. Not all of the bits that can be set in the `viewFlags` slot pertain to recognition, however.

Note

Newton ToolKit displays recognition-oriented view flags in an area of the screen called Entry Flags because these flags control the entry of recognized data. However, in the code that NTK generates (and when you set these flags procedurally) they are placed in the view's `viewFlags` slot. Hence, this chapter refers to these flags as "view flags." ◆

Accepting Pen Input

When setting up any view, you must specify whether it accepts user input at all. If you set the value of the view's `viewFlags` slot to `vNothingAllowed` the view does not accept pen input. If you want the view to accept pen input you must set the `vClickable` flag in its `viewFlags` slot. Setting this flag only causes the view to accept pen taps and call `ViewClickScript` methods; it does not enable ink-handling or calls to any of the unit-handling scripts that provide recognition behavior. To provide these behaviors, you must use view flags or `recConfig` frames to set additional bits in the input mask.

Setting the `vClickable` flag specifies that the view system is to send the `viewClickScript` message to the view once for each pen tap (click) that occurs within the view. Note that this is the case only when vClickable is the only flag set for the view—to cause the view to recognize user input, you will need to set additional flags.

Additional system messages are sent to the view to signal taps, strokes, gestures and words.

All pen input is first signaled by the `ViewClickScript` message (page 10-95). This message indicates that the pen contacted the tablet within the boundaries of the view. If this message is not handled by the view and additional recognition flags are set, other messages may be sent to the view, depending on what was written. These other messages include `ViewStrokeScript` (page 10-96), `ViewGestureScript` (page 10-98), and `ViewWordScript` (page 10-100)—in that order if all are sent.

Each of the input-related messages has at least one parameter—a unit. The unit contains information about the pen input. There is no way to look at the unit directly, but you can pass it to other system-supplied functions to get information from it such as the beginning and ending points of the stroke, an array of stroke points, the stroke bounds, and so on. Functions that operate on units are described in this chapter and in Chapter 8, "Text and Ink Input and Display."

IMPORTANT

You must set the `vClickable` flag for any view that is to accept pen input; no taps or strokes are passed to the view if this flag is not set. ▲

Using Dictionary-Based Text Recognition

To enable dictionary-based recognition in a view that accepts pen input, add the `vCharsAllowed` flag to the values already present in the view's `viewFlags` slot.

The `vCharsAllowed` flag specifies that the recognition system is to match the user input as closely as possible with the pool of words specified by the set of dictionaries currently available to the application. This set of dictionaries includes those enabled by the `vCharsAllowed` flag and any other flags that are set for the view.

The `vCharsAllowed` flag enables the user dictionary and a default set of dictionaries. The default dictionaries provide vocabulary used in common speech, names of days, names of month and proper names.

The set of dictionaries available to a view may also include other system-supplied dictionaries as well as custom dictionaries supplied by the application developer. The next section describes how various sets of dictionaries are enabled for views that perform dictionary-based recognition.

Using Predefined Sets of Dictionaries

In most cases, it is not appropriate for the recognition system to work with the entire set of available dictionaries. The system-supplied view flags specify sets of dictionaries that are well-suited to recognizing particular kinds of input such as dates, phone numbers and so on. The system shows no preference towards any single dictionary in the set except for a slight weighting of results in favor of words found in the User Dictionary.

The use of the word "Field" in the names of some flags and the word "Allowed" in others does not indicate any functional difference. It merely

reflects the common usage of these flags. The so-called “field” flags are more likely to be used singly, in views designed for only one type of input. The “allowed” flags, on the other hand, are more likely to be used in combination with one another, where several types of input are possible or “allowed” as input to a particular view.

Despite differences in naming conventions (and despite certain limitations imposed by NTK), the “field” and “allowed” flags can be mixed in any combination. Keep in mind, though, that the more choices the recognizer has, the more opportunity it has to pick the wrong choice.

When you need precise control over the dictionaries used for recognition, you can specify explicitly which dictionaries the view must use, as discussed in the next section.

Customized Dictionaries

Setting the `vCustomDictionaries` flag for the view allows you to specify explicitly which dictionaries the view must use for text recognition. This approach is useful for views that must use custom dictionaries to recognize specialized terms, such as medical or legal vocabulary. It’s also useful for views that must restrict the dictionaries used to a specific set; for example, a view that recognizes city names might use this approach to match input against only the dictionaries that store such information.

When the `vCustomDictionaries` flag is present in the view’s `viewFlags` slot, the view must contain a slot named `dictionaries`, which you must create. The `dictionaries` slot stores an array of values that identify the dictionaries to be used for recognition. The system uses these dictionaries in addition to those specified by the remaining flags.

To use custom dictionaries along with those supplied by the system, you need to

- use the `Bor` function to bitwise OR the `vCustomDictionaries` flag with those flags already present in the `viewFlags` slot.
- store the identifiers of your custom dictionaries in the `dictionaries` slot.

To restrict the view to using only dictionaries specified in the `dictionaries` slot, place only the `vCustomDictionaries`, `vClickable` and `vGesturesAllowed` flags in the view's `viewFlags` slot. Note that the printed recognizer can always return words not present in dictionaries. Only the cursive recognizer allows you to restrict the system to returning only words that are present in dictionaries.

For more details, see the section "Using Dictionaries" beginning on page 10-40.

Recognizing Text Not In Dictionaries

Setting the `vLettersAllowed` flag in the view's `viewFlags` slot allows the cursive recognizer to perform letter-by-letter recognition in addition to or in place of dictionary-based recognition. Note that the printed recognizer can always return words not present in dictionaries.

When the `vLettersAllowed` flag is used in combination with other flags that enable dictionaries, the recognizer shows preference to dictionary-based words but may resort to letter-by-letter recognition when dictionary-based recognition does not produce acceptable results. When used by itself, this flag may facilitate the recognition of more unusual combinations of letters, with a resultant increase in the possibility of error when recognizing normal words.

Recognizing Special Characters

The `vPunctuationAllowed` flag permits the cursive recognizer to return common punctuation marks such as the period (`.`), comma (`,`) question mark (?), single quotation marks (‘ and ’), double quotation marks (“ and ”) and so on. Views restricted to the entry of phone numbers, dates or times only need not set the `vPunctuation` only flag because the `vPhoneField`, `vDateField` and `vTimeField` flags already allow the entry of appropriate punctuation.

The cursive recognizer can also apply some simple rules when deciphering ambiguous characters; for example, it can make use of the fact that most punctuation marks follow rather than precede words.

Suppressing Extraneous Spaces Within Words

Setting the `vSingleUnit` flag causes the text recognizer to ignore spatial information when grouping input strokes as words; instead it relies on temporal cues to determine when the user has finished writing a word. When this flag is set, the recognizer ignores short delays, such as those that occur between writing the individual characters in a word. Longer delays cue the recognizer to group the most recently-completed set of strokes as a word. The amount of time considered to be a longer delay is a function of the speed of the processor and the recognition system as well as the user preference settings for handwriting recognition.

The `vSingleUnit` flag is useful for views in which the presence of gratuitous spaces may confuse the recognizer; for example, phone number entry fields usually suppress the recognition of spaces. If you want to suppress all spaces in the displayed text, you can use the `vNoSpacesFlag` in conjunction with the `vSingleUnit` flag, as described in the section "Suppressing All Spaces" beginning on page 10-25.

Post-processing Recognition Results

The flags described in this section direct the system to modify the text returned by the recognition system before displaying it in the view. These flags do not restrict the interpretation of the input strokes nor do they assist the recognition system in choosing between alternative interpretations of the input.

Suppressing All Spaces

The `vNoSpaces` flag suppresses the display of spaces in recognized text. The `vNoSpaces` flag must appear in an evaluate slot named `textFlags` that you create in the view. The `vNoSpaces` flag is often used in conjunction with the `vSingleUnit` flag; the `vSingleUnit` flag appears in the view's `viewFlags` slot, as usual.

Capitalized Words Only

The `vCapsRequired` flag directs the system to capitalize the first letter of each word returned by the recognizer before displaying the text in the view.

Processing Non-Text Tablet Input

The `vGesturesAllowed`, `vShapesAllowed` and `vStrokesAllowed` flags enable gesture recognition, shape recognition and custom processing of input strokes. These three flags are independent of each other and can be used in any combination.

Recognizing Shapes

The `vShapesAllowed` flag enables the recognition of geometric shapes such as circles, straight lines, polygons and so on. Do not set this flag for views that handle text input only. This flag is intended for use only in edit views that handle geometric shapes. An example of such a view is the `clEditView` view that provides the built-in Notepad application with much of its recognition behavior.

Recognizing Gestures

Setting the `vGesturesAllowed` flag supplies system-defined behavior for the gestures tap, double-tap, highlight and scrub. Most input views set the `vGesturesAllowed` flag, as they need to respond to standard gestures such as scrubbing to delete text or ink. When the `vGesturesAllowed` flag is set, the gesture recognizer invokes the view's `viewGestureScript` method before handling the gesture. Note that you normally don't need to supply a `viewGestureScript` method because the system handles the entire set of predefined gestures automatically in these views.

If you do implement a `viewGestureScript` method, it is called before the system-supplied method. Your `viewGestureScript` method should return the value `True` to avoid passing the unit to the `viewGestureScript` supplied by the system. If you do want to pass the unit to the

system-supplied `viewGestureScript` method, your method should return the value `nil`.

Customized Handling of Input Strokes

Setting the `vStrokesAllowed` flag provides the view with a means of intercepting raw input data for application-specific processing. When this flag is set, the view's `viewStrokeScript` method is invoked when the pen is lifted from the screen at the end of the user's input stroke. Strokes completed within the amount of time specified by the value of the `timeoutCursiveOption` value are passed one at a time as the argument to the view's `viewStrokeScript` method. Your `viewStrokeScript` method can then process the strokes in whatever manner is appropriate.

Note
Both the `vGesturesAllowed` and `vStrokesAllowed` flags invoke methods that can be used to provide application-specific handling of gestures. However, the `vGesturesAllowed` flag supplies system-defined behavior for the gestures tap, double-tap, highlight and scrub, while the `vStrokesAllowed` flag does not provide any behavior that you don't implement yourself. ◆

The next section describes how to combine view flags to produce different kinds of recognition behaviors.

Combining View Flags

Because multiple recognizers are often enabled for a single view, you may configure multiple view flags to specify the recognizers to be used. Keep in mind that "Less is more" when configuring view flags. That is, the fastest and most accurate recognition occurs when the fewest recognizers necessary to successfully analyze the input are enabled. Enabling unnecessary recognizers can potentially slow down recognition and introduce errors.

The following examples describe how to combine view flags to produce various kinds of recognition behavior.

Recognizing Numbers and Gestures

Use the flags in the following example to enable the recognition of numbers and gestures.

```
vClickable+vGesturesAllowed+vNumbersAllowed
```

The `vNumbersAllowed` flag enables the recognition of numbers. Numeric recognition includes monetary amounts (for example, $12.25), decimal points, and signs (+ and -).

Note

Note that all of these combinations of view flags include the `vClickable` flag. You must set `vClickable` or the view will not accept pen input! ◆

Recognizing Dictionary Words, Punctuation and Gestures

Use the flags in the following example to enable the recognition of punctuation, gestures and words in the system-supplied dictionaries (including the user dictionary.)

```
vClickable+vGesturesAllowed+vCharsAllowed+vPunctuationAllowed
```

The vAnythingAllowed Flag

The `vAnythingAllowed` flag can be used only for views derived from the `clEditView` class. When this flag is present in the view's `viewFlags` slot, the recognition system replaces the set of view flags specified by the developer with a set of flags derived from the current settings of user preferences that affect recognition. The recognition behavior of views that use this flag varies according to current user preference settings. The built-in Notepad application provides a good example of the behavior of view that use this flag.

The `vAnythingAllowed` flag is a mask that enables all of the recognizers in the system if used by itself. Although setting the `vAnythingAllowed` flag

potentially enables all recognizers, the actual set of recognizers enabled for the view is controlled by

- user settings
- the application's `protoRecToggle` view if it has one.
- the view's `recConfig` frame if it has one.

The settings specified by the `recConfig` frame override those specified by the `protoRecToggle` view; thus, the `protoRecToggle` view can only provide those choices specified by the `recConfig` frame.

The settings specified by the `protoRecToggle` view override those specified by user preferences; in other words, the user can only set from preferences those things that the `protoRecToggle` view allows.

Thus, in practice the `vAnythingAllowed` flag usually is not what its name implies: if any bit in this mask is turned off (by another flag, for example) the mask is no longer `vAnythingAllowed`.

Text Flags

To control certain aspects of text-handling and recognition behavior for any view derived from the `clParagraphView` class, you can add an evaluate slot named `textFlags` to the view and set the following flags in this slot.

`vNoSpaces` Spaces are suppressed as input to this field; that is, the spaces are removed before the word is displayed.

`vWidthIsParentWidth` The right boundary of the text field is extended to match that of its parent. You do not need to set this value yourself; the system sets it for you in views that use it.

Using RecConfig Frames

This section describes how to use a `recConfig` frame to control recognition behavior. Note that the use of view flags is generally the best (and simplest) way to configure the recognition system to accept common input such as

words and shapes.You need not use a `recConfig` frame unless you require some recognition behavior that cannot be achieved by using the view's `viewFlags` and `dictionaries` slots. For example, the use of a `recConfig` frame is required for views that support ink text, constrain character-based recognition, or implement customized forms of deferred recognition.

Creating a recConfig frame

You must create a slot named `recConfig` in any view that is to use a `recCofig` frame.

The frame residing in this slot must contain an `inputMask` slot. For more information, see the description of this slot under "Input Mask Slots" beginning on page 10-54.

This frame may include a `_proto` slot that references one of the system-supplied `recConfig` frames.For complete descriptions of these frames, see "System-Supplied RecConfig Frames" beginning on page 10-60.

The `recConfig` frame may also contain additional slots described in this section. For a complete listing of the slots that may appear in the recConfig frame, see the section "protoRecConfig" beginning on page 10-54.

Using protoRecToggle Views With recConfig Frames

The `protoRecToggle` view changes the values of the slots in the `userConfiguration` frame to effect changes in recognition system behavior for a view. Note that the settings specified in the view's `recConfig` frame may override the `protoRecToggle` settings, which in turn may override settings in the `userConfiguration` frame.

The `protoRecToggle` view is only used for

- views that set the `vAnythingAllowed` flag
- views that support ink text.

To change the recognition behavior for a view controlled by a `protoRecToggle` view, add the `doTextRecognition`,

doShapeRecognition and doInkWordRecognition slots to your recConfig frame as necessary. These slots are described in the "Recognizer Configuration Slots" section of the description of the protoRecConfig frame on page 10-54.

For some operations, you may wish to restrict the recognizers that the user can enable in a view while still respecting the rest of the preferences indicated in the global userPreferences frame. The optional slots allowTextRecognition, allowShapeRecognition and allowInkWordRecognition in the view's recConfig frame are intended for use with views having an input mask that sets the vAnythingAllowed bit. These slots are described in the "recToggle Configuration Slots" section of the description of the protoRecConfig frame on page 10-54.

Views that use the allow*Something*Recognition slots allow the user to turn on only the recognizers that you specify while respecting all other user preferences. Any subset of allow*Something*Recognition slots can be specified to allow the user to enable any appropriate combination of recognizers from the protoRecToggle view or user preferences.

For example, setting the value of the allowTextRecognition slot to True allows the view to recognize ink as text if the user turns on the text recognizer. When the value of the allowTextRecognition slot is nil, the view cannot recognize text regardless of the user preference settings. The view cannot recognize shapes unless the allowShapeRecognition slot has the value True and the user turns on shape recognition from user preferences.

Controlling the Cursive Recognizer

You can add the optional speedCursiveOption, timeoutCursiveOption and letterSpaceCursiveOption slots to the recConfig frame to control various aspects of the cursive recognizer. Generally these are set by the user through a preferences mechanism, and shouldn't be set by the application. However, for those occasions on which the application needs to override recognition preferences, these slots may be used. These slots are described in the "Cursive Recognizer Configuration

Slots” section of the description of the protoRecConfig frame on page 10-57.

Controlling the Ink Recognizer

The doInkWordRecognition and doRawInkRecognition slots in the recConfig frame directly affect the recognition of ink words. Normally, the view tries to recognize input by enabling recognizers. If no recognizers are enabled or recognition fails for some reason—for example, due to messy input or some sort of error—then the view system converts the input strokes into ink. The doInkWordRecognition and doRawInkRecognition slots control the kind of ink that the system creates from the input strokes.These slots are described in the “Ink Recognizer Configuration Slots” section of the description of the protoRecConfig frame on page 10-58.

The big difference between sketching ink and inkText ink is not the ink itself, but what the view does with it. Text turned into inkText is treated as text, and inserted into existing text as if it is a recognized word. On the other hand, sketching ink is treated as a graphic and inserted into the view as a polygon. As a rule of thumb, consider ink and inkText to be mutually exclusive when configuring recognition in views; that is, set the view up to recognize only one of these two data types for best results.

Note that clEditView views handle ink and inkWord text automatically. For other views, the system invokes the view’s viewInkWordScript or viewRawInkScript method when ink arrives. For more details, see Chapter 8, “Text and Ink Input and Display.”

Controlling the Shape Recognizer

To cause the view to recognize shapes, you must set the vShapesAllowed bit in its input mask. Recognized shapes are added properly to views derived from the clEditView class. These views have no viewShapeScript method, nor is there any way to examine the shapes returned by the recognizer.

Your recConfig frame may include the symmetryShapeOption, curveShapeOption and gravityShapeOption slots when the shape

recognizer is enabled. These slots are described in the “Cursive Recognizer Configuration Slots” section of the description of the `protoRecConfig` frame on page 10-57.

Manipulating Dictionaries

Your `recConfig` frame may include the `dictionaries`, `rcSingleLetters` and `inhibitSymbolsDictionary` to control the view’s use of dictionaries. These slots are described in the “Dictionary configuration slots” section of the description of the `protoRecConfig` frame on page 10-59.

Specifying the Size and Location of Single Letter Input

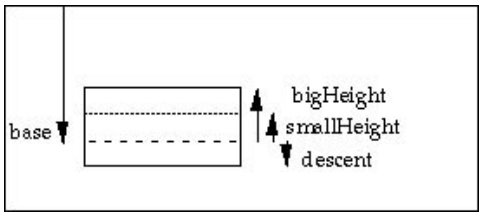
When recognizing single letters it is sometimes difficult for the recognizer to determine individual characters’ baseline or size. If you know precisely where the user will be writing with respect to a baseline, you can provide an `rcBaseInfo` frame or `rcGridInfo` frame to specify to the recognition system precisely where the characters are written.

The rcBaseInfo Frame

If you know where the user will be writing with respect to a baseline—a visible guideline on the screen with which the user can align written input—then you can provide an `rcBaseInfo` frame that specifies to the recognizer more precisely where the characters are written. These frames are especially valuable in improving the recognition of single letters, for which it is sometimes difficult to derive baseline or letter size values from user input.

Figure 10-6 depicts the editing box that an `rcBaseInfo` frame defines.

Figure 10-6 Single-character editing box specified by `rcBaseInfo` frame



The NewtonScript code used to create the baseline information for the editing box shown in Figure 10-6 looks like the following example.

```
rcBaseInfo := {
   base:    140,// y coordinate of baseline
   smallHeight:15,// height of a lower case x
   bigHeight:30,// height of an upper case X
   descent:15,// size of descender below baseline
   };
```

To obtain the best performance and to conserve available memory, create your `rcBaseInfo` frame by cloning the frame stored in the `ROM_canonicalBaseInfo` constant. Store your frame in a slot named `rcBaseInfo` in your view's `recConfig` frame.

For a complete description of the rcBaseInfo frame see the section "rcBaseInfo" beginning on page 10-82.

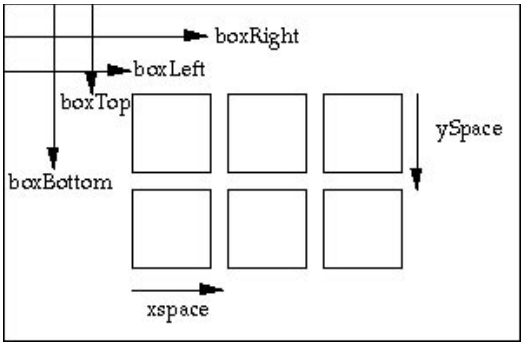
The rcGridInfo Frame

If you provide a grid in which the user is to write characters or words, you need to use an `rcGridInfo` frame to define the grid to the text recognizer. An example of such a grid is the one displayed by the corrector when the user double-taps a previously-recognized word. Recognition in the most recently-used grid box begins as soon as the user writes in a new box in the grid.

The recognizer requires the information in an `rcGridInfo` frame in order to make character-segmentation decisions. You can use the `rcGridInfo` frame in conjunction with an `rcBaseInfo` frame to provide more accurate recognition within boxes.

Figure 10-7 depicts the two-dimensional array of boxes that an `rcGridInfo` frame defines.

Figure 10-7 Two-dimensional array of input boxes specified by rcGridInfo frame



The NewtonScript code used to create the grid shown in Figure 10-7 looks like the following example.

```
rcGridInfo := {
   boxLeft: 100,// x coordinate of left of topleft box
   boxRight:145,// x coordinate of right of topleft box
   xSpace:55,// x distance from boxLeft to boxLeft
   boxTop: 50,// y coordinate of top of topleft box
   boxBottom:95,// y coordinate of bottom of topleft box
   ySpace:55// y distance from boxTop to boxTop
   };
```

To obtain the best performance and to conserve available memory, create your `rcGridInfo` frame by cloning the frame stored in the `ROM_canonicalGridInfo` constant. Store your frame in a slot named `rcGridInfo` in your view's `recConfig` frame.

IMPORTANT ▲
The `ROM_canonicalGridInfo` constant is not yet available from the Dante Platforms file as of the date of this printing. To access `canonicalBaseInfo`, use the `@682` magic pointer directly. ▲

Changing Recognition Behavior Dynamically

A small number of recognition areas are kept in a cache. If you want to change the recognition behavior of a view dynamically, you can do so by invalidating the area cache. Using the global function `SetValue` to change either the `viewFlags` or `recConfig` slot invalidates the area cache automatically. You can also use the `PurgeAreaCache` function to invalidate this cache. This approach works for views configured by view flags, too.

IMPORTANT
The view's `viewFlags` slot must contain the same recognition flags as the `inputMask` slot in its `recConfig` frame. Certain view system operations depend on the `viewFlags` slot being set up properly. ▲

Using protoCharEdit Views

This section describes how to position protoCharEdit views and restrict the set of characters they recognize.

Positioning protoCharEdit Views

There are two ways to position a `protoCharEdit` view on the screen. You can set the values of its `top` and `left` slots to the desired values for the top

left corner of the view, or you can just provide a value for the view's `viewBounds` slot.

If you specify the values of the `top` and `left` slots then the `viewSetupFormScript` method of the `protoCharEdit` view supplies an appropriate value for its `viewBounds` slot for you. On the other hand, if you provide the value of the `viewBounds` slot, then this method supplies appropriate values for the `maxChars` and `cellHeight` slots for you.

The technique you use depends on how you want to set the slots provided by this proto. For detailed information, see the section "protoCharEdit" beginning on page 10-67.

Text in protoCharEdit Views

The initial value of the text displayed by the `protoCharEdit` view is specified by the value of its `text` slot or by the value of the `string` slot in the template used to restrict the view's input. At any time, you can change the text string that this view displays by sending the `SetupWordScript` message to the view. Similarly, you can change the template by sending the `SetupTemplateScript` message.

To get the current value of the text in the view, you need to send the `CleanWord` message to the view. You must not use the value of the `text` slot directly because it contains extra characters that pad the string for editing.

Restricting Characters Recognized By protoCharEdit Views

The recognition behavior of a `protoCharEdit` view is controlled by its `template` slot. If the template slot is `nil`, then the view's recognition behavior is similar to that of the corrector used by the built-in Notepad application: all characters are allowed, insertion and deletion are fully supported, and spaces are added at the ends of the words to allow words to be extended. The character entry fields are said to be **free-form entry fields** because input is not restricted in any way.

You can also define templates that restrict the set of characters recognized in each position of the `protoCharEdit` view. For example, a phone number is likely to restrict the set of permissible characters to numerals. A field in which a template limits the set of characters to be recognized is a **restricted entry field**. When the user taps on a character in a restricted entry field, it simply displays the list of characters defined by the template, rather than choices based on what the user wrote, if the list defined by the template is sufficiently small.

The template specifies an initial value for the string that the view displays, the editing characteristics for each position in the comb field, and filters that restrict the values recognized in each of these positions.

The template may contain any of the `format`, `filters`, `string` and `setupString` slots, as well as optional methods you supply to manipulate data in the `string` slot.For complete descriptions of these slots, see the "Template slots" portion of the section "protoCharEdit" beginning on page 10-67.

The system provides date and time templates that handle the various international date and time conventions properly. The templates defined by the following code fragments are intended to serve as examples only.

The first code example defines a template for a date field.

```
digits := "0123456789";// filter[0]
digits1 := "01";// filter[1]
digits3 := "0123";// filter[2]

dateTemplate := {
   string:"  /  /  ",// slashes locked by "_" in format
   format:"10_20_00",// digits are indexes into filters
array
   filters:[digits, digits1, digits3],
   };
```

You can see that the numeric fields have filters that restrict their input; the positions with slashes do not allow input because the format string has underscores in those positions.

Note that this template is not and should not be smart about the various days allowed within each month. Because the user might enter the day before the month, or because there may be a recognition error in the month, it is not wise to restrict the input of the day based on the current value of the month. The application can look at the current content of the comb view and indicate an error condition if necessary.

Here is an example of a time field that defines `SetupString` and `CleanupString` functions.

```
digits := "0123456789";// filter[0]
digits1 := "01";// filter[1]
digits5 := "012345";// filter[2]
ap := "ap";    // filter[3]

timeTemplate := {
   format:"0000001",
   string:"      0",
   filters: [kMyDigitsOrSpace, kMyDigits],

   SetupString: func(str) begin
      // pad string to 5 digits
      if StrLen(str) < 7 then
         StrMunger(str,0,0,string,0,7-StrLen(str));
      str;
   end,

   CleanupString: func(str) begin
      // replace spaces with zeros
      StrReplace(str, " ", "0", nil);
      // trim leading 0's
```

```
      str := NumberStr(StringToNumber(str));
      str;
   end,
};
```

Accessing Correction Information

The correction information frame constructed from the unit is stored in the `correctInfo` global variable. The correction information frame contains the alternate interpretations returned by the recognizer, the raw stroke data, and information that the system uses to alter its internal handwriting model when the user corrects misrecognized words.

For more information, see the descriptions of the `protoCorrectInfo`, `protoWordInterp` and `protoWordInfo` prototypes in the section "Recognition System Reference."

Using Dictionaries

Besides using the system-supplied dictionaries, your application can use custom dictionaries to recognize specialized terms, such as medical or legal vocabulary. Currently, you can create RAM-based custom dictionaries. Future versions of NTK will allow developers to create dictionaries residing on PCMCIA cards (ROM or RAM.)

It's relatively easy to create a RAM-based enumerated dictionary at run time. The ability to create a lexical dictionary or ROM-based dictionaries (lexical or enumerated) will be included in a future release of NTK.

To create a custom enumerated dictionary, you must populate a blank RAM-based dictionary with your dictionary items. Dictionary items can come from a number of places: they might be elements of your own array of strings stored in the application's Project Data file; they could be represented as binary resource data in your application's NTK project; they might be supplied by the user in an input line view; they might even arrive as serial data. Because dictionary items can originate from a number of sources, the

example here presumes that you know how to store your word strings and pass them, one at a time, to the `AddWordToDictionary` function.

The `AddWordToDictionary` function does not place any restrictions on the strings to be entered in the dictionary. For non-recognition purposes, any string is a valid dictionary entry; for example, you might use a dictionary to restrict entries in a data field to a specified set of valid entries. For use in stroke recognition, however, each string in an enumerated dictionary must have the following characteristics:

- The string must consist entirely of lowercase alphabetic characters.
- The string must not include spaces, digits, or non-alphabetic characters.

You can take the following steps to create a RAM-based enumerated dictionary at run time:

1. Use the global function `NewDictionary` to create a new empty dictionary.
2. Use the global function `AddWordToDictionary` to add dictionary items to the new dictionary.
3. Use the global function `GetDictionaryData` to create a binary representation of the completed dictionary, which can then be stored in a soup.

Another way to do this is to create a new dictionary and restore its data from a soup.

The next several sections describe the numbered steps in greater detail. Following this discussion, the section “Restoring Dictionary Data From a Soup” describes how to restore an existing dictionary from soup data.

Creating the Blank Dictionary

You can create a blank RAM-based dictionary anywhere in your application that makes sense; a common approach is to take care of this in the `viewSetupFormScript` method of the application’s base view. You must also create a slot in which to store the RAM-based dictionary. The following code fragment creates a dictionary in the `mySpecialDictionary` slot.

```
viewSetupFormScript:func()
begin
   local element;
   mySpecialDictionary := NewDictionary('custom);
end,
```

This code example uses the `NewDictionary` function to create a blank dictionary in the `mySpecialDictionary` slot. The `NewDictionary` function accepts the symbol `'custom` as its argument, which specifies that the new dictionary is for this application's use only.

Note

Although the token returned by the `NewDictionary` function currently evaluates to an integer in the Inspector, the type of value returned by this function may change on future Newton devices. Do not rely on the `NewDictionary` function returning an integer. ◆

Populating the Blank Dictionary

Once you have created a blank dictionary, you need to populate it with your dictionary items. You can use the `AddWordToDictionary` function to add a specified string to a specified RAM-based dictionary.

The first argument to this function is the identifier of the dictionary to which the string is to be added; this identifier is returned by the `NewDictionary` function. The previous code example stored this identifier in the `mySpecialDictionary` slot.

The second argument to this function is the string to be added to the dictionary. If this argument is not a string, the `AddWordToDictionary` function throws an exception.

If the word is added successfully, this function returns `True`. If the specified word cannot be added because it is already present in the specified dictionary, this function returns `NIL`.

To populate the dictionary, you need to call the `AddWordToDictionary` function once for each item to be added. There are many ways to iteratively

call a function; the best approach for your needs is an application-specific detail that cannot be anticipated here. The following code example shows one way to populate a blank dictionary.

```
myAdder:func()
begin
local element;
// items slot contains an array of dictionary strings
foreach element in items do
   AddWordToDictionary(mySpecialDictionary, element);
end
```

This approach works well for small dictionaries; for most large dictionaries, however, it is far more efficient to populate the dictionary from saved soup data. You should store custom dictionary data in a soup so that it is safely stored and persistent across warm boots.

Saving Dictionary Data to a Soup

Once you have added all of your dictionary entries, your RAM-based custom dictionary is ready for use. However, it would be rather inefficient to have to build it from scratch each time you need it, especially if the dictionary is large. Instead, you can store a binary representation of the dictionary data in a soup and use this soup data to restore the custom dictionary.

The `NewDictionary` function returns an identifier used to reference the dictionary; in the previous example, this identifier was stored in the slot `mySpecialDictionary`, which is defined in the base view of the application. You can pass a path expression to this slot as the argument to the `GetDictionaryData` function, which returns a frame containing a binary representation of the dictionary data (the words or items). You can then store this frame in a soup. The following code fragment assumes that the soup `kSoupName` is a valid soup created according to the Newton DTS soup-creation guidelines.

```
// make sure the soup is open & available
mySoup := GetUnionSoup (kSoupName);
```

```
// create binary representation of dictionary data
local dict := GetRoot().appSym.mySpecialDictionary;
local theObj:=  GetDictionaryData(dict);
// store the dictionary data
dictData := {data:theObj};
mySoup:Add(dictData);
```

Restoring Dictionary Data From a Soup

To use the dictionary, your application needs to retrieve the dictionary data object from the soup and use the global function `SetDictionaryData` to install the data in an empty dictionary. This is typically done in the application package's `installScript` method or in the `viewSetupFormScript` method of the view that uses the custom dictionary, as in the following code example.

```
// make new blank dictionary
mySpecialDictionary := NewDictionary('custom);
// get the dictionary data from the soup
// structure of query depends on how you store data
dataCursor:=dictDataSoup:Query(querySpec);
// how you get entry depends on how you store data
myBinaryData := dataCursor:entry();
// put data in dictionary
SetDictionaryData(mySpecialDictionary, myBinaryData);
```

Using Your RAM-based Custom Dictionary

This section describes how to make a RAM-based dictionary available to your view. The dictionary is made available to the view by means of a `dictionaries` slot that the view can access.

You need to take the following steps to make your RAM-based dictionary available:

1. Set the `vCustomDictionaries` flag for the view that is to use the dictionary. This flag indicates that the view has a slot named `dictionaries` which contains the identifiers for all dictionaries to be used in recognition for this view.
2. Create a `dictionaries` slot that is accessible to the view and store your dictionary's identifier in it.

In the view that is to use the custom dictionary, you need to create a slot called `dictionaries` which stores an array of dictionary identifiers. You need to install the newly created dictionary in this slot using code similar to the following example.

```
// vCustomDictionaries flag already set in NTK
// search custom dictionary only
dictionaries := [mySpecialDictionary];
```

If no flags other than `vCustomDictionaries` are set in the entryFlags editor in NTK the default system dictionary chain is empty and the only dictionary used for recognition in this view is the custom dictionary.

If you want to search the custom dictionary in addition to the system-supplied dictionaries, just check the appropriate view flags in the Entry Flags editor in NTK and add only your custom dictionary to the `dictionaries` slot. (Don't forget to check the `vCustomDictionaries` flag as well.)

If you want to enable dictionaries procedurally, rather than in the NTK `entryFlags` editor, you can add the appropriate flags to the view's `viewFlags` slot.

Each view flag enables a particular combination of system dictionaries; if you want to explicitly control which system dictionaries are used, check only the `vCustomDictionaries` flag in NTK and use the constants described previously in the "System-Supplied Dictionaries" section to add specific dictionaries to the `dictionaries` slot. For example,

```
dictionaries :=[mySpecialDictionary, kUserDictionary,
                kCommonDictionary]
```

The order in which dictionaries appear in the `dictionaries` array is not critical; the recognition system searches all of the dictionaries at once.

Removing the RAM-based Dictionary

It is recommended that you remove your custom dictionary when it is no longer needed, such as when your application is removed. The `DisposeDictionary` function removes a specified RAM-based dictionary and always returns NIL.

The `DisposeDictionary` function accepts one argument, the dictionary identifier returned by `NewDictionary`. In an earlier example, this identifier was stored in the `mySpecialDictionary` slot; hence, a line of code similar to the following example would be used to remove the custom dictionary.

```
DisposeDictionary(mySpecialDictionary);
```

Manipulating the User Dictionary

The user dictionary is stored in the root view.To obtain a reference to the user dictionary, you can use code similar to the following example.

```
local reviewDict := GetRoot().ReviewDict;
```

Note

Future versions of the system are not guaranteed to have this slot. Make sure you verify that the returned value is non-`nil` before using it. ◆

The system supplies methods that you can use to load and save this dictionary as well to add and remove words. For complete descriptions of these functions, see the section “User Dictionary Methods” beginning on page 10-101.

Disabling AutoAdd

Either of the following techniques prevent words written in a particular view from being automatically added to the user dictionary and the autoAdd dictionary.

- Set a `_noautoadd` slot in the view to a non-`nil` value.
- Set a `_noautoadd` slot in the word's `wordInfo` frame to a non-`nil` value. You can get a word's `wordInfo` frame by calling the `GetCorrectionWordInfo` function from within the view's `viewWordScript` method.

Recognition System Reference

This section describes in detail the flags that control recognition in views, the system supplied prototypes related to recognition, the system-supplied dictionaries and related methods and global functions.

System-wide Settings

You can use the following slots in the `userConfiguration` frame to

- specify the use of a particular recognizer.
- enable and disable the system's ability to modify its handwriting model.
- enable and disable the automatic addition of words to the user dictionary and the autoAdd dictionary.

Note that the values of these slots are set by the user in various preferences slips; generally, you should not change them. Do not access slots in the `userConfiguration` frame directly; instead, use the functions `GetUserConfig` (page 18-101) and `SetUserConfig` (page 18-102).

`letterSetSelection`

Sets the recognizer currently in use. This value may be any of the following constants. Although the

recognizers built into Newton platforms through version 2.0 of system software support these values, not all recognizers are guaranteed to support them.

`kStandardCharSetInfo`
: The dictionary-based cursive recognizer. This value is used when the user selects the Cursive radio button in the Handwriting Recognition preferences slip.

`kUCBlockCharSetInfo`
: The character-based printed recognizer. This value is used when the user selects the Printed radio button in the Handwriting Recognition preferences slip.

`learningEnabledOption`
: The value `NIL` specfies that writing in this view does not modify the system-defined handwriting model. This feature would be appropriate for a handwriting practice area or for protecting the machine from guest users.

`doAutoAdd` Specifies whether words are automatically added to the user dictionary and the autoAdd dictionary.

View Flags for Recognition

This section describes the flags that specify which kinds of pen input the view recognizes and handles. These flags are specified in the view's `viewFlags` slot, as described in the next section, "Combining View Flags."

Note that every option may not be available in every kind of view. For example, a view of the `clView` class can accept clicks (taps) but can't recognize words.

Note

Newton ToolKit displays these flags in an area of the screen called Entry Flags because these flags control the entry of recognized data. However, in the code that NTK generates (and when you set these flags procedurally) they are placed in the view's `viewFlags` slot. Hence, this chapter refers to these flags as "view flags." ◆

Non-word options

`vNothingAllowed` The view accepts no handwritten or keyboard input.

`vAnythingAllowed`

This flag turns on all recognizers, theoretically allowing the view to accept any kind of input; however, the recognition actually performed for the view at run time is controlled by the user preferences settings and the settings of the recognition buttons at the bottom of the screen (the user can tap these buttons to enable recognition of text and graphics). You must be certain that the recognition buttons are visible when you use this flag, because it allows the creation of a state in which nothing is recognized: if recognition is turned off and the recognition buttons are not displayed, the user cannot enable recognition in the view.

Note

The `vAnythingAllowed` flag is a mask that sets and clears other flags. To avoid unexpected results when specifying flags programmatically, you need to use the `Bor` function to logically OR this flag with any others you specify for the view that uses `vAnythingAllowed`. (See the description of the `Bor` function in Chapter 17, "Utility Functions.") If you use the entryFlags editor in NTK to specify view flags for the view, you need not do anything special to use this flag—the editor makes appropriate settings with respect to other flags for you. ◆

Note that you'll obtain faster and more accurate recognition using the correct set of individual flags for the types of data that your view accepts. If you want to control specific recognizers, you need to use a combination of the other `viewFlags` settings.

`vClickable` The system sends the `viewClickScript` message to the view once for each pen tap (click) that occurs within the view. The unit is passed as the argument to the `viewClickScript` method.

IMPORTANT

You must set the `vClickable` flag for any view that is to accept pen input; no taps or strokes are passed to the view if this flag is not set. ▲

Views that explicitly handle clicks (such as buttons) or track the pen themselves can set this flag and use the `viewClickScript` method to implement their handling of pen input. The `viewClickScript` method can track the position of the pen by calling the `GetPoint` function; for more information, see the description of the `GetPoint` function in the section "Recognition Functions,"later in this chapter.

Electronic ink is turned on or off depending on the `vClickable` flag's interaction with the `viewClickScript` method and the settings of view

flags for views in the parent chain. If `vClickable` is the only view flag set for the view, inking is automatically turned off; that is, no electronic ink is drawn in the view at pen-down time. However, if vClickable is not set for the view, any of its parent views may handle clicks or draw ink.

Ink is turned on in views having a `viewClickScript` method. To turn off inking, the `viewClickScript` can call `InkOff`. Note also that the `TrackHilite` and `TrackButton` methods also turn off ink.

`vStrokesAllowed`

The view accepts handwritten strokes, and is sent the `viewStrokeScript` message at the end of each stroke. The only time you need to set this flag is when the view has a `viewStrokeScript` method. This method would be used to do something application-specific with strokes, such as recognizing your own gestures. Don't set this flag if your view does not have a `viewStrokeScript` method—you'll only waste battery power!

You must also set the `vClickable` flag when using this flag; otherwise, the view accepts no input.

For more information on the `viewStrokeScript` method, see the *Different Strokes* code sample from Newton DTS.

`vGesturesAllowed`

The view accepts gesture strokes such as scrub, hilite, tap, double tap, caret, and line. It is sent the `viewGestureScript` message when one of these

gestures is written. Most views that accept input also set this flag so that gestures such as scrub can be used.

You must also set the `vClickable` flag when using the `vGesturesAllowed` flag; otherwise, the view accepts no input.

`vShapesAllowed` Enables shape recognition within the view. You must also set the `vClickable` flag when using this flag; otherwise, the view accepts no input.

`vSingleUnit` Disables recognition based on spatial cues (distance between gestures or strokes), forcing the recognition system to rely on temporal cues (time between the end of one stroke and the beginning of the next one) to determine when the user has completed a group of strokes. Using this flag may result in better recognition of complex stroke groups in which users tend to put large spaces, such as phone numbers.

Word options

`vCharsAllowed` Enables word recognition using the default dictionary chain defined for the view. The default dictionary chain for a particular view is defined according to the view class or system prototype from which the view is derived, as well as the current locale. Setting this flag includes in the chain dictionaries for common words, proper names, the user dictionary, and any user-installed default dictionaries. For more information, see the section “How Locale Affects Recognition” in Chapter 15, “Localizing Newton Applications.”

`vLettersAllowed` Enables letter-by-letter character recognition without using dictionaries; thus enabling this flag allows the view to recognize combinations of letters that are not dictionary items. Take care to use this flag only when necessary, as it can slow down recognition and make it

less reliable. For example, non-word combinations of characters such as “xyz” can be recognized when the `vLettersAllowed` flag is set.

The `vLettersAllowed` flag can be used in conjunction with any combination of dictionaries; however, it does not allow numeric input and therefore cannot be used with the `vPhoneField`, `vNumbersAllowed`, `vDateField`, `vTimeField`, or `vAddressField` flags.

`vNameField` Enables word recognition using the Names dictionary as the primary dictionary.

`vCustomDictionaries`
Enables word recognition using custom dictionaries. This is used for views that accept custom sets of data (e.g., company names or plant species, etc.). You must include in the template a dictionary slot which contains a reference to a custom dictionary (or an array of references to multiple dictionaries). Or you can specify the id of one or more built-in dictionaries to limit recognition to that set of words (e.g., state names).

`vPunctuationAllowed`
Enables recognition of punctuation marks in addition to any other recognizers that are enabled. This flag enables recognition of the following marks preceding a word: single quotation mark, double quotation mark, left parenthesis, and hyphen. This flag also allows the recognition of the following marks at the end of a word: single quotation mark, double quotation mark, right parenthesis, hyphen, period, comma, exclamation point, question mark, colon, and semicolon.

`vCapsRequired` Forces the first character of each word to be capitalized when it is returned from the recognition system.

Lexical word options

These options can be used simultaneously with all other options except for `vLettersAllowed`.

`vNumbersAllowed` Enables recognition of numbers. Numeric recognition includes monetary amounts (for example, $12.25), decimal points, and signs (+ and -).

`vPhoneField` Enables recognition of phone numbers.

`vDateField` Enables recognition of dates.

`vTimeField` Enables recognition of times.

Recognition System Prototypes

This section describes prototypes used for configuring the recognition system or providing a user interface to it.

protoRecConfig

This prototypes is used to configure the recognition system when a particular configuration is not available through the use of view flags. It is also used to support features such as ink text and specialized behavior such as restricting recognition to a particular character set.

These slots affect the input mask that the view constructs.

Input Mask Slots

`inputMask` Required. The bit field that controls the view's recognition behavior. The recognition portion of the view's `viewFlags` slot should be set to the same value as the `inputMask` slot in the `recConfig` frame. There is one exception to this rule: to make it easy to enable ink words, you can put the system-supplied `recConfig` frame `rcInkOrText` in your view's

recConfig slot, leaving everything else the same. Refer to the section on using ink words for more details.

`buildInputMask` Optional. Set the value of this slot to `True` when you want the view to recognize ink according to the user preferences and the settings of the global recognition buttons rather than from the settings specified by the `inputMask` slot. A value of `True` forces the system to build a new input mask based on `userConfiguration` settings and the on-screen recognizer controls set by the user. For related information, also see the description of the `recConfig` frame's `baseInputMask` slot.

`baseInputMask` Required when the value of the `buildInputMask` slot is `True`. This slot stores a mask specifying the appropriate recognition bits to be set in addition to those calculated from user preferences. This mask is intended to be used when the value of the `buildInputMask` slot is `True`. If the `baseInputMask` slot is not defined or its value is `nil`, the system supplies a default value of `vClickable + vGesturesAllowed` for the value of this slot. For a typical deferred recognition view, the value of the `baseInputMask` is usually `0`. Flags not automatically set by the `buildInputMask` slot include `vSingleUnit`, `vClickable`, `vStrokesAllowed`, and `vGesturesAllowed`; you may wish to set some or all of these flags in the `baseInputMask` slot.

These slots enable the view's use of a particular recognizer.

Recognizer Configuration Slots

`doTextRecognition`

The value `True` enables word recognition. The system

sets the value of this slot to `True` when the user turns on text recognition from the `protoRecToggle` view.

`doShapeRecognition`.

The value `True` enables shape recognition. The system sets the value of this slot to `True` when the user turns on shape recognition from the `protoRecToggle` view.

`doInkWordRecognition`

The value `True` causes the recognizer to change ink to ink words if text and shape recognition is disabled. The system sets the value of this slot to `True` when the user turns on ink word recognition from the `protoRecToggle` view. If this value is not `True` the recognizer leaves strokes as ink.

These slots specify the choices that the `protoRecToggle` view provides to the user.

recToggle Configuration Slots

`allowTextRecognition`

A value of `True` in this slot specifies that word recognition is enabled when the value of the `doTextRecognition` slot is `True.` (The `doTextRecognition` slot is `True` when the user chooses the text recognition setting from the on-screen recognizer buttons.) The view respects all remaining user preferences not related to recognizer settings.

`allowShapeRecognition`

A value of `True` in the `allowShapeRecognition` slot specifies that only shape recognition is enabled when the value of the `doShapeRecognition` slot is `True.` (The `doShapeRecognition` slot is `True` when the user chooses the shape recognition setting from the on-screen recognizer buttons.) The view respects all

remaining user preferences not related to recognizer settings.

`allowInkWordRecognition`

A value of `True` in the `allowInkWordRecognition` slot specifies that only text recognition as ink word text is enabled when the value of the `doInkWordRecognition` slot is `True`. (The `doInkWordRecognition` slot is `True` when the user chooses the ink text recognition setting from the on-screen recognizer buttons.) The view respects all remaining user preferences not related to recognizer settings.

Note

As of this publication date, version 2.0a7c5 of Newton system software ignores the value of the `allowInkWordRecognition` slot. ◆

These slots modify the behavior of the cursive recognizer.

Cursive Recognizer Configuration Slots

`speedCursiveOption`

This value affects the amount of time the cursive recognizer spends recognizing input. The user's preference (set by a slider in the Handwriting Settings preference) is used as the default value of this slot. This value ranges from 0 through 9, with 0 representing the slowest and most accurate recognition and 9 representing the fastest and least accurate recognition. This value does not affect the character recognizer.

Note

These slots are not guaranteed to affect all future cursive recognizers. ◆

`timeoutCursiveOption`

This value affects the amount of time the recognizer

waits from the completion of a stroke for subsequent strokes that might belong to the same character or word. The value of this slot is a delay expressed in ticks (60ths of a second). The slider in user preferences sets values for this slot ranging from 15 ticks (.25 second) to 60 ticks (1 second). Your view can use larger or smaller values, although it is not recommended.

`letterSpaceCursiveOption`

The value of this slot affects the amount of space required to consider sets of strokes as belonging to separate letters or words. The user's preference (set by a slider in the Handwriting Recognition preference), is used as the default value of this slot. This value ranges from 0 through 9, with 0 representing widely spaced words or characters and 9 representing closely spaced words or characters. If the value of this slot is `nil` the recognizer performs no segmentation.

These slots specify the kind of ink to be returned to the view.

Ink Recognizer Configuration Slots

`doInkWordRecognition`

When the value of this slot is `True` the view turns unrecognized ink into `inkText`. If the value of this slot is `nil` or the slot is absent, the view turns unrecognized ink into sketching ink.

`doRawInkRecognition`

When the value of this slot is `True`, the view turns unrecognized ink into sketching ink.

These slots modify the behavior of the shape recognizer.

Shape Recognizer Configuration Slots

`symmetryShapeOption`

A value that is not `NIL` causes the shape recognizer to find symmetry within the shapes that are drawn,

including the recognition of edges that are similar in length and angle. This slot derives the default value of `True` from the global `userConfiguration` frame.

`curveShapeOption`

A value that is not `NIL` causes the shape recognizer to allow curved segments within the shapes that it recognizes. This slot derives the default value of `True` from the global `userConfiguration` frame.

`gravityShapeOption`

A value that is not `NIL` causes the shape recognizer to allow new shapes to stick to old shapes. This slot derives the default value of `True` from the global `userConfiguration` frame.

These slots affect the view's use of dictionaries for recognition.

Dictionary configuration slots

`dictionaries` Specifies custom dictionaries to be used by the view. This slot may contain a single dictionary id or an array of dictionary id's. When this slot is present, the view's `dictionaries` slot is ignored. Although not always necessary, it is still a good idea to set the `vCustomDictionaries` bit in the `recConfig` frame's `inputMask` slot when the `recConfig` frame provides a `dictionaries` slot.

`rcSingleLetters`

Set the value of this slot to `True` for a view that is to recognize only single letters. For example, this feature would be useful in a corrector view, in a crossword puzzle or when overwriting letters in a previously-recognized word. Note that you still need to

provide a dictionary; in this case, one having entries that are single letters.

`inhibitSymbolsDictionary`

Set the value of this slot to `True` when the symbols dictionary is not to be added to the dictionary list used for recognition in the view. The symbols dictionary contains single letters, punctuation marks and miscellaneous characters and is normally enabled. It is used by the recognition system when the user overwrites single characters in a misrecognized word.

System-Supplied RecConfig Frames

The `_proto` slot of your view's `recConfig` frame must reference one of the system-supplied `recConfig` frames defined by the constants described here. The `recConfig` frames supplied by the constants `ROM_rcInkOrText`, `ROM_rcPrefsConfig` and `ROM_rcRerecognizeConfig` require no modification to produce useful behavior. You must provide appropriate initial values for some slots in the `recConfig` frames supplied by the constants `ROM_rcDefaultConfig`, `ROM_rcSingleCharacterConfig` and `ROM_rcTryLettersConfig`.

To use any constant described in this section, place it in the `_proto` slot of your `recConfig` frame and call the `PurgeAreaCache` function to build a recognition area that uses your new `recConfig` frame.

`ROM_rcInkOrText`

This general-purpose `recConfig` frame can be used as it is for views that accept text input. It allows the user to turn on text recognition only; when text recognition is disabled, the system returns `inkWord` text to the view. This `recConfig` frame is generally used with a `protoRecToggle` view to allow the user to specify

whether the view displays ink text or normal text. The `rcInkOrText` frame provides the following slots.

`allowTextRecognition`
: Default value of `True` allows user to turn on the text recognizer from on-screen buttons. See the description of the `allowTextRecognition` slot on page 10-56 for more information.

`doInkWordRecognition`
: Default value of `True` enables recognition of input as `inkWord` text when text recognizer is off.

`ROM_rcPrefsConfig`
: This frame can be used as is to configure views for performing recognition according to user preference settings. Views having recognition behavior based on this frame permit the user to enable or disable any recognizer for which the system provides a user interface. The `rcPrefsConfig` frame provides the following slots.

`allowTextRecognition`
: Default value of `True` allows the user to turn on the text recognizer from on-screen buttons. See the description of the `allowTextRecognition` slot on page 10-56 for more information.

`allowShapeRecognition`
: Default value of `True` allows the user to turn on the shape recognizer from on-screen buttons.

`allowInkWordRecognition`
: Default value of `True` allows the user to

turn on the `inkWord` recognizer from on-screen buttons.

`allowRawInkRecognition`
Default value of `True` allows the user to turn on sketching ink from on-screen buttons.

`ROM_rcDefaultConfig`
This frame can be used as a generic prototype for any `recConfig` frame; it provides a set of useful slots for which the application developer must supply values. The `rcDefaultConfig` frame provides the following slots; note that the prototype supplies the value `nil` for all of these slots; the application must initialize them with useful values.

`punctuationCursiveOption`
A value of `True` specifies that the view recognizes punctuation marks. The prototype supplies a default value of `nil`.

`dictionaries`
The list of dictionaries to use for recognition. This slot holds an array of dictionary identifiers, a single dictionary identifier, or the value `nil`. The prototype supplies a default value of `nil`. For more information, see the description of the `dictionaries` slot in the section "Using Your RAM-based Custom Dictionary" beginning on page 10-44.

`rcSingleLetters`
A value of `True` specifies that the view recognizes single letters only, rather than dictionary words. The prototype supplies a default value of `nil`.

`rcBaseInfo`
Holds an `rcBaseInfo` frame, which describes the coordinates of an editable

field suitable for single-character input. The prototype supplies a default value of `nil`. For more information, see the section "The rcBaseInfo Frame" beginning on page 10-33.

`inputMask` A bit field specifying the configuration of the recognition system for this view. The prototype supplies a default value of `nil`. For more information, see the section "View Flags" beginning on page 10-7.

`ROM_rcSingleCharacterConfig`

This frame can be used as it is to configure recognition in views accepting single-character input. For example, use this frame to configure the entry fields in a crossword puzzle or the entry fields in a single-character corrector view similar to the `protoCharEdit` system prototype. The `rcSingleCharacterConfig` frame provides the following slots.

`_proto` This frame is based on the system-supplied `rcdefaultconfig` frame. Do not change the value of this slot.

`letterSpaceCursiveOption`
A Boolean value indicating whether the recognition system segments strokes into groups by interpreting spatial and temporal cues.The default value of `nil` specifies that the system performs no segmentation, which is appropriate for a field in which all strokes are to be interpreted as a single character.

`rcSingleLetters`
The default value of 1 indicates that the

system is to recognize single letters rather than dictionary words.

`inputMask` This view's input mask. The default value of `vCustomDictionaries` indicates that the view uses the dictionaries specified in the view's `dictionaries` slot. For more information, see the description of the `dictionaries` slot in the section "Using Your RAM-based Custom Dictionary" beginning on page 10-44.

`dictionaries`
The default value of `kSymbolsDictionary` specifies that this view uses the system-supplied symbols dictionary for recognition. The symbols dictionary is used to recognize single alphanumeric characters, punctuation marks, mathematical symbols, diacritical marks and so on.

`inhibitSymbolsDictionary`
The default value of `1` specifies that the system is not to use the symbols dictionary for recognizing characters written over previously-recognized ones. (The symbols dictionary was used when the characters were first recognized—presumably incorrectly if the user is overwriting.)

`ROM_rcTryLettersConfig`
This frame can be used as it is to configure a view for recognition of single letters and numbers; it is intended for views that implement their own form of deferred recognition. The main difference between this frame and the `ROM_rcSingleLetters` frame is that views using this `recConfig` frame include the Try Letters

button in the picker that is displayed when the user double taps a previously-recognized word.

`_proto` The default value of this slot is `ROM_rcDefaultConfig`. For more information regarding the slots that this frame acquires through prototype inheritance, see the description of the `ROM_rcDefaultConfig` constant beginning on page 10-62.

`letterSpaceCursiveOption`
A Boolean value indicating whether the recognition system segments strokes into groups by interpreting spatial and temporal cues.The default value of `nil` specifies that the system performs no segmentation, which is appropriate for a field in which all strokes are to be interpreted as a single character.

`inputMask` The default value of `vLettersAllowed+vNumbersAllowed` configures this view to recognize single letters and numbers. For more information, see the description of the `vLettersAllowed` flag beginning on page 10-24; also see the description of the `vNumbersAllowed` flag beginning on page 10-54. For information regarind the use of the NewtonScript plus (+) operator to combine view flags, see the section "Combining View Flags" beginning on page 10-27.

`ROM_rcRerecognizeConfig`

This frame may be used as it is by views that implement their own form of deferred recognition. It builds an input mask from user preference settings and allows the

user to turn on text recognition from the on-screen buttons.

`allowTextRecognition`
: Default value of `True` allows the user to turn on text recognition from on-screen buttons. See the description of the `allowTextRecognition` slot on page 10-56 for more information.

`doTextRecognition`
: The default value `True` enables word recognition. The system sets the value of this slot to `True` when the user turns on text recognition from user preferences.

`speedCursiveOption`
: The amount of time the cursive recognizer spends recognizing input. The prototype provides a default value of 2. For more information see the description of this slot that begins on page 10-57.

`letterSpaceCursiveOption`
: A Boolean value indicating whether the recognition system segments strokes into groups by interpreting spatial and temporal cues.The default value of `nil` specifies that the system performs no segmentation, which is appropriate for a field in which all strokes are to be interpreted as a single character.

`baseInputMask`
: This slot stores a mask specifying the appropriate recognition bits to be set in addition to those calculated from user preferences. The default value of `0` sets no bits in the `baseInputMask` bit field.

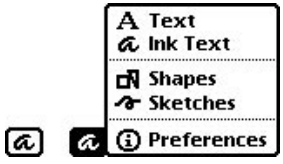
`buildInputMask`
: The default value of `True` forces the system to build a new input mask based

on `userConfiguration` settings and the on-screen recognizer controls set by the user. For more information, see the description of this slot beginning on page 10-55.

protoRecToggle

The `protoRecToggle` prototype provides a popup menu that controls recognition in views that set the `vAnythingAllowed` flag. When this view is opened, its appearance reflects the current configuration of the recognition system in the view that the `protoRecToggle` view controls. Figure 10-8 shows the `protoRecToggle` view with the menu open and collapsed.

Figure 10-8 protoRecToggle view closed and expanded



protoCharEdit

The `protoCharEdit` system prototype provides a comb-style entry view in which the user can edit text. The recognition system uses this proto as a means of allowing the user to easily correct single letters in a misrecognized word. The `protoCharEdit` system prototype is shown in Figure 10-9.

Figure 10-9 The protoCharEdit system prototype

In a `protoCharEdit` view, each character position that can be edited has a dotted line beneath it to indicate that it can be changed. The user can edit a character by overwriting a new character into the cell, which displays the recognized value of the new character. The user can delete one or more characters with the scrub gesture. The user can insert a space for a new character with the caret gesture.

When the user taps on a cell, the system displays a list of alternate guesses for that character; the list also includes a Delete item and an Add item. The user can perform the following operations by tapping on items in the list.

- Tapping on a letter in the pick list replaces the character in the cell with the one selected from the list.
- Tapping the Delete item in the pick list removes the character from the cell and shifts the remaining characters to close up the space in the word that this operation creates. The Delete item can also be used on empty cells to close spaces between characters.
- Tapping the Add item in the pick list adds a space in the position indicated by the cell that was tapped to display the picker. The remaining characters are shifted to accommodate insertion of the new space.

Tapping on blank comb fields may have meaning as well; for example, tapping on the space before or after a word displays a list of punctuation marks that may be inserted at that position.

The `protoCharEdit` prototype provides the following slots of interest to developers:

Slot name	
`maxChars`	The number of cells to be displayed in the view. The default value is `8`. If you specify the values of the `top` and `left` slots, then you'll also need to specify the value of the `maxChars` slot. If instead you specify the value of the `viewBounds` slot, the value of the `maxChars` slot is calculated for you.
`cellWidth`	The width of each cell. This value must be an even number; the default value is `24`. If you specify the values of the `top` and `left` slots, then the total width of the field is calculated as the value `(cellWidth * maxChars)` and is set for you.
`cellGap`	The number of pixels comprising the blank space between cells. This value must be an even number; the default value is `6`.
`viewLineSpacing`	The distance in pixels from the top of the view to the dotted line on which the user enters written input. The default value is `30`.
`cellHeight`	The total height of the cell, expressed in pixels. The default value is `50`. If you specify the values of the `top` and `left` slots, then the height of the view as expressed by its `viewBounds` value is set to the value of the `cellHeight` slot. If you specify the value of the `viewBounds` slot explicitly, the value of the `cellHeight` slot is set to the height expressed by the value of the `viewBounds` slot.

You can use a template containing the following slots to limit `protoCharEdit` view input to a specified set of values.

Template slots

`format`	Optional. A string that specifies the editing characteristics of each position in the comb field. Each character in the format string corresponds to a position in the comb field: a digit specifies the element of the filters array to use as the filter for that position in the comb field; an underscore specifies that the corresponding position in the comb field cannot be edited. The presence of the `format` slot specifies that the number of characters in the field is fixed: scrubbing clears a field in the comb, rather than deleting it, and comb fields cannot be inserted or deleted. If the `format` slot is missing or its value is `nil`, the comb field acts as a free-form entry field, like the corrector in the built-in Notepad application. If the template has a `format` slot, then it must also provide `filters` and `string` slots.
`filters`	Required if a `format` slot is provided. An array of filters, each of which specifies characters that may be entered in fields of the `protoCharEdit` view. The value of the `format` slot associates entry fields in the `protoCharEdit` view with filters in this array.
`string`	Required if a `format` slot is provided.The string initially displayed in the comb field when the `protoCharEdit` view opens.

System-supplied protoCharEdit Templates

The system provides predefined templates for dates, times, phone numbers and integers. The templates for dates, times and phone numbers may change when you switch locales, so you should be sure to get the appropriate template from the current locale.

Phone Number Template

The template for phone numbers is stored in `GetLocale().phoneFilter`. This template lets you enter phone numbers in an acceptable format for the current locale, excluding area code. It is expected that the area code will be entered in a separate field.

Date Template

The template for dates is stored in `GetLocale().dateFilter`. This template lets you enter a date in "mm/dd/yy" format (in which the ordering may change depending on the locale).

TimeTemplate

The template for times is stored in `GetLocale().timeFilter`. This template lets you enter a time in "HH:MM xm" format. For locales that use 24 hour time, the format will be simply "HH:MM".

Number Template

A general-purpose number template is defined in `ROM_numberFilter`. This lets you enter a variable length integer containing only digits.

protoCharEdit Functions and Methods

The system provides these `protoCharEdit` functions and methods. You can also define the `SetupString` and `CleanupString` methods, described in the next section.

CurrentWord

```
CurrentWord()
```

Returns the current word displayed in the comb view. This function strips leading and trailing spaces, so it doesn't return the precise string displayed in unformatted comb views. Always use this function (or `GetWordForDisplay`) to retrieve the text from the comb view.

GetWordForDisplay

```
GetWordForDisplay()
```

Returns the current word in the comb view, but cleaned up for display. Some templates provide functions for cleaning up the string before displaying it. For example, a date or time template might strip out internal spaces or leading zeros. This is the best function to call if you want to display a nice readable version of the string.

DeleteText

```
DeleteText(left, right)
```

Delete text causes text to be deleted (or cleared) from the comb view. Left is the index of the leftmost character to be deleted, and right is the index of the rightmost character to be deleted, plus one. The index of the leftmost character is stored in the slot wordLeft and the rightmost character plus one is stored in wordRight.

Normally deletion is accomplished either by scrubbing or using the popup for a character. This function is primarily documented as a way to clear the entire view. This would be done by the following call:

```
view:DeleteText(view.wordLeft, view.wordRight);
```

Scroll

```
Scroll(direction)
```

The Scroll function can be used to scroll the comb view to the left or to the right. If direction is greater than zero, then characters to the right of the currently displayed characters are shown, if necessary. If direction is less than zero, then the display scrolls back to the beginning of the word (not to the next chunk to the left), if necessary. If scrolling takes place, then true is returned.

UseTextAndTemplate

`UseTextAndTemplate()`

UseTextAndTemplate is the function for changing your comb view text and/or template. Before calling it, you must first set the text and template slots to their desired values. It does all initialization of internal variables. If you are changing the text, but not the template, you can use SetNewWord/UseNewWord for greater speed.

It is not necessary to call UseTextAndTemplate when first initializing the comb view. The viewSetupFormScript does the equivalent work.

SetNewWord

`SetNewWord(`*str*`, `*dontUse*`)`

SetNewWord can be used to set the text that is displayed in the comb view after the viewSetupFormScript has already been called.

For a formatted comb view, str must be of the appropriate length and format; no reformatting will be performed. Thus `SetNewWord("", nil)` should not be used to clear a formatted comb view (see `DeleteText`, above).

`SetNewWord` should be followed by a call to `UseNewWord`.

If you are changing both the text and the template, you should use the `UseTextAndTemplate` function.

str	The new text to be displayed. This string must not contain leading or trailing spaces.
dontUse	For system use only; always set this value to nil.

UseNewWord

`UseNewWord()`

UseNewWord should be called after SetNewWord. UseNewWord ensures that all internal parameters are correctly initialized and used.

FixedWord

`FixedWord()`

FixedWord returns true if the comb view has a template slot, and that template has a non-nil format slot. When `FixedWord` is `true`, characters are cleared rather than deleted; when it is `false`, leading and trailing spaces are added to the displayed word, as necessary.

FixexdWordLength

`FixedWordLength()`

`FixedWordLength` returns the length, in characters, of the template's format slot. If it is not a formatted word (`FixedWord()` returns nil), then `FixedWordLength()` returns nil.

DisplayExternal

`DisplayExternal(`*doIt*`)`

Whenever the text in the comb view is edited, either manually, or by picking an alternate from a cell's popup, the `DisplayExternal()` function is called. The parameter *doIt* may sometimes be `false`, and you should not need to do any drawing in this case.

SaveUndoState

`SaveUndoState(`*appState*`)`

This function is called internally to save the state of the comb view for undo operations. It creates a frame in which important internal state is stored, saves the passed *appState* in a slot called `parm`, and then adds an undo action that will call `RestoreUndoState` with the created frame as a single parameter.

If you need to provide additional undo behavior, which may be necessary if you are displaying the text externally, then you can override this function, put private information into a frame you define, and then call the inherited `SaveUndoState` with your frame as parameter.

RestoreUndoState

```
RestoreUndoState(savedState)
```

RestoreUndoState is called by an undo operation. SavedState is the frame that was created by the SaveUndoState function. If you added your own information to the undo frame, it will be in savedState.parm.

Application-Defined protoCharEdit View Methods

SetupString

charEditView`:SetupString(`*str*`)`

Optional. A method you provide which sets the initial value of the `string` slot.

str The string to place in the `string` slot.

CleanupString

charEditView`:CleanupString(`*str*`)`

Optional. A method you provide to do postprocessing of the value in the `string` slot. For example, you can use this function to strip extraneous spaces or leading zeros from the string before it is displayed in the comb view.

str The string on which this method operates.

protoCorrectInfo

The main repository for correction information is a frame stored in the `correctInfo` global variable. This frame's `_proto` slot references the `protoCorrectInfo` system prototype.

The `correctInfo` frame contains the following slots of interest to developers.

`info`	An array of frames based on the `protoWordInfo` system prototype. Each frame stores the correction information for a recognized word.
`max`	the maximum number of items for which the system stores `correctInfo` frames. Currently the maximum value for this slot is `10`; it may change in the future.

This proto provides the following methods of interest to developers.

RemoveView

correctInfo`:RemoveView(`*view*`);`

Deletes all *correctInfo* entries having a `viewID` value matching the ID of the specified view. This method is useful when an entire view is being deleted, and you want to delete all *correctInfo* entries corresponding to that view.

view	The view from which this method extracts a `viewID` value.

Find

correctInfo`:Find(`*view*`, `*offset*`);`

Returns the `wordInfo` frame at the specified offset in the specified view.

view	The view from which to extract a `wordInfo` frame.
offset	The offset at which this method is to find a wordInfo frame.

FindNew

correctInfo`:FindNew(`*view*`, `*offset*`, `*length*`);`

Returns the `wordInfo` frame at the specified offset in the specified view. If this method doesn't find a wordInfo frame at that location, it creates a new wordInfo frame for the word, adds it to the correctInfo array, and returns it.

Offset and length describe the position of the specified word within the view. This routine assumes that the view has a 'text slot that contains the text to which offset and length refer.

AddUnit

correctInfo:`AddUnit(`*view*`,` *start*`,` *stop*`,` *unit*`);`

Extracts the `wordInfo` frame from the specified unit and adds it to the *correctInfo*. frame. This method does not move existing elements. If you are doing an insertion or replacement, then you should invoke the Offset method before using the AddUnit method. The `AddUnit` method can be called from a `viewWordScript` method to save a word's info into a `correctInfo` frame.

AddWord

correctInfo:`AddWord(`*wordInfo*`);`

Adds the specified wordInfo frame to the *correctInfo* array. This method doesn't do anything to the wordInfo frame—it just adds the frame to the *correctInfo* array.

wordInfo The `wordInfo` frame to add.

Clear

correctInfo:`Clear(`*view*`,` *offset*`,` *length*`);`

Clears all `wordInfo` frames that overlap the specified range. If *view* is nil, this method removes any elements that overlap the range, regardless of view.

Extract

`correctInfo:Extract(view, start, stop);`

Creates a new correctInfo frame, copies all the entries that overlap the specified range into it and returns the new correctInfo frame. This method is useful for supporting undo or drag and drop. It does not remove the

`correctInfo` frames from the source—you would normally use the Offset method to do so when necessary. The `Extract` method clones the entries that are copied, in case the receiver wants to offset them.

view	The view from which to extract text.
start	Offset to the beginning of the range of text to copy.
start	Offset to the end of the range of text to copy.

Insert

correctInfo`:Insert(`*newCorrectInfo*`,` *newView*`);`

Copies all the `wordInfo` frames from the *correctInfo* frame and inserts them in the *newCorrectInfo* frame.

newCorrectInfo	The correctInfo frame that is to hold the copied entries.
newView	The view associated with the *newCorrectInfo* frame.

protoWordInfo

The `protoWordInfo` frame stores the correction information for a particular word on the Newton screen. This frame contains slots specifying the view that contains the word, the position of the word in its `clParagraphView` view, the list of alternate words supplied by the recognition system and a reference to the recognizer that recognized the word. Optionally, this frame may also contain strokes, ink and learning data.

Each `protoWordInfo` frame contains the following slots of interest to developers.

`ID`	Not documented; for system use only.
`start`	An integer index into the `clParagraph` view text that specifies the position of the word's first character. The index of the first element in the array of characters is zero. You can determine the number of characters in the

	word by subtracting the value of the `stop` slot from the value of the `start` slot.
`stop`	An integer index into the `clParagraph` view text that specifies the position of the space after the word's first character. The index of the first element in the array of characters is zero.You can determine the number of characters in the word by subtracting the value of the `stop` slot from the value of the `start` slot.
`flags`	Not documented; for system use only.
`unitID`	Not documented; for system use only.
`words`	The list of words returned by the recognition system as an array of `protoWordInterp` frames. The `protoWordInterp` system prototype is described later in this chapter, in the section "protoWordInterp" beginning on page 10-81.
`strokes`	The stroke bundle associated with the word. For more information, see Chapter 9, "Stroke Bundles."
`ink`	The comrpessed ink representing the writen word. `Nil` if no ink is present for the word. This slot is rarely used and this information is provided for debugging use only; commercial applications must not rely on this value.
`unitData`	Not documented; for system use only.

A typical `protoWordInfo` frame looks like the following code example.

```
[{id: 267, // ID of view that owns this data
   Start: 0, // first char's offset into clParagraph view
   Stop: 5, // last char's offset into clParagraph view
   flags: forSystemUseOnly, // do not use this slot
   unitID: forSystemUseOnly, // do not use this slot
// list of words & associated data returned by recognition system
   words: [ {word: "Lunch", score: 130, label: -1, index: 0},
```

```
            {word: "lunch", score: 0, label: -1, index: -2},
            {word: "Lunar", score: 290, label: -1, index: 1},
            {word: "Sundv", score: 300, label: -1, index: 2}],
// the original input's stroke data
   strokes: {class: strokeBundle,
             bounds: {left: 176, top: 289, right: 338, bottom: 336},
             strokes: [<stroke, length 2040>]},
   unitData: forSystemUseOnly}, // do not use this slot
```

Note the negative index values in the second interpretation of the word "lunch." The `-1` value is the default value for this slot and the `-2` value indicates that the word was synthesized by the system; in other words, it's an alternate capitalization or something of the sort. Use these values for debugging purposes only; commercial applications must not rely on them.

This proto provides the following methods of interest to developers.

SetWords

wordInfo`:SetWords(`*words*`)`

Alters the list of words held in a `wordInfo` frame. For each element in the *words* array, this method clones the `protoWordInterp` frame and sets its `word` slot to the value of that array element.

words An array of strings.

GetWords

wordInfo`:GetWords()`

Returns an array of strings, one for each `wordInterp` frame stored in the *wordInfo*`.words` array.

AutoAdd

wordInfo:AutoAdd()

Adds the first item in the *wordInfo* frame's word list to the autoAdd dictionary and the user dictionary. If the *wordInfo* frame has a slot named _noAutoAdd, this method does nothing.

AutoRemove

wordInfo:AutoRemove()

If a word has previously been added by the AutoAdd function, this method removes the first word in the *wordInfo* frame's word list from the user dictionary.

protoWordInterp

The words slot in the protoCorrectInfo frame stores an array of protoWordInterp frames returned by the recognition system. Each protoWordInterp frame contains data associated with a possible interpretation of the original stroke data. For an example of a typical protoWordInterp frame, see the words slot in the protoCorrectInfo code listing on page 10-79.

Each protoWordInterp frame containing the following slots.

word	The text string to which the values of the other slots in this frame apply
score	An integer indicating the accuracy level of the match between this word and the original ink. A low score indicates a good match; conversely, a higher score indicates a poorer match.
label	For system use only. -1 is the default value; -2 indicates a synthesized word. These values are provided for

debugging use only. Commercial applications must not rely on them.

`index` An integer indicating the position of this word in the original list of matches returned by the recognition system. The list of words in the correction picker is ranked ordinally by this value, with the word having the lowest `index` value displayed at the top of the pick list. This value is initialized to `-1` and may have the value `-2` if the word is an alternate capitalization added to the list by the system.

rcBaseInfo

This frame specifies to the recognizer more precisely where characters are written with respect to a well-defined baseline.The rcBaseInfo frame is especially valuable in improving the recognition of single letters, for which it is sometimes difficult to derive baseline or letter size values from user input.

The `rcBaseInfo` frame has following slots.

`base` The y-coordinate of the view's baseline, expressed in screen coordinates (global coordinates)

`smallHeight` Positive offset, expressed in pixels, from `base` to the top of a lowercase "x". Set to `nil` if you aren't sure what value this slot should have.

`bigHeight` Positive offset, expressed in pixels, from `base` to the top of an uppercase "X". Set to `nil` if you aren't sure what value this slot should have.

`descent` Positive offset, expressed in pixels, from `base` to the bottom of a lowercase "g". Set to `nil` if you aren't sure what value this slot should have.

If you aren't sure of appropriate values for the `smallHeight`, `bigHeight`, or `descent` slots, it's better to set them to `nil` than to provide inaccurate values. In general, you shouldn't specify these values unless there is a visible guideline on the screen with which the user can align handwritten input.

Note
If the user can drag the view around on the screen, you'll need to offset the value of the `base` slot when the window is moved. ◆

rcGridInfo

You can use the `rcGridInfo` frame in conjunction with an `rcBaseInfo` frame to provide more accurate recognition within boxes. The `rcGridInfo` frame can be used to define a single box, a horizontal array of boxes, a vertical array of boxes or a two-dimensional array of boxes.

If you provide a grid in which the user is to write characters or words, you need to use an `rcGridInfo` frame to define the grid to the text recognizer. The recognizer requires the information in an `rcGridInfo` frame in order to make character-segmentation decisions.

The `rcGridInfo` frame has the following slots.

`boxLeft`	The global (screen) coordinate of the left edge of the `topleft` box
`boxRight`	The global (screen) coordinate of the right edge of the topleft box
`xSpace`	The distance from one `boxLeft` to the next `boxLeft`
`boxTop`	The global (screen) coordinate of the topmost edge of the `topleft` box
`boxBottom`	The global (screen) coordinate of the bottom edge of the `topleft` box
`ySpace`	The distance from one `boxTop` to the next `boxTop`

The definition of a horizontal array requires the presence of the `boxLeft`, `boxRight` and `xSpace` slots. The definition of a vertical array requires the presence of the `boxTop`, `boxBottom` and `ySpace` slots. The definition of a two-dimensional array requires all six slots be defined.

Note
If the user can drag the view around on the screen, you'll need to offset the values of the `boxLeft`, `boxRight`, `boxTop` and `boxBottom` slots when the window is moved. ◆

System-Supplied Dictionaries

The system supplies various enumerated and lexical dictionaries for the recognition system's use. The set of dictionaries used by a particular view is specified by a combination of the default settings, the locale specified in user preferences and the set of view flags specified for the view.

Table 10-2 describes the system-supplied dictionaries accessible from NewtonScript. These dictionaries are available in every locale.

Table 10-2 System-supplied enumerated dictionaries

Dictionary ID (constant)	Contents
`kUserDictionary`	words added by the user
`kCommonDictionary`	commonly-used words
`kCountriesDictionary`	names of countries
`kDaysMonthsDictionary`	names of days and months
`kFirstNamesDictionary`	first names
`kLastNamesDictionary`	surnames
`kSharedPropersDictionary`	proper names

Note
Although these constants currently evaluate to integers, do not rely on these values. Use only the appropriate constant names to reference these dictionaries. ◆

Locale-specific Dictionaries

In addition to the dictionaries described in Table 10-2, several enumerated dictionaries containing location-specific information are configured according to the locale specified in user preferences.These dictionaries are listed in Table 10-3.

Table 10-3 Locale-specific enumerated dictionaries

Dictionary ID (constant)	Contents
`kLocalPropersDictionary`	proper names
`kLocalCitiesDictionary`	names of local cities
`kLocalCompaniesDictionary`	names of local companies
`kLocalStatesDictionary`	names of states, provinces, etc.
`kLocalStatesAbbrevsDictionary`	abbreviations of states, provinces, etc.

Note
Although these constants currently evaluate to integers, do not rely on these values. Use only the appropriate constant names to reference these dictionaries. ◆

The system also supplies lexical dictionaries that define certain location-specific formats; these dictionaries are listed in Table 10-4.

Table 10-4 Locale-specific lexical dictionaries

Dictionary ID (constant)	Contents
`kLocalDateDictionary`	date formats
`kLocalTimeDictionary`	time formats
`kLocalNumberDictionary`	currency and numeric formats
`kLocalPhoneDictionary`	phone number formats

Note

Although these constants currently evaluate to integers, do not rely on these values. Use only the appropriate constant names to reference these dictionaries. ◆

Recognition Functions

These functions operate on ink, points, strokes, and words.

InkOff

`InkOff(`*unit*`)`

unit	The unit passed to the `viewClickScript` and `viewStrokeScript` methods when the user touches the pen to the screen.

Turns off the display of "electronic ink" for the stroke contained in the specified unit.

This function always returns `True`.

SetInkerPenSize

`SetInkerPenSize(`*size*`)`

size	The width of the pen, in pixels.

Sets the thickness of the electronic ink drawn on the screen. The pen size can range from 1 to 4 pixels wide; the system default is 2.

This function returns NIL if the pen size was set successfully, otherwise an error code is returned.

Note

This function only changes the width of ink as it is drawn by the system. To ensure that ink is properly displayed and updated under all circumstances, your application must set the value of `userConfiguration.userPenSize` appropriately and then pass this value as the argument to the global function `SetInkerPenSize`. ◆

The following code example sets the inker pen size to the value 4.

```
sysSoup:=Getstores()[0]:GetSoup(ROM_SystemSoupName);
configCursor := Query(sysSoup, {type:'index,
         indexPath:'tag, startKey:"userConfiguration"});
userConfigs := configCursor:Entry();
userConfigs.userPenSize := 4
SetInkerPenSize(userConfigs.userPenSize);
```

GetPoint

`GetPoint(`*selector*, *unit*`)`

selector specifies which point coordinate is returned. You must specify one of the following predefined values for *selector*:

`firstX`	The x coordinate of the first point in the stroke
`firstY`	The y coordinate of the first point in the stroke
`finalX`	The x coordinate of the last read point in the stroke
`finalY`	The y coordinate of the last read point in the stroke

unit The unit passed to the `viewClickScript` and `viewStrokeScript` methods when the user touches the pen to the screen.

Returns one of two different kinds of point coordinates that are part of the stroke contained in the specified unit. All points returned are in global (screen) coordinates.

Note that if you specify `finalX` or `finalY`, the coordinate returned could be that of a point in the middle of a stroke in progress, rather than the final point in the stroke. If the user is still drawing the stroke when this function is called, the coordinate of the last point read by the system may not be the last

point in the stroke. You can test if the stroke is actually done by using the function `StrokeDone`.

StrokeDone

`StrokeDone(`*unit*`)`

unit	The unit passed to the `viewClickScript` and `viewStrokeScript` methods when the user touches the pen to the screen.

Returns True if the stroke contained in the specified unit has been completed by the user (they have lifted the pen from the display). Returns NIL if the stroke is not done.

StrokeBounds

`StrokeBounds (`*unit*`)`

unit	The unit passed to the `viewClickScript` and `viewStrokeScript` methods when the user touches the pen to the screen.

Returns a bounds frame describing the boundaries of the unit in its view. A bounds frame has this structure:

`{left: `*n*`, top: `*n*`, right: `*n*`, bottom: `*n*`}`

GetPointsArray

`GetPointsArray(`*unit*`)`

unit	The unit passed to the `viewClickScript` and `viewStrokeScript` methods when the user touches the pen to the screen.

Returns an array of points in the unit.The array consists of coordinate pairs describing the points. The first element contains the Y coordinate of the first point and the second element contains the X coordinate, and so on. (Note that this is reversed from the usual way that coordinate pairs are

written.) Coordinates are global; that is, they are relative to the upper-left corner (0, 0) of the screen.

GetWordArray

`GetWordArray(`*unit*`)`

unit	The unit passed to the `viewClickScript` and `viewStrokeScript` methods when the user touches the pen to the screen.

Returns an array of strings that are the recognition choices for the unit passed as an argument. The first element in the array is the word with the highest probability of matching the stroke that the user wrote. The following words are alternate choices in descending order of matching confidence. Note that the “words” returned aren't necessarily alphabetic words. They can be numbers, phone numbers, times, or any other special kind of recognized characters.

GetScoreArray

`GetScoreArray(`*unit*`)`

unit	The unit passed to the `viewClickScript` and `viewStrokeScript` methods when the user touches the pen to the screen.

Returns an array of numbers that are the recognition confidence scores for each of the words returned by `GetWordArray`. There is one score for each word. A score can range from 1 to 1000, with a lower number representing a higher recognition confidence.

RecConfig Functions

These functions allow you to manipulate `recConfig` frames.

PurgeAreaCache

`PurgeAreaCache()`

Call the `PurgeAreaCache` function when you have changed your view's `recConfig` frame, view flags or dictionaries list.This function causes the recognition behavior for all views to be recalculated for subsequent recognition operations. Recognition of any strokes that were started before this command was executed is not affected.

Deferred Recognition Functions

These functions allow you to implement your own form of deferred recognition.

RecognizePara

`RecognizePara(`*para*`, `*start*`, `*end*`, `*hilite*`, `*config*`)`

Recognize ink in the paragraph view from the start index to the end index, replacing the ink with the recognized text. All ink within the range is converted. All text within the range is left as it is. This function returns an integer that indicates the new end value for the range.

para	The `clParagraph` view containing the ink to be recognized.
start	Offset from the beginning of the paragraph to the first ink character to be recognized; indexed from zero.
end	Offset from the beginning of the paragraph to the last ink character to be recognized; indexed from zero.
hilite	The value `True` specifies that the view is to highlight each ink word as it is passed to the recognition system. If this value is `nil`, the words are not highlighted as they are recognized.
config	A `recConfig` frame or nil.

RecognizePoly

`RecognizePoly(`*poly*`, `*hilite*`, `*config*`)`

Recognize the ink in the polygon view, and replace it in the parent view with the text returned by the recognition system.

poly	The `clPolygon` view containing the ink to be recognized.
hilite	A non-zero integer value specifies that the view is to highlight each ink shape as it is passed to the recognition system. If this value is zero, the shapes are not highlighted as they are recognized.
config	A non-nil value specifies that the information in the recognition configuration frame is to be used when performing the recognition.

Correction Functions

These functions allow you to get correction information from the system.

GetViewID

`GetViewID(`*viewRef*`)`

viewRef is the view whose id you want. The returned id is a unique identifier for the view, and it is saved in the `protoWordInfo.id` slot.

GetCorrectionWordInfo

`GetCorrectionWordInfo(`*wordUnit*`)`

Returns the `wordInfo` frame for a newly written word unit. This function may be used to inspect or alter the `wordInfo` list from within your `viewWordScript` method before the word is actually added to the paragraph. This function creates a new `wordInfo` frame and caches it in the `wordUnit`, so that the same `wordInfo` frame can be used later to add `wordInfo` to the paragraph.

Dictionary Functions

These functions allow you to look up words in the built-in dictionaries and to work with your own custom dictionaries.

GetRandomWord

`GetRandomWord(`*minLength, maxLength*`)`

minLength	the minimum number of characters in words returned by this function
maxLength	the maximum number of characters in words returned by this function. This function does not return words longer than 20 characters, regardless of the value specified by this argument.

Returns a string that is a word chosen at random from the common word dictionary. The string returned by this function is at least *minLength* characters long but not more than *maxLength* characters long. This function does not return any word longer than 20 characters.

The `GetRandomWord` function uses random numbers generated by the system to select words from the dictionary. To begin a new sequence of random words you must first initialize the random number generator using the `SetRandomSeed` function. You need only call `SetRandomSeed` once to begin a new sequence of random words. For more information, see the discussion of the `SetRandomSeed` function in Chapter 17, "Utility Functions."

LookupWordInDictionary

`LookupWordInDictionary(`*dictID*`,` *word*`)`

dictID	identifier specifying the dictionary to be searched
word	the string to be found in the specified dictionary

This function looks up the specified string in the dictionary indicated by dictID, and returns True if the word is there, NIL if it was not.

DeleteWordFromDictionary

`DeleteWordFromDictionary(`*dictID*`,` *word*`)`

dictID identifier specifying the dictionary to be searched

word the string to be removed from the specified dictionary

This function removes the specified word from the RAM-based dictionary indicated by *dictID*, returning True if the word is removed and NIL if it is not. A NIL result usually indicates that the specified word was not found in the dictionary indicated by *dictID*.

It is an error to call this function on a ROM-based dictionary.

AddToUserDictionary

`AddToUserDictionary(`*wordString*`)`

wordString The string to be added to the user dictionary

Adds the word to the user dictionary. This function returns True if the word was added, or NIL if the word already exists in the dictionary.

SaveUserDictionary

`SaveUserDictionary()`

Saves the user dictionary to the permanent system soup. This saves any changes that have been made to the dictionary.

NewDictionary

`NewDictionary(`*dictionaryKind*`)`

Creates a new RAM-based dictionary and returns a dictionary identifier for it. The dictionary identifier is used in the other custom dictionary functions.

dictionaryKind specifies how the dictionary is to be used. Currently, only the symbol `'custom` has any meaning as an argument to this function. If you pass `'custom`, the dictionary is used for recognition only in views where it is specified in a dictionaries slot in conjunction with the

`vCustomDictionaries` viewFlag (refer to the previous section, “Configuring Views for Recognition.”)

Note

Although the token returned by the `NewDictionary` function currently evaluates to an integer in the Inspector, the type of value returned by this function may change on future Newton devices. Do not rely on the `NewDictionary` function returning an integer. ◆

DisposeDictionary

`DisposeDictionary(`*dictionary*`)`

dictionary The dictionary to be deleted

This function deletes the specified RAM-based dictionary and always returns NIL.

AddWordToDictionary

`AddWordToDictionary(`*dictionary, wordString*`)`

dictionary The dictionary to which this function adds the specified string

wordString The string to be added to the specified dictionary

Adds the word to the specified RAM-based dictionary, returning True if the word was added successful. If the word could not be added, this function returns NIL.

GetDictionaryData

`GetDictionaryData(`*dictionary*`)`

dictionary The dictionary from which this function extracts data.

Returns a binary representation (an object of class `'dictdata`) of the specified dictionary’s word list. You can use this function to store dictionary data in a soup.

SetDictionaryData

`SetDictionaryData(`*dictionary*, *binaryObject*`)`

dictionary The dictionary into which this function loads data.

binaryObject The binary object from which this function extracts data.

Retrieves dictionary data (the words) from the specified binary object and loads them into the specified dictionary. This function always returns NIL. You can use this function to populate a blank dictionary with dictionary items stored in a soup.

Application-Defined Methods

These messages are sent to your view during pen input.

ViewClickScript

`ViewClickScript(`*stroke*`)`

This message is sent when the user places the pen down on the screen inside the view (assuming the view has the `vClickable` flag set). This message is sent before the view system does any processing of the pen input.

stroke A unit object. This object contains information describing the interaction of the pen with the display

If the `viewClickScript` method returns `true`, the pen interaction is considered done, and the system performs no further processing of the pen input, and no other stroke-related messages are sent to the view (such as `viewStrokeScript`, `viewGestureScript`, and so on).

If the `viewClickScript` method returns `nil`, the system continues to process the pen input. The message is passed up the parent view chain, until it is handled by a `viewClickScript` method, or ignored. If the `viewClickScript` message is not handled and there are other recognition flags set, then additional system messages may be sent to the view. For example, if the `vStrokesAllowed` flag is set, then the `viewStrokeScript` message may be sent; this may be followed by the `viewGestureScript`

message, if the vGesturesAllowed flag is set; and this may be followed by the viewWordScript message, if word recognition is enabled.

You can determine the coordinates of the pen-down location using the function GetPoint, and you can control the display of electronic ink using the functions InkOff and InkOn. That is, if you want to prevent the display of electronic ink in the view, you must call InkOff from within the viewClickScript method. Alternatively, you can call the view methods TrackHilite or TrackButton; both of these methods call InkOff internally.

The StrokeDone function will tell you if the user has finished making the stroke. The Drag method will automatically track the pen on the screen and drag the view to where the pen is lifted.

Here is an example of using this method:

```
viewClickScript: func(unit)
   begin
   :Drag(unit, nil); // drag the view when it's tapped
   end
```

ViewStrokeScript

ViewStrokeScript(*stroke*)

This message is sent when the pen is first lifted after being placed down within the view (assuming the view has the vStrokesAllowed flag set). You can do whatever processing you want as a result of this event. The view system does no default processing as a result of this event.

stroke A unit object. This object contains information describing the interaction of the pen with the display

If the viewStrokeScript method returns true, the pen interaction is considered done, and the system performs no further processing of the pen input, and no other stroke-related messages are sent to the view (such as viewGestureScript, viewWordScript, and so on).

If the `viewStrokeScript` method returns `nil`, the system continues to process the pen input. The message is passed up the parent view chain, until it is handled by a `viewStrokeScript` method, or discarded. If the stroke is not handled and there are other recognition flags set, then additional system messages may be sent to the view. For example, if the `vGesturesAllowed` flag is set, then the `viewGestureScript` message may be sent, and this may be followed by the `viewWordScript` message, if word recognition is enabled.

The system regards multiple pen-down / pen-up events that are close in time as a single stroke (for writing letters and words). The `viewStrokeScript` message is sent to the view only the first time the pen is lifted during a stroke. If the pen is lifted more than once during a single stroke, only one `viewStrokeScript` message is sent for that stroke.

Note that this message will be preceded by a call to `viewClickScript`, if the view has that method.

You can determine the coordinates of the stroke using the function `GetPoint`, and you can control the display of electronic ink using the functions `InkOff` and `InkOn`. The `StrokeDone` function will tell you if the user has finished making the stroke (only the first stroke if the pen interaction actually consists of several strokes close together in time). See the chapter “Written Input and Recognition” for details on these functions.

Here is an example of using this method:

```
viewStrokeScript: func(unit)
   begin
   local bounds, points;
   bounds := StrokeBounds(unit);
   print("Bounds of stroke are "); print(bounds);
   points := GetPointsArray(unit);
   print("Points are "); print(points);
   true;
   end
```

ViewGestureScript

`ViewGestureScript(`*stroke*`, `*gesture*`)`

This message is sent when the user writes a recognizable gesture inside the view (assuming the view has the `vGesturesAllowed` flag set).

stroke A unit object. This object contains information describing the interaction of the pen with the display

gesture An integer code that identifies the gesture that was recognized. The following gestures are supported:

Gesture	Constant	Integer Value
tap	`aeTap`	49
double tap	`aeDoubleTap`	50
scrub	`aeScrub`	13
highlight	`aeHilite`	47
caret	`aeCaret`	15
horizontal line (makes a new page in the notepad)	`aeLine`	16

This message is sent after the view system recognizes the gesture, and only if the gesture is one not normally handled by the view. For example, views of the `clParagraphView` class handle all gestures except a tap, so for this kind of view, the `viewGestureScript` message will not usually be sent (except for a tap). (However, if you set the `vReadOnly` flag in the `viewFlags` slot, the `viewGestureScript` message will be sent for all gestures except highlight.)

Note

You can work around the limitation that this message is sometimes not sent. For example, you may want a view to receive this message no matter what kind of view it is, or what kinds of input it handles. You can do this by creating a child view of the `clView` class that is transparent and the same size as the input view you are working with. If you set the appropriate input flags for the `clView`, it will receive the input-related messages first. For any particular message, the `clView` can take some action and return `true` to prevent the message from being passed to the parent, or it can return `nil` to pass the message on to the parent. ◆

If the `viewGestureScript` method returns `true`, the pen interaction is considered done, and the system performs no further processing of the pen input, and no other stroke-related messages are sent to the view (for example, `viewWordScript`).

If the `viewGestureScript` method returns `nil`, the system continues to process the pen input. The message is passed up the parent view chain, until it is handled by a `viewGestureScript` method, or discarded. If the tap is not handled and there are word recognition flags set, then the `viewWordScript` message may be sent to the view.

Note that this message will be preceded by a call to `viewClickScript`, if the view has that method. It may or may not also be preceded by a call to `viewStrokeScript`, if the view has that method.

Here is an example of using this method:

```
viewGestureScript: func(unit, gestureKind)
   begin
   if gestureKind = aeLine then  // If it was a line
      begin
      // Make a new data item in our app
      end;
   end
```

ViewWordScript

`ViewWordScript(`*stroke*`)`

This message is sent when a word is recognized and passed to the view (assuming the view has some kind of word recognition flag set).

stroke	A unit object. This object contains information describing the interaction of the pen with the display

You can get the word that was recognized by calling the function `GetWordArray`. The first string returned in the word array is the one with the highest recognition probability.

The `viewWordScript` message is sent after the system recognizes the word, and only if the view is not one that normally supports word recognition. For example, views of the `clParagraphView` and `clEditView` class support word recognition, so will not normally receive this message. (However, if you set the `vReadOnly` flag for these views, the `viewWordScript` message will be sent.)

Note

You can work around the limitation that this message is sometimes not sent. For example, you may want a view to receive this message no matter what kind of view it is, or what kinds of input it handles. You can do this by creating a child view of the `clView` class that is transparent and the same size as the input view you are working with. If you set the appropriate input flags for the `clView`, it will receive the input-related messages first. For any particular message, the `clView` can take some action and return `true` to prevent the message from being passed to the parent, or it can return `nil` to pass the message on to the parent. ◆

If the `viewWordScript` method returns `true`, the event is considered handled. If the `viewWordScript` method returns `nil`, the message is passed up the parent view chain, until it is handled by a `viewWordScript` method, or discarded.

Here is an example of using this method:

```
viewWordScript: func(unit)
   begin
   local matchedWords, recognizedWord;
   matchedWords := GetWordArray(unit);
   recognizedWord := matchedWords[0];
   print("The recognized word was " & recognizedWord);
   true;
   end
```

Note

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

User Dictionary Methods

This section describes methods you can use manipulate the user dictionary and the expand dictionary.

These dictionaries are accessible from the root view. You can use code similar to the following example to get a reference to an object to which you can send the messages described in this section.

```
local reviewDict := GetRoot().ReviewDict;
```

AddWord

reviewDict:`AddWord(`*word*`)`

Adds the specified word to the user dictionary. If the word is added successfully, this method returns the value `True`.If the word is already in the user dictionary or one of the standard system dictionaries, then the word is not added and the return value of this method is unspecified.

If the Personal Word List is open, the display will be automatically updated. An undo action is posted for this operation. For performance reasons, the dictionary is not flushed to the user store for each word that is added.

word The word to be added to the user dictionary.

RemoveWord

reviewDict:`RemoveWord(`*word*`)`

Call this function to remove the specified word from the user dictionary. If the call succeeds, then it returns true; otherwise it returns nil. Note that the case of the word to be removed must match exactly. If the Personal Word List is open, the display will be automatically updated. An undo action is posted for this operation. For speed the dictionary is not flushed to the user store.

word The word to be removed from the user dictionary.

LoadUserDictionary

`LoadUserDictionary()`

Loads the Review Dictionary into RAM from the system soup entry having the `UserDictionary` tag.

SaveUserDictionary

`SaveUserDictionary()`

Writes the Review Dictionary from RAM into the system soup entry having the `UserDictionary` tag.

AddExpandWord

reviewDict:`AddExpandWord(`*word*`,` *expandedWord*`)`

Adds a word and its expanded version to the expand dictionary. The word must be recognized before it can be expanded, so you must first invoke the `AddWord` method to add the word to the user dictionary. If the word is not already in the expand dictionary and can be successfully added, the

`AddExpandWord` method returns the value `True`; otherwise its return value is unspecified. If the Personal Word List is open, its display is updated automatically. An undo action is posted for this operation. For performance reasons, the dictionary is not flushed to the user store.

word The abbreviated version of *expandedWord* to be added to the the expand dictionary.

expandedWord The word to be added to the user dictionary.

GetExpandedWord

reviewDict`:GetExpandedWord(`*word*`)`

Looks for the specified word in the expand dictionary and returns the expansion if the word is found. If the word is not found in the expand dictionary, the return value of this method is unspecified.

word The word to be found in the expand dictionary.

RemoveExpandedWord

reviewDict`:RemoveExpandedWord(`*word*`)`

Looks for the specified word in the expand dictionary and removes it, if present.

Because each word in the expand dictionary has a matching word in the user dictionary, you need to invoke the the `RemoveWord` method whenever you use the `RemoveExpandedWord` method.

LoadExpandDictionary

`LoadExpandDictionary()`

Loads the expand dictionary from the user store into RAM. The expand dictionary is stored in the user store in the system soup entry with the `ExpandDictionary` tag.

SaveExpandDictionary

```
SaveExpandDictionary()
```

Writes the expand dictionary from RAM into the user store. The expand dictionary is saved in the user store in the system soup entry with the `ExpandDictionary` tag.

Summary of Recognition

Prototypes

```
protoCharEdit := {
CurrentWord() 92
GetWordForDisplay() 93
DeleteText(left, right) 93
Scroll(direction) 93
UseTextAndTemplate() 94
SetNewWord(str, dontUse) 94
UseNewWord() 94
FixedWord() 95
FixedWordLength() 95
DisplayExternal(doIt) 95
SaveUndoState(appState) 95
RestoreUndoState(savedState) 96
charEditView:SetupString(str) 96
charEditView:CleanupString(str) 96
...}
protoWordInterp := { ... }
protoCorrectInfo := { ... }
protoRecToggle := { ... }
```

protoCharEdit Templates

```
ROM_numberFilter // general-purpose numeric template
GetLocale().timeFilter // time template
GetLocale().dateFilter// date template
GetLocale().phoneFilter // phone numnber template
```

Correction Information Functions

correctInfo:RemoveView(*view*)
correctInfo:Find(*view*, *offset*)
correctInfo:FindNew(*view*, *offset*, *length*)
correctInfo:AddUnit(*view*, *start*, *stop*, *unit*)
correctInfo:AddWord(*wordInfo*)
correctInfo:Clear(*view*, *offset*, *length*)
correctInfo:Extract(view, start, stop)
correctInfo:Insert(*newCorrectInfo*, *newView*)
wordInfo:SetWords(*words*)
wordInfo:GetWords()
wordInfo:AutoAdd()
wordInfo:AutoRemove()
GetViewID(*viewRef*) 85
GetCorrectionWordInfo(*wordUnit*) 85

Inker Functions

InkOff(*unit*)
SetInkerPenSize(*size*)

Recognition Functions

`BuildRecConfig(`*viewRef*`)`
`GetPoint(`*selector*`, `*unit*`)`
`GetPointsArray(`*unit*`)`
`GetScoreArray(`*unit*`)`
`GetViewID(`*viewRef*`)`
`GetWordArray(`*unit*`)`
`PurgeAreaCache()`
`RecognizePara(`*para*`, `*start*`, `*end*`, `*hilite*`, `*config*`)`
`RecognizePoly(`*poly*`, `*hilite*`, `*config*`)`
`StrokeBounds (`*unit*`)`
`StrokeDone(`*unit*`)`

Dictionary Functions

`AddToUserDictionary(`*wordString*`)`
`AddWordToDictionary(`*dictionary, wordString*`)`
`DeleteWordFromDictionary(`*dictID,word*`)`
`DisposeDictionary(`*dictionary*`)`
`GetDictionaryData(`*dictionary*`)`
`GetRandomWord(`*minLength, maxLength*`)`
`LookupWordInDictionary(`*dictID,word*`)`
`NewDictionary(`*dictionaryKind*`)`
`SaveUserDictionary()`
`SetDictionaryData(`*dictionary, binaryObject*`)`

User Dictionary Functions and Methods

reviewDict:AddWord(*word*)
reviewDict:RemoveWord(*word*)
LoadUserDictionary()
SaveUserDictionary()
reviewDict:AddExpandWord(*word*, *expandedWord*)
reviewDict:GetExpandedWord(*word*)
reviewDict:RemoveExpandedWord(*word*)
LoadExpandDictionary()
SaveExpandDictionary()

Application-Defined Methods

ViewClickScript(*stroke*)
ViewStrokeScript(*stroke*)
ViewGestureScript(*stroke*, *gesture*)
ViewWordScript(*stroke*)

Recognition

Data Storage and Retrieval

This chapter describes the Newton data storage model and the key concepts that underlie its use. The Newton operating system provides a suite of objects that interact with one another to

- manage one or more persistent stores
- read and write data and objects residing on RAM stores
- read data and objects residing on ROM stores

This chapter describes the use of these objects to store and retrieve data. If you are developing an application that stores data, retrieves data or provides a pre-existing data set to the user, you should become familiar with the Newton data storage system and the concepts discussed in this chapter.

This chapter is divided into four main parts: an introduction, a conceptual section, a practical section and a reference section.

The introductory portion of the chapter consists of three sections.

- “Introduction to Data Storage on Newton” beginning on page 11-3 describes the principal data storage objects that most Newton applications use, such as stores, soups, cursors, entries and frames, as well as important operations such as data queries and soup change notifications.

- “Working With Storage Objects” beginning on page 11-10 provides a high-level overview of the interaction of storage objects used by most Newton applications.
- “Special-Purpose Storage Objects” beginning on page 11-12 describes specialized objects such as virtual binary objects, packages, parts, package stores and mock entries.

The conceptual portion of the chapter consists of several parallel sections of equal importance. Where applicable, these sections also provide compatibility information for older Newton devices.

- “About Stores” discusses stores and the objects that reside on them, such as packages, parts, store parts and soups.
- “About Soups” discusses issues and strategies pertaining to soup-based storage, alternatives to soup-based storage and related data structures such as soup definitions, indexes and tags.
- “About Virtual Binary Objects” compares and contrasts virtual binary objects with normal NewtonScript objects.
- “About Queries” describes the retrieval of soup data.
- “About Cursors” describes the data-access object that queries return.
- “About Entries” describes the properties of the individual data items returned by a cursor object.

The practical portion of the chapter consists of several sections that present code examples describing the use of various Newton data storage objects. These sections assume familiarity with the conceptual material presented earlier in the chapter.

- “Using Stores” describes how to create and retrieve store-relatedobjects and information.
- “Using Soups” describes how to create, retrieve and remove soups, how to get and set soup-related information, how to add indexes to existing soups and how to perform certain soup-mantenance tasks.
- “Using Queries, Indexes and Tags” describes how to create a cursor object that iterates over a specified subset of soup data.

- "Using Cursors" describes how to retrieve individual data items from the cursor object that a query returns.
- "Using Entries" describes how to add data to soups and how to use individula data items extracted from a soup.
- "Using Soup Change Notification" describes how your application can notify the system that it has made changes to shared soup data and how your application can register a callback function to execute when changes are made to the contents of specified soups.
- "Using Virtual Binary Objects" describes how to use virtual binary objects to store large amounts of data.
- "Using Store Parts" describes how to build read-only soup data into packages.
- "Using Mock Entries and Mock Soups" describes how developers can define special-purpose objects that act like the built-in suite of store, soup, cursor and entry objects.

Although these sections are equal in importance, they are presented in the order one is likely to use them when writing a NewtonScript application. For example, because one sends a message to a store to retrieve a soup object, the "Using Stores" section precedes the "Using Soups" section. Similarly, each section presents its topics in the order they are likely to be needed; for example, information on creating soups is presented before information on sending messages to soups.

The last portion of this chapter, "Data Storage Reference,"provides complete descriptions of all constants, data structures, functions and methods in the Newton data storage system.

Introduction to Data Storage on Newton

Newton devices utilize one or more persistent stores to store data permanently. The Newton operating system implements a suite of interdependent objects that manage these stores and the objects that reside

on them.This section introduces Newton data storage objects and provides an overview of their use.

Memory

The Newton object storage system and the NewtonScript programming language free application developers from having to allocate and free memory explicitly. For more information, see the discussion of garbage collection in *The NewtonScript Programming Language*.

Data Objects

Newton devices store data as objects.The NewtonScript programming laguage provides four basic object types that applications can use to store data.

Immediate	A small, immutable object such as a character, integer or Boolean value.
Binary	Raw binary data. This kind of object can be used to store data such as bitmaps.
Array	A collection of object references accessed from a numerical index.
Frame	A collection of object references accessed by name.

Because immediates, binaries and arrays are object representations of data types that are common in many programming languages, they are not discussed further here. For a complete description of these objects, see *The NewtonScript Programming Language*.

The frame is of particular interest, however, as it can contain any of the other objects and is the only object to which one can send messages. The next section provides a more detailed look at the characteristics of frames.

Frames and Slots

At the heart of the Newton object system's shared data model is an object called the **frame** which offers a flexible and efficient means of storing data.

- Frames are sized dynamically—they grow and shrink as necessary.
- A common set of predefined NewtonScript data types that all frames support allows Newton applications to share most data virtually transparently.
- Dissimilar data types can be stored in a single frame.

Like a database record, a frame stores data items. An individual data item in the frame is stored in a **slot,** which may be thought of as a field in the database record. Unlike database records, however, frames need not all contain the same slots.

Slots can store any NewtonScript data type, such as strings, numeric formats, arrays and binary objects. Note that NewtonScript does not require that slots declare a datatype. Slots are untyped because every NewtonScript object stores datatype information as part of the object itself. (NewtonScript variables need not declare a type, either, for the same reason.)

Slots can also store other frames and references to frames, slots, and binary objects. The ability to reference other frames from slots allows the typical Newton application to inherit attributes and behaviors from ROM-based objects known as system prototypes or "protos." This feature of the object system also provides dynamic slot lookup and message-passing among frames. Slot-based inheritance and method dispatching in NewtonScript is described in detail in *The NewtonScript Programming Language.*

Frames don't impose any structure on their data other than the requirement that it must reside in a slot. In practical use, though, the slots in a frame are usually related to each other in some way. They generally store related data and methods that operate on the data, embodying in many ways the classic object-oriented programming definition of an "object,"Frames do not implement data hiding, however, nor do they necessarily encapsulate their data.

RAM-based frames are not persistent until they are stored in an object known as a soup. The next section describes soup objects in more detail.

For detailed discussions of frame and slot syntax, system-supplied data types, dynamic slot lookup, message-passing and inheritance in NewtonScript, see *The NewtonScript Programming Language.*

Soups

Frames can be stored in a persistent data structure called a **soup**, which is an opaque object that provides a dynamic repository for data. Unless removed intentionally, soups remain resident on the Newton device even when the application that owns them is removed.

A soup is simply a collection of frames. Multiple data types can reside in a single soup; the object system does not impose any limitations on the number of frames or the kinds of data that may reside in a soup. In practical use, though, the items in a soup generally have some relationship to one another.

Soups are made available to the system in a variety of ways. Applications may create them on demand, they may be installed along with the application itself, or the user may introduce them on a PCMCIA card.

Stores

The soup resides on a **store,** which is a logical data repository on a physical storage device. A store may be likened to a disk partition or volume on a conventional computer system; just as a disk can be divided logically into multiple partitions, a physical storage device can house multiple stores. The Newton platform supports a single internal store and one or more external stores on PCMCIA devices.

Each store is identified by a name, which is not necessarily unique, though a store has a nearly-unique random number identifier called a **signature**.

Soups can reside on internal or external stores; a special kind of soup, the union soup, is composed of multiple soups residing on multiple stores.

Union Soups

An application's data is often not stored in a single soup. For example, when a PCMCIA card is installed, data may be spread out amongst the internal and card soups. The object system provides a way to address multiple soups as a single "virtual" soup called a union soup**.**

A **union soup** is an object that represents multiple same-named soups residing on different stores. Union soups allow applications to treat multiple soups as a single entity, regardless of their locations on various physical stores.

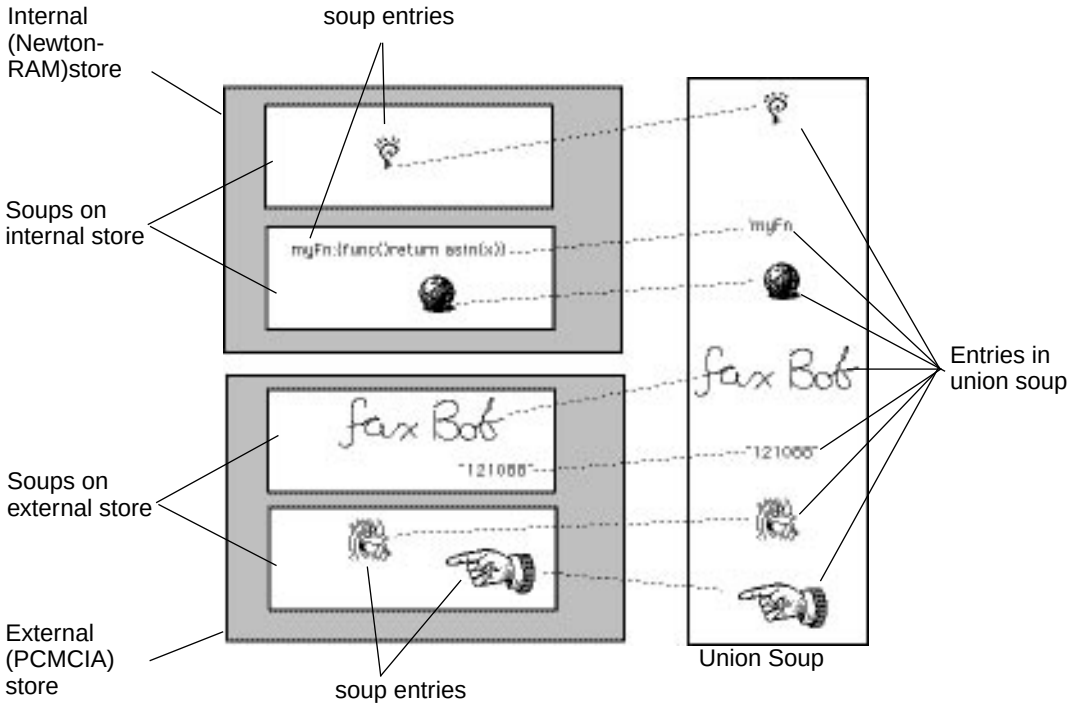
It's important to understand that there is only one kind of soup object in the system; union soups are simply a logical association of multiple soup objects. In other words, aside from their logical association with other soups in the union, a union soup's constituent soups are no different than soups that are not part of a union. Unless specifically noted otherwise, anything said about soups in this text applies equally to union soups.

In general, you'll want to use union soups for most purposes, and certainly for most data that the user creates with your application. Applications must allow the user to choose whether new data is to be stored on the internal or external store; using union soups makes this easy to do.

Regardless of whether they participate in a union, all soups store data as objects called entries.

Figure 11-1 depicts the concept of a union soup graphically.

Figure 11-1 Stores, soups and union soups



Entries

A frame stored in a soup is referred to as an **entry** in that soup. Returning to the database analogy, you can think of entries as the individual records in the database, and you can think of the soup as the database itself. Like a database, a soup is opaque—you retrieve soup entries by querying the soup, rather than by examining its records directly.

Queries and Cursors

Applications retrieve entries from a soup by asking for them; the request takes the form of a `Query` message sent to the soup object. Union soups and single soups respond to queries in exactly the same manner.

The soup method `Query` accepts as its argument a frame known as the **query specification** or **query spec**. The query spec describes the kind of information that is to be returned as well as the order in which it is to be returned.

The `Query` method returns a **cursor,** which is an object that iterates over the set of entries meeting the criteria defined by the query spec. Cursors are updated dynamically: if soup entries meeting the search criteria are added or deleted after the original query is made, these changes are automatically reflected in the set of entries referenced by the cursor.

Soup Change Notification

Because a primary purpose of the frames-based storage model is to facilitate the sharing of data, the system provides a means of notifying other objects when soup data changes. This mechanism is described in the section “Using Soup Change Notification,” later in this chapter.

Inter-Package Magic Pointers

Inter-package magic pointers enable application packages to reference objects in other packages. The “magic” is that the object referenced by the pointer can change without changing the pointer’s address. Thus, one can use magic pointers to make new objects available to a previously-installed application. For example, one could create an installer that adds new kinds of stationery to an existing application.

Working With Storage Objects

The preceding section, “Introduction to Data Storage on Newton,” described the principal players in the Newton data storage system. This section describes how they interact in a typical application.

Frame objects perform the lion’s share of work in Newton applications. They are full-fledged objects in the classic object-oriented programming sense, with the exception that they do not hide data. Your application can store data in a frame’s slots, retrieve data from those slots directly or through inheritance and send messages to the frame itself to invoke methods (again either directly or via dynamic slot lookup).

Recall that frames are not persistent unless they are stored in a soup. Thus, most applications create and retrieve soup objects that hold frame data. Applications can use as many soups as they need, subject to the availability of memory space on the permanent store and in the NewtonScript heap. For most uses, union soup objects are preferable, providing the most benefits for the least effort.

The application creates a union soup by registering a soup defiinition frame with the system; registering the soup definition causes the system to return a union soup object to which the application can send messages. This object may be a new soup or one previously created by this or another application.

Frames are stored in soups as soup entries. To add a frame to a soup, your application sends a message to the soup to transform the frame into a soup entry.

To retrieve soup data, the application queries the soup for entries meeting a specified set of criteria; the soup returns a cursor object that iterates over the set of entries having the specified characteristics. The cursor object responds to messages that permit the extraction of individual entries from the search results.

At this point, the entries are still compressed soup data residing on one or more stores and the application can manipulate the entries—move them,

copy them, delete them, and so on—without reading them into the NewtonScript heap.

The first time a slot in the entry is referenced—whether to read its value, set its value, or to print its value to the Inspector—the system decompresses the entire entry and creates from it a “normal” frame in a special area of RAM known as the entry cache. Changes to the entry are actually made in the cached frame, not the compressed soup entry; hence, changes to a soup entry are not persistent until the cached frame is written back to the soup entry from which it was created. This scheme also makes it simple to undo the changes to a soup entry—the system simply throws away the cached frame and restores references to the original soup entry.

Applications can register callback functions to be executed when data in a particular soup changes. Soup changes that applications might require notification of include creating soups, deleting soups and adding, removing or changing soup entries.

In summary, most applications that work with dynamic data will peform the following operations described in this chapter.

- creating and using frames
- storing frames as soup entries
- querying soups to retrieve sets of entries
- using cursor objects to manipulate sets of soup entries
- extracting individual entries from cursor objects
- manipulating individual soup entries as frame objects
- returning modified entries to the soup from which they came
- notifying other applications of changes to soups

The next section describes objects that the Newton data storage system provides for special purposes.

Special-Purpose Storage Objects

The special-purpose data storage objects described here can be used to augment the behavior of stores, soups, cursors and entries as required.

Virtual Binary Objects

The size of any NewtonScript object is limited by the amount of memory available in the NewtonScript heap. The system provides virtual binary objects as a means of working around this restriction.

A **virtual binary object** or **VBO** is a special kind of object used to create binary objects larger than the available space in the NewtonScript heap. VBOs are used to store large amounts of raw binary data, such as large bitmaps, the samples of a large digitized sound, fax data, packages or application-specific binary data.

A VBO is “virtual” because it does not actually reside in the NewtonScript heap as other NewtonScript objects do; instead, the system dynamically maps data from the VBO into NewtonScript heap memory as required. When creating a VBO, one must specify a permanent store in which its associated binary data is to reside. The system compresses and decompresses this binary data automatically as it is paged in and out of the NewtonScript heap.

Packages

A **package** is the basic unit of downloadable Newton software: it provides a means of loading code, resources, objects, and scripts into a Newton device. Most Newton applications are shipped as packages that can be installed on a Newton device by the Newton Connection Utility or Newton Package Installer applications.

Packages can be read either from a data stream or directly from memory. For example, Newton Connection Kit uses a data stream protocol to load a package into the Newton system from a Macintosh or Windows computer. However, it is much more common to use packages directly from memory—which is what the user does once the package has been installed by Newton Connection Utility or Newton Package Installer.

Packages are actually a special kind of virtual binary object, but they are read-only and are created in a slightly different manner than normal VBOs.

Packages are composed of one or more units of software known as **parts**. The various kinds of parts are discussed in more detail in the section "Parts" beginning on page 11-27. However, one kind of part, the store part, deserves special mention as a useful means of storing read-only data.

Store Parts

A **store part** is a part that encapsulates a read-only store. This store may contain one or more soup objects. Store parts permit soup-like access to read-only data residing in an application package.

Entry Aliases

An **entry alias** is an object that provides a standard way to save a reference to a soup entry. Entry aliases themselves may be stored in soups. Because queries must be sent to a single soup or a union soup, entry aliases are also useful for providing access to items in dissimilar soups.

Entry aliases are discussed in more detail in the section "About Entry Aliases" beginning on page 11-51.

Mock Entries

A **mock entry** is a NewtonScript object that mimics the behavior of a soup entry. The mock entry is a foundation object that developers can use to build up a suite of objects that acts like the system-supplied store, soup, cursor and entry objects. For example, one could create a mock entry object that uses a

serial communications link to return a record from a remote database; additional objects could implement methods to provide cursor-like access to these mock entries, just as if they resided in a local soup.

The current implementation of the Newton object system provides only mock entries; the developer must implement appropriate mock cursors, mock soups and mock stores as required.

Evaluating Your Data Storage Needs

When choosing schemes for storing your application's data, you need to consider factors such as

- the kind of data to be stored
- the quantity of data to be stored
- how the application accesses the data

The most important factor to consider with respect to the kind of data is whether the data is static or dynamic. You must use soups to store dynamic data, but a number of options are available for storing static data. You will probably find that certain structures lend themselves more naturally than others to working with your particular data set.

Especially for large data sets, space-efficiency may influence your choice of one data structure over another. In some cases, you may need to consider tradeoffs between space requirements and speed or ease of access.

Data access issues include questions such as whether the data structure under consideration facilitates searching or sorting the data. For example, soups provide powerful and flexible searching and sorting mechanisms.

Dynamic Data

Data that your application gathers from the user must be stored in soups. Within individual soup entries you are free to store data in whatever manner best suits your application's needs.

Because each entry in a soup is a frame, the price you pay for using soup-based storage can be measured in terms of

- the time required to find slots at run time
- the memory space required to expand soup entries
- the memory space required to store the expanded entry frames on the NewtonScript heap

For many uses, the benefits offered by the use of soups outweigh these costs; however, other approaches may be more suitable for certain data sets, especially large sets of read-only data.

Static Data

Read-only or static data can be stored in packages held in protected memory on the Newton. There are a variety of reasons you might store data in a package rather than in a soup.

- Storing static data in a package rather than in a soup helps to conserve store space and NewtonScript heap space.
- Although the user might enter data dynamically, there might be a large initial set of data your application needs to provide. Again, it's more efficient to supply this as package data rather than as soup data.
- You can supply multiple static data sets as separate packages to allow the user to load some subset of that data. For example, a travel guide application might keep the data for each country in its own package.

The next section describes various approaches to storing static data.

Storing Static Data in Packages

This section describes several approaches to storing static data in packages. It is not intended to present a single strategy that is appropriate for every use, but rather to illustrate the the strengths and weaknesses of various approaches.

You can use the information presented here as a starting point for choosing storage strategies and data structures that best suit your application's needs. For best results, you are encouraged to experiment with various approaches to storing a data set that is representative of the kind and quantity of data your application manipulates. For a description of the tests you can conduct on your data structures, including sample code, see the Newton DTS "Lost In Space" code sample. The information presented here is based on the "Lost In Space" code sample.

This section begins with descriptions of several approaches to structuring Newton data. Following these descriptions is a comparison of the space efficiency and data access speed they provide.

Structuring Newton Data

As you know, arrays, frames, binaries and soups can be combined; for example, you can create an array of frames and store it as a soup entry. For the purposes of discussion, consider an array of 300 frames structured as in the following example.

```
gData := [ // a naive array of frames
{slot1: 000, slot2: "foo 000", slot3: 42, slot4: true},
{slot1: 001, slot2: "foo 001", slot3: 42, slot4: true},
{slot1: 002, slot2: "foo 002", slot3: 42, slot4: true},
// several array elements (frames) not shown
{slot1: 297, slot2: "foo 297", slot3: 242, slot4: true},
{slot1: 298, slot2: "foo 298", slot3: 242, slot4: true},
{slot1: 299, slot2: "foo 299", slot3: 242, slot4: true},
];
```

This array of frames is considered naive because it does not utilize any space- or speed-optimization techniques.

Naive Array of Frames

Associated with each frame is another data structure called a **map**, which is used to locate the value associated with a specified slot name. Frames with common structure can save heap space by sharing maps. Unfortunately, an array of frames defined as in the `gData` array does not utilize map sharing. Even though the frames in the `gData` array are structured identically, they each have their own map because each frame constructor in your code creates its own map. Note that map sharing is a technique that you can apply to all of the frames your code creates, not just those holding static data.

Although it is possible to detect frames with identical maps and cause them to share maps, the Newton Toolkit compiler does not currently perform this optimization. Thus, use of the `gData` array "as is" with no further processing would be extremely wasteful of NewtonScript heap space.

Use of a naive array of frames is not recommended as a serious candidate for representing your application's data. It is included to illustrate a common pitfall in NewtonScript programming. The next section describes how frames can share maps to avoid wasting NewtonScript heap space.

Shared Frame Map

The easiest way to ensure that your frames share maps is to make sure they are all generated by the same frame constructor. You can do this by defining a function used to create all of the frames that share a common map. The following example illustrates such a function.

```
func MakeFrame(s1,s2,s3,s4){slot1:s1,slot2:s2,slot3:s3,slot4: s4};
```

You may be familiar with the `SetBounds` and `RelBounds` global functions which return view bounds frames. You might have thought them pointless since it's almost as easy to write

```
local bounds := {left: 0, top: 0, right: 100, bottom: 100};
```

as it is to write

```
local bounds := SetBounds(0,0,100,100);
```

Now you know that using the `SetBounds` function creates view bounds frames that share maps, which is a good reason to use this function. In fact, the frames returned by the run-time version of `SetBounds` use a shared map that resides in ROM. You can write similar functions for creating your own types of frames that share frame maps.

An alternative way to create frames with a shared map is to make them by cloning a common frame, as illustrated in the following example.

```
constant kDummy := '{s1: nil, s2: nil};

local frame1 := Clone(kDummy);
frame1.s1 := 42;
frame1.s2 := true;

local frame2 := Clone(kDummy);
frame2.s1 := 43;
```

At run-time, both `frame1` and `frame2` share a common map.

Note that the technique of cloning a common frame is described only for the sake of completeness. Writing functions to build your frames is a better approach because it makes code more legible and easier to maintain. In addition, the use of a frame constructor function is generally faster than a call to `Clone` followed by multiple assignment statments.

Array Of Arrays

This data structure uses an arrays rather than frames to represent the elements of the `gData` array. For example, the following element of the `gData` array

```
{slot1: 002, slot2: "foo 002", slot3: 42, slot4: true}
```

is represented as

```
[002, "foo 002", 42, true]
```

Using an array of frames provides a more legible data structure than using an array of arrays. The price you pay for this additional legibility is small: As long as map sharing is utilized, the array of frames does not require much more additional storage space than the array of arrays The time required to access data items in the array of frames is not much more than that required to access items in the array of arrays. (Note that figures generated by testing a representative data set are included in the summary at the end of this section.)

Frame Of Frames

This data structure is a frame containing a slot for each element of the `gData` array. The `slot2` values in the `gData` array are used as the names of the slots in the frame of frames. For example, consider the following portion of the `gData` array:

```
[
…
{slot1: 297, slot2: "foo 297", slot3: 242, slot4: true},
{slot1: 298, slot2: "foo 298", slot3: 242, slot4: true},
…
]
```

The corresponding part of the frame-of-frames data structure is:

```
{
…
|foo 297|: {slot1: 297, slot3: 242, slot4: true},
|foo 298|: {slot1: 298, slot3: 242, slot4: true},
…
}
```

Representing the data as a frame instead of as an array allows you to utilize NewtonScript slot lookup to retrieve data items; that is, to retrieve the data for the "foo 297" entry we can use code like the following example.

```
frameOfFrames.(Intern("foo 297"))
```

Note the use of the `Intern` global function to turn the string key value into a symbol. This is necessary because slot names are symbols, not strings. It is also important to note that characters in symbols (and therefore, in slot names) are restricted to 7-bit ASCII values. This means that if you need to include special characters in your slot names you will need to transform them into 7-bit ASCII values using your own encoding scheme. (For an example of using special characters in symbols, see the section “Application Symbol” beginning on page 2-10)

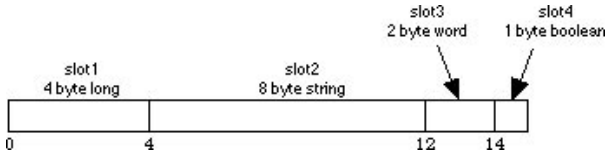
Consider storing symbols for repeated strings or frames rather than storing the strings or frames themselves. When you define a symbol for an object, only one copy of the object is stored in the application package, and all of the symbols reference it.

Remember that symbols are limited to 7-bit ACII values. Symbols (slot names) can include non-alphanumeric characters if the name is enclosed by vertical bars; for example, the space in the symbol `'|Chicken Little|` would normally be illegal syntax because it contains a space, but the vertical bars suppress the usual evaluation of all characters they enclose.

Binary Objects

The most space-efficient approach is to encode the entire `gData` array as a single binary object. Each element of the `gData` array occupies 15 bytes of the binary object, as depicted in Figure 11-2.

Figure 11-2 Binary object representation of an element of the `gData` array



Not altogether uncoincidentally, you pay for savings of space in difficulty accessing and manipulating the data. You must use the various `Stuff`*Xxx* and `Extract`*Xxx* binary functions to access the data. No searching or other object-level data manipulation functions are available.

Soups

It is not particularly desirable in any way to create a soup containing an entry for each element of a large static data set such as the `gData` array: it is not efficient in terms of speed or space. However, such a soup was tested for purposes of comparison and contrast with the other approaches described here.

Store Parts

Store parts provide a way to include a read-only soup as part of your package. Because you can find other approaches that will yield a smaller data set or faster access to data, store parts are useful mainly when you want to avoid recoding your data in a more efficient representation.

A representative data set was built into a store part for inclusion in the comparative tests described at the end of this section. The store part that was constructed contains a soup having an entry for each element of the `gData` array.

The next two sections summarize the results of tests conducted on the example data structures to rate their speed and compactness.

Space Results

Repeatedly compiling the same Newton Toolkit project with different representations of the `gData` data set produced the package size results shown in Table 11-1.

Table 11-1 Package sizes in bytes

	Package Size	
Data Structure	**Uncompressed**	**Compressed**
Array of frames (naive)	35538	11884
Array of frames (shared map)	25970	8932
Array of arrays	25866	8860
Frame of frames	22386	9176
Binary object	9962	5328
Soup	N / A[1]	37260
Store part	35012	14900

[1] Soups are always compressed.

As you can see, storing the data as a binary object produced the most compact representation. The results also show that map sharing produces a substantial difference in data size. This difference becomes even more pronounced as the number of slots in your frames increases.

Note also that using arrays instead of frames to represent the elements of our data set did not yield a great savings of space over representing the same data as frames utilizing a shared map.

Finally, note that the store part compresses more than the normal soup, due to the system's use of a different compression scheme for read-write soups.

For a complete description of how these tests were conducted, including sample code you can modify for use with your own data set, see the Newton DTS "Lost In Space" code sample.

Speed Results

The packages produced for the space testing were also used in tests to determine which data structures provided the fastest access to data. The benchmark test used for timing data access was to retrieve each of the elements of the data set using the `slot2` value as the key. The data was sorted on `slot2` to allow the use of a standard binary search algorithm to retrieve items from the array of frames, array of arrays and binary object data structures.

The frame of frames test used NewtonScript slot lookup to retrieve values rather than an explicit binary search; however, it should be noted that for large frames the current implementation of NewtonScript slot lookup is a binary search on slot-name hash values. The soup searching code was based on the use of `slot2` as the index path, although `slot1` could work as well. The source code used to conduct these tests is supplied with the Newton DTS “Lost In Space” code sample.

Table 11-2. shows the times required to run the benchmark test on each of the data structures discussed here.

Table 11-2 Data access times

Data Structure	Benchmark time (ticks)
Array of frames (shared map)	755
Array of arrays	725
Frame of frames (first access)	97
Frame of frames (subsequent accesses)	60
Binary object	1001
Soup	420
Store part	343

For a complete description of how these tests were conducted, including sample code you can modify for use with your own data set, see the Newton DTS “Lost In Space” sample code.

The values in Table 11-2 show that the frame-of-frames provides the fastest lookup. Note also that repeated lookups in the frame-of-frames are faster than initial lookups because Newton caches frame maps and because the slot name symbols have already been created from the initial lookup.

The use of compression did not affect data access times appreciably, probably because the packages are small enough that they can fit in working memory in their entirety when they are paged-in and uncompressed. When performing your own tests, run them on a representative quantity of data so that they reflect the amount of time required to page data in and out of memory.

Conclusions

There are a wide variety of trade-offs to consider when choosing a structure to represent your application data. You are strongly advised to conduct realistic tests with the actual data set your application manipulates before committing to the use of a particular data structure. It's also recommended that you design your application in a way that allows you to experiment with the use of various data structures at any point in its development.

The rest of this section provides brief summaries of the strengths and weaknesses of the approaches described here.

Map Sharing

Clearly, you should be sure to take advantage of map sharing—it can make a substantial difference in package size.

Array of Frames

An array of frames is a good general-purpose data structure. If speed of access is a concern, consider using the frame of frames as an alternative. As a general rule, keep arrays sorted so that you can perform binary searches on them.

Array of Arrays

The array of arrays provides no real advantage over the array of frames, and it has the disadvantage of being a less-legible data structure. Unless your data is more naturally represented as an array, there is probably no reason to prefer an array of arrays over an array of frames.

Frame of Frames

A frame of frames is a significantly faster alternative to an array of frames. The only disadvantage is that if your key values contain special characters you will have to deal with the complexity of encoding them in 7-bit ASCII – not too great a disadvantage.

Binary Objects

Binary objects are definitely the most complex and least legible data structure discussed here; however, if space is of paramount importance, then you should definitely consider using binaries to store your data. One way to make them easier to use is to write your own accessor (setter and getter) methods.

Extracting your data from binary-objects entails a certain amount of overhead, especially in the case of strings. This overhead is reflected in the slower lookup times shown for binaries in Table 11-2. Remember, too, that extracting a string from a binary object requires that the system allocate a new object in the NewtonScript heap. If you encode arrays or frames within your binary objects, these too will require the creation of objects in the NewtonScript heap when they are extracted. In contrast, the other data structures discussed here allow you to reference their sub-parts directly (but, remember, they're still read-only).

Soups

Although soups are not the most space-efficient data structure discussed here, they provide faster data access than any of the other data structures

described here except for the frame-of-frames.Soups also provide great flexibility:

- Soups can be modified
- Soups can have multiple indexes.
- Soup indexes can be added and removed dynamically.

To provide read-only soup data in your package, you can use a store part.

Store Parts

Store parts provide most of the same features and behavior as soups, with the exception that store parts are read-only. Their primary advantage over soups is they offer better compression than soups do, but store parts are still not as space- efficient as the other data structures described here.

About Stores

This section provides background information about store objects and objects that reside on stores, including packages, parts, store parts and soups.

Packages

A **package** is the basic unit of downloadable Newton software; it provides a means of loading code and data into the Newton system.

Packages can be loaded into the system from a variety of sources:

- From a Macintosh or Windows computer using Newton Connection Kit
- From a PCMCIA card
- From Newton Toolkit

Packages can be loaded either as a data stream or directly from memory. For example, Newton Connection Kit uses a data stream protocol to load a package from a Macintosh or Windows computer. However, it is much more

common to use packages directly from memory on PCMCIA RAM or ROM cards.

Parts

A package consists of one or more constituent units called **parts.** The format of a part is identified by a four-character identifier called its **type** or its **part code**.

Various subsystems of the Newton operating system are notified when a particular type of part is installed or removed. Table 11-3 lists the various kinds of parts and their associated type identifiers.

Table 11-3 Parts and type identifiers

Part	Type	Description
Application	`form`	General-purpose application created by Newton Toolkit
Book	`book`	Books created by Newton Book Maker
Auto part	`auto`	Non-visible background application
Store part	`soup`	Read-only soup
Dictionary	`dict`	Custom dictionary for Newton recognition subsystem
Font	`font`	Additional font

Some of the parts described in Table 11-3 may already be familiar to you. `Form` parts are the Newton application packages you create with Newton Toolkit. `Book` parts are the interactive digital books described in the *Newton Book Maker User's Guide*. Store parts (parts of type `soup`) are useful for the storage of read-only data and are discussed later in this chapter. Dictionary parts (parts of type `dict`) supplement the built-in word lists used by the recognition subsystem; for more information, see Chapter 10, "Recognition." `Font` parts are used to add new typefaces to Newton devices; for more information, contact Newton Developer Technical Support. `Auto` parts are described in the *Newton Toolkit User's Guide.*

Frame Parts

Except for `soup` parts, all of the parts listed in Table 11-3 are called **frame parts** because they include a `part` frame which holds the items comprising the frame part. Such items may include icons, scripts, other parts, binary data and so on. A `soup` part, on the other hand, does not have a `part` frame and is comprised of soup data only.

When a frame part is loaded, the system disperses the contents of its `part` frame to the appropriate subsystems. For example, in addition to the application itself, which is a `form` part used by the Extras drawer, the part frame in an application package might include a custom icon used by the Extras drawer, a custom dictionary used by the recognition subsystem, a soup part that provides application data, and an `InstallScript` method that performs some application-specific setup tasks.

Store Parts

A **store part** is a part that encapsulates a read-only store. Because you can build store parts into application packages, the store part is sometimes referred to as a **package store**.

Soups can reside on package stores,just as they do on “normal” stores; however, because package stores are read-only, the soups that reside on them must also be read-only. Thus, store parts can be used to provide soup-like access to read-only data residing in the application package.

For more information about the characteristics of soups, see the sections “About Soups” beginning on page 11-29, “Using Stores” beginning on page 11-58 and “Using Soups” beginning on page 11-63.

Package Compatibility

Packages written according to the 1.x Newton interfaces and guidelines can be run on 2.0 systems with no modification. 2.0 packages have the same

structure or format as 1.*x* packages, but they can now be managed directly from NewtonScript as virtual objects.

New Arguments To RemovePackage Global Function

Prior to version 2.0 of Newton system software, the `RemovePackage` function required a package frame as its argument. The `RemovePackage` global function now accepts as its argument a package frame or a package ID. The `GetPackages` global function returns a package frame; the package ID is stored in this frame's `ID` slot.

Soups

The store object that many applications interact with most frequently is the soup object. Soup objects, commonly called soups, provide random access data storage on Newton devices. The next section, "About Soups," describes their characteristics.

About Soups

This section provides important background information about soup objects. Topics discussed here include

- strategies for soup-based storage
- related data structures such as constants, soup definitions, indexes, index specifications and the alternate sort table
- guidelines for naming and managing soups
- when soups are created
- system overhead required to access soup entries
- effects of system resets on soup data
- alternatives to soup-based storage

- soup change notification
- compatibility information

Soup Strategy

Applications using soup-based data storage must

- respect the user's default store preferences for writing soup entries
- create soups only as necessary

The effort to write your own code that observes these requirements can be significant. It is also not easy to anticipate every possible scenario that might arise as the user moves your application from one Newton device to another, changes preferences or changes the stores available to your application.

The use of union soups provides the easiest and best solution to this problem. Union soups provide methods that make it easy for the application developer to respect the user's default store preferences. These ROM-based methods are also much faster than equivalent NewtonScript code.

Another good reason to use union soups is that applications almost never need to create them explicitly. Once the appropriate soup definition is registered with the system, individual soups in the union are created automatically only as needed. However, if for some reason you do need to create a union soup explicitly, you can do that as well. The "Using Soups" section, later in this chapter, describes the creation of soups in detail.

Thus, when you need to allow the user to control where data is stored, you are strongly encouraged to use union soups, for reasons of performance and convenience. Their use is simpler than managing your own single soups and faster than trying to duplicate system-provided behavior in NewtonScript code.

For those occasions when the application developer must control the store on which soup data resides, the union soup method `AddToStoreXmit` is available for your use.

Soup Definitions

A **soup definition** is a frame that provides information about a particular soup. Soup definitions simplify the creation and management of union soups. They provide a way for the system to create individual soups in the union automatically as necessary. They also allow soups to supply information about themselves for use by the system, applications and the user.

The soup definition frame specifies a name that identifies the soup to the system, a user-visible name for the soup, a symbol identifying the application that "owns" the soup, a user-visible string that describes the soup and an array of index specification frames defining the set of indexes with which the soup is created.

Once a soup definition has been registered with the system, the system and other applications can use it to create new instances of that soup on demand. For more information, see the section "Registering and Unregistering Soup Definitions" beginning on page 11-65.

The information in the soup definition frame may also be used to present information about a soup and its function to the user; for example, this information can be used to make the user aware of a particular soup's owner and function before allowing the user to delete the soup.

NewtApp applications also make use of soup definitions; for more information, see Chapter 4, "NewtApp Applications."

For a complete description of the slots in the soup definition frame, see the section "Soup Definition Frame" beginning on page 11-107.

Indexes

An **index** is a data structure that provides random access to the entries in a soup as well as a means of ordering those entries. A designated value extracted from each soup entry is stored separately in the soup's index as the index key for that entry. The system can use this index data to retrieve and sort soup entries without actually reading the entries into memory. Hence,

indexes provide a particularly fast and efficient means of manipulating soup entries.

The system provides a default index for each soup when the soup is created. This index ranks entries in roughly the order in whch they were created. (For more information, see “Queries On Single-Slot Indexes” beginning on page 11-73.) Additionally, you can create your own specialized indexes for any soup. You should create an index for each slot on which the soup will be searched frequently.

The system maintains all indexes automatically as soup entries are added, deleted or changed. Thus index data is always up-to-date and readily available, further adding to the speed and efficiency that its use provides.

Multiple-slot Indexes

Multiple-slot indexes index soup entries on the values of multiple slots. Each soup entry can be indexed on a primary key, a secondary key, and so on. Each multiple-slot index can support a total of six keys.

Index Specification Frames

The **index specification frame** or **index spec** specifies the characteristics of a soup index. The values in the index spec frame indicate the kind of index to build, which slot values to use as index data and the kind of data stored in those slots. The specific content of the index spec frame varies according to the type of index; for complete descriptions of single-slot and multiple-slot index specs, see the “Data Storage Reference” section at the end of this chapter.

Internationalized Sort Order

To better support the use of languages other than English, soup indexes and queries can be made sensitive to case and diacritical marks. This behavior is intended to allow applications to easily support the introduction of non-English data when, for example, a PCMCIA card containing data from a

different locale is inserted. To take advantage of this behavior, the application must create an internationalized index for the soup and the query must explicitly request the alternate sorting behvior in the query spec. For more information, see the sections “Using Soups” and “Using Queries, Indexes and Tags,” later in this chapter.

Tags Index

A **tag** is an optional developer-defined symbol used to mark one or more soup entries. Tags reside in a developer-specified slot that can be indexed, with the results stored in a special index called the **tags index**.

The tags index is used to select soup entries according to their associated symbolic values without reading the entries themselves into memory; for example, one could select the subset of entries tagged `'business` from the `"Names"` soup. In fact, “filing” Newton data items in “folders” is a user-interface illusion—the data really resides in soups and its display is filtered for the user according to the tags associated with each soup entry.

Note that the system allows only one tags index per soup. Each soup can contain a maximum of 624 tags. A tag may be a symbol, an array of symbols or the value `nil`. The system treats missing tags as `nil` values.

Tags Index Specification Frames

A **tags index specification frame**, or **tag spec**, defines the characteristics of a soup’s tags index. Like index specs, tag specs can be used to

- create a default tags index on a new soup.
- add a tags index to an existing soup.

Tag specs look quite similar to index specs because a tag spec is actually a special kind of index spec. For a complete description of the slots in a tag spec frame, see the section “Tags Index Specification Frame” beginning on page 11-112 in the “Data Storage Reference” section at the end of this chapter.

Storing User Preference Data in the System Soup

Most of the time you'll want to store data in union soups, but one task for which union soups are not suitable is the storage of user preferences for your application. There are several good reasons to always store the application's user preferences data on the internal store:

- If your application is on a card and is moved from one Newton device to another, it should acts the way the users of the respective Newton devices think it should. The obvious way to keep preference data with a particular device is to store it in internal RAM.
- It rarely makes sense to spread out preferences data across several PCMCIA cards.
- It's difficult to guarantee that your application would always have access to even one particular card.
- If your application is in internal memory and it simply adds preference data to the default store, the preference data could wind up on the card and become unavailable to the application when the card is ejected.

Hence, the built-in `ROM_System` soup on the internal store would seem to be the ideal place to store your application's preference data. The `GetAppPrefs` function allows you to get and set your application's preferences frame in this soup. For more information, see the description of this function in Chapter 20, "Utility Functions." For more information about the `ROM_System` soup itself, see See Chapter 18, "Built-In Applications and System Data."

Effects of System Resets On Soup Data

The microprocessor in a Newton device never shuts down completely, but instead "sleeps" when it is not in active use; thus, RAM does not normally need to be reinitialized when the Newton device is powered up. This allows the device to return to a working state quickly—virtually the same one last seen by the user— because soups remain in RAM until they are removed

explicitly, even if the machine is powered down. This working state is preserved unless a misbehaving application should corrupt RAM, or a hard reset occurs.

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** reinitializes the 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.

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 stylus 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.

Unfortunately, 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 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 Newton device is documented in its user guide.

Alternatives to Using Soups

Despite the many advantages offered by soup-based data storage, there are times when storing data in other formats may improve your application's performance or reduce its RAM requirements. For example, a large list of read-only data, like a list of states and postal codes, is best stored as single array, frame, or binary object in a slot in the application's base view or even in the application package itself. Information stored in this way is compressed along with your application package and is not brought into the NewtonScript heap when it is accessed. The primary disadvantages of this scheme are that the data set is read-only and that you cannot take advantage of the conveniences provided by soup queries.

If your application makes use of a large initial data set to which the user can make additions, you might consider a hybrid approach: keep the initial data set in your base view and use a soup only for the user's additions.

Soup Compatibility

Because 2.0 soup formats are incompatible with earlier versions of the Newton data storage model, the system implements the following soup-conversion strategy:

- When a 1.*x* data set is introduced to a 2.0 system, the system allows the user to choose read-only access or permanent conversion of the 1.*x* soup data to the 2.0 format.
- Older systems display a slip that says “This card is too new. Do you want to erase it?” when a 2.0 soup is introduced to the system.

RegCardSoups and UnRegCardSoups Methods Obsolete

The automatic creation of union soups by the system has changed with version 2.0 of Newton system software. In previous versions of the system, any soup registered by the `RegCardSoups` method was created automatically on any PCMCIA card lacking that soup, even when the user specified that new items be written by default to the internal store. The result was a proliferation of unused “empty” soups on any PCMCIA card introduced to the system.

Version 2.0 of Newton system software ensures that elements of union soups are created automatically only when they are actually needed to store data. For more information, see the “Soup Strategy” section in this chapter.

New Soup Change Notification Mechanism

Applications no longer modify the global `SoupNotify` array directly to register and unregister with the soup change notification mechanism. Instead, they use the `RegSoupChange` and `UnRegSoupChange` global functions provided for this purpose.

In addition to the new registration and unregistration functions, the soup change mechanism provides additional information about the nature of the change and allows applications to register callback functions to be executed whenever a particular soup changes.

The `BroadcastSoupChange` global function is obsolete. It is replaced by a new soup change notification mechanism that provides applications with more information about changes, more control over how and when notifications are sent and the ability to react more precisely to changes. For more details, see the section "Using Soup Change Notification" beginning on page 11-92.

Soup Information Frame

Soups created from a soup definition frame carry a default soup information frame that holds a copy of the soup definition frame. Soups created by the global function `RegisterCardSoup` have a default soup information frame that contains the slots `applications` and `itemNames`.

Soups created by the *store*`:CreateSoup` method do not contain a soup information frame.

For more information, see the descriptions of the `RegisterCardSoup` function and the `CreateSoup` method.

Null Union Soups

Under unusual circumstances a 1.x application may encounter a union soup that doesn't contain any constituent soups. A soup in this state is referred to as a **null union soup**. Queries on a null union soup fail. Attempts to add entries to a missing constituent soup also fail if a soup definition for that soup has not been registered. Null union soups should not normally occur with 1.x applications and cannot occur with applications that use the 2.0 union soup interface correctly.

Null union soups are most often found in the aftermath of a debugging session—for example, if in the Inspector you have deleted various soups to test the cases in which your application needs to create its own soups, and neglected to restore things to their normal state afterwards.

Null union soups can also occur as a result of the application soup not being created properly. Normally, when a card is ejected, the internal store constituent of a union soup is left behind or a soup definition for creating that soup is available. When this is not the case, the union soup reference to

the internal store constituent will be null when the card is ejected. If you follow the guidelines outlined in the section “Registering and Unregistering Soup Definitions,” this problem will not occur.

Null union soups can also occur when another application deletes one or more soups that your application uses. Hopefully, any application that deletes soups would at least transmit a soup notification, thereby allowing your application to deal with the change appropriately.

When your application is running on a 1.x unit or when no soup definition exists for a union soup, it is appropriate to test for the constituent soup's validity before trying to add an entry to it. Simply loop through the array of stores returned by the `GetStores` function, sending the `IsValid` message to each of the consituent soups in the union.

About Virtual Binary Objects

A **virtual binary object** or **VBO** is a special kind of binary object that does not actually reside in the NewtonScript heap as other NewtonScript objects do; instead, the system maps data from the VBO into NewtonScript heap memory as required. For this reason, a permanent store for the binary data associated with the VBO must be specified when the VBO is created.

Except for the way in which it is created, a VBO is in most ways just like any other NewtonScript binary object:

- The VBO can be inserted in or removed from a frame or soup entry.
- The VBO is not persistent until the cached soup entry that stores it is written to the soup.
- The store occupied by the VBO is made available for garbage collection when there are no more references to the VBO.
- References to VBOs residing on external stores require that you exercise the usual precautions for avoiding the “Newton still needs the card” alert.

- The data associated with a VBO is automatically compressed and decompressed by the system as it is paged in and out of the NewtonScript heap.
- Binary data—including VBO data—is not shared between soups, even if the soups reside on the same store. As a result, you may need to consider space issues when copying and duplicating entries that hold VBO data. For more information, see "Copying An Entry Into Another Soup" beginning on page 11-90.

VBOs are different from normal NewtonScript binaries in the following ways:

- You cannot use a value stored in a virtual binary object as a soup index key.

About Queries

To retrieve entries from a soup, you perform a **query** by sending the `Query` message to the soup. This section describes the various queries that can be performed on soups.

The Query Spec

A soup's `Query` method accepts as its argument a frame specifying the kind of data to test and the characteristics that entries must have to be included in the results of the search. This frame is known as a **query specification** or **query spec**.

The `Query` method tests two broad categories of data: indexes and strings. Queries based on indexes are fast, powerful and flexible; as a result, they are the preferred choice whenever feasible, and shall be the focus of this discussion. For information on other kinds of queries, see the section "Other Kinds of Queries" beginning on page 11-45.

Index Queries

An index provides random access to a set of items as well as a means of ranking them. Thus, index-based queries can search for items according to the presence of an index key and can test the value associated with the key as well. A query that tests for the presence or value of an index key is called an **index query**.

An index query can be used to obtain from a soup all entries in which a particular slot is present. For example, one could use an index query to retrieve every entry that has a `foo` slot. Conversely, entries not having a `foo` slot are not included in the set of entries referenced by the `foo` index and are never even passed to the `Query` method for testing. Thus, index queries provide an efficient way to select a subset of a soup's entries.

Index queries can be based only on slot names for which an index has been generated. For example, to query on the presence of the `foo` slot, the soup must be indexed on the `foo` slot. A soup's indexes are usually specified in the soup definition used to create it, although indexes can be added to already-existing soups as well.

Because the system maintains each soup's indexes automatically as entries are added, changed and deleted, index information is always up-to-date and readily available. Thus, index queries do not require reading soup entries into memory. Reading soup entries into memory consumes both time and RAM space; thus, the index query's ability to select a subset of entries without reading any of them into memory makes it space-efficient and fast.

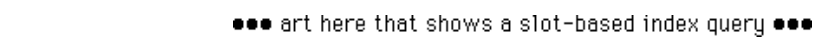
To understand index queries well, it helps to know something about indexes themselves. You can envision the contents of an index as a sequence of entries arranged in key order. Figure 11-3 depicts such an arrangement, with asterisks (*) representing individual entries in the index.

Figure 11-3 An index

****

The purpose of a query is to select some subset of the entries and generate a cursor that can move around in that subset. There are a number of ways to do this; so far, we've discussed only one, the index query, which is illustrated in Figure 11-4.

Figure 11-4 Slot-based index query



The query spec can optionally define additional qualifications that narrow this subrange further to make the query speedier. Other options allow you to define your own comparison functions for determining which entries are to be included in the results of the search.The next several sections discuss these options.

Begin Keys and End Keys

Because index keys are ranked by value, you can improve the speed of an index query significantly by narrowing the range of entries to search. One way to do this is to eliminate from the search any index key values that fall outside of specified minimum or maximum values. For example, you can specify a minimum index key value used to select the first entry to be tested, causing the query to "skip over"all lesser-valued index keys. A minimum value used in this way is defined in the query spec as a `beginKey` value.

Similarly, you can specify a maximum index key value to be used in selecting the last entry to be tested, causing the query to ignore subsequent index entries having index keys of greater value. A maximum value used in this way is defined in the query spec as an `endKey` value.

You define `beginKey` values and `endKey` values in the query spec. Both kinds of keys are typically used together to specify a subrange of valid index entries. You can also define a special kind of begin key or end key that is itself excluded from the valid subrange. Figure 11-5 depicts these special keys, which are defined as the `startExclKey` and `endExclKey` in the

query spec. The figure also depicts their relationship to start keys and end keys defined on the same index key values.

Figure 11-5 Selecting an index subrange

Tags

After sufficiently narrowing the working subrange of the query, one can further narrow the set of entries to be tested by eliminating entries according to the presence of one or more tags. A **tag** is an optional developer-defined symbol that resides in a specified slot in the soup entry.

The symbols used as tags are stored as the key values in the soup's **tags index**. As with any other index, the system maintains the tags index automatically and queries can test values in this index without reading soup entries into memory. Thus, tag-based queries are quick and efficient.

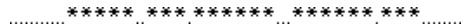
Unlike other indexes, the tags index alone cannot be used as the basis of an index query. However, you need not specify an additional index in order to query on tag values because every soup carries a default index that includes all entries in the soup. In lieu of a separate index specification, tags queries use this default index to sort and retrieve the subrange of entries to filter on tag values.

The tags for which the query tests are specified by a tags specification frame supplied as part of the original query spec. Tags can be used as the entire basis of a query; for example, one could query for all the entries that have the tag `'blue` or all the entries that don't have the tag `'blue`.

Tags can also be used to filter a range of entries specified by an index query. For example, Figure 11-6 depicts a set of soup entries that is the result of an index query on the `timeStamp` slot, with a subrange of entries defined by a

`startExclKey` and an `endExclKey`. That subrange is narrowed further by filtering on the presence of the `'business` tag.

Figure 11-6 Filtering on tag values



The example above is only one of several tests that may be conducted on tags. The `tagQuerySpec` can specify set operators such as `not`, `any`, `equal`, and `all`; these operators are described in greater detail in the section "Tag Specifications for Queries" beginning on page 11-115.

Customized Tests

The use of indexes, begin keys, end keys and tags provide sufficient control over query results for many uses; however, the developer can specify additional customized tests when necessary. These tests take the form of an `indexValidTest` function that the application developer defines in the query spec.

The `Query` method passes to the `indexValidTest` function the index values associated with each entry in the working subrange that the query tests. The `indexValidTest` function must return `nil` for an entry that is to be rejected. This function returns any value that is not `nil` for an entry that is to be included in the results of the search.

Another kind of customized test, the `validTest` function, works like the `indexValidTest` function but actually reads the soup entry into memory in order to perform some test on the entry data. Thus, for performance reasons, `validTest` functions should be used only when absolutely necessary. It is strongly suggested that you use index-based approaches to limit the working subrange of entries passed to the `validTest` function.

Other Kinds of Queries

Queries can also test values other than those described in the previous section. This section describes multiple-slot index queries, string queries, and the use of internationalized sorting tables for ranking the results of a query. A single query spec can combine any of the query types; for example, a single query can test `beginKey` and `endKey` values, tag values, index values and string values.

Multiple-slot Index Queries

A **multiple-slot index query** uses multiple index keys to search for soup entries. A multiple-slot index query can be performed only on a soup that has a multiple-slot index.

Words Queries

A **words query** tests all strings in each soup entry for a word beginning or for an entire word.

The default behavior for a words query is to test for word beginnings. For example, a words query on the string `"smith"` would find the words `"smithe"` and `"smithereens"`. The word `"blacksmith"` would not be included in the results of the search because the string `"smith"` is not at a word beginning. (A text query, described in the next section, tests for strings not at word beginnings.) Because words queries are not case-sensitive, the word `"Smithsonian"` would also be found by this query.

If you specify that the words query only match entire words, it would return only entries containing the entire word `"smith"` and would not return any of the variations described previously.

A words query is slower than a similar index query because it takes some time to test all of the string values in a soup entry.

You might think that words queries require more space than index queries because they must read soup entries into memory. Because words queries

make use of VBOs they do not require significantly more heap space than other kinds of queries.

Text Queries

A **text query** is similar to a words query but its test is not limited to word boundaries; that is, it tests all strings in each soup entry for one or more specified strings, regardless of where they appear in the word. For example, a words query on the string `"smith"` would find the words `"smithe"` and `"smithereens"` as well as the word `"blacksmith"`. Because text queries are not case-sensitive, the words `"blackSmith"` and `"Smithsonian"` would also be found by this query.

A text query is slower than its words query counterpart.

You might think that text queries require more space than index queries because they must read soup entries into memory. Because text queries make use of VBOs they do not require significantly more heap space than other kinds of queries.

Accessing Search Results

All queries return search results as a **cursor** object. The cursor is an object that iterates over the set of entries meeting the criteria defined by the query spec. To retrieve an entry, you simply send messages to the cursor object.

For more information about cursors, see the section “About Cursors” beginning on page 11-48.

Query Compatibility Information

Version 2.0 of Newton system software provides a more powerful query mechanism while at the same time simplifying the syntax of queries. Although old-style query syntax is still supported, you’ll probably want to revise your application code to take advantage of the features that new-style queries provide. This section explores query compatibility issues in more detail.

Query Global Function Is Obsolete

Queries are now performed by the `Query` soup method; the `Query` global function still exists for compatibility purposes. The `Query` method accepts as its argument a frame describing the criteria for the inclusion of soup entries in the cursor returned by the query. This frame is called a **query specification** or **query spec**. For a complete description of this frame and its slots, see the section "Query Specification Frame" beginning on page 11-113. For examples of the use of the `Query` method, see "Using Queries, Indexes and Tags" beginning on page 11-72.

Query Types Merged

Query specs no longer require a `type` slot; if one is present it is ignored. The `Query` method determines the query's type according to the slots present in the query spec and the types of data on which the soup has been indexed.

StartKey and EndTest Obsolete

Because the order of the entries in the cursor is determined entirely by the index values, specifying key values is sufficient to determine a range. Hence, the use of an end test is unnecessary in a version 2.0 query spec.

The end test may sometimes be used for other purposes, such as stopping the cursor after the visible portion of a list has been filled; however, this sort of test is best performed outside the cursor to optimize performance. The caller of the cursor's `Next` method should be able to determine when to stop retrieving soup entries without resorting to the use of an end test.

When a cursor is generated initially and when it is reset, it references the entry having the lowest index value in the set of entries in the selected subset. Thus, it is also usually unnecessary to use a start key, although this operation still works as in earlier versions of system software. For those occasions when it is necessary to start the cursor somewhere in the middle of the range, the use of a start key can be simulated easily by invoking the cursor's `GotoKey` method immediately after generating or resetting the cursor.

Queries on Nil-Value Slots

In Newton system software prior to version 1.05, storing a value of `nil` in the indexed slot of an entry returns `nil` to the query for that entry; that is, the query fails to find the entry. To work around this problem in older Newton systems, make sure your indexed slots store appropriate values.

Queries in system software version 2.0 ignore slots having a `nil` value, as if they are not indexed.

In Newton system software prior to version 2.0, indexed slots containing a type of data other than that specified by the index caused queries to return an “index inconsistency” error; for example, if a slot that should store text holds an integer value would cause this kind of query error. Version 2.0 queries throw an exception in this situation.

Heap Space Requirements of Words and Text Queries

On systems prior to version 2.0, words and text queries generally require more RAM space than index queries, because each entry to be tested must first be read into RAM. System software version 2.0 uses VBOs to significantly reduces the RAM requirements of words and text queries.

About Cursors

The `Query` function returns a **cursor,** which is an object that iterates over the set of entries satisfying the query spec and returns entries in response to the messages it receives. As entries in the soup are added, deleted and changed, the set of entries to which the cursor points is updated dynamically.

The cursor ranks the set of entries it returns according to the value of the index key used in the original query; in other words, cursors return entries in index order. If no other index is specified when the soup is created, its entries are indexed on an internally-assigned value. Because these values are assigned in incremental order as the entries are created, the result is that the default index sorts entries in roughly the order that they were created.

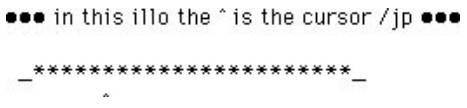
However, this ordering is not guaranteed because the system recycles these internal identifier values as necessary.

One of the most interesting and useful aspects of cursor objects is that they update themselves dynamically as the set of soup entries they reference changes. That is, as soup entries meeting the original query spec are added, deleted and modified, the set of entries returned by the cursor reflects these changes, even after the original query has been performed.

The cursor's dynamic updating behavior can be a programmer's pitfall as well: unless the query spec used to generate the cursor specifies an `endKey` value, the cursor's `validTest` method may be run any time the current entry is modified or deleted.

Recall that after selecting a subrange of all entries in the soup, our example query used various tests to eliminate certain entries within that range. If viewed within the context of the entire soup index, the final set of valid entries would include gaps occupied by entries that were eliminated, similar to the depiction in Figure 11-6. As seen from the cursor, however, this subset looks like the continuous range depicted in Figure 11-7.

Figure 11-7 Selected subset as "seen" by the cursor



Initially, the cursor points to the first soup entry in the data set that satisfies the query. The cursor supplies methods that allow you to determine its current position, retrieve the entry referenced by its current position or specify a new position. The cursor may be moved left and right, moved to a specifc entry or key position and reset to an initial position that is not necessarily the first entry in the valid set.

Note that the cursor may occupy "virtual" positions before the beginning and after the end of the set of valid entries. These positions are "virtual"

because they appear to occupy positions contiguous with either end of the valid set of entries even though these positions do not actually exist in the index referenced by the cursor.

About Entries

An **entry** is a special kind of frame that resides in a soup. You cannot create a valid entry by simply creating a frame having certain slots and values—the methods *soup*`:AddXmit` and *unionSoup*`:AddToDefaultStoreXmit` are used to create new entries. The entry that these methods create consists of the frame presented as the argument to the method and the transitive closure of that frame's pointers (that is, everything the frame points to, everything those data structures point to, and so on).

All frames are automatically compressed when they are stored as soup entries and soup entries are decompressed when they are referenced. The automatic compression and decompression of soup data reduces the amount of storage space and run-time memory required by Newton applications.

Circular references within the entry are supported, as are references to symbols. The entry-creation methods store the new entry in the soup that received the entry-creation message.

The only way to retrieve a valid entry from its soup is by performing a query on the soup. The query returns a cursor, which returns entries in response to messages it receives.

As first returned by the cursor, the entry is not a frame, but compressed soup data. It is not decompressed until it is referenced by getting or setting the value of one of its slots. At this time, the system constructs in RAM the entire entry frame and the transitive closure of its pointers (that is, everything the entry frame points to, everything its data structures point to, and so on). This scheme reduces both the time necessary to perform a query and the amount of storage required by a cursor object.

Decompressed entries are cached in RAM until they are written back to the soup. Applications can modify these cached entry frames directly. The

system supplies functions for modifying entries, writing them back to the soup and manipulating them in other ways.

For more information, see the section “Using Entries” beginning on page 11-87.

About Entry Aliases

An entry alias is an object that provides a standard way to save a reference to a soup entry. Entry aliases themselves may be saved in soups.

Entry aliases are also useful for providing easy access to items in dissimilar soups. For example, the built-in Find service uses entry aliases to present entries from multiple soups in a single overview view.

You must not assume that an entry alias is valid. When the entry to which an alias refers is deleted or is moved to another store, the alias becomes invalid.

Aliases can be created for any entry that resides in a soup or union soup. Aliases cannot be created for mock entry objects residing in mock soups.

About Mock Entries

A **mock entry** is an object that has many of the properties of a soup entry. Mock entries provide a foundation object that developers can use to build up a suite of objects that act like the built-in system of store, soup, cursor and entry objects. The current implementation provides only mock entries; the developer must implement mock cursors, mock soups and mock stores as appropriate for the application.

The mock entry’s methods are implemented by a NewtonScript object known as its **handler**, which is a frame that implements methods corresponding to those that the system supplies for normal soup entries. These methods can be implemented in the handler frame itself or made available to the handler through parent or prototype inheritance. The handler also holds information local to a specific entry or enough information to get at the real soup entry data.

Like a normal soup entry, the mock entry caches its data in RAM when the entry is accessed; thus, the data associated with a mock entry is called its **cached object**. As with normal soup entries, the cached data is seen by other NewtonScript objects as the mock entry itself. If an entry's corresponding cached object is present, entry accesses are transparently forwarded by the system to the cached object. If no cached object is present for the entry, the entry's handler is asked to generate one before the access is forwarded.

As you might expect, the mock entry resides in a mock soup, which , in turn, resides on a mock store. The **mock store** is a developer-supplied frame that responds appropriately to all the messages that might normally be sent to a store object. For example, when the *mockStore*`:GetSoup(`*soupName*`)` method is invoked, it must return a mock soup.

The **mock soup** is just a frame, again implemented by the application developer, that responds appropriately to all the messages that might normally be sent to a real soup object. For example, when the mock soup's `Query` method is called, the mock soup must return a mock cursor.

Like the other mock objects, a **mock cursor** is just a frame that can respond appropriately to all the normal cursor messages; it is implemented by the application developer. When the mock cursor's `Entry` method is invoked, it must return a mock entry. Mock entries must look to the system like any other frame, so the usual `_proto` and `_parent` inheritance cannot be used.

Using Newton Data Storage Objects

This section provides a code-level overview of the interaction of the most common Newton data storage objects and methods. It presumes knowledge of the material in preceding sections and roughly parallels the section "Working With Storage Objects" beginning on page 11-10.

Most applications store data as frames that reside in soup entries. To create a frame, simply define it and store it in a slot, variable or constant. The following code fragment defines a frame containing the `aSlot` and

anotherSlot slots. The frame itself is stored in the myFrame local variable. For all practical purposes you can treat variables that hold NewtonScript objects as the objects themselves; hence, the following discussion refers to the frame stored in the myFrame variable as the myFrame frame.

```
local myFrame := {aSlot: "some string data", anotherSlot: 'aSymbol};
```

The myFrame frame contains two slots: the aSlot slot stores the "some string data" string and the anotherSlot slot stores the 'aSymbol symbol. (Note the single quotation mark used in all symbols.)

Frames are not persistent unless stored as soup entries. To add the myFrame frame to a soup, you must send a message to the appropriate soup object. You can obtain a soup object by creating a new one or by retrieving a refrence to an existing soup. (This discussion provides examples of both operations.) Some applications use more than one soup; this discussion will limit itself to the use of a single soup for the sake of simplicity.

To create a new union soup, use the RegUnionSoup function to register its soup definition with the system. The system uses this definition to create the union's constituent soups as needed to store soup entries.

The following code fragment passes the kMySoupDef frame to the RegUnionSoup function, which registers this soup definition and returns a union soup object. The union soup object that this function returns is stored in the myUSoup local variable. The rest of this discussion refers to this object as the myUSoup object.

```
// kAppSymbol is defined by NTK - see NTK User Guide
// assume kMySoupDef is a valid soup definition
// see Data Storage Reference section for example
// register kMySoupDef soup definition
   local myUsoup := RegUnionSoup(kAppSymbol,kMySoupDef);
```

Because the system does not create any soup until it is needed to store an entry, the constituent soups of myUSoup may not exist if an entry has never been added to this soup. As a result, messages other than AddToStoreXmit or AddToDefaultStoreXmit may fail when sent to the myUSoup object.

The AddToStoreXmit and AddToDefaultStoreXmit methods are safe to send to the myUSoup object because these methods create consituent soups on the appropriate stores when necessary to store a soup entry.

Soup validity is not really as great a problem as it might seem at first. As long as you use the AddToStoreXmit and AddToDefaultStore methods properly, you should have few problems adding entries to union soups. Note that you can also use the IsValid soup method to determine whether a soup resides in valid memory (an example is provided later in this section.) When necessary, you can use the GetMember method to force the creation of a specified soup in the union.

The next code fragment uses the AddToDefaultStoreXmit function to add the myFrame frame to the myUSoup union soup. Because no entries have been stored in this soup yet, this function creates a new soup to hold the entry. The soup is created on the store indicated by the user preference specifying where new items are kept.

```
// add the entry and transmit notification
// soup is created if it doesnt exist on default store
   myUSoup:AddToDefaultStoreXmit(myFrame, kAppSymbol);
```

Note that at this point we have created a soup on a store specified by the user and added an entry to that soup without ever manipulating the store directly.

Because you'll often need to notify other applications—or even your own application—when you make changes to soups, all of the methods that modify soups or soup entries are capable of broadcasting an appropriate soup change notification message automatically. In the preceding example, the AddToDefaultStoreXmit method notifies applications registered for changes to the myUSoup soup that the kAppSymbol application made an entryAdded change to this soup.

Most of the time your application will need to work with existing soups as opposed to creating them afresh. You can use the GetUnionSoupAlways function to get a reference to an existing soup, as the following code fragment shows.

```
local myUSoup := GetUnionSoupAlways("My Union Soup");
```

The `GetUnionSoupAlways` function retrieves the union soup named `"My Union Soup"`. Do not assume that this object necessarily represents an existing soup—the soup may not yet have been created or the PCMCIA card on which it resides may have been removed. You can use the `IsValid` function to determine whether the object references a soup in valid memory, as the following code fragment illustrates.

```
local myUSoup:= GetUnionSoupAlways("My Union Soup");
if IsValid(myUSoup) then begin
      local myCursor := anotherUSoup:Query(myQuerySpec);
   end
```

Once you have a valid soup object, you can send the `Query` message to it to retrieve soup entries. The `Query` method returns a cursor object that iterates over the set of soup entries satisfying the query specification passed as the argument to the `Query` method. You can send messages to the cursor to manipulate its position and to retrieve specified entries, as shown in the following example.

```
// move the cursor two positions ahead in index order
myCursor:Move(2);
// retrieve the entry at the cursor's current position
local myEntry := myCursor:Entry()
```

For the purposes of discussion, let's assume that the cursor returned the entry holding the `myFrame` frame. To access this frame, simply treat the entry as if it were the original frame and use the NewtonScript dot operator (`.`) to dereference any of its slots. As soon as any slot in the entry is referenced, the system reads the entire entry into a cache in memory and sets the `myEntry` variable to reference the cache, rather than the soup entry. This is important to understand for two reasons: First, referencing a single slot in an entry costs you time and memory space, even if you only examine or print the slot's value without modifying it. Second, when you change the value of a slot in the entry, you really change the cached entry frame, not the

original soup entry; changes to the real soup entry are not persistent until the cached entry frame is written back to soup, taking the place of the original entry.

Recall that the `aSlot` slot in the original `myFrame` frame stored a string; the following code fragment retrieves this string by using the dot operator to dereference the `aSlot` slot in `myEntry` soup entry. You can set a new value for this slot by storing it directly in the `aSlot` slot, as shown in the following code fragment.

```
myEntry.aSlot := "new and improved string data";
```

The dereferenced entry frame is cached in RAM at this point. The cache must be written back to the soup entry for the changes to persist. You can use the `EntryChangeXmit` function to write the cached entry frame back to the soup, as in the following example.

```
EntryChangeXmit(myEntry, kAppSymbol);
```

As with all the auto-transmit functions, the `EntryChangeXmit` function transmits an appropriate soup-change notification message after writing the entry back to the soup; in this case, the notification specifies that the `kAppSymbol` application made an `entryChanged` change to the `myUSoup` soup. (All entries store a reference to the soup in which they reside, which is how the `EntryChangeXmit` method determines which soup changed.)

You can use the `EntryUndoChangesXmit` function to undo the changes to the soup entry. This function throws away the contents of the entry cache and assigns a reference to the original soup entry to the `myEntry` variable. Note that subsequently referencing a slot in this entry will again cause the entire entry to be read into memory.

Most applications unregister their soup definitions when they close; use the `UnRegUnionSoup` function to do so, as shown in the following example. Because a single application can create multiple soups and soup definitions, soup definitions are unregistered by name and application symbol.

```
UnRegUnionSoup(kMySoupDef.name,kAppSymbol);
```

Using Memory

This section describes techniques for reducing your application's RAM requirements and making efficient use of available RAM.

Releasing Unused References

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 the application closes, it needs to clean up after itself as much as possible, removing its references to soups, entries or cursors. Again, this usually means setting to `nil` any references to these objects.

IMPORTANT

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

Avoiding Unnecessary Caching

Reading a soup entry into memory consumes more RAM than simply testing its tag or index values. Whenever possible, work with index keys and tags rather than the contents of soup entries. Some suggested techniques for doing so include the following.

- Avoid using `validTest` functions in favor of using `indexValidTest` functions in your queries, as the latter can be performed without reading soup entries into memory.
- Query on index key values or tag values rather than on values that require reading soup entries into RAM, such as strings.

- Use the cursor method `EntryKey` to retrieve an entry's key value without reading it into RAM.

Reclaiming Cache Memory

Whenever you dereference a slot in a soup entry you read that entry into memory if it is not already present. That is, simply testing or printing the value of a single slot causes the entire soup entry in which it resides to be read into the entry cache. Thus, applications must reference soup entries only when necessary and flush the entry cache as soon as is appropriate. The `EntryChangeXmit` and `EntryUndoChangesXmit` functions both flush the entry cache.

If you intend to actually modify the value of the slot you've dereferenced, call the `EntryChangeXmit` function to write the new value back to the soup and flush the entry cache. On the other hand, if you are just testing or printing slot values without modifying them, you can use the `EntryUndoChangesXmit` method to flush the cache without writing the entry back to its soup. The `EntryUndoChangesXmit` function flushes the cache and restores references to the original uncompressed soup entry.

Using Stores

Because the system manages stores automatically, most NewtonScript applications' direct interaction with store objects is limited to manipulating pre-existing soup objects that reside on them. (The `RegUnionSoup` function returns references to new union soups without requiring that the application interact with any stores directly.) In order to retrieve pre-existing soups, the application must message the store object on which they reside. The "Referencing Stores" section, which follows immediately, describes how to get store objects from the system.

Unless you are creating a specialized backup / restore application, the "Referencing Stores" section is the only part of the "Using Stores"section that

you must read. The store methods described in subsequent sections are intended for use by special backup / restore applications.

This section describes the use of system-supplied functions and methods for

- getting references to store objects from the system
- accessing information about stores and their contents
- reading and writing store data
- loading packages procedurally

Referencing Stores

The `GetStores` global function returns an array of references to all currently available stores. You can send the messages described in this section to any of the store objects in this array.

```
local allStores := GetStores();
```

Note

Do not modify the array that the `GetStores` function returns. ◆

You can reference individual stores in the array by appending an array index value to the `GetStores` call, as in the following code example.

```
local internalStore := GetStores()[0];
```

The first element of the array returned by the `GetStores` function is always the internal store; however, the ordering of subsequent elements in this array cannot be relied upon, as it may vary in future Newton devices.

IMPORTANT

Don't call the `GetStores` function from your application's `RemoveScript` method, or you may find yourself looking at the "Newton needs the card…" slip. You can avoid this situation by using the `IsValid` store method to test the validity of a store object before sending messages to it. ▲

Retrieving Objects From Stores

As you know, the objects that may reside on a store include packages, store parts, soups and virtual binary objects.

The `GetPackages` global function returns an array of packages currently available to the system; this array contains packages that reside on any currently-available store.

To determine the store on which a specified package resides, test the value of the store slot in package reference information frame associated with the package. This frame is returned by the `GetPkgRefInfo` function.

Procedures for retrieving other objects are described in those objects' respective "Using" sections; for more information , see the sections "Using Soups," "Using Virtual Binary Objects," "Using Store Parts" and "Using Mock Entries and Mock Soups," as appropriate.

Testing Stores For Write-Protection

The store methods `CheckWriteProtect` and `IsReadOnly` determine whether a store is write-protected. The former throws an exception when it is passed a reference to a write-protected store, while the latter simply returns the value `nil` for such stores.

Getting or Setting the Default Store

The default store is that store designated by the user as the one on which new data items are created. The system-supplied functions that accept union-soup arguments handle the details of storing and retrieving soup data according to the preferences specified by the user. Thus, applications using union soups normally do not need to be concened with getting or setting references to the default store.

If for some reason you need to get or set the default store yourself, you can utilize the `GetDefaultStore` and `SetDefaultStore` global functions.

Note
Do not change the default store without first notifying the user. ◆

Loading Packages Procedurally

The system provides two methods for loading packages procedurally.

The store method `SuckPackageFromEndpoint` allows you to load a package from the network onto a card or into RAM.

You can also create packages from binary data by using the store method `SuckPackageFromBinary` to stream in package data from any kind of PCMCIA card.

For more information, see the descriptions of these functions in the section “Data Storage Reference” beginning on page 11-106.

Getting and Setting Store Information

This section describes how to get information associated with the store itself, such as its name, kind, signature and an optional information frame. This section also describes how to retrieve packages residing on the store.

Note
Most NewtonScript applications never need to use the functions and methods described here, which are intended for use only by specialized backup and restore applications. ◆

Getting and Setting the Store Signature

Each store is identified to the system by its **signature**, which is a value assigned to the store when it is created. Note that most applications do not need to manipulate a store’s signature. Only special-purpose backup and restoration applications need to get or set a store’s signature.

To get a store's signature, send the `GetSignature` message to the store object. The following code fragment gets the internal store's signature.

```
local theSig := GetStores()[0]:GetSignature();
```

The default signature assigned to a store is a random integer generated by the system; however, you can use the `SetSignature` method to assign a specified signature to a particular store.

IMPORTANT

Do not alter a store's signature unless you intend to completely replace all data on the store. The *store*`:SetSignature()` method is a special-purpose method intended for use only by backup / restore applications. ▲

To set a store's signature, send the `SetSignature` message to the store object. Before attempting to set the store's signature or write any other data to it, you can use the store methods `IsReadOnly` or `CheckWriteProtect` to determine whether the store can be written.

Getting and Setting the Store Name

Associated with each store is a string that identifies the store in slips seen by the user. The default name for the internal store is "Internal" and a PCMCIA store is named "Card" by default. The store methods `GetName` and `SetName` are used to get and set the store's name.

The following example uses the `GetName` method to obtain a string that is the name of the internal store.

```
//returns the string "Internal"
GetStores()[0]:GetName();
```

The special-purpose `SetName` method sets the name of a store as specified; note that this method is intended for use only by backup / restore applications.

IMPORTANT

It's generally unwise to change the name of a store; there is no way to determine whether other applications are using its current name to refer to it. ▲

The following code example sets the name of the internal store to `"dontTryThisAtHome"`.

```
//sets internal store's name to "dontTryThisAtHome"
getStores()[0]:SetName("dontTryThisAtHome");
```

Accessing the Store Information Frame

Each store may contain an optional information frame that applications can use to hold information associated with the store itself. Note that unless an application stores data in it, the information frame may not exist on every store.

The `GetInfo` store method retrieves the value of a specified slot in the store information frame. Its corollary, the `SetInfo` store method, writes the value of a specified slot in this frame. These methods are intended for use by backup / restore applications only; most applications need not use them at all.

Using Soups

This section discusses the functions and methods used to manipulate soup objects. Individual entries in soups and union soups are manipulated with queries, cursors and entry functions, as described in subsequent sections of this chapter. This section describes procedures for

- creating soups and indexes
- indexing existing soups
- reading soup data from the store
- writing soup data to the store

- accessing information about the soup itself and the store on which it resides
- removing soups

Using Soup Definitions

Soup definitions encapsulate the information required to create a soup. These definitions are made available to the system by the `RegUnionSoup` global function. Once a soup definition is registered, union soup methods require only the soup's name to create soups having a specified default set of indexes and tags. Applications can use this mechanism to let the user specify the store on which soups are created at run time.

The next several sections describe how to use soup definitions. The "Naming Soups" section, immediately following, describes naming conventions to use when creating soup definitions. The following section, "Registering and Unregistering Soup Definitions" beginning on page 11-65, describes how to make soup definitions available to the methods that create soups automatically. The next section, "Creating and Accessing Union Soups," describes the use of union soup methods that create soups automatically.

For complete descriptions of the contents of various soup definition frames, as well as examples of soup definitions that specify single-slot, multiple-slot and tags indexes, see the section "Data Structures" beginning on page 11-107.

Naming Soups

When creating soups, you need to follow certain naming conventions in order to avoid name collisions with other applications' soups. Following these conventions also makes your own soups more easily identifiable.

If your application creates only one soup, you can use your package name for the soup name. Your package name is created by using a colon (`:`) to concatenate your package's Extras drawer name with your unique developer signature. For example, if your developer signature is `myCompany` and you are creating a package that appears in the Extras drawer with the

name `foo`, concatenating these two values produces the `foo:myCompany` package name.

If your application creates multiple soups, use your package name as a suffix to a descriptive soup name. For example, `soup1:foo:myCompany` and `soup2:foo:myCompany` would be acceptable soup names unlikely to collide with those used by other applications.

For more information, see the section "Developer Signature Guidelines" beginning on page 2-8.

Registering and Unregistering Soup Definitions

The `RegUnionSoup` global function registers a soup definition with the system and returns a union soup object to which the application can send messages. Once the soup definition is registered, various methods create soups in the union as needed. If you attempt to register a soup definition that is already registered, the `RegUnionSoup` function does not replace the currently-registered soup definition.

A corollary function, `UnRegUnionSoup`, unregisters the soup definition with the system.

Using the RegUnionSoup Function

You can register a soup definition with the system any time before your application needs to access the soup it defines. If your application is the only one using your soup, you need only ensure that its definition is registered while the application is actually open.

You can use the `RegisterCardSoup` function by placing code like the following example in the `viewSetupFormScript` method of your application's base view.

```
// example assumes kAppSymbol and kMySoupDef are valid
// register kMySoupDef soup definition
   local myUsoup := RegUnionSoup(kAppSymbol,kMySoupDef);
// a frame to add to the soup
```

```
   local myFrm := {text:"Insert clever aphorism here"}
// add the entry and transmit notification
// soup is created if it doesnt exist on default store
   myUSoup:AddToDefaultStoreXmit(myFrame, kAppSymbol);
```

Note that the soups comprising the union are not created until actually needed to store an entry. Nonetheless, you can send messages to the object that the `RegUnionSoup` function returns, as shown in the last line of the code example.

The `AddToDefaultStoreXmit` method adds the specified frame to the union soup, creating a single soup on the default store if a constituent of the union is not already present. After creating the new soup entry, this method also transmits a soup change notification message. Note that passing `nil` as the last argument to this method causes no change notification message to be sent. For more information about change notification, see “Soup Change Notification” beginning on page 11-9; also see the description of this method in the section “Soup Functions and Methods” beginning on page 11-135.

Methods that create soup entries—such as the `AddXmit`, `AddToStoreXmit`, and `AddToDefaultStoreXmit` methods—destructively modify the frame presented as their argument to transform it into a soup entry. Thus, if your application defines a constant frame for use as a default soup entry, you need to pass its clone to these methods to avoid error. Similarly, if the original frame must remain unmodified, pass its clone to these methods.

Using the UnRegUnionSoup Function

You can unregister a soup definition anytime you no longer need to create the soup it defines. If your application is the only one that uses your soup, you need only ensure that its definition is registered while the application is actually open. If other applications use your soup, you may wish to leave its definition registered even after your application is closed or removed; however, most applications unregister their soup definitions at one of these times, if only to make that much more RAM available to other applications.

The following code fragment illustrates the use of the `UnRegUnionSoup` function.

```
// uregister the soup def
   UnRegUnionSoup (kMySoupDef.Name, kAppSymbol);
// don't forget to set all unused references to nil
   myUsoup := nil;
```

Creating and Accessing Union Soups

This section describes functions and methods that create or retrieve union soup objects from the system. You can send messages to these objects to manipulate soup entries, create additional soup indexes or get and set information pertaining to the union soups themselves.

Note that the soup-creation functions and methods use previously-registered soup definitions to create the union's constituent soups as necessary. This feature frees the application developer from most concerns about whether the union soup's constituents are present on a particular store.

Most applications need not use any soup-creation functions other than the `RegUnionSoup` function. The object that this function returns can be treated as a valid union soup, regardless of whether its constituent soups are present. Your application simply saves this object and sends any appropriate union soup messages to it. For an example of the use of the `RegUnionSoup` function, see "Using the UnRegUnionSoup Function" beginning on page 11-66.

The `GetUnionSoupAlways` global function returns the specified union soup. Note that the constituent soups of this union may not necessarily exist. When an appropriate soup definition is registered with the system, the `AddToStore`, `AddToStoreXmit`, `AddToDefaultStore` and `AddToDefaultStoreXmit` methods create constituent soups on the appropriate store as necessary.

Use of the `GetUnionSoupAlways` function is straightforward, as the following example shows.

```
// "Gazpacho" soup does not exist
// on any currently-available store
local theUnionSoup := GetUnionSoupAlways("Gazpacho");
```

```
// “Gazpacho” soup is created on default store
// when it is actually needed to store an entry
theUnionSoup:AddToDefaultStoreXmit(myFrame, kAppSymbol);
```

The `GetMember` union soup method provides another way to create a union soup on a specified store. This method returns the union soup’s constituent from the specified store, creating the constituent soup from its soup definition if it is not already present.

Adding Entries to a Specified Constituent Soup

When you need to control the store on which soup data is saved, use the `AddToStoreXmit` union soup method. This method works like the `AddToDefaultStoreXmit` method but allows you to specify the store to which the entry is added. Like `AddToDefaultStoreXmit`, this method returns the new soup entry.

```
// the frame to add to the soup
local frameToAdd := {Boston : true, Manhattan :nil}
// get reference to internal store
local intStore := GetStores()[0];
// assume "Chowder" soup is not on internal store
local e:= chowder:AddToStoreXmit(frameToAdd, intStore, kAppSymbol);
```

As with all the auto-transmit methods, passing `nil` as the last argument to the `AddToStoreXmit` method causes no change notification message to be sent. For more information about change notification, see “Soup Change Notification” beginning on page 11-9; also see the description of this method in the section “Soup Functions and Methods” beginning on page 11-135.

Adding an Index to an Existing Soup

Normally, applications create an index for each slot on which a soup will be searched frequently. Although the soup's indexes are usually created along with the soup itself, you may occasionally need to use the `AddIndexXmit` method to add indexes to already-existing soups. Indexes must be added individually—you can't pass arrays of index specs to the `AddIndexXmit` method. Note that sending the `AddIndexXmit` message to a union soup adds the specified index to all soups in the union.

The following code fragment adds an index to the `"My Soup"` soup, allowing queries to search on strings in that soup's `Postal_code` slot.

```
// get "My Soup" soup
local mySoup:= GetUnionSoupAlways("My Soup");
// force creation of constituent soup on internal store
mySoup:GetMember(GetStores()[0]);
// add a new single-slot index on the'Postal_code slot
local myIndexSpec := {structure:'slot,
                      path:'Postal_code,type:'string}
local namesSoup:AddIndexXmit(myIndexSpec, nil);
```

IMPORTANT ▲

Only one tags index is allowed per soup; if you add a tags index to a soup that already has one, you will replace the original tags index. For more information, see the description of the `AddIndexXmit` method on page 11-141. ▲

Adding Tags to an Existing Soup

You can add tags only to a soup that has a tags index. Adding a tags index to an existing soup is like adding any other kind of index: simply pass the appropriate index spec to the soup's `AddIndex` method. Remember,

however, that the system allows only one tags index per soup. If you try to add another tags index to that soup, you'll replace the original tags index. It's quite easy to add new tags to a soup that already has a tags index, so you'll rarely need to actually replace a soup's tags index.

To add new tags to a soup that already has a tags index, simply add to the soup an entry that uses the new tags—the tags index is updated automatically to reflect the change.

Invoking Internationalized Index Order

Indexes are not normally sensitive to case, diacritical marks or ligatures. To make sorting sensitive to these characteristics, the index specification frames passed as input to a store object's soup-creation methods or a soup object's `AddIndex` method may include an optional `sortId` slot to specify the use of a ROM-based alternate sort table that respects letter case and diacritical marks. When internationalized sorting is invoked, lowercase letters sort first, followed by diacritical marks, ligatures and capital letters. Thus, the letter `a` sorts before the letter `á`, which sorts before the letter `A`, which sorts before the letter Á, which sorts before the ligature æ.

If the index spec frame does not include a `sortId` slot, the default sort table is used, causing case and diacritics to be ignored. For example, when the default sort table is used, the values `apple` and `Apple` may sort differently on different occasions. When the index spec frame includes a sortID slot that holds the value 1, the index built for that soup supports internationalized ordering.

If a soup has internationalized index ordering, the query spec passed to its `Query` method can also include a `sortID` slot. When the value of this slot is `1`, the query uses the internationalized index.

For more information about internationalized sorting, see the section "Internationalized Sort Order" beginning on page 11-32. For a description of the `sortID` slot, see the descriptions of index spec frames on pages 11-109 through 11-111.

Removing Soups

When the user scrubs your application's icon in the Extras drawer, the system sends a `DeletionScript` message to your application. The `DeletionScript` method is an optional method accepting no arguments that you supply. You can remove your application's soups from within this method by calling the `RemoveFromStoreXmit` soup method. This method is defined only for single soups—you must remove each constituent of a union soup separately.

For more information on the `DeletionScript` method, see the *Newton Toolkit User's Guide*. For more information on the `RemoveFromStoreXmit` method, see its description on page 11-146.

Do not delete soups from within your application's `viewQuitScript` method— the user data needs to be preserved until the next time the application is run, unless the application is being removed permanently. For similar reasons, do not remove soups from within your applications RemoveScript method. This method does not distinguish between removing software permanently (scrubbing its icon in the Extras drawer) and removing software temporarily (ejecting the PCMCIA card.)

Sharing Soups With Other Applications

The shared data model on which Newton soups are based relies on the application developer's following certain conventions to avoid name-space collisions. For more information, see "Naming Soups" beginning on page 11-64.

Newton applications are encouraged to reuse existing system-supplied soups for their own needs and to share their own soups with other applications. Refer to Chapter 18, "Built-In Applications and System Data," to see descriptions of the soups used by the applications built into the MessagePad ROM. You can also use these descriptions as a model for documenting the structure of your application's shared soups.

Making Changes to Other Applications' Soups

You should avoid changing other applications' soups if at all possible. If you must make changes to an entry in another application's soup, be sure to respect the format of that soup as documented by its creator. When possible, confine your changes to a single slot that you create in any soup entry that you modify.

When naming slots you add to other applications' soups, exercise the same caution you would in naming soups themselves—use your application name and developer signature in the slot name to avoid name-space conflicts.

This approach provides the following benefits:

- It prevents your application from inadvertently damaging another application's data.
- It helps your application avoid name conflicts with other applications' slots.
- It keeps soups from becoming cluttered with excessive numbers of entries.
- It allows easier removal of your application's data.

Note that when you makes changes to other applications' soups you should transmit notification of the changes by means of the mechanism described in "Using Soup Change Notification" beginning on page 11-92

Using Queries, Indexes and Tags

Once you have a reference to a valid soup object, you can retrieve soup entries from it by performing a query. The most efficient queries are those which test index and tag values; hence these items are explained together .

This section describes how to perform various queries to retrieve soup data. It includes examples of

- simple queries on indexes, tags, word beginnings and full text
- the use of ascending and descending indexes

- complex queries that combine the characteristics of multiple simple queries

Combining Query Types

The various kinds of queries are discussed separately only for pedagogical purposes. You can combine in the query spec any query types for which a particular soup has appropriate indexes; for example, you can create a single query spec that searches index key values, strings and tags.

Queries On Multiple Soups

Because pointers between soups are not supported, you must use multiple queries to obtain information from dissimilarly-named soups. For example, to remind yourself of someone's upcoming birthday, you need to query the "Names" soup with the name of the person (or other identifying information) to get their birthdate and query the "Dates" soup to find the entry to which you want to add the birthday reminder.

Queries On Single-Slot Indexes

You can do quite a lot with the entries in a soup by testing single slot values. This section provides code examples illustrating a variety of queries on single-slot indexes.

The following code fragment presents an example of the simplest kind of index query—it returns all entries in the soup.

```
local stuffSoup := GetUnionSoup("BunchOStuff");
allEntriesCursor := stuffSoup:Query(nil);
```

Note that `nil` is passed as the query spec. Because all soups carry a default index created for the system's own use, the implied specification is an index query on the default index. The order that this index imposes on the soup's entries is roughly that in which they were added to the soup; however, this ranking is not guaranteed. Because the system recycles the key values used

for this default index it may not rank entries in the order they were added under all circumstances. The only way to guarantee the ranking of entries in the order they were added is to index them on your own time stamp slot.

You'll generally be querying for a subset of soup entries, rather than all soup entries, however. That is, you'll want to include or exclude entries according to criteria you define. For example, you might want to find only entries that have a certain slot, or a certain range of values stored in that slot. The next several examples illustrate the use of single-slot index queries.

To find all entries that have a particular slot, simply query on that slot. Note that in order to query on the presence of a slot, the soup must be indexed on that slot. For example, the following example of a query returns a cursor to all soup entries that have a `name` slot. The cursor ranks the entries according to the value of this slot. As first returned by the query, the cursor points to the first entry in index order.

```
// mySoup is a valid soup indexed on the 'name slot
nameCursor:= mySoup:Query({indexPath:'name});
```

You can also use the cursor method `GoToKey` to go directly to the first indexed slot that has a particular name or value. For examples of the use of this method, see "Positioning the Cursor Incrementally" beginning on page 11-84.

Using index values and `beginKey` values to limit your search can speed up your searches significantly. The following example is an index query that uses a `beginKey` value and an `endKey` value to return only entries beginning with the `"Bob"` string.

```
// mySoup is a valid soup indexed on the 'name slot
local bobCursor := mySoup:Query({indexPath: 'name,
                                 beginKey: "Bob",
                                 endKey: "Bp"));
```

The index on the `name` slot guarantees that all of the `name` slots having values beginning with the string `"Bob"` are presented to the `Query` method

in index order. Thus the query can use a `beginKey` value to skip over any entries ranked before the first `"Bob"` entry. Similarly, the test can be concluded quickly by specifying an `endKey` modeling the next entry that could theoretically appear after `"Bob"`; in this case, the `"Bp"` string is sufficient.

Another way to find all entries having a particular value in a particular slot is to use an `indexValidTest`, which can test any indexed value without reading its corresponding entry into RAM. Thus, assuming that the soup is indexed on the `name` slot, you could use code like the following to search for entries having a string beginning with `"Bob"` in the `name` slot. (The `BeginsWith` function tests the beginnings of strings.)

```
// mySoup is a valid soup indexed on the 'name slot
// thus, the name itself is passed to the indexValidTest
local myCursor :=
      mySoup:Query({ indexValidTest: func (aName)
                                BeginsWith(aName, "Bob")});
```

A less-preferred way to test entries is to provide a `validTest` function to test entries individually. The use of a `validTest` increases the RAM requirements of the search drastically because the query must read each soup entry into RAM in order to pass it to the `validTest` function; hence, its use is less-preferred than the use of an indexValidTest method.

Because the query passes index key values to the `indexValidTest`, the `BeginsWith` function in the previous example can test these values directly. Performing the same operation as a `validTest` requires a slightly different syntax for the arguments to the `BeginsWith` function because the query passes the entire entry to the `validTest` method, rather than just the value of the indexed slot. Thus, the next code example must dereference the entry's `name` slot in order to pass its value to the `BeginsWith` function.

```
// mySoup is a valid soup indexed on the 'name slot
// the entire entry is passed to a validTest
// so dereference the 'name slot to pass its value
```

```
local myCursor :=
      mySoup:Query({ validTest: func (entry)
                        BeginsWith(entry.name, "Bob")});
```

You can also use a 'words query to test for word beginnings.

Queries On Word Beginnings

You can use a 'words query to obtain a set of entries having specified word beginnings. For example, the following code fragment illustrates a 'words query that returns entries having strings beginning with "bob".

```
// it's a words query because it has a 'words slot
// find words beginning with "bob"
myCursor := namesSoup:Query({words: ["bob"]});
```

This query finds entries containing the words "Bob", "Bobby", and so forth, but not words such as "JoeBob". Note that 'words queries are not case-sensitive—the original query spec is all lower-case.

Because the words slot contains an array, a'words query can test for multiple word beginnings. For example, the following code fragment returns entries that contain both of the string beginnings "Bob" and "Apple". Thus, an entry containing the strings "Bobby" and "Applegate" would be included in the results of the search.

```
// returns entries that contain both "bob" and "apple"
// won't return entries having only one of these strings
appleBobCurs:= namesSoup:Query({words:["Bob", "Apple"]});
```

Because each element in the array is a string, each "word" in a words query can actually contain multiple words and punctuation. For example, the following code fragment returns entries that contain both of the string beginnings "Bob" and "Apple Computer, Inc.".

```
// return entries that contain
// "bob" and "Apple Computer, Inc."
wdsPuncCurs:= namesSoup:Query({words:["Bob",
                               "Apple Computer, Inc."]});
```

Note

The more unique the search string is, the more quickly a `'words` search proceeds. Thus, `'words` queries are slow for search words that have only one or two characters in them. ◆

Queries on Entire Strings

To search for entire strings, rather than word beginnings, add to your `'words` query spec an `entireWords` slot that holds the value `true`, as in the following example.

```
// return entries holding entire words "bob" and "Apple"
entWdsCurs:= namesSoup:Query({ entireWords: true,
                               words:["Bob", "Apple"]});
```

This particular query returns entries that contain both of the words `"Bob"` and `"Apple"`. Because the `entireWords` slot holds the value `true`, this query does not match strings such as `"Bobby"`. Entries containing only one of the specified words are not included in the results of the search.

Queries On Any Text

You can use a `'text` query to conduct string searches not constrained by word boundaries. For example, the following code fragment illustrates a `'text` query that returns entries having strings that contain `"Bob"`.

```
// find strings containing the substring "Bob"
myCursor := namesSoup:Query({text: "Bob"});
```

This query finds entries containing the words "Bob"and "Bobby"as well as words such as "JoeBob". Note that unlike 'words queries, 'text queries search only for single strings rather than for arrays of strings.

Queries On Multiple-Slot Indexes

A multiple-slot query can be performed only on a soup that has a multiple-slot index. For a description of how to create a multiple-slot index, see “Using Soup Definitions” beginning on page 11-64 and “Adding an Index to an Existing Soup” beginning on page 11-69.

For the most part, queries on multiple-slot indexes are specified like single-slot queries and behave like single-slot queries. There are several notable differences, however, which are described here.

- The indexPath slot in the query spec frame (input to the soup’s Query method) must be an array of path expressions matching one of the multiple-slot indexes.
- Each of the slots beginKey, endKey, beginExclKey, and endExclKey may contain an array of sub-keys. Similarly, the argument to the GoToKey method may contain an array of sub-keys. The number of elements in this array need not match the total number of sub-keys. Sub-keys may be specified as single key values as well; in this case the single key is considered to be the primary key.

  For example, with the following multiple-slot index

  ```
  {structure: 'multiSlot,
  path:['name, 'company, 'age],
  type:['string, 'string, 'int]}
  ```

  one can use keys singly or in arrays, as in the following examples.

  ```
  // this array of beginKeys has "bob" as primary key
  beginKey:["bob", "Apple", 55];
  ```

```
// you can use a single endKey with a multi-slot index
endKey:"bob";
// go to first "bob" entry using "Apple" as tie-breaker
cursor:GotoKey(["bob", "Apple"]);
```

- The input to the `indexValidTest` function is an array of sub-key values. Like `beginKey` and `endKey` arrays, the first key in the array passed as input to an `indexValidTest` function is the primary key. Subesquent keys are used to distinguish entries having key values in common.

Note

Index keys are limited to a total of 39 unicode characters (78 bytes) per soup entry. Keys that exceed this limit may be truncated when passed to an `indexValidTest` function. For more information, see the description of the `indexValidTest` function in the section "Query Specification Frame" beginning on page 11-113. ◆

Queries on Descending Indexes

Queries on soups having descending order indexes can produce what may seem like surprising results.

For example, consider an index generated on the following query spec

```
descIndSpec := {structure: 'slot, path:'name,
               type:'string, order:'descending};
```

with the following data (sorted by the descending index):

```
"noun", "name", "axe", "able"
```

Sending the message *cursor*`:GotoKey("a")`returns the value `nil`. Sending the message *cursor*`:GotoKey("az")` returns the value `"axe"`. If you specify the value `b` as the query's `beginExclKey`, sending the message `cursor:GotoKey("n")` returns the value `"axe"`.

Keep in mind the sorting order when specifying `beginKey` and `endKey` values for soups sorted in descending order—they are the reverse of what

they would be if the same soup was indexed in ascending order. For example, the following `beginKey` and `endKey` values would be appropriate for the previous example.

```
{..., beginKey:"name", endKey:"able", ...}
```

Note

The sort order for symbol-based indexes is the ASCII order of the symbol's lexical form. This sorting behavior is made available in NewtonScript by the `SymbolCompareLex` global function. ◆

Queries On Tags

The `querySpec` frame presented as input to the `Query` method can include a `tagSpec` slot to specify queries on tags indexes. For a detailed description of the tag specification frame that resides in a query spec's `tagSpec` slot, see "Tag Specifications for Queries" beginning on page 11-115.

Recall that the results of a query are sorted in index order; thus, queries can use a tag spec in conjunction with an index to retrieve a specified subset of entries in a specified order. For example, you could use this kind of query to retrieve all entries having the tag `'business` in the order that they were created.

The following exceptions may be thrown when attempting to query using a tag spec. If the soup does not have a tags index, a `kFramesErrNoTags` exception (`evt.ex.fr.store -48027`) is thrown. If the tag spec passed as an argument to the `Query` method has none of the slots `equal`, `any`, `all` or `none`, a `kFramesErrInvalidTagSpec` exception (`evt.ex.fr.store -48028`) is thrown.

The next several examples presume that the `mySoup` soup is a valid soup that has a tags index on the `labels` slot.

The following query matches entries having only the `'tree` tag. The cursor returned by this query sorts the entries on the value of the `height` slot.

```
soup:query({indexPath:'height, tagSpec:{equal:'tree}});
```

The following query matches entries that have the string `"my text"` in any slot and any of the symbols `'tree` or `'flower` in the tag slot. Note that you need not specify the path to the tag slot because only one tags index is allowed per soup.

```
// match ("my text") AND ('tree OR 'flower)
soup:query({text:"my text",
            tagSpec: {any:['tree, 'flower]}});
```

The following query matches entries having the word beginnings `"abcde"` and `"wxyz"` in the `text` slot and both of the tags `'tag1` and `'tag0` but not the `'tag2` tag.

```
// match ("abcde…" AND "wxyz…") AND ('tag1 AND 'tag0) AND NOT 'tag2
soup:query({words:["abcde", "wxyz"],
         tagSpec: {all:['tag1, 'tag0], none:'tag2}});
// the following frame matches this query
{text: “"abcdefg and wxyzwyg swam home", tags: ['tag0, 'tag1]}
```

Invoking Internationalized Sorting Order

The `querySpec` frame passed as input to the `Query` method may include an optional `secOrder` slot. If the value of the `secOrder` slot is `true`, the query is case- and diacritics-sensitive. Internationalized indexing order is available only for queries of type `'index` having index keys of type `'string`. It affects the use of the `beginKey` and `endKey` slots, as well as the `GoToKey` and `GotoEntry` cursor functions.

Using Cursors

This section discusses the functions and methods used to manipulate cursor objects returned by soup queries. Individual entries in soups and union

soups are manipulated by the entry functions described in the section "Using Entries," later in this chapter. This section describes

- getting the cursor
- getting the current soup entry from the cursor
- positioning the cursor
- testing validity of the cursor
- getting an index key from the cursor
- cloning cursors
- getting the number of entries in cursor data

Some discussions in this section refer to the "first" entry in a soup. As you know, the only order imposed on soup entries is that defined by the soup's indexes. When you see references to the first entry in a soup, be aware that this phrasing really means "the first entry as defined by index order."

Getting a Cursor

Cursor objects are returned by the `Query` method. For more information, see "Using Queries, Indexes and Tags" beginning on page 11-72.

Getting the Current Entry From the Cursor

To obtain the soup entry that the cursor currently references, send the `Entry` message to the cursor, as shown in the following code fragment.

```
// assume myCursor is valid cursor returned from a query
local theEntry := myCursor:Entry();
```

Manipulating the Cursor

This section describes various ways to position the cursor within the range of entries it references.

The `Reset` method sets the cursor to the entry having the lowest index value in the query results and returns that entry; this method provides a reliable means of obtaining the first entry (as defined by the index—soup entries themselves are not really ordered) in the set of data referenced by the cursor.

```
// assume mySoup is a valid soup or union soup
local cursor := mySoup:Query({path: 'labels,
                              beginKey: 'business});
local firstCursEntry := cursor:Reset();
```

Note that if the query spec includes a `beginKey` value, the `Reset` method positions the cursor on the entry specified by the `beginKey` value.

An alternative approach relies upon the fact that when first returned by a query, the cursor points to the first soup entry in the data set that satisfies the query. Thus, to obtain the first entry in the data set referenced by a newly-created cursor, just send the `Entry` message to the cursor, as shown in the following code fragment.

```
// assume mySoup is a valid soup or union soup
local cursor := mySoup:Query({path: 'labels});
local firstCursEntry := cursor:Entry();
```

Resetting the cursor will not return the entry having the lowest index value in the soup if this entry does not satisfy the query spec that was used to generate the cursor. In such a situation, you need to obtain a cursor from a more broad-based query that includes all the entries in the soup. For example, the following query returns all entries in the `mySoup` soup; hence, the cursor returned by this query initially points to the “first” soup entry as defined by the soup’s index.

```
// assume mySoup is a valid soup or union soup
// the cursor initially points to the first soup entry
// this query returns all indexed entries
local cursor := mySoup:Query({});
// get the entry
local firstEntry := cursor:Entry();
```

To obtain the last entry in the set of data referenced by the cursor, send it the `ResetToEnd` message, as shown in the following example.

```
local cursor := mySoup: Query({endKey: 'stuff});
local lastCursorEntry := cursor:ResetToEnd();
```

If the query used to generate the cursor specifies an `endKey` value, the last entry in the cursor data may not necessarily be the last entry in the index. To obtain a cursor to the last entry in a soup, you need to generate the cursor on a query that includes all entries in the soup. You can then simply send the `ResetToEnd` message to the cursor generated by that query, as shown in the following example.

```
local cursor := mySoup: Query(nil);
local lastSoupEntry := cursor:ResetToEnd();
```

Positioning the Cursor Incrementally

The cursor can be advanced to the next entry in the set or moved back to the previous entry by the `Next` and `Prev` methods, respectively. After these methods move the cursor, they return the current entry.

You can use the `Next` method as in the following example.

```
local nextEntry := myCursor:Next();
```

You can use the `Prev` method as in the following example.

```
local previousEntry := myCursor:Prev();
```

You can use the `Move` method to move the cursor multiple positions. For example, instead of coding incremental cursor movement as in the following example

```
for i := 1 to n do
   myCursor:Next();
```

you can obtain faster results by using the `Move` method, as in the following example. After positioning the cursor, the `Move` method returns the current entry.

```
local theEntry := myCursor:Move(n)
```

To move the cursor in large leaps, it's faster to use `GoTo` and `GoToKey` methods to position the cursor directly, as described in the next section.

Positioning the Cursor Directly

You can use the `GoToKey` method to go directly to the first indexed slot that has a particular value and return the entry containing that slot, as shown in the following examples.

```
// assume index spec for soup that generated myCursor is
// {structure: 'slot, path: 'name, type: 'string}

// go to the first entry that has
// the value "Nige" in the name slot
local theEntry := myCursor:GotoKey("Nige");
```

The example's `myCursor` cursor was generated by querying a soup indexed on the `name` slot; thus any value passed to the `GoToKey` method must be the kind of data that resides in the `name` slot (string). For example,

```
// won't work - argument doesn't match index data type
myCursor:GotoKey('name);
```

Testing Validity of the Cursor

When a storage card is inserted or when a soups is created, union soups automatically include the new soup in the union as appropriate. A cursor on a union soup may not be able to include the new soup when

- the new soup does not have the index specified in the `indexPath` of the query spec used to generate the cursor
- query spec used to generate the cursor included a `tagSpec` and the new soup does not have a tags index.

In such cases, the cursor becomes invalid. An invalid cursor returns `nil` when sent messages such as `Next`, `Prev`, `Entry` and so on.

The `Status` cursor method can be used to test the cursor's validity; it returns the `'valid` symbol for cursors that are valid and returns the `'missingIndex` symbol when a soup referenced by the cursor is missing an index. Your application needs to call this method when it receives either of the `'soupEnters` or `'soupCreated` soup change notification messages. If the `Status` method does not return the `'valid` symbol, the application must correct the situation and recreate the cursor.

For a detailed description of the `Status` cursor method, see the section "Query and Cursor Functions" beginning on page 11-154. For a discussion of soup change notification messages, see the section "Callback Functions for Soup Change Notification" beginning on page 11-116.

Getting the Current Entry's Index Key

The `EntryKey` cursor method returns the index key data associated with the current cursor entry without reading the entry into RAM. Note, however, that under certain circumstances the data returned by this method does not match the entry's index key data exactly. For more information, see the description of the `indexValidTest` function in the section "Query Specification Frame" beginning on page 11-113.

Copying Cursors

You can clone a cursor to use for rummaging around in the soup without disturbing the original cursor. Do not use the global functions `Clone` or `DeepClone` to clone cursors. Instead, use the `Clone` method for cursors, as shown in the following code fragment.

```
local namesSoup:=GetStores()[0]:GetSoup("Names");
local namesCursor := namesSoup:Query({});
local cursorCopy:=namesCursor:Clone();
```

Counting the Number of Entries in Cursor Data

Because the user can add or delete entries at any time, it's difficult to determine with absolute certainty the number of entries in a cursor. With that in mind, you can use the `CountEntries` cursor method to discover the number of entries present in the set referenced by the cursor at the time the `CountEntries` method executes.

To discover the number of entries in the entire soup, you can execute a very broad query that includes all soup entries in the set referenced by the cursor and then send a `CountEntries` message to that cursor. For example,

```
// assume mySoup is valid
local allEntriesCursor := mySoup:Query(nil);
local numEntries := allEntriesCursor:CountEntries();
```

Note that if the query used to generate the cursor specifies a `beginKey` value, the `CountEntries` method starts counting at the entry that the `beginKey` value specifies.

Using Entries

This section discusses the functions and methods used to manipulate soup entry objects returned by cursors. This section describes

- adding entries to soups
- saving references to entries
- modifying entries
- replacing entries
- sharing entry data
- copying entry data

Adding Entries to Soups

To create a new soup entry, pass to the `AddXmit` soup method the frame to be stored as the new soup entry. The `AddXmit` method transforms the frame presented as its argument into a soup entry, returns the entry and transmits a change notification message. The following code example illustrates the use of this method.

```
local myFrame := {text: "Some info", color: 'blue};
// assume mySoupDef is a valid soup definition
local myUSoup := RegUnionSoup(mySoupDef)
myUSoup:AddToDefaultStoreXmit(myFrame, kAppSymbol);
```

The new soup entry that this method creates consists of the frame originally passed to the `AddXmit` method plus the transitive closure of that frame's pointers (that is, everything that frame points to, everything those data structures point to, and so on). Thus, you must be very cautious about making soup entries out of frames that include pointers to other data structures. In general, this practice is to be avoided—it can result in the creation of entries that are extremely large or entries missing slots that were present in the original frame.

For example, the presence of a `_parent` slot in the frame presented as the argument to `AddXmit` causes the whole `_parent` frame (and its parent, and so on) to be stored in the resulting entry, potentially making it extremely large. An alternative approach is to store a key symbol in the data and look up the parent object in a frame of templates at runtime.

Do not include `_proto` slots in frames presented to methods that create soup entries. These slots will not be written to the soup entry and will be missing when the entry is read.

Circular pointers within an entry are supported, and an entry can refer to another by using an entry alias.

Keeping these cautions in mind, you can put any slots you need into your soup entries. Entries within the same soup need not have the same set of slots. The only slots to which you must pay special attention are those that are indexed. When you create a soup, you specify which of its entries' slots

are to be indexed. Indexed slots must contain data of the type specified by the index. For example, if you specify that an index is to be built on slot `foo` and that `foo` contains a text string, then it's important that every `foo` slot in every entry in the indexed soup contains a text string or `nil`. Entries that do not have a `foo` slot will not be found in queries on the `foo` index. Entries having a `foo` slot that contains data of some type other than text will cause the query to throw an exception.

Modifying Entries

Only one copy of a particular entry exists at any time, regardless of how the entry was obtained. That is, if two cursors from two different queries happen to be pointing at identical entries, they are actually both pointing at the same entry.

When first retrieved from a soup, an entry is just an identifier. When the entry is accessed as a frame (by getting or setting one of its slots), the complete entry frame is constructed. The frame is specially marked to identify it as a member of the soup from which it came.

When the frame is constructed from the entry, it is cached in memory. At this point, you can add, modify and delete slots just as you would in any other frame; however, the changes will not persist until the `EntryChangeXmit` function is called for that particular entry. The `EntryChangeXmit` function writes the cached entry frame back to the soup, replacing the original entry with the changed one.

If the `EntryUndoChangesXmit` method is called, the changes are thrown out and the entry is restored to its original state. This function disposes of the cached entry frame and restores the reference to the original uncached entry, just as if the original entry was never referenced.

The following code example gets an entry from the `cardSoup` soup, changes it, and then writes the changed entry back to the soup.

```
cardSoup:=getStores()[0]:GetSoup("Names");
cardCursor:= cardSoup:Query(nil);
theEntry:=cardCursor:Entry();
```

```
theEntry.cardType:=4;
EntryChangeXmit(theEntry, nil);
```

Deciding when to call the `EntryChangeXmit` function is sometimes difficult. For example, it would be inappropriate to call this function from the `viewChangedScript` method of a `protoLabelInputLine` view. When the user enters data on the input line with the keyboard, the `viewChangedScript` is called after every key press. Calling the `EntryChangeXmit` function with every key press would be noticeably slow.

In some situations, the appropriate time to call `EntryChangeXmit` is more obvious. For example, a natural time to call `EntryChangeXmit` is when the user dismisses an input slip.

Copying An Entry Into Another Soup

You can use the `EntryCopyXmit` function to copy an entry from a specified source soup to a specified destination soup and transmit a soup change notification message, as the following code fragment illustrates.

```
// get store references
local intStore := GetStores()[0];
local otherStore := GetStores()[1];
// get soup references
local myIntSoup := intStore:GetSoup("My Soup");
local myOtherSoup := otherStore:GetSoup("My Other Soup");
// get all entries having 'hot in 'temperature slot
local cursor := myIntSoup:Query({indexPath: 'temperature,
                                 beginKey: 'hot
                                 endKey: 'g});
// copy the entries to destination soup
while e := cursor:Entry() do
            EntryCopyXmit(e,destinationSoup,kAppSymbol);
```

Note
The `EntryCopyXmit` method copies the cached entry into the destination soup, not the original soup entry.

Sharing Entry Data

Shared soups (and therefore their entries) need to be in a well-documented format to allow other applications to use them. For an example of how to document the structure of your soup entries, refer to Chapter 18, “Built-In Applications and System Data,” to see descriptions of the soups used by the built-in applications on the MessagePad.

Using Entry Aliases

This section describes how to make entry aliases, store them and resolve them.

The `MakeEntryAlias` function returns an alias to a soup entry, as shown in the following code fragment.

```
// return entries that contain "bob" and "Apple"
local entWdsCurs:= namesSoup:Query({ entireWords: true,
                                words:["Bob", "Apple"]});
// keep an alias to bob around
local ebertAlias := MakeEntryAlias(entWdsCurs:Entry());
// but get rid of the cursor
entWdsCurs := nil;
```

To store an entry alias, simply add it to a soup like any other frame.

You can use the `ResolveEntryAlias` function to obtain the entry to which the alias refers, as shown in the following code fragment.

```
// continued from previous example
local bobEntry := ResolveEntryAlias(ebertAlias);
```

Note that the `ResolveEntryAlias` function returns `nil` if the original store, soup or entry to which the alias refers is unavailable.

You can use the `IsSameEntry` function to compare entries and aliases to each other; this function returns true for any two aliases or references to the same entry. For example,

```
// return entries that contain "bob" and "Apple"
local entWdsCurs:= namesSoup:Query({ entireWords: true,
                                words:["Bob", "Apple"]});
local aBob:= entWdsCurs:Entry();
// keep an alias to bob around
local ebertAlias := MakeEntryAlias(aBob);
// the following comparison returns true
IsSameEntry(aBob, ebertAlias)
```

Not sure what you've got? The `IsEntryAlias` function returns `true` if its argument is an entry alias, as shown in the following example.

```
// return entries that contain "bob" and "Apple"
local entWdsCurs:= namesSoup:Query({ entireWords: true,
                                words:["Bob", "Apple"]});
// keep an alias to bob around
local ebertAlias := MakeEntryAlias(entWdsCurs:Entry());
// the following test returns true
IsEntryAlias(ebertAlias);
```

Using Soup Change Notification

When your application changes an entry in a shared soup, it must notify other applications using that soup to allow them to take appropriate action

for dealing with the change. Similarly, your application may need to take action when soups that it uses are changed by other applications. The system-supplied soup change notification service allows applications to

- notify each other when they make changes to soup entries
- react precisely to notifications
- control how and when notifications are sent

The first part of this section describes how to register and unregister a callback function for execution in response to changes in a particular soup. The next part describes the various notifications that may be sent. The last part of this section describes how applications send soup change notifications.

Registering Your Application for Change Notification

The `RegSoupChange` global function registers a callback function for execution in response to changes in a particular soup. If your application needs to deal with changes to more than one soup, you'll need to call the `RegSoupChange` function once on each soup for which your application requires change notification.

You can call the `RegSoupChange` function at any time that makes sense for your application. For example, you might do so from within your base view's `viewSetupDoneScript` method; however, this is only a suggested guideline. In order to conserve available memory, your application should minimize the amount of time that soup definitions remain registered.

The following code example shows how to register your application for notification of changes to the System soup. The `kAppSymbol` constant that this code uses is defined for you by Newton Toolkit.

```
// this example callback prints info to the Inspector
local myFn :=
   func (soupName, kAppSymbol, changeType, changeData )
   begin
```

```
      print(changeType & " happened in the " & soupName & " soup.");
         if (changeData) then
            print ("the new data is " & changeData);
         else
            print ("the change data is nil");
         end;
// register the callback for changes to "Names" soup
RegSoupChange("Names", kAppSymbol, myFn);
```

Unregistering Your Application for Change Notification

When your application no longer needs to be notified of changes to a particular soup, it needs to call the `UnRegSoupChange` function to unregister its callback function for that soup, as shown in the example immediately following.

```
// unregister my app's Names soup callback
UnRegSoupChange("Names", kAppSymbol);
```

Normally, you can unregister your soup change callbacks in the `viewQuitScript` method of your application's base view.

Receiving Notifications

When a soup changes in some way, the system executes the callback functions registered for that soup. Note that the system does not consider the soup to have changed until an entry is written to the soup. Thus, changing an entry itself is not considered a change to the soup until the `EntryChangeXmit` function writes the cached entry back to the soup.

Your callback function must take any action that is appropriate to respond to the change. Most applications have no need to respond to soup changes unless they are open, which is why it is recommended that you register your callbacks when your application opens and unregister them when it closes.

The arguments passed to your callback function include the name of the soup that changed, the symbol identifying the callback function to execute, the kind of change that occurred and optional data such as changed soup entries. For a complete description of the callback function and its parameters, see the section "Callback Functions for Soup Change Notification" beginning on page 11-116.

IMPORTANT

The system-supplied Preferences application sends soup change notifications only if your application uses the `RegUserConfigChange` function to register for such notifications. ▲

Sending Notifications

When your application alters a shared soup, it may need to notify other applications that the soup has changed. The best means of doing so depends on the exact nature of the change.

The system provides functions and methods that change notification messages automatically after altering soups, union soups or entries. The names of these auto-transmit routines end with the `-Xmit` suffix. They are described throughout this chapter in sections pertaining to the main behaviors they provide.

The auto-transmit routines provide the easiest and best way to handle soup notification when making a limited number of changes to a soup. For example, to add a new soup entry and then transmit an appropriate notification message automatically, use the `AddXmit` method as shown in the following code fragment.

```
// example assumes catSoup is valid
// frame to add as new entry
local myFrame := {name: Daphne, color: tabby};
// add the entry and transmit change notification
// application making the change is kAppSymbol
local err := catSoup:AddXmit(myFrame, kAppSymbol);
```

Note that the auto-transmit methods and functions accept a *changeSym* parameter specifying which application changed the soup. If you pass `nil` for the value of the *changeSym* parameter, the change notification is not sent, but the function or method does everything else that its description specifies.

Sometimes it may not be not desirable to send notifications immediately after making each change to a soup; for example, when changing a large number of soup entries, you might want to wait until after you've finished making all the changes to transmit notification messages. You can use the `XmitSoupChange` global function to send soup change notifications explicitly.

The first argument to the `XmitSoupChange` function specifies the name of the soup that has changed and the second argument specifies the application making the change. The third argument is a predefined symbol specifying the kind of change that was made, such as whether an entry was added, deleted or changed. Where appropriate, the final argument is change data, such as the new version of the entry that was changed. For a more detailed discussion of change type and change data, see the section "Callback Functions for Soup Change Notification" beginning on page 11-116.

Using Virtual Binary Objects

This section describes how to use a virtual binary object to store a data set that is too large to fit into a soup entry. Topics discussed include how to

- create a VBO
- make the VBO persistent
- undo changes to VBO data

In addition to the subjects discussed here, several VBO utility functions are also described in the section "VBO Functions and Methods" beginning on page 11-166.

Creating Virtual Binary Objects

You can use the store method NewVBO to create new, uncompressed, “blank” virtual binary objects, as shown in the following code example. Because the VBO is not persistent until it is stored in a soup entry, the example assigns the result of the VBO-creation method to a soup entry.

Whether you create compressed or uncompressed VBO objects is a question of space versus speed: uncompressed objects provide faster access to data but require more store space than their compressed counterparts.

```
// create new uncompressed VBO of size 5 KB and class 'samples in
// the 'binData slot of the myFrm frame
local myFrm := {binData: GetDefaultStore():NewVBO('samples,5000),
                  text: “empty vbo”};
// add myFrm to the “foo” soup and transmit change notification
GetUnionSoupAlways(“Foo”):AddToDefaultStoreXmit(myFrm, kAppSymbol);
```

Note that you can use the NewCompressedVBO method to create uncompressed virtual binary objects by passing nil as the values of the *companderName* and *companderData* parameters.

```
// another way to create an uncompressed VBO
newEntry.binaryData := GetDefaultStore():NewCompressedVBO('samples,
                                          soundSize, nil, nil);
```

The system provides two built-in companders, which expand and compress raw binary data on demand as the data is paged beteween the NewtonScript heap and the store on which the VBO resides. The LZ compander is a suitable for most data types; its use is specified by passing the string "TLZStoreCompander" as the value of the *companderName* parameter to the NewCompressedVBO method. The pixel map compander is specialized for use with pixel map data; its use is specified by passing the string "TPixelMapCompander" as the value of the *companderName* parameter to the NewCompressedVBO method.

▲ WARNING
The pixel map compander makes certain assumptions about the data passed to it; do not use it for any data other than pixel maps. For more information, see the description of the `NewCompressedVBO` method beginning on page 11-166. ▲

Because the companders provided by the current implementation of the system initialize themselves automatically, you must always set the value of the *companderArgs* parameter to `nil`.

To create a new compressed VBO, specify a compander and a store in the arguments to the `NewCompressedVBO` method, as shown in the following example.

```
// create a compressed VBO for pixmap data
thePixMap := GetDefaultStore():NewCompressedVBO('pixMap,
                              theSize, "TPixelMapCompander", nil);
```

Making Virtual Binary Objects Persistent

Recall from the "About Virtual Binary Objects" discussion that VBO data is not persistent until the VBO itself is stored in a soup entry. Once stored in the soup entry, VBO data acts much like any other soup entry data. Like other soup entry data, the VBO data is cached in RAM when it is accessed. Thus, after changing VBO data stored in a soup entry, you need to call the `EntryChangeXmit` function to write the changed VBO data back to the soup.

Undoing Changes to VBO Data

The `EntryUndoChanges` global function undoes changes to VBO data residing in a cached soup entry. The following code example adds sound sample data to an empty VBO and demonstrates the use of the EntryUndoChanges function to undo those changes.

```
// create a temporary soup
mySoup := RegUnionSoup('|foo:Temp:PIEDTS|,
```

```
        {name: "foo:Temp:PIEDTS", indexes: '[]}) ;

// get a soup entry that is a sound
anEntry := mySoup:AddToDefaultStoreXMIT('{sndFrameType: nil,
                                          samples:nil,
                                          samplingRate:nil,
                                          dataType:nil,
                                          compressionType: nil,
                                          userName: nil}, nil) ;

// make a VBO to use for the samples
myVBO := GetDefaultStore():NewCompressedVBO('samples,5000,nil, nil);

// grab some samples from ROM and fill in most of sound frame
romSound := Clone(ROM_FunBeep) ;
anEntry.sndFrameType := romSound.sndFrameType ;
anEntry.samplingRate := romSound.samplingRate ;
anEntry.dataType := romSound.dataType ;
anEntry.compressionType := romSound.compressionType ;
anEntry.samples := myVBO ;

// put the samples in the VBO
BinaryMunger(myVBO, 0, nil, romSound.samples, 0, nil) ;

// write the VBO to the soup
EntryChangeXMIT(anEntry, nil);

// listen to the sound to verify what we did
PlaySound(anEntry);

// change the sound
BinaryMunger(anEntry.samples,0, nil, ROM_PlinkBeep.samples, 0, nil);
```

```
PlaySound(anEntry) ;

// decide to go back to the original
EntryUndoChanges(anEntry);

PlaySound(anEntry);

// clean up
foreach store in GetStores() do
begin
        mySoup := store:GetSoup("foo:Temp:PIEDTS") ;
        if  mySoup then
                mySoup:RemoveFromStoreXMIT(nil);
end ;
UnregUnionSoup("foo:Temp:PIEDTS", '|foo:Temp:PIEDTS|);
```

Using Store Parts

This section describes how to create a store part and add soup data to it. This discussion is followed by a description of how to access the store part's soups from your application.

Do not use store parts unless you are familiar with the information provided in the section "Storing Static Data in Packages" beginning on page 11-15. Because other approaches may provide better space-efficiency or faster access to data, store parts are useful mainly to avoid recoding soup data in a more efficient representation.

Creating A Store Part

To create a store part, take the following steps using version 1.5 or later of Newton Toolkit.

- Create a new project.
- Select the Store Part radio button in the Output Settings dialog box.NTK disables all of the other settings in this dialog box when the Store Part option is selected.
- Configure the Package Settings dialog box as you normally would. The name specified in this dialog box identifies the store part to the system in much the same way that a package name identifies a package; thus, you need to ensure the uniqueness of this identifier by basing it on your developer signature in some way.
- Add a new text document to the project. You'll add to this document the NewtonScript code that creates one or more soups to reside the store part.

At compile time, NTK provides a global variable named `theStore` which represents the package store (store part) you are building. Any changes made to the `theStore` variable are reflected in the store part that is produced as the output of the build cycle. Thus, to create your read-only soup, you can add to the text file some NewtonScript code similar to the following example:

```
// some useful consts; note use of developer signature
constant kStoreName := "MyStore:MYSIG" ;
constant kSoupName := "MySoup:MYSIG" ;
constant kSoupIndices := '[] ;

// theStore is a global var provided by NTK
theStore:SetName(kStoreName) ;

// use a function to setup the soup so we can use locals
call func() begin
// create the soup ;
local soup:=theStore:CreateSoup(kSoupName, kSoupIndices);

// add a couple entries
soup:Add({anInteger: 1}) ;
soup:Add({anInteger: 2}) ;
end;
```

See the “Lost In Space” Newton DTS code sample for a more complex example.

When your application packages is built, NTK incorporates your store part in it.

Getting the Store Part

Package stores are made available by the `GetPackageStore` function. Package stores do not appear in the `GetStores` array, which is reserved for normal store objects.

The `GetPackageStore` function looks up the store by name, so each package store must be given a unique name when it is built. Generally, this is ensured by including the unique package symbol in the store name.

Accessing Data in Store Parts

Although store parts support all the messages that a normal store does, remember that they are read-only. Sending to the store part messages that normally would change the store (such as `CreateSoup`, `SetName` and so on) will cause an exception.

Another thing to keep in mind is that soups on store parts do not participate in Union soups. You need to check explicitly for the presence of your store and soup.

The GetPackageStore and GetPackageStores functions provide two different ways to find a store part. Generally, you will use `GetPackageStore` and pass the name of the store you created as its argument. Assuming the code above was used to create the Store part, you could use code similar to the following example to check for the existence of the soup.

```
local pStore := GetPackageStore(kStoreName) ;
if pStore then
   local pSoup := pStore:GetSoup(kSoupName) ;
```

The following code example shows how you can use the soup data in a store part.

```
// here is an example of using the soup on a Store part
call func()
begin
   local myStore := GetPackageStore("StorePart:PIEDTS") ;
   local mySoup :=
myStore:GetSoup("StorePartSoup:PIEDTS") ;

   local q := Query(mySoup, {type: 'index}) ;
   print(q:Entry()) ;

   local i := 1 ;
   while q:Next() do
      i := i + 1 ;

   Write("Total items in soup = ") ;
   Print(i) ;
end with () ;
```

This code prints `Total items in soup = 300` in the Inspector.

Using Mock Entries and Mock Soups

A mock entry is really a kind of a "holder" that has two parts: one is a cached frame which the NewtonScript interpreter treats as the real frame when doing assignment, slot lookup, and so on; the other is the handler frame that retrieves the actual entry data and implements a suite of methods that manipulate it. Because these handler methods must have intimate knowledge of the application's data structures, they must be implemented by the application developer.

This section describes the definition and use of the most important functions and methods that the application developer must supply to implement mock entries and mock soups. For detailed descriptions of these methods, as well as a complete list of additional mock entry handler methods, see the section "Application-Defined Mock Entry Handler Methods" beginning on page 11-178.

Topics discussed in this section include

- creating a mock entry
- testing the validity of a mock entry
- getting entry cache data
- writing data to the entry cache
- getting and setting mock entry handlers

Creating a New Mock Entry

The `NewMockEntry` global function creates a new mock entry object having the specified handler and cached object. The interface to this function looks like the following example.

`NewMockEntry(`*handler*`,` *cachedObject*`)`

Your application can use this method to create a new mock entry; for example, in response to a *mockSoup*`:Add()` or *mockCursor*`:Entry()` call.

Testing the Validity of Mock Entry Objects

The `IsMockEntry` global function returns `true` for objects that are valid mock entries. You can use this function to distinguish between mock entry objects and other objects such as cache frames or soup entries. Note that the `IsSoupEntry` function returns `true` for both mock entries and normal soup entries.

Getting the Cached Object

The `EntryCachedObject` global function returns the cached object associated with the specified mock entry. The interface to this function looks like the following example.

`EntryCachedObject(`*mockEntry*`)`

Your mock cursor's `Entry` method may call this function to obtain the data associated with a specified mock entry. However, this method must first check the mock soup's cache to make sure the entry does not already exist—the objects returned by the Entry methods of two mock cursors referencing the same mock entry must test equal; that is, the cursors must return the same mock entry when their `Entry` method is invoked.

Your handler must supply an `EntryAccess` method that does anything necessary to install in the cache a valid NewtonScript frame containing the entry data. If the entry's cached object is not `nil`, your handler's `EntryAccess` method can call the `EntrySetCachedObject` function to install the *cachedObject* data in the cache.

Installing Entry Data in the Cache

The `EntrySetCachedObject` global function installs a specified frame in the cache. The interface to this function looks like the following example.

`EntrySetCachedObject(`*mockEntry*`, `*newCachedObj*`)`

Changing the Mock Entry's Handler

The `EntrySetHandler` function is a special-purpose function that you can use to install a mock entry in a different handler than the one in which it was originally installed. For example, you can use this function to install a mock entry in a handler that implements debug versions of the methods present in a normal handler frame. Such methods might include breakpoints and print statements that would not be present in the shipping version of an application.

The interface to the `EntrySetHandler` function looks like the following example.

`EntrySetHandler(`*mockEntry*`, `*newHandler*`);`

This method installs the mock entry specified by the value of the *mockEntry* parameter in the handler frame specified by the value of the *newHandler* parameter.

Getting The Mock Entry's Handler

Because applications manipulate mock entries by sending messages to their associated handler frames, the circumstances under which you would have a reference to a mock entry without having a reference to its handler frame are rare. However, because such situations can occur when debugging code that uses mock entries, the system supplies the `EntryHandler` function for debugging purposes.

The interface to this function looks like the following example.

`EntryHandler(`*mockEntry*`)`

The `EntryHandler` function returns a reference to the handler frame associated with the mock entry specified by the value of the *mockEntry* parameter.

Data Storage Reference

This section describes Newton data storage functions, methods and related data structures. This section begins with a description of important constants and data structures, including soup-creation constants, soup definition frames, index specification frames, query specification frames, soup change notification callback functions and package reference information frames. Subsequent sections provide descriptions of functions and methods grouped according to the subject with which they are most closely associated

including stores, VBOs, soups, tags, soup changes, queries and cursors, soup entries and mock entries. To facilitate easy lookup, each of these sections present their associated functions and methods in alphabetical order. The chapter concludes with a task-oriented summary of functions and methods.

Data Structures

This section describes various data structures related to the Newton data storage system, including soup definitions and specification frames for single-slot indexes, multiple-slot indexes and tags indexes. This section also describes query specification frames, tag specification frames for queries, callback functions for change notification and package reference information frames.

Soup Definition Frame

Soup definition frames are used to create soups on demand and to provide information about soups to the system, to other applications and to the user. This section describes the slots present in soup definition frames. For more information about soup definitions, see “Soup Definitions” beginning on page 11-31. For a description of how to use soup definitions, see “Registering and Unregistering Soup Definitions” beginning on page 11-65.

The soup definition frame specifies the soup’s name, its user-visible name, the application to which it belongs, descriptive strings used to present information to the user, and a default set of indexes to be created along with any soup of the kind defined by this frame. A typical soup definition looks like the following code fragment.

```
{  // string that identifies this soup to the system
   name: "soupName:appSym",
   // string that is user visible name
   userName: “Vital Statistics”,
   // application symbol
   ownerApp: '|Frobbitz:PIEDTS|,
```

```
   // user-visible name of app that owns this soup
   ownerAppName: "Frobbitz",
   // user-visible string describing soup
   userDescr: "This soup is used by
               the Frobbitz application.",
// array of indexSpecs - default indexes
   indexes: [anIndexSpec, anotherIndexSpec]
// optional - used to initialize the soup
   initHook: symbolOrCallBackFn
}
```

The soup definition frame contains the following slots.

`name`	Required. A string unique to the store on which the soup resides. This string identifies the soup to the system. For more information about naming soups, see "Naming Soups" beginning on page 11-64.
`userName`	Optional. A string that is the user-visible name for this soup; for example, this string is displayed as the soup's name in the Extras drawer.
`ownerApp`	Required. The application symbol identifying the application to which this soup belongs. For more information about application symbols, see the section "Application Symbol" beginning on page 2-10
`ownerAppName`	Required. The user-visble string identifying the application to which this soup belongs.
`userDescr`	Optional. A string that is the user-visible description of this soup. This string contains information about the purpose of the soup and the data it contains; for example, this string might be used to help the user to avoid deleting the soup accidentally.
`indexes`	Optional. An array of index specification frames. Each frame in this array describes one index in the default set of indexes with which this soup is created. Regardless of

whether additional indexes are specified, every soup has a system-supplied default index that includes every entry in the soup. This index ranks entries in roughly the order they were created. It can be accessed by passing `nil` as the argument to the `Query` method. For more information about index specification frames, see "Index Specification Frames" beginning on page 11-32. For detailed descriptions of index spec frames, see the sections "Single-slot Index Specification Frame" and "Multiple-Slot Index Specification Frame," immediately following.

`initHook` Optional. A callback function or a symbol. If this slot contains a function, it is called when the soup is created. If the slot contains a symbol, it is sent to the base view of the application specified by the value of the `ownerApp` slot. That application must define in its base view a slot containing a function to be executed when this soup is created. This function can do anything you require; however, the intent of the `initHook` slot is to provide a means of seeding the soup with initial values.

For related information, see the description of the NewtApp framework's `newtSoup` object, which provides methods for creating soups and filling them with entries.

Single-slot Index Specification Frame

This section describes the slots present in single-slot index specification frames. For general information about index specification frames, see "Indexes" beginning on page 11-31. For a description of how to use index specification frames, see "Using Soup Definitions" beginning on page 11-64 and "Adding an Index to an Existing Soup" beginning on page 11-69.

The index specification frame specifies the kind of index to create, the slot from which to extract index key values and the type of data found in the index key slot.

The index spec frame contains the following slots.

`structure` Required. Specifies whether the soup is indexed on a single slot or on multiple slots. For a single-slot index, this value must be the `'slot` symbol.

`path` Required. A path expression specifying the slot from which index key values are extracted. A path expression is either a symbol (for a simple path) or an array of class `PathExpr` in which individual elements are symbols. For example, a simple path could be expressed as

```
PathExpr:'x.y.z
```

and an array of path expressions could be expressed as

```
[PathExpr:'x, 'y, 'z]
```

For a complete explanation of path expressions, see *The NewtonScript Programming Language*.

IMPORTANT

You cannot use a value stored in a virtual binary object as an index key. ▲

`type` Required. The type of data stored in the index key slot. This can be an integer (specify `'int`), a string (specify `'string`), a character (specify `'char`), a real number (specify `'real`), or a symbol (specify `'symbol`).

`order` Optional; used only for multiple-slot indexes. Specifies the sorting order for the index; the only permissible values are either of the `'ascending` or `'descending` symbols. If this slot is missing or has the value `nil`, the index keys are assumed to be in ascending order.

`sortID` Optional. The value `1` specifies the use of the alternate sort table in ROM, which provides a case- and diacritic-sensitive ranking more suitable for non-English languages. If this slot is missing or has the value `nil`, the default sort table is used. For more information, see “Internationalized Sort Order” beginning on page 11-32

and “Invoking Internationalized Index Order” beginning on page 11-70.

A typical single-slot index spec looks like the following code fragment.

```
{
   structure:'slot,
   path:'Postal_code,
   type:'string
}
```

Multiple-Slot Index Specification Frame

This section describes the index description frame for a multiple-slot index. For general information about multiple-slot indexes, see “Multiple-slot Indexes” beginning on page 11-32. For a description of how to use index specification frames, see “Using Soup Definitions” beginning on page 11-64.

The multiple-slot index specification frame specifies the kind of index to create, the slots from which to extract index key values and the types of data found in those index key slots. The multiple-slot index spec frame contains the following slots.

`structure` Required. Specifies whether the soup is indexed on a single slot or on multiple slots. For a multiple-slot index, this value must be the `'multiSlot` symbol.

`path` An array of path expressions specifying the slots from which index key values are extracted. The first element in the array contains the path to the primary key, the second element contains the path to the secondary key, and so on. Note that the `path` array and the `type` array must be equal in length when measured by the `Length` function.

`type` Required. An array having any of the symbols `'string`, `'char`, `'int`, `'real` or `'symbol` as its elements. Each element of this array specifies the type of the data stored in the corresponding element of the

	`path` array in this index spec frame. Note that the `type` array and the `path` array must be equal in length when measured by the `Length` function.
`order`	Optional. An array of any of the possible values `'ascending` or `'descending`. Each element of this array specifies the sorting order for the key stored in the corresponding element of the `path` array in this index description frame. If the `order` array is missing, all subkeys are assumed to be in ascending order. Note that the values `Length(order[`*i*`])` and `Length(path[`*i*`])` must be equal.

A typical multiple-slot index spec looks like the following code fragment.

```
{
   structure: 'multiSlot,
   path:['name, 'company, 'age],
   type:['string, 'string, 'int]
}
```

Tags Index Specification Frame

The index defined by a tag spec stores the tag symbols associated with each entry in the soup. The tags index specification frame specifies the kind of data to index (in this case, symbols used as tags) and the slot from which to extract this data. The tags index spec frame contains the following slots.

`structure`	Required. Specifies whether the soup is indexed on a single slot or on multiple slots. For a tags index, this value must be the `'slot` symbol.
`path`	Required. A path expression specifying the slot from which index key values are extracted. In this case, the

	index key values are tags, so this expression specifies the slot in which this soup stores its tags.
`type`	Required. A symbol specifying the type of the data stored in the index key slot. For a tags index, this value must be the `'tags` symbol.

A typical tag spec frame looks like the following code fragment.

```
{
structure:'slot,
type:'tags,
path:'labels,
}
```

Query Specification Frame

The query specification frame or query spec describes the criteria which soup entries must meet to be included in the set of entries returned by the query. For more information regarding queries and their results, see the section “About Queries” beginning on page 11-40.

A query spec frame includes the following required slots; optional slots that may be included follow this listing of required slots.

`indexPath`	Required for queries based on any index other than the default. This value specifies the path to the slot used to generate index data. Search results are ranked according to the value of the slot specified by this value. Can be used by any of the query types.
`indexValidTest`	Optional. A function accepting the entry’s index key value and returning non-nil if the entry should be included in the query result. Like `validTest`, but tests the index key value without reading the entry into memory. Note that the system invokes the

	`indexValidTest` method before the `validTest` method. Note that in the following situations the input to the `IndexValidTest` function may not exactly match the entry's actual index key: Keys of type '`string` are truncated after 39 unicode characters. Ink data is stripped from '`string` keys. Sub-keys in multi-slot indexes may be truncated or missing when the total key size is greater than 78 bytes.
`validTest`	Optional. Used only for index queries. The value of this slot is a developer-supplied function accepting a soup entry as its argument. The function must return any non-`nil` value for an entry that is to included in the subset of entries returned by the cursor and return `nil` for an entry that is not to be included in the subset of entries returned by the cursor. Note that the use of an `indexValidTest` is preferable to the use of a `validTest`, for performance reasons.
`words`	Optional. Used only for words queries; it specifies an array of one or more strings to match with word beginnings in any slot in each entry. This query will not find a string in the middle of a word in the entry. Because each element in the array is a string, each "word" in a word query can actually contain multiple words and punctuation. Words queries are not case-sensitive. If you specify multiple array elements, each string must appear somewhere in the entry for it to be selected.
`entireWords`	Optional. Used for words queries only. When a `words` slot exists the value `true` in the `entireWords` slot specifies that query matches the entire string in the

`words` slot instead of matching strings beginning with the string in the `words` slot.

`text` Optional. Used for text queries only. Specifies a string for which the query searches. This search is not confined to word boundaries; that is, the search string will be found if it appears anywhere (beginning, middle or end) in any string in any slot in the entry.

`beginKey`, `beginExclKey`, `endKey`, `endExclKey`
Optional. These slots specify the range of a query in terms of key values. Each end of the range may be inclusive or exclusive of a given key value; that is, you can specify `key >= beginKey`, `key > beginExclKey`, `key <= endKey` or `key < endExclKey`. Either end of the range may be unspecified, in which case the range extends all the way to that end of the index. You can't specify both the inclusive and exclusive forms of the same end of the range.

`tagSpec` Optional. Contains a tag query specification frame as described in the section “Tag Specifications for Queries.”.

Tag Specifications for Queries

The frame described here specifies the use of tags by the `Query` method. This frame is placed in the `tagSpec` slot of the query spec frame presented as the argument to the `Query` method. In addition to specifying the tags on which to test, the tag spec frame specifies how the query is to use the specified tags; for example, whether the query includes or excludes entries having the specified tags.

The `tagSpec` slot contains a frame holding one or more of the following slots, each of which contains a symbol, an array of symbols or the value `nil`. The order in which symbols appear in the array is unimportant.

`equal`	The entry's tags must equal the set of tags specified in this slot. Entries with additional tags or missing tags are not matched. Note that `equal: [ ]` returns all non-tagged entries.
`all`	The entry's tags must include all of the tags specified in this slot. Entries having additional tags are matched as well.
`none`	Entries having none of the tags specified in this slot are matched, including entries that have no tags.
`any`	Entries having one or more of the tags specified in this slot are matched.

For examples of the use of tag specs, see the section "Queries On Tags" beginning on page 11-80.

Callback Functions for Soup Change Notification

This section describes the callback function that your application registers with the soup change notification mechanism. For more information about soup change notification, see the sections "Soup Change Notification" beginning on page 11-9 and "Using Soup Change Notification" beginning on page 11-92.

Your callback function is passed as the value of the *callBackFn* parameter to the `RegSoupChange` global function. The `RegSoupChange` global function registers this callback to be executed in response to changes in a specified soup. The value your callback function returns is currently ignored; however, it is recommended that this function return the value `nil`.

Your callback function must be of the form
`func(`*soupName*`,`*appSymbol*`,`*changeType*`,`*changeData*`);`

soupName	A string that is the name of the soup that changed
appSymbol	A unique symbol identifying the application that caused the change. If this information is not available, the system passes the `'_unknown` symbol to your callback function.
changeType	A symbol indicating the kind of change that occurred; for possible values, see Table 11-4, immediately following.
changeData	The data that changed. The data passed as this argument varies varies according to the value of the *changeType* parameter; for more information, see Table 11-4, immediately following.

Table 11-4 Change messages and associated change data

changeType	when sent	changeData
`entryAdded`	Entry added to soup or union soup	The entry that was added to the specified soup
`entryRemoved`	Entry deleted from soup or union soup	A frame having the soup the entry came from in its `oldSoup` slot and the (former) entry that was removed in its `entry` slot. For example, `{oldSoup:`*theSoup*`,entry:`*theEntry*`};`
`entryChanged`	Any change to entry data	The new version of the changed soup entry
`entryMoved`	Alteration changes entry's index position	A frame having the soup the entry came from in its `oldSoup` slot and the entry that moved in its `entry` slot. For example, `{oldSoup:`*theSoup*`,entry:`*theEntry*`};`

changeType	when sent	changeData
`entryReplaced`	Entry replaced with another having same internal identifier	A frame having the entry to replace in its `oldEntry` slot and the new version of that entry in its `entry` slot. For example, `{oldEntry:`*oldOne*`,entry:`*newOne*`}`; note that the `entry` slot can hold either a normal frame or a soup entry.
`soupInfoChanged`	Any change to soup information frame	The soup that changed
`soupEnters`	Soup becomes available to union, usually because card inserted	The soup that became available to the union
`soupLeaves`	Soup becomes unavailable to union, usually because card ejected	The soup that is no longer available; don't use this soup, as it is invalid when this message is sent
`soupCreated`	New soup created	The soup that was created
`soupDeleted`	Existing soup deleted	The store from which the soup was removed
`soupTagsChanged`[1]	Several tags changed	The soup that changed

changeType	when sent	changeData
`soupIndexAdded`	New soup index or tags index added	A frame having the new version of the soup in its `soup` slot and the new index path in its `index` slot; for example, `{soup:`*reIndexedSoup*`,index:`*newIndexPath*`}`
`soupIndexRemoved`	Existing soup index or tags index removed	A frame having the new version of the soup in its `soup` slot and the removedindex path in its `index` slot; for example, `{soup:`*reIndexedSoup*`,index:`*newIndexPath*`}`
`whatThe`	multiple changes to soup or 1.x application made change	Value is unspecified.Used when it's impractical to report all of the individual changes to a soup ; also used by 1.x applications that still use the `BroadCastSoupChange` function.

[1] Send soup notification only when tags are added by the `AddTags` method; otherwise, it's unnecessary.

Package Reference Information Frame

The `GetPkgRefInfo` function provide information about a specified package by returning an information frame containing the following slots of interest to NewtonScript developers. Do not rely on the values of any slots in

this frame that are not documented here; they are for system use only and subject to change without notice.

`size`	An integer specifying the package's uncompressed size, expressed in bytes
`store`	The store on which the package resides.
`title`	The string that is the name of the package
`version`	The integer that is the version number of the package
`timeStamp`	The date and time the package was loaded, expressed as an integer returned by the `Time` global function
`creationDate`	An integer specifying the date the package was created
`copyProtection`	Non-`nil` value specifies that the package is copy-protected.
`dispatchOnly`	The value `True` specifies that this package is a dispatch-only package.
`copyright`	Copyright information string
`compressed`	Non-`nil` value specifies that the package is compressed
`cmprsdSz`	Integer specifying the compressed size of package,m expressed in bytes
`numParts`	Integer specifying the number of parts in the package
`parts`	Array of parts comprising this package.
`partTypes`	Array of part type symbols; each element in this array specifies the part type of the corresponding element in the `parts` array.

Functions and Methods

This section describes all Newton data storage functions and methods. Methods are listed alphabetically under the object that defines them or the object on which they operate. Global functions are listed under the object on

which they operate. For a task-oriented summary of functions and methods, see “Summary” beginning on page 11-172.

Package Functions

The following functions and methods allow you to obtain information about the packages present on a specified store.

GetPackageNames

`GetPackageNames(`*store*`)`

Returns an array having elements that are the names of all packages present on the specified store.

GetPackages

`GetPackages()`

Returns an array of packages currently active in the Newton system. Each array element is a frame containing the following slots:

`id`	An integer that is the package id.
`size`	An integer that is the uncompressed size of the package, expressed in bytes
`title`	A string that is the name of the package.
`store`	The store on which the package resides
`version`	An integer that is the package’s version number.
`timeStamp`	The time the package was created, expressed as an integer returned by the `Time` global function
`copyProtection`	The value `True` specifies that this package is not to be replicated by the Newton system

A package is a bundle of resources that has been installed in the system. For example, a package can contain an application, fonts, sounds, images, and any other data needed by an application.

IMPORTANT

Because the package installation process is not re-entrant, you cannot call the `GetPackages` function from your application's `InstallScript` method or `RemoveScript` method. (These and other methods are invoked by the system in the midst of installing or removing a package.) ▲

GetPackageStore

`GetPackageStore(`*name*`)`

Returns the package store having the specified name; otherwise, returns nil. As always in NewtonScript, string comparison is not case sensitive. If more than one store has the specified name this function returns one of them arbitrarily.

GetPackageStores

`GetPackageStores()`

Returns an array of all available package stores.

▲ **WARNING**

Do not modify this array. ▲

GetPkgRef

`GetPkgRef(`*name*`, `*store*`)`

Returns a reference to the specified package on the specified store.

name	The string that is the name of the package
store	The store on which the package resides.

GetPkgRefInfo

`GetPkgRefInfo(`*pkgRef*`)`

Returns a frame containing information about the specified package. For a complete description of this frame, see the section "Package Reference Information Frame" beginning on page 11-119.

pkgRef Package reference specifying the package for which this function returns information.

Store Functions and Methods

The following functions and methods allow you to get information about all of the currently available stores; get and set the information frame that describes the store and its contents; create soups; write soups and packages to the store; obtain lists of soups present on the store and execute functions as transactions with respect to the store.

AtomicAction

store`:AtomicAction(`*myAction*`)`

Executes the *myAction* function as a single transaction, meaning that if its operations do not all succeed, the changes caused by the others are undone and the store is returned to the state it was in before the *myAction* function executed.

In order to provide this service, the system caches the changes made by the *myAction* function before making them permanent. Therefore, you must avoid doing large amounts of work from within the *myAction* function or the `AtomicAction` method will fail due to insufficient cache space.

Changing a small number of logically-related entries falls within this method's intended use, while changing every entry in a soup does not. For example, you might change the Names soup entries for the company name of all the members of a company as an atomic action—that way, if an error occurs, you are ensured that the entries are not left in an inconsistent state

(where some members of the company have the old name and some have the new name.)

myAction Application-defined function object accepting no arguments. The system calls this function with the store marked busy until the function returns.

BusyAction

store:`BusyAction(`*appSymbol*`,` *appName*`,` *action*`)`

Calls the *action* function object with the store marked busy until the *action* function returns. Unlocking the card switch on a store marked busy causes the "Newton needs the card…" slip to be displayed. The `BusyAction` method returns the result of the *action* function.

appSymbol Unique symbol identifying the application that posted the busy action.

appName String that is the user visible name of the application that posted the busy action.

action Application-defined function object accepting no arguments. The system calls this function with the store marked busy until the function returns.

CheckWriteProtect

store:`CheckWriteProtect()`

Returns `nil` if the store can be written; throws an exception if the store is locked or in ROM. If the store is in ROM, this function throws `|evt.ex.fr.store|(-48020)`. If the store is not in ROM, but is write-protected, it throws `|evt.ex.fr.store|(-10605)`.

This function is used to test whether the store can be written before executing a complicated operation; for example,

```
myStore:CheckWriteProtect(); //exit if we can't write
myStore:DoSomething();
```

CreateSoupXmit

store:`CreateSoupXmit (`*soupName*`,` *indexArray*`,` *changeSym*`)`

Creates a soup called *soupName* on the specified store, returns a reference to the newly-created soup object and transmits a soup change notification. Any existing union soups with the same name are updated to include the newly created soup, as are any existing cursors.

soupName — A case-insensitive string up to 39 characters long that specifies the name with which the soup is to be created. This name must be unique among all soups on the store.

indexArray — An array of index specification frames. For more information, see “Index Specification Frames” beginning on page 11-32. For detailed descriptions of various kinds of index spec frames, see “Data Structures” beginning on page 11-107.

changeSym — A unique symbol identifying the application that created the soup; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

Erase

store:`Erase()`

Erases all contents of the specified store. Returns `nil` if successful; else, throws an exception.

GetAllInfo

store:`GetAllInfo()`

Returns the store’s information frame. This special-purpose method is intended for use by backup / restore applications only; most applications need not use it. Unless an application stores data in it, this slot may not exist on every store. Applications can use the `GetInfo` store method to get their

own slot from the store's information frame. For more information, see the description of the `GetInfo` method, later in this section.

GetDefaultStore

`GetDefaultStore()`

Returns a reference to the store on which new items are created by default. The default store is specified by the user.

GetInfo

store`:GetInfo(`*slotSymbol*`)`

Returns the contents of the specified slot in the store's information frame. Unless an application stores data in it, the information frame may not exist on every store. This function returns `nil` if the store information frame does not exist. Applications can create a slot in the information frame to store card data, such as the last time the application encountered a particular card. For more information, see the description of the *store*`:SetInfo` method.

slotSymbol The slot to be returned. This value must be a symbol. Applications should create only a single slot in the store information frame and store minimal amounts of data in it.

The following example gets the contents of the `myText` slot from the information frame on the `myStore` store.

```
local t := myStore:GetInfo('myText);
```

GetName

store`:GetName()`

Returns as a string the name of the store referenced by *store*.

GetSignature

store:GetSignature()

Returns an integer that is the store's signature. The store signature is a random integer assigned when the store is created unless a specific signature was assigned by using the SetSignature store method.

GetSoup

store:GetSoup(*soupNameString*)

Returns the soup object specified by *soupNameString* from the store specified by *store*. If the soup doesn't exist, nil is returned. For example, the following code fragment retrieves the "Frobbitz" soup from the internal store.

```
local mySoup := GetStores()[0]:GetSoup("Frobbitz");
```

Note

Applications should use union soups for most purposes. You cannot use the GetSoup function to retrieve a union soup; instead, use the GetUnionSoupAlways function. ◆

GetSoupNames

store:GetSoupNames()

Returns an array of strings that are the names of the soups in the store specified by *store*. For example, the following code fragment returns an array of the names of soups present on the default store.

```
local s := GetDefaultStore():GetSoupNames();
```

GetStores

GetStores()

Returns an array containing references to all existing stores. Do not modify this array. The elements of this array are store objects to which you can send the messages described in the rest of this section. The element occupying the first position in the array returned by this function (GetStores()[0]) is

always the internal store; however, the meaning of the positions occupied by other stores in this array cannot be relied upon, as it may vary in future Newton devices.

The following example shows how you can use the `GetStores` function to pass the internal store to the `GetSoupNames` function, which then returns a list of all the soup names in the internal store.

```
GetStores()[0]:GetSoupNames();
```

HasSoup

store`:HasSoup(`*soupName*`)`

Returns a non-`nil` value if the store specified by *store* contains the soup having the name specified by *soupName*; otherwise, returns `nil`.

soupName — A string that is the name of the soup for which this method tests.

IsReadOnly

store`:IsReadOnly()`

Returns a non-`nil` value if the store is read-only (it could be a card that is write-protected), or `nil` if the store can be written.

IsValid

`IsValid(`*obj*`)`

Returns the value `true` if its argument references an object in valid memory. Returns `nil` for invalid objects such as references to objects residing on a card that is no longer available. Note that soup and store objects have their own `IsValid` methods; do not use the global function `IsValid` to test these kinds of objects.

obj — The object to be tested. Note that soup and store objects have their own `IsValid` methods; use these methods as appropriate to test these kinds of objects.

Note
Use this function judiciously; overuse can hamper your application's performance. ▲

▲ **WARNING**
This function tests only whether the object passed as its argument resides in valid memory; it does not follow references that the object may contain. Thus, its use does not cause the display of the "Newton needs the card" slip. However, if the object to be tested is a frame containing a slot that references an object on a PCMCIA card that has been removed, the frame itself may test valid even though it contains an invalid reference. You would need to test each slot in the frame with the `IsValid` function to find the slot containing the invalid reference. ▲

IsValid

store`:IsValid()`

Returns `true` if the store can be used. A store becomes invalid when it is removed, such as when the PCMICA card on which the store resides is removed.

ObjectPkgRef

`ObjectPkgRef(`*obj*`)`

Returns the package containing the specified object. This function returns `nil` if the object does not reside in a package; for example, this function returns `nil` for objects that reside in the NewtonScript heap.

obj The NewtonScript object to be tested

SafeRemovePackage

`SafeRemovePackage(`*pkgFrmOrID*`)`

Removes the specified package. Do not rely on the value returned by this function; it is unspecified.

pkgFrmOrID The package to be removed. This value can be a package frame as returned by the `GetPackages` function or it can be a package ID.

IMPORTANT

You cannot call this function from your application's `InstallScript` method or `RemoveScript` method. ▲

SetAllInfo

store`:SetAllInfo(`*frame*`)`

Writes the specified frame as the store's information frame. The value returned by this method is unspecified; do not rely on it. This special-purpose method is intended for use by backup / restore applications only; most applications need not use it. Instead, applications should use the *store*`:SetInfo` method to store their data in a single slot in the information frame. For more information, see the description of the `SetInfo` method.

frame The frame to be written as the store's information frame.

SetInfo

store`:SetInfo(`*slotSymbol*`,` *value*`)`

Sets the value of the specified slot in the store's information frame. If the slot does not exist, this function creates it and sets it to the specified value.

Applications can create a slot in the information frame to store card data, such as the last time the application encountered this card. Because the store information frame is shared by all applications, it is strongly recommended that your application follow the same guidelines for creating its slot in the store information frame as for creating a slot in another application's soup.

The value returned by this method is unspecified; do not rely on it.

slotSymbol	The slot to be set (or created if necessary.) This value must be a symbol. Applications should create only a single slot in the store information frame and store minimal amounts of data in it. To help avoid slot name collisions with other applications storing data in the store information frame, the name of this slot needs to be suffixed with your developer signature.
value	The value to be stored in the specified slot.

SetName

store:SetName(*storeNameString*)

Sets the name of the specified store to the value of *storeNameString*. and returns the new name of the store. This special-purpose method is intended for use only by backup / restore applications.

IMPORTANT

It's generally unwise to change the name of a store; you have no way of telling whether other applications are using its current name to refer to it. ▲

SetSignature

store:SetSignature(*signature*)

Sets the store signature to the specified integer value and returns the signature.

signature	The integer value to assign as the store's signature

IMPORTANT

Do not call this function unless you intend to completely replace all data on the store. This special-purpose method is for use only by backup / restore applications. ▲

SuckPackageFromBinary

store:SuckPackageFromBinary(*binary*, *paramFrame*)

Reads data from the specified binary object and returns a package created from the data. The package created by this method is suitable for loading on a Newton device.

binary The binary object supplying this package's data. This data is duplicated in the package that this method returns.

paramFrame A frame containing information used to build the package. This frame may contain the following slots and values:

callback Optional callback routine; set to `nil` if no callback is supplied. This callback is commonly used for the implementation of a progress indicator. The callback function must be a function object of the form

```
func(callbackInfo)
begin
    //do something w/callbackInfo
end
```

callbackFrequency The number of bytes to read before

executing the callback function again; set to `0` if no callback function is supplied.

callbackInfo A frame containing the following slots:

`packageSize`
Number of bytes in the package

`numberOfParts`
Number of parts in the package

`packageName`
name of the package

`currentPartNumber`
index of the part currently being read

`amountRead`
number of bytes read so far

SuckPackageFromEndpoint

store`:SuckPackageFromEndpoint(`*endPoint*`,` *paramFrame*`)`

Reads data from the specified endpoint and returns a package created from the data. The package created by this method is suitable for loading on a Newton device.

endPoint The endpoint supplying this package's data.This data is duplicated in the package that this method returns.

paramFrame A frame containing information used to build the package. This frame may contain the following slots and values:

callback Optional callback routine; set to `nil` if no callback is supplied. This callback is commonly used for the implementation of a progress indicator. The callback function must be a function object of the form

```
func(callbackInfo)
begin
   //do something w/callbackInfo
end
```

callbackFrequency
The number of bytes to read before executing the callback function again; set to `0`if no callback function is supplied.

callbackInfo A frame containing the following slots:

`packageSize`
Number of bytes in the package

`numberOfParts`
Number of parts in the package

`packageName`
name of the package

`currentPartNumber`
index of the part currently being read

`amountRead`
number of bytes read so far

TotalSize

store`:TotalSize()`

Returns the total size in bytes of the physical medium on which the specified store resides.

For example, the following line of code returns the total size of the internal store.

```
local size := getStores()[0]:TotalSize();
```

UsedSize

store`:UsedSize()`

Returns the number of bytes used in the store, including all overhead.

The following code example returns the number of bytes used in the main store.

```
getStores()[0]:UsedSize();
```

Soup Functions and Methods

The functions and methods described here allow you to get information about the currently available soups and union soups; get and set the information frame that describes a soup and the store on which it resides; obtain lists of soups present on the store; create soups and union soups; make copies of soups; and write soups and packages to the store.

RegUnionSoup

`RegUnionSoup(`*appSymbol*`, `*soupDef*`);`

Registers the specified soup definition for automatic soup creation by the `AddToDefaultStoreXmit` and `AddToStoreXmit` methods and returns the union soup.

appSymbol	Unique symbol identifying the application to which this soup belongs.
soupDef	A soup definition frame as specified in the section “Soup Definition Frame” beginning on page 11-107.

For example, the following code fragment registers the `kMySoupDef` soup definition and returns the myUsoup union soup object.

```
local myUsoup := RegUnionSoup(kAppSymbol,kMySoupDef);
```

UnRegUnionSoup

`UnRegUnionSoup (`*name*`, `*appSymbol*`);`

Unregisters the specified soup definition from automatic soup creation. The return value of this method is unspecified; do not rely on this value.

name	The name of the soup to unregister.
appSymbol	Unique symbol identifying the application to which this soup belongs.

AddToDefaultStoreXmit

unionSoup`:AddToDefaultStoreXmit(`*frame, changeSym*`)`

Adds the specified frame to the specified union soup and transmits a soup change notification message. This method returns `true` if the frame is added successfully and throws an exception if the frame cannot be added. The frame is added to the appropriate constituent of the specified union soup according to the user's default store preferences. (The user can specify either the internal store or a store on a storage card as the default store.)

frame	The frame to be made into an entry in the specified union soup. This frame must be writeable.
changeSym	A unique symbol specifying the application that added the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

IMPORTANT

The `AddToDefaultStoreXmit` method destructively modifies the frame that is presented to it. Thus, if it's important that the original frame remain unmodified, you may want to clone it first and pass the copy as the argument to this method. For more information see the description of the `AddXmit` method. ▲

AddToStoreXmit

uSoup:AddToStoreXmit (*frame*, *store*, *changeSym*)

Adds the specified frame to the constituent of the specified union soup on the specified store and transmits a soup change notification message. If necessary, this method creates the constituent soup to which the frame is added. This method returns `true` if the frame is added successfully and throws an exception if the frame cannot be added.

frame	The frame to be made into an entry in the specified soup.
store	The store containing the union soup constituent to which this method adds the specified frame as an entry.
changeSym	A unique symbol specifying the application that added the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

IMPORTANT

The `AddToStoreXmit` method destructively modifies the frame that is presented to it, creating special slots in the entry to identify it as belonging to a soup. Thus, if it's important that the original frame remain unmodified, you may want to clone it first and pass the copy as the argument to this method. For more information see the description of the `AddXmit` method. ▲

AddFlushedXmit

soupOrUSoup:AddFlushedXmit(*frameOrEntry*, *changeSym*)

Adds the specified frame or entry to the specified soup and transmits a soup notification message. The `AddFlushedXmit` method is similar to the `AddXmit` soup method except that the `AddFlushedXmit` method does not

`EnsureInternal` the frame presented to it, nor does it create a cached entry; thus, the `AddFlushedXmit` method uses less memory.

frameOrEntry The frame or entry to be added to the specified soup as an entry.

changeSym A unique symbol specifying the application that added the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

AddToStoreFlushedXmit

uSoup`:AddToStoreFlushedXmit(`*frameOrEntry*`,` *store*`,` *changeSym*`)`

Adds the specified frame to the constituent of the specified union soup on the specified store and transmits a soup change notification message. The `AddToStoreFlushedXmit` method is similar to the `AddToStoreXmit` soup method except that the `AddToStoreFlushedXmit` method does not `EnsureInternal` the frame presented to it, nor does it create a cached entry; thus, the `AddToStoreFlushedXmit` method uses less memory

frameOrEntry The frame or entry to be added to the specified soup as an entry.

store The store containing the union soup constituent to which this method adds the specified frame as an entry.

changeSym A unique symbol specifying the application that added the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

IMPORTANT

The `AddToStoreFlushedXmit` method destructively modifies the frame that is presented to it. Thus, if it's important that the original frame remain unmodified, you may want to clone it first and pass the copy as the argument to this method. For more information see the description of the `AddXmit` method. ▲

AddXmit

soup`:AddXmit(`*frame, changeSym*`)`

Adds the specified frame to the specified soup and transmits a change notification.

frame The frame to be made into an entry in the specified soup.

For example, the following code fragment adds the `aFrame` frame to the `mySoup` soup.

```
local aFrame:={name: "Casey", kind:'DMH, color: 'black};
local mySoup:= GetUnionSoupAlways("Critters");
mySoup:AddXmit(aFrame, kAppSymbol);
```

IMPORTANT

The `AddXmit` method destructively modifies the frame that is presented to it, creating special slots in the entry to identify it as belonging to a soup. If it's important that the original frame remain unmodified, you may want to clone it first and pass the clone as the argument to this method. ▲

GetUnionSoupAlways

`GetUnionSoupAlways(`*soupNameString*`)`

Returns a reference to the union soup object named by the value of the *soupNameString* parameter, even if none of its constituent soups exist or no soup definition for the specified soup exists. This function never returns `nil`. For more information, see the section "Using Newton Data Storage Objects" beginning on page 11-52.

For example, the following code fragment retrieves the "Frobbitz" union soup.

```
local myUSoup := GetUnionSoupAlways ("Frobbitz");
```

Query

soup:Query (*querySpec*)

Performs a query on the soup *soupRef*, according to the query specification frame *querySpec*, and returns a cursor that points to the first entry in the set of results satisfying the query.

soup	A valid reference to a soup object as returned by the GetSoup store method or the global GetUnionSoup function.
querySpec	A query specification frame as described in the section “Query Specification Frame” beginning on page 11-113.

IsValid

soup:IsValid()

Returns true if the soup can be used. A soup becomes invalid when the store on which it resides is removed, such as when a PCIMCIA card is removed, or when the soup itself is deleted.

GetMember

uSoup:GetMember(*store*)

Returns the specifed union soup member (single soup) from the specified store, creating that soup if it doesn't already exist.

CopyEntriesXmit

soup:CopyEntriesXmit (*destSoup*, *changeSym*)

Copies the entries in the source soup to the destination soup and transmits a change notification. The copied entries preserve the values of the original entries' unique identifiers. If one of the these identifiers is already used by an entry in the destination soup, this method throws an exception.

destSoup The soup in which the copied entries are written. This soup must be empty; this method does not perform error checking for duplicate entries in the soup specified by *destSoup*. This soup must not be a union soup, else a kFramesErrCantCopyToUnionSoup exception is thrown.

changeSym A unique symbol identifying the application that copeid the entries; usually this value is the application symbol or some variation on it. Pass nil for the value of this parameter to avoid transmitting a soup change notification.

AddIndexXmit

soupOrUsoup:AddIndexXmit (*indexSpec*, *changeSym*)

Adds an index to the specified soup or union soup and transmits a soup change notification. If this message is sent to a union soup, the index is added to all soups in the union. If the specified soup or union soup resides on a read-only store, this method throws an exception.

indexSpec An index specification frame. For more information, see "Index Specification Frames" beginning on page 11-32. For detailed descriptions of various kinds of index spec frames, see "Data Structures" beginning on page 11-107.

changeSym A unique symbol identifying the application that added the index; usually this value is the application symbol or some variation on it. Pass nil for the value of this

parameter to avoid transmitting a soup change notification.

CreateSoupFromSoupDef

`CreateSoupFromSoupDef (`*soupDef*`, `*store*`, `*changeSym*`)`

Creates a single soup on the specified store using the specified soup definition and transmits a `'soupChanged` notification. This special-purpose function is intended for use by backup and restoration applications only; normally, soups are created by the `RegUnionSoup` function (see p. 11-135).

GetAllInfo

soup`:GetAllInfo()`

Returns the soup's information frame. This special-purpose method is intended for use by backup / restore applications only; most applications need not use it. Unless an application stores data in it, this frame may not exist in every soup. Applications can use the `GetInfo` method to get their own slot from the soup information frame. For more information, see the section "Soup Compatibility" beginning on page 11-37, and the description of the `GetInfo` method on page 11-143.

GetIndexes

soup`:GetIndexes()`

Returns an array of index specification frames corresponding to the indexes that exist in the soup. This function does not return the default index. (Every soup has a default index which includes every entry in the soup.)

For example, the following code fragment returns the indexes from the `mySoup` soup.

GetIndexModTime

soup:GetIndexModTime()

Returns the time when the soup indexes were last changed. Soup index information is set when the soup is created, when indexes are added or removed and when indexed soup entries are added, deleted or changed.

GetInfo

soup:GetInfo(*slotSymbol*)

Returns the contents of the specified slot in the soup's information frame; this function returns `nil` if the slot does not exist.

soup	The soup having the information frame to be returned. This soup must be a single soup, not a union soup. This function is undefined for union soups.
slotSymbol	The slot to be returned. This value must be a symbol.

GetInfoModTime

soup:GetInfoModTime()

Returns the time when the soup info was last changed. Values in the soup information frame are set when the soup is created and when the soup methods `SetInfo` and `SetAllInfo` are executed.

GetName

soup:GetName()

Returns a string that is the name of the soup or union soup object to which this message is sent. For example,

```
local theName := aSoup:GetName();
```

GetNextUid

soup:GetNextUid()

Returns the unique identifier to be assigned to the next entry added to the soup. This special-purpose method is intended for use by backup / restore applications. Because the *soup*:AddXmit method and the AddToDefaultStoreXmit function automatically assign these identifiers, most applications do not need to use the GetNextUid method.

GetSignature

soup:GetSignature()

Returns an integer that is the signature for the soup. Using this signature, you can tell if a soup has been deleted and replaced by another with the same name. The signature is a random integer assigned when the soup was created, unless a specific signature was set using the SetSignature soup method. Normal applications never need to set a soup's signature—only specialized backup and restore applications may change this value.

GetSize

soupOrUsoup:GetSize()

Returns the size of the specified soup, expressed in bytes.

GetStore

soup:GetStore()

Returns a reference to the store on which the soup object *soup* resides. For example:

```
local store:= aSoup:GetStore();
```

IsSoupEntry

`IsSoupEntry(`*object*`)`

Returns `true` if the data object passed to this function is a soup entry. Otherwise, returns `nil`.

MakeKey

soup`:MakeKey(`*string*`, `*indexPath*`)`

Returns the index key for the specified string at the specified index path. Under certain conditions this key does not match the source string exactly:

- Keys of type '`string` are truncated after 39 unicode characters.
- Ink data is stripped from '`string` keys.
- Sub-keys in multi-slot indexes may be truncated or missing when the total key size is greater than 78 bytes.

You can use this method to determine precisely the index key used for a specified string.

string The string for which this method retrieves an index key.

indexPath The index path from which to extract the key value.

RemoveAllEntriesXmit

soup`:RemoveAllEntriesXmit (`*changeSym*`)`

Deletes all entries from the specified soup and transmits a change notification. The soup object to which this message is sent must be a single soup; this method is not implemented for union soups.

changeSym A unique symbol identifying the application that removed the entries; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

RemoveFromStoreXmit

soup:`RemoveFromStoreXmit (`*changeSym*`)`

Removes the specified soup from its store, deletes all of its entries and sends a soup change notification. This method cannot be used on a union soup.

changeSym — A unique symbol identifying the application that removed the soup; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

RemoveIndexXmit

soupOrUsoup:`RemoveIndexXmit (`*indexPath*`, `*changeSym*`)`

Removes an index from the specified soup or union soup object and transmits a soup change notification.

indexPath — The index path symbol on which the index to remove was generated; that is, the same index path used to create the index.

changeSym — A unique symbol identifying the application that removed the index; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

AddWithUniqueIDXmit

soup:`AddWithUniqueIDXmit (`*entry*`, `*changeSym*`)`

Adds the *entry* frame to the specified soup as a soup entry having the unique identifier specified in the *entry* frame. Transmits a soup change notification after adding the entry. This method throws an exception if the specified unique identifier is already used by an entry in the destination soup.

IMPORTANT

Use with extreme caution to avoid soup corruption. ▲

This special-purpose function is intended only for restoration of soup data; most applications should not use it. Normally, applications use the *soup*:AddXmit method to add a frame to a specified soup. The *soup*:AddXmit method generates a new unique identifier for the entry it adds.

entry — The entry to be added to the specified soup. This value must be a soup entry rather than a normal frame.

changeSym — A unique symbol identifying the application that added the entry; usually this value is the application symbol or some variation on it. Pass nil for the value of this parameter to avoid transmitting a soup change notification.

SetAllInfoXmit

soup:SetAllInfoXmit (*frame*, *changeSym*)

Writes the specified frame as the soup's information frame and transmits a soup change notification. This special-purpose method is intended for use by backup / restore applications only; most applications need not use it. Instead, applications should use the *soup*:SetInfo method to store data in a single slot in the soup information frame. For more information, see the description of the SetInfoXmit method, later in this section.

frame — The frame to be written in the soup's info slot.

changeSym — A unique symbol identifying the application that changed the soup information frame; usually this value is the application symbol or some variation on it. Pass nil for the value of this parameter to avoid transmitting a soup change notification.

IMPORTANT

The soup information frame holds the soup definition frame used to create the soup. Loss of the soup definition frame can lead to the presence of a null union soup. For more information, see the section "Null Union Soups" beginning on page 11-38. ▲

SetInfoXmit

soup:SetInfoXmit (*slotSymbol*, *value*, *changeSym*)

Sets the value of the specified slot in the soup information frame and transmits a soup change notification. If the slot does not exist, this function creates it and sets it to the specified value. This method's return value is unspecified.

slotSymbol — The slot to be set (or created if necessary.) This value must be a symbol. Applications should create only a single slot in the soup information frame and store minimal amounts of data in it. To help avoid slot name collisions with other applications storing data in the soup information frame, the name of this slot needs to be suffixed with your developer signature.

For more information, see the sections "Soup Information Frame" beginning on page 11-38 and "Making Changes to Other Applications' Soups" beginning on page 11-72.

value — The value to be stored in the specified slot.

changeSym — A unique symbol identifying the application that changed the soup information frame; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

SetName

soup:SetName(*soupNameString*)

Sets the name of the soup to the *soupNameString* string. This method's return value is unspecified. If you try to set the name to an invalid value (for example, one that is already in use) this method throws an exception. Generally, you should exercise extreme caution when changing the names of soups, even your own; other applications may be using them.

IMPORTANT

Do not under any circumstances change the names of the built-in soups. ▲

SetSignature

soup:SetSignature(*signature*)

Sets the soup signature to the specified integer value and returns the signature.

IMPORTANT

Use this method cautiously—it does not test whether the signature you pass as its argument is in use by another soup. ▲

Soup Notification

These functions allow you to register and unregister callback functions for execution when a soup changes in some way; for example, when entries are added or removed, when the soup itself is created or removed, and so on. Note that many of the soup methods themselves transmit soup-change notification messages; these methods end in the -xmit suffix and are described in the previous section ,“Soup Functions and Methods,” which begins on page 11-135.

RegSoupChange

RegSoupChange(*soupName*, *callbackID*, *callBackFn*)

Registers a callback function to be executed whenever the specified soup changes. The return value of this method is unspecified; do not rely on it.

soupName	A string that is the name of the soup that changed
callbackID	A unique symbol identifying the *callBackFn* function to the soup change mechanism. Because this symbol must be unique among the symbols registered with this soup,

your application's `appSymbol` or some variation on it is normally used as this parameter's value.

callBackFn A function that is called when the specified soup changes. The value this function returns is currently ignored; however, it is recommended that this function return the value `nil`. For a detailed description of this function, see "Callback Functions for Soup Change Notification" beginning on page 11-116.

UnRegSoupChange

`UnRegSoupChange(`*soupName*`,` *callbackID*`)`

Unregisters the specified callback function with the soup change notification service for the specified soup only. The return value of this method is unspecified; do not rely on this value.

soupName A string that is the name of the soup with which the callback function is registered for change notification

callbackID Unique symbol identifying the *callBackFn* function to the soup change mechanism. Because this symbol must be unique among the symbols registered with this soup, your application's `appSymbol` or some variation on it is normally used as this parameter's value.

XmitSoupChange

`XmitSoupChange(`*soupName*`,` *appSymbol*`,` *changeType*`,` *changeData*`)`

Notifies applications registered with the soup change mechanism that the specified soup has changed. Use this function when you don't want to

transmit separate notifications for every change to a soup, or to support change notification on older Newton devices.

soupName	A string that is the name of the soup that changed
appSymbol	Unique symbol identifying the application that caused the change.
changeType	A symbol indicating the kind of change that occurred; this value must be one of the symbols listed in Table 11-4 on page 11-117.
changeData	The data that changed. The data passed as this argument varies according to the value of the *changeType* parameter; see Table 11-4 on page 11-117 for more information.

Methods for Manipulating Tags

The methods described here allow you to add, remove and modify tags in soups and union soups as well as get information on the currently-existing tags in a specified soup. Methods that modify soups can transmit change notifications automatically.

AddTagsXmit

soupOrUsoup:AddTagsXmit (*tags*, *changeSym*)

Adds the specified tags to the soup's tags array as necessary and transmits a soup change notification. This method requires that the soup already have a tags index. This method always returns `nil`. If this message is sent to a union soup, the tags are added to each soup in the union. Note that the soup entries themselves are not changed by this method.

Normally you do not need to add tags to a soup explicitly - when you add an entry that uses the new tags the system adds them to the tags index automatically. You should use the `AddTags` method only when unused tags must be added to the tags index for some reason. For example, if you wanted to allow the user to file items in a folder category that was not yet used, you could use the `AddTags` method to add the not-yet-used tag to the tags index.

Subsequently, you could use the GetTags method to retrieve all of the currently-available tags (including the unused ones) for display to the user.

This method throws the `kFramesErrNoTags` exception (`evt.ex.fr.store -48027`) if the soup has no tags index. If the operation causes the maximum number of tags for the specified soup to be exceeded, this method throws a `kFramesErrInvalidTagsCount` exception (`evt.ex.fr.store -48026`) and does not add any of the new tags.

tagsToAdd	An array of symbols or a single symbol.
soupOrUSoup	The soup or union soup to which this method adds the specified tags.
changeSym	A unique symbol identifying the application that added the tag(s); usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

Note
Applications rarely need to use this method; when an entry with one or more new tags is added to the soup, the new tags are added to the tags index automatically. ◆

GetTags

soupOrUSoup`:GetTags()`

Returns an array containing the specified soup's tags. Returns `nil` if the soup does not have a tags index. If the specified soup is a union soup, the array returned by this method contains the tags for all soups in the union.

soupOrUSoup	The soup or union soup from which this method retrieves tags.

HasTags

soupOrUSoup:HasTags()

Returns true if the specified soup has a tags index. If the specified soup is a union soup, this method returns true only if each of the union's constituent soups have a tags index.

soupOrUSoup The soup or union soup to be tested.

ModifyTagXmit

soupOrUsoup:ModifyTagXmit (*oldTag*, *newTag*, *changeSym*)

Changes the symbol specified by *oldTag* to that specified by *newTag*, updates the soup entries and transmits a soup change notification. If this message is sent to a union soup, the specified tag is modified in all soups in the union.This method returns the value nil if successful. This method also returns nil and does nothing if the *oldTag* tag is not one of the tags in the specified soup.

Note

If the only difference between the *oldTag* tag and the *newTag* tag is case, this method does nothing because symbolic values are case-insensitive. For example, changing a tag from 'foo to 'Foo has no effect. ◆

This method throws the kFramesErrNoTags exception (evt.ex.fr.store -48027) if the soup has no tags index. If the *newTag* tag is already present in the soup's tags index, this method throws a kFramesErrInvalidTagSpec exception (evt.ex.fr.store -48028).

soupOrUSoup The soup or union soup for which this method modifies the specified tag.

oldTag A symbol specifying an existing soup tag.

newTag The new symbol for the tag specified by the *oldTag* argument.

changeSym A unique symbol identifying the application that modified the tag(s); usually this value is the application

symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

RemoveTagsXmit

soupOrUsoup`:RemoveTagsXmit (`*tagsToRemove*`,` *changeSym*`)`

Removes the specified tags as necessary from the specified soup, updates the soup entries and transmits a soup change notification. If this message is sent to a union soup, the specified tags are removed from all soups in the union. This method returns the value `nil`.

This method throws the `kFramesErrNoTags` exception if the soup has no tags index.

soupOrUSoup	The soup or union soup from which the specified tags are removed.
tagsToRemove	An array of symbols or a single symbol.
changeSym	A unique symbol identifying the application that removed the tag(s); usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

Query and Cursor Functions

This section describes the global function `Query`, which returns a cursor referencing a set of soup entries, the cursor method `Clone`, which duplicates a cursor, and various other methods that manipulate the cursor to obtain individual entries.

Clone

cursor`:Clone()`

This method clones (makes a copy of) the specified cursor and returns the clone.

Note

Do not use the global functions `Clone` or `DeepClone` to clone cursors. Instead, use the `Clone` method for cursors, as described here. ◆

CountEntries

cursor`:CountEntries()`

Returns the count of the entries matching the query specification. Note that if the query spec used to generate the cursor has a `beginKey` slot, the entries are counted beginning at the `beginKey` value.

cursor Soup cursor returned by the `Query` function.

Note

Use this method only when necessary—counting a large number of entries may be time-consuming and may require relatively large amounts of heap space. ◆

Entry

cursor`:Entry()`

Returns the current soup entry referenced by *cursor*.

If the current entry is deleted from the soup, the entry reverts to a plain frame (rather than a soup entry), and the method *cursor*`:Entry` returns the symbol `'deleted`. Make sure your code can handle gracefully a return value of `'deleted` from the *cursor*`:Entry` method.

If the cursor is advanced past the last entry or moved before the first entry in the set, the current entry pointed to by the cursor has the value `nil`. Make sure your code can also handle gracefully a `nil` value returned from the *cursor*`:Entry` method.

If the current entry is altered in a way that causes it to move to a different place within the set of entries returned from the query, the cursor moves with it.

EntryKey

cursor:`EntryKey()`

Returns the current entry key without reading the entry into memory.

Note
The value that this method returns may be different from the actual index key value for a particular entry; for more information, see the description of the `indexValidTest` function in the section "Query Specification Frame" beginning on page 11-113. ◆

Goto

cursor:`GoTo(`*entry*`)`

If the specified entry is valid, this method moves the cursor to the specified entry and returns `true`. If the specified entry is not valid the cursor does not move and this method throws an exception.

entry	The entry to which the cursor is moved. You cannot create an entry procedurally by creating a frame with the appropriate attributes. The only valid entries are those returned by the various cursor and entry methods.

GotoKey

cursor:`GoToKey(`*key*`)`

Moves the cursor to the first valid entry with the specified key, or to the next entry in key order if there is no entry with the specified key, and returns the entry. If no entries have the specified key, or the key is invalid, the cursor tests each entry until it runs out of entries, at which point this method returns `nil`.

key	A value residing in an indexed slot in the soup that *cursor* references. The data type of this value must match that specified by an index on the soup that *cursor* references.

MapCursor

`MapCursor(`*cursor*`, `*function*`)`

Applies the specified function to each of the cursor's entries in turn and returns an array of the results. If *function* is nil, the returned array consists simply of the cursor entries themselves.

function — The function that is to be mapped to the cursor's entries. This function you supply for this argument must accept one parameter, an entry.

Note that if the function returns a `nil` result for an entry, that entry is not added to the return array. `nil` results are discarded.

Move

cursor`:Move(`*n*`)`

Moves the cursor *n* entries forward from its current position and returns that entry. If *n* is negative, the cursor is moved backwards. If the cursor is advanced past the last entry or moved before the first entry in the set of entries it references, the current entry pointed to by the cursor has the value `nil`.

Next

cursor`:Next()`

Moves the cursor to the next entry in the set of entries referenced by the cursor and returns the entry. If the cursor is advanced past the last entry in the set of entries it references, the current entry pointed to by the cursor has the value `nil`.

Prev

cursor`:Prev()`

Moves the cursor to the previous entry in the set of entries referenced by the cursor and returns the entry. If the cursor is moved before the first entry in

the set of entries it references, the current entry pointed to by the cursor has the value `nil`.

Reset

cursor`:Reset()`
Resets the cursor to its initial state (pointing to the first entry satisfying the query).

ResetToEnd

cursor`:ResetToEnd()`

Resets the cursor to the entry having the highest index value in the valid subset.

cursor	Soup cursor returned by the `Query` function.

Status

cursor`:Status()`

Returns a symbol expressing the validity of the cursor. Cursors on union soups become invalid when including a soup that is missing one or more indexes common to the rest of the union. For more information, see the section "Testing Validity of the Cursor" beginning on page 11-85.

This method returns the following symbols.

`'valid`	No problems with the soups or indexes used by this cursor.
`'missingIndex`	At least one soup referenced by this cursor is missing one or more indexes common to the other soups in the union. The missing index may have been specified in the `indexPath` or `tagsSpec` slot of the query spec used to generate the cursor.

WhichEnd

cursor:WhichEnd()

Returns 'begin or 'end when the cursor is in a "virtual" position outside the range of valid entries. When the cursor is within the valid range of entries, this function's return value is unspecified.

Entry Functions

This section describes the functions used to manipulate individual entries that have been returned by a cursor.

EntryChangeXmit

EntryChangeXmit (*entry*, *changeSym*)

Writes a cached *entry* back to its soup and transmits a change notification. Returns an error if *entry* is not a valid soup entry.

entry — The cached entry that this method writes back to its soup.

changeSym — A unique symbol identifying the application that changed the entry; usually this value is the application symbol or some variation on it. Pass nil for the value of this parameter to avoid transmitting a soup change notification.

EntryUndoChanges

EntryUndoChanges(*entry*)

Disposes the cached *entry* frame. Any changes made to the frame are lost and the entry frame reverts to the version stored in the soup.

entry — The soup entry. If this entry contains VBO data, this function undoes its changes also.

EntryFlushXmit

`EntryFlushXmit(`*entry*`, `*changeSym*`)`

Writes the entry cache back to the specified soup entry and transmits a change notification. The `EntryFlush` function is similar to the `EntryChange` function; however, because the `EntryFlush` function does not pass its argument to the `EnsureInternal` function, it requires less NewtonScript heap space to write a changed entry back to its soup. Use of this function can result in dramatic savings of time and heap space when writing a large frame or many smaller frames to a soup.

entry The entry from which the cached frame was originally extracted.

changeSym A unique symbol identifying the application that changed the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntryIsResident

`EntryIsResident(`*entry*`)`

Returns `True` if the specified entry is cached. For more information about the entry cache, see the section “About Entries” beginning on page 11-50.

entry The entry to be tested.

EntryCopyXmit

`EntryCopyXmit (`*entry*`, `*newSoup*`, `*changeSym*`)`

Copies the specified entry into the specified soup, returns the new entry and transmits a change notification.

Note
This function copies the cached entry frame into the new soup, rather than the original soup entry. ◆

entry	The entry to be copied
newSoup	The soup into which the specified entry is to be copied.
changeSym	A unique symbol identifying the application that copied the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntryMoveXmit

`EntryMoveXmit (`*entry*`,`*newSoup*`,`*changeSym*`)`

Safely moves the specified entry into the specified soup and transmits a soup change notification message. This method copies the cached entry into the new soup, verifies the integrity of the duplicate entry and then deletes the original soup entry.

entry	The entry to be moved.
newSoup	The soup into which the specified entry is to be moved.
changeSym	A unique symbol identifying the application that moved the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntryReplaceXmit

`EntryReplaceXmit (`*original*`,`*replacement*`,`*changeSym*`)`

Replaces the contents of the *original* soup entry with the *replacement* entry and transmits a soup change notification.

original — The soup entry to be replaced. This value must be an entry, not a normal frame.

replacement — The soup entry to be added. This value can be an entry or a normal frame. In the latter case, this function makes the frame into a soup entry and adds the new entry to the soup.

changeSym — A unique symbol identifying the application that replaced the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntryRemoveFromSoupXmit

`EntryRemoveFromSoupXmit(`*entry*`, `*changeSym*`)`

Removes *entry* from its soup and transmits a soup change notification. The entry frame is converted to a plain frame (unmarked as belonging to a soup). The return value of this function is unspecified—do not rely on it.

entry — The soup entry to be removed and converted to a plain frame.

changeSym — A unique symbol identifying the application that removed the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntrySize

`EntrySize(`*entry*`)`

Returns the number of bytes that *entry* occupies in the store.

EntrySoup

`EntrySoup(`*entry*`)`

Returns a reference to the soup in which *entry* resides.

EntryStore

`EntryStore(`*entry*`)`

Returns a reference to the store in which *entry* resides.

EntryTextSize

`EntryTextSize(`*entry*`)`

Returns the number of bytes of *entry* that are occupied by text.

FrameDirty

`FrameDirty(`*frame*`)`

Returns `true` if the specified frame in memory has been modified since it was retrieved from its soup. Otherwise, returns FALSE.

EntryModTime

`EntryModTime(`*entry*`)`

Returns the time that the specified entry was last modified. The time is expressed as an integer that is the number of minutes passed since midnight, January 1, 1904. This function get this information directly from the soup, which is faster than referencing the `_modTime` slot in the entry; the latter approach would require that the entire entry frame be constructed.

EntryChangeWithModTimeXmit

`EntryChangeWithModTimeXmit (`*entry*`, `*changeSym*`)`

Writes a cached *entry* back to its soup using the modification time you specify, and transmits a soup change notification. This special-purpose method is intended for use by backup / restore applications only; most applications need not use it.

entry The cached entry that this method writes back to its soup.

changeSym A unique symbol identifying the application that changed the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntryReplaceWithModTimeXmit

`EntryReplaceWithModTimeXmit(`*original*`,`*replacement*`,`*changeSym*`)`

Replaces the *original* entry with the *replacement* entry, sets the modification time of the *replacement* entry to match that of the *original* entry and transmits a soup change notification.

This special-purpose method is intended for use by backup / restore applications only; most applications need not use it.

original The soup entry to be replaced. This value must be an entry, not a normal frame.

replacement The soup entry to be added. This value can be an entry or a normal frame. In the latter case, this function makes the frame into a soup entry and adds the new entry to the soup.

changeSym A unique symbol identifying the application that replaced the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntryUniqueId

`EntryUniqueId(`*entry*`)`

Returns the value that identifies the specified entry to the system. This function get this information without reading the entry into the cache.

Entry Alias Functions

The functions described here allow you to create aliases (references or links) to soup entries.

IsEntryAlias

`IsEntryAlias (`*object*`)`

Returns true if the specified object is an entry alias.

object The object to be tested.

MakeEntryAlias

`MakeEntryAlias(`*entry*`)`

Returns an entry alias object representing the specified soup entry. This object can be saved in a soup and later used as input to the `ResolveEntryAlias` function to retrieve the soup entry.

entry The soup entry to which this method creates a reference.

ResolveEntryAlias

`ResolveEntryAlias(`*alias*`)`

Returns the soup entry referenced by the specified alias. Returns `nil` if the entry cannot be retrieved, typically because the original store, the original soup or the original entry is not found.

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

IsSameEntry

`IsSameEntry(`*entryOralias1*`, `*entryOralias2*`)`

This method returns the value `True` only if its arguments evaluate to the same soup entry. Passing two distinct entries with identical content to this function does not cause it to return the value `True`. This method can compare soup entries, entry aliases, or combinations of the two.

entryOralias1	The soup entry or entry alias compared to the value of the *entryOralias2* parameter.
entryOralias2	The soup entry or entry alias compared to the value of the *entryOralias1* parameter.

VBO Functions and Methods

In addition to the new functions described in this section, VBOs support all standard object system functions such as `ClassOf`, `SetClass`, `Length`, `SetLength`, `Clone`, `BinaryMunger` and so on. Remember that the VBO is not persistent until it is put in a soup entry and written to the soup.

NewVBO

store`:NewVBO(`*class*`, `*size*`)`

Creates a virtual binary object of the specified class and size on the specified store. This function returns a reference to the object it creates.

class	A symbol specifying the class of the binary object that this method creates.
size	The amount data to be stored in this object, expressed in bytes.

NewCompressedVBO

store`:NewCompressedVBO(`*class*`, `*size*`, `*companderName*`, `*companderArgs*`)`

Creates on the specified store a compressed binary object large enough to store the specified number of bytes. This function returns a reference to the object it creates.

The specified compander is instantiated using the values specified by the *companderArgs* parameter. The compander transparently compresses the data as it is stored in the VBO and expands the data when it is read. This method creates an uncompressed binary object when `nil` is specified as the value of the *companderName* parameter.

class A symbol specifying the class of the binary object that this method creates.

size The amount of data to be stored in this object, expressed in bytes.

companderName A string value specifying the implementation of the store compander protocol used when the VBO created by this object is written to or read from a soup entry. If the value of this parameter is `nil`, an uncompressed object is created. The following strings are valid values for this parameter

`"TLZStoreCompander"`
Specifies the use of LZ compression and decompression

`"TPixelMapCompander"`
Specifies the use of a compander specialized for pixel map data. (A bitmap is a pixel map having a bit depth of 1.) This compander assumes that the data in the VBO is a pixel map and that the pixel map data is 32-bit aligned; that is, the length of the rows in the pixel map is an even multiple of 4 bytes.

For a description of the Newton bitmap format, see the `MakeBitmap` function in Chapter 12, “Drawing and Graphics.”

companderArgs Aguments for instantiating the specified compander. In the current implementation, always pass `nil` as the value of this parameter.

IsVBO

`IsVBO(`*vbo*`)`

Returns a non-`nil` value if the object to be tested is a virtual binary object; otherwise, returns `nil`.

vbo The object to be tested

GetVBOStore

`GetVBOStore(`*vbo*`)`

Returns the store object on which the specified virtual binary object resides. This function returns `nil` if its argument is not a VBO.

vbo The virtual binary object to be tested

GetVBOStoredSize

`GetVBOStoredSize(`*vbo*`)`

Returns the number of bytes the specified VBO actually uses in the store; for example, if the VBO is compressed, this function returns its compressed size.

vbo The VBO to be tested. Do not use objects other than VBOs as the value of this parameter.

IMPORTANT ▲
This method writes the VBO back to its store. ▲

GetVBOCompander

`GetVBOCompander(`*vbo*`)`

Return the name of the compander used for the specified object. If the object is not a VBO, this function returns an unspecified value.

vbo The VBO to be tested

Mock Entry Functions

The global functions described here create and manipulate mock entries. They do not work on normal soup entries.

NewMockEntry

`NewMockEntry(`*handler*`, `*cachedObj*`)`

Creates a new mock entry having the specified handler and cached object.

handler The frame implementing this mock entry's methods.

cachedObj The frame containing the data associated with this mock entry. You may pass `nil` for this value and fill in the entry data later from the `EntryAccess` method of this mock entry's handler frame.

IsMockEntry

`IsMockEntry(`*object*`)`

Returns a non-`nil` value if the specified object is a mock entry; otherwise, returns `nil`. This function returns the value `nil` when the object to be tested is a normal soup entry; in contrast, the `IsSoupEntry` function returns `true` for mock entries and for normal soup entries.

object The object to be tested.

EntrySetCachedObject

`EntrySetCachedObject(`*mockEntry*`, `*newCachedObj*`)`

Installs the specified frame as the cached object associated with the specified mock entry. The cached object is the object to which the system forwards accesses to the specified mock entry.

mockEntry The mock entry object for which the *newCachedObject* frame is to be entry data. If the value of the *mockEntry*

parameter is not a mock entry object an error is signalled.

newCachedObj The frame to be installed as the entry data for the specified mock entry.

EntryCachedObject

`EntryCachedObject(`*mockEntry*`)`

Returns the cached object associated with the specified mock entry object.

EntrySetHandler

`EntrySetHandler(`*mockEntry*`, `*newHandler*`)`

Installs the specified frame as the handler for the *mockEntry* mock entry object. This special-purpose method is intended for debugging purposes only; normally, one would not change a mock entry's handler.

mockEntry The mock entry object for which the *newHandler* frame is to be installed.

newHandler The new handler frame with which the *mockEntry* frame is to be associated.

EntryHandler

`EntryHandler(`*mockEntry*`)`

Returns specified mock entry's handler frame. This special-purpose method is intended for debugging purposes only.

mockEntry The mock entry object to be tested.

Developer-Defined Entry Handler Methods

The methods described here must be supplied by the application developer.

EntryAccess

handler:`EntryAccess(`*mockEntry*`)`

This method must be implemented by the application developer. It is called when the frame system needs to access a slot in a mock entry and the mock entry's cached object is `nil`. The method must create a frame representing the entry and use the `EntrySetCachedObject` function to assign that frame to the *mockEntry* object.

handler	The handler frame for the specified mock entry.
mockEntry	The mock entry being accessed. Do not rely on this value—it is not always passed.

Additional Developer-Defined Handler Methods

Mock entry handlers may also implement the following methods. The message is sent to the handler; the arguments are the same as the arguments passed to the global Entry*XXX* function with the corresponding name that caused the message to be sent.

handler:`EntrySoup(`*mockEntry*`)`
handler:`EntryStore(`*mockEntry*`)`
handler:`EntrySize(`*mockEntry*`)`
handler:`EntryTextSize(`*mockEntry*`)`
handler:`EntryUniqueID(`*mockEntry*`)`
handler:`EntryModTime(`*mockEntry*`)`
handler:`EntryChange(`*mockEntry*`)`
handler:`EntryChangeWithModTime(`*mockEntry*`)`
handler:`EntryRemoveFromSoup(`*mockEntry*`)`
handler:`EntryReplace(`*original*`, `*replacement*`)`
handler:`EntryReplaceWithModTime(`*original*`, `*replacement*`)`
handler:`EntryUndoChanges(`*mockEntry*`)`
handler:`EntryCopy(`*mockEntry*`, `*newSoup*`)`
handler:`EntryMove(`*mockEntry*`, `*newSoup*`)`
handler:`EntryValid(`*mockEntry*`)`

Summary

This section categorizes the data storage functions and methods by task.

Packages

`GetPackageNames(`*store*`)`
`GetPackages()`
`GetPkgRef(`*name*`, `*store*`)`
`GetPkgRefInfo(`*pkgRef*`)`
`IsValid(`*obj*`)`
`ObjectPkgRef(`*obj*`)`
`SafeRemovePackage(`*package*`)`
store`:SuckPackageFromBinary(`*binary*, *callback*, *callbackFrequency*`)`
store`:SuckPackageFromEndpoint(`*endPoint*, *callback*, *callbackFrequency*`)`

Stores

store`:AtomicAction(`*function*`)`
store`:BusyAction(`*appSymbol*`, `*appName*`, `*action*`)`
store`:CheckWriteProtect()`
`GetDefaultStore()`
store`:GetInfo(`*slotSymbol*`)`
store`:GetName()`
store`:GetSignature()`
store`:GetSoup(`*soupNameString*`)`
store`:GetSoupNames()`
`GetStores()`
store`:HasSoup(`*soupName*`)`
store`:IsReadOnly()`
store`:IsValid()`

store:SetInfo(*slotSymbol*, *value*)
store:SetName(*storeNameString*)
store:TotalSize()
store:UsedSize()

Package Stores

GetPackageStore(*name*)
GetPackageStores()

Soups

These functions and methods allow you to manipulate soup-level data such as frames, soup indexes, soup information frames and soup signatures.

Creating Soups

store:CreateSoupXmit (*soupName*, *indexArray*, *changeSym*)
CreateSoupFromSoupDef (*soupDef*, *store*, *changeSym*)
RegUnionSoup(*appSymbol*, *soupDef*);
UnRegUnionSoup (*name*, *appSymbol*);
uSoup:GetMember(*store*)

Change Notification

RegSoupChange(*soupName*, *callbackID*, *callBackFn*)
UnRegSoupChange(*soupName*, *callbackID*)
XmitSoupChange(*soupName*, *appSymbol*, *changeType*, *changeData*)

Adding and Copying Entries

soupOrUsoup:AddFlushedXmit(*frameOrEntry*, *changeSym*)
soupOrUsoup:AddIndexXmit (*indexSpec*, *changeSym*)
uSoup:AddToDefaultStoreXmit(*frame*, *changeSym*)
uSoup:AddToStoreXmit (*frame*, *store*, *changeSym*)
soup:AddXmit (*entry*, *changeSym*)
soup:CopyEntriesXmit(*destSoup*, *changeSym*)
soup:Flush()

Additional Functions and Methods

soup:GetIndexes()
soup:GetIndexModTime()
soup:GetInfo(*slotSymbol*)
soup:GetInfoModTime()
soup:GetName()
soup:GetSignature()
soupOrUsoup:GetSize()
soup:GetStore()
GetUnionSoupAlways(*soupNameString*)
soup:MakeKey(*string*, *indexPath*)
IsSoupEntry(*object*)
soup:IsValid()
soup:RemoveAllEntriesXmit(*changeSym*)
soup:RemoveFromStoreXmit(*changeSym*)
soupOrUsoup:RemoveIndexXmit(*indexPath*, *changeSym*)
soup:SetInfoXmit(*slotSymbol*, *value*, *changeSym*)
soup:SetName(*soupNameString*)

Tags

soupOrUsoup:AddTagsXmit (*tags*, *changeSym*)
soup:HasTags()
soup:GetTags()
soupOrUsoup:RemoveTagsXmit (*tags*, *changeSym*)
soupOrUsoup:ModifyTagXmit (*oldTag*, *newTag*, *changeSym*)

Queries and Cursors

These functions allow you to query a soup and manipulate the cursor that the query returns.

soup:Query(*querySpec*)

cursor:Clone()
cursor:CountEntries()
cursor:Entry()
cursor:EntryKey()
cursor:GoTo(*entry*)
cursor:GoToKey(*key*)
cursor:IsValid()
MapCursor(*cursor*, *function*)
cursor:Move(*n*)
cursor:Next()
cursor:Prev()
cursor:Reset()
cursor:ResetToEnd()
cursor:Status()
cursor:WhichEnd()

Entries

These functions allow you to manipulate individual soup entries.

EntryChangeXmit(*entry*, *changeSym*)
EntryCopyXmit(*entry*, *newSoup*, *changeSym*)
EntryFlushXmit(*entry*, *changeSym*)
EntryIsResident(*entry*)
EntryModTime(*entry*)
EntryMoveXmit(*entry*, *newSoup*, *changeSym*)
EntryRemoveFromSoupXmit(*entry*, *changeSym*)
EntryReplaceXmit(*original*, *replacement*, *changeSym*)

`EntrySize(`*entry*`)`
`EntrySoup(`*entry*`)`
`EntryStore(`*entry*`)`
`EntryTextSize(`*entry*`)`
`EntryUndoChanges(`*entry*`)`
`EntryUniqueId(`*entry*`)`
`FrameDirty(`*frame*`)`
`IsSameEntry(`*entryOralias1*`, `*entryOralias2*`)`

Entry Aliases

These functions allow you to create and manipulate entry aliases.

`IsEntryAlias (`*object*`)`
`MakeEntryAlias(`*entry*`)`
`ResolveEntryAlias(`*alias*`)`

Virtual Binary Objects

`GetVBOCompander(`*vbo*`)`
`GetVBOStore(`*vbo*`)`
`GetVBOStoredSize(`*vbo*`)`
`IsVBO(`*vbo*`)`
store`:NewVBO(`*class*`, `*size*`)`
store`:NewCompressedVBO(`*class*`, `*size*`, `*companderName*`, `*companderArgs*`)`

Data Backup and Restore Functions

These functions are intended for use by special-purpose data backup and restoration programs only. Many of them intentionally defeat the error-checking features upon which the system relies to maintain values that identify entries to the system and specify when they were last modified.

store`:Erase( )`
store`:GetAllInfo()`

store:SetAllInfoXmit(*slotSymbol*, *value*, *changeSym*)
store:SetSignature(*signature*)

soup:AddWithUniqueIdXmit(*entry*, *changeSym*)
soup:GetAllInfo()
soup:GetNextUid()
soup:SetSignature(*signature*)
soup:SetAllInfoXmit (*frame*, *changeSym*)
EntryChangeWithModTimeXmit(*entry*, *changeSym*)
EntryReplaceWithModTimeXmit (*original*, *replacement*, *changeSym*)

Mock Entries

The global functions described here create and manipulate mock entries. They do not work on normal soup entries.

EntryCachedObject(*mockEntry*)
EntryHandler(*mockEntry*)
EntrySetCachedObject(*mockEntry*, *newCachedObj*)
EntrySetHandler(*mockEntry*, *newHandler*)
IsMockEntry(*object*)
NewMockEntry(*handler*, *cachedObj*)

Application-Defined Mock Entry Handler Methods

Application developers must implement the following mock entry handler methods themselves.

handler:EntryAccess(*mockEntry*)
handler:EntryChange(*mockEntry*)
handler:EntryChangeWithModTime(*mockEntry*)
handler:EntryCopy(*mockEntry*, *newSoup*)
handler:EntryModTime(*mockEntry*)
handler:EntryMove(*mockEntry*, *newSoup*)
handler:EntryRemoveFromSoup(*mockEntry*)
handler:EntryReplace(*original*, *replacement*)
handler:EntryReplaceWithModTime(*original*, *replacement*)
handler:EntrySize(*mockEntry*)
handler:EntrySoup(*mockEntry*)
handler:EntryStore(*mockEntry*)
handler:EntryTextSize(*mockEntry*)
handler:EntryUndoChanges(*mockEntry*)
handler:EntryUniqueID(*mockEntry*)
handler:EntryValid(*mockEntry*)

CHAPTER 12

Drawing and Graphics

This chapter describes how to draw graphical objects such as lines and rectangles in Newton applications.

You should read this chapter if you are attempting to draw complex or primitive graphical objects in a view. Before reading this chapter, you should be familiar with the information in Chapter 3, “Views.”

This chapter describes:

- the types of graphical objects supported and how to draw objects.
- drawing methods and functions used to perform specific tasks.
- drawing classes and protos that operate on graphics and drawing methods and functions.

About Drawing

Drawing provides a number of functions, methods, and protos that allow you to create graphical objects in Newton applications. Objects can be shapes, pictures, or rendered bitmaps. Additional functions and methods provide ways to scale, transform, or rotate the images. All objects are drawn into views. See “View Instantiation” on page 3-36 for complete details.

This section provides detailed conceptual information on drawing functions and methods. Specifically, it covers the following:

- supported shape objects
- the style frame
- new functions, methods, and messages added for 2.0 as well as modifications to existing pieces of the drawing code.

Note that for all of the functions described in this chapter:

- the coordinates you specify are interpreted as local to the view in which the object is drawn
- the origin of the coordinate plane (0,0,) is the upper-left corner of the view in which the object is drawn
- positive values are towards the right or towards the bottom of the screen from the origin. For additional information on the Newton coordinate system see the section “Coordinate System” beginning on page 3-8 in Chapter 3, “Views.”

Shape-Based Graphics

Newton system software provides functions for drawing primitive graphical objects in a view. These drawing functions return a data structure called a **shape** that is used by the drawing system to draw an image on the screen. The drawing system supports the following shape objects:

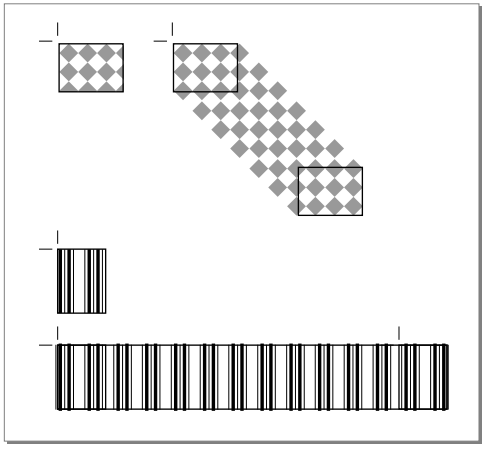
- lines
- rectangles
- rounded rectangles
- ovals (including circles)
- polygons
- wedges and arcs
- regions
- text
- pictures
- bitmaps

Complex graphics can be drawn by passing arrays of shapes to the various drawing functions. Primitive shapes can be combined procedurally by collecting them into a shape called a picture. The appearance will be the same except that, when drawn, the picture will not be affected by any style specifications. The styles are recorded into the picture when you make it with `MakePict`—with the exception of any transform or clipping slot. See “Controlling Clipping” on page 12-18 and “Transforming a Shape” on page 12-19 for more information.

Each type of shape is described on the following pages.

A **line** is defined by two points: the current x and y location of the graphics pen and the x and y location of its destination. The pen hangs below the right of the defining points, as shown in Figure 12-1, where two lines are drawn with two different pen sizes.

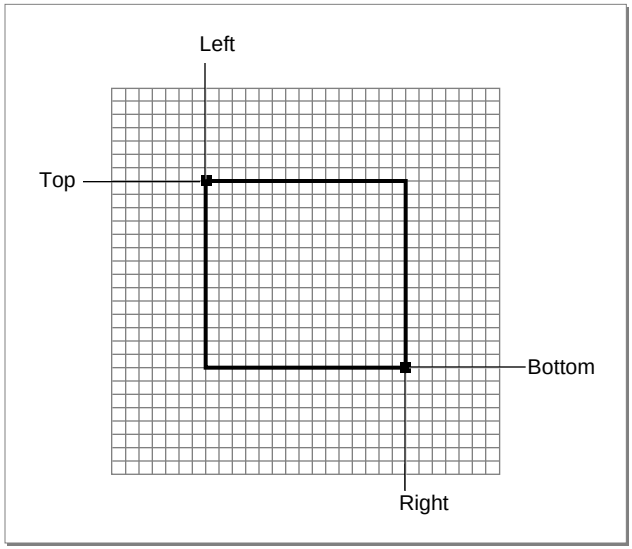
Figure 12-1 A line drawn with different bit patterns and pen sizes



A **rectangle** can be defined by two points—its top-left and its bottom-right corners, as shown in Figure 12-2, or by four boundaries—its upper, left, bottom, and right sides. Rectangles are used to define active areas on the screen, to assign coordinate systems to graphical entities, and to specify the locations and sizes for various graphics operations.

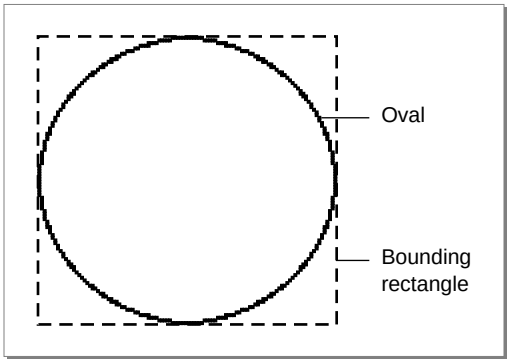
Drawing also provides functions that allow you to perform a variety of mathematical calculations on rectangles—changing their sizes, shifting them around, and so on.

Figure 12-2 A rectangle



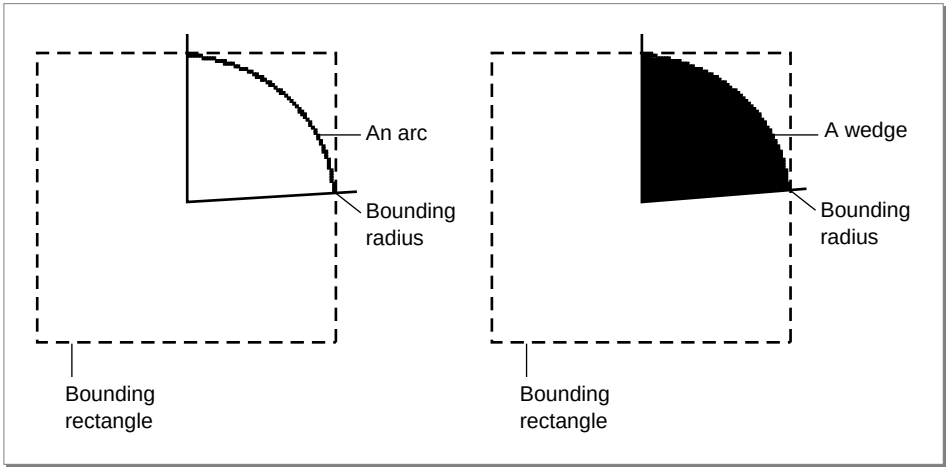
An **oval** is a circular or elliptical shape defined by the bounding rectangle that encloses it. If the bounding rectangle is a square (that is, has equal width and height), the oval is a circle, as shown in Figure 12-3.

Figure 12-3 An oval



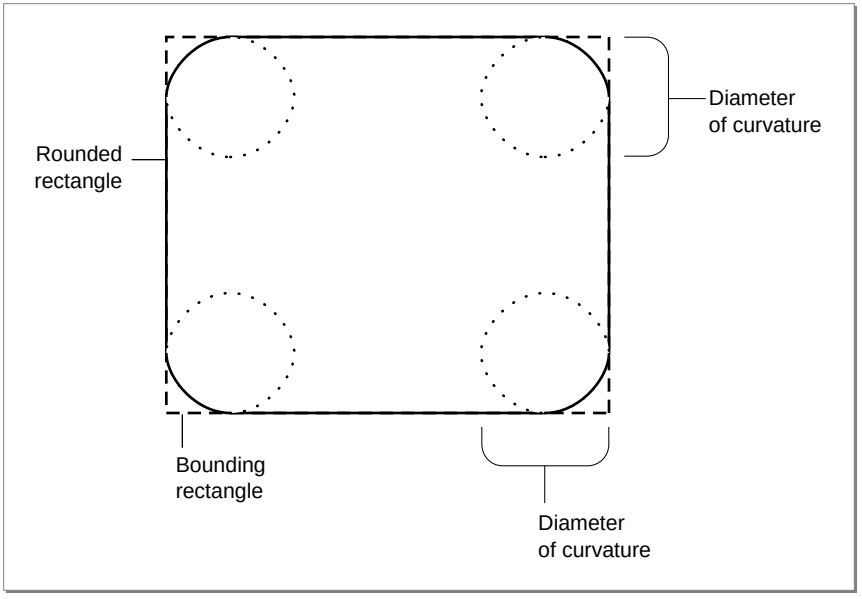
An **arc** is a portion of the circumference of an oval bounded by a pair of radii joining at the oval's center; a wedge includes part of the oval's interior. Arcs and wedges are defined by the bounding rectangle that encloses the oval, along with a pair of angles marking the positions of the bounding radii, as shown in Figure 12-4.

Figure 12-4 An arc and a wedge



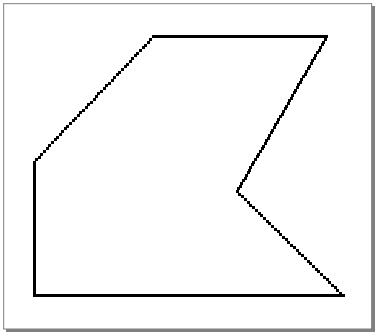
A **rounded rectangle** is a rectangle with rounded corners. The figure is defined by the rectangle itself, along with the width and height of the ovals forming the corners (called the diameters of curvature), as shown in Figure 12-5. The corner width and corner height are limited to the width and height of the rectangle itself; if they are larger, the rounded rectangle becomes an oval.

Figure 12-5 A rounded rectangle



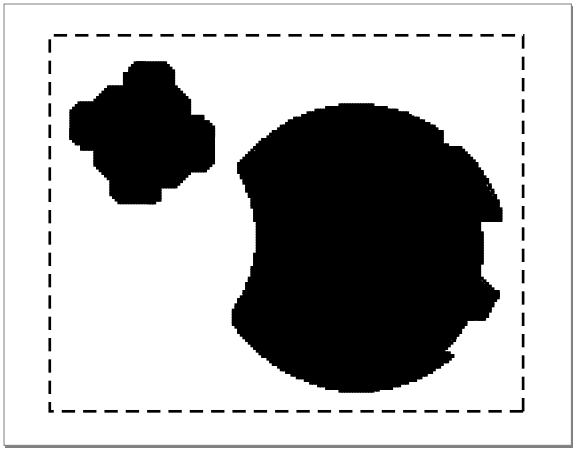
A **polygon** is defined by a sequence of points representing the polygon's vertices, connected by straight lines from one point to the next. You define a polygon by specifying an array of x and y locations in which to draw lines and passing it as a parameter to `MakePolygon`. Figure 12-6 shows an example of a polygon.

Figure 12-6 A polygon



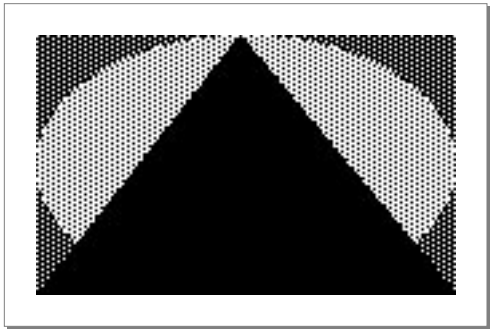
A **region** is an arbitrary area or set of areas, the outline of which is one or more closed loops. One of drawing's most powerful capabilities is the ability to work with regions of arbitrary size, shape, and complexity. You define a region by drawing its boundary with drawing functions. The boundary can be any set of lines and shapes (even including other regions) forming one or more closed loops. A region can be concave or convex, can consist of one connected area or many separate areas. In Figure 12-7, the region consists of two unconnected areas.

Figure 12-7 A region



Your application can record a sequence of drawing operations in a **picture** and play its image back later. Pictures provide a form of graphic data exchange: one program can draw something that was defined in another program, with great flexibility and without having to know any details about what's being drawn. Figure 12-8 shows an example of a picture containing a rectangle, an oval, and a triangle.

Figure 12-8 A simple picture



Manipulating Shapes

In addition to drawing shapes, you can perform operations on them. You can

- **offset** shapes; that is, change the location of the origin of the shape's coordinate plane, causing the shape to be drawn in a different location on the screen. Note that offsetting a shape modifies it; for example, the offset shape will have different `viewBounds` values than the original shape.
- **scale** shapes; that is, draw the shape to fill a destination rectangle of a specified size. The destination rectangle can be larger, smaller, or the same size as the original shape. Note that scaling a shape modifies it; for example, the scaled shape has different `viewBounds` values than the original shape.
- **hit-test** shapes to determine whether a pen event occurred within the boundaries of the shape. This operation is useful for implementing button-like behavior in any shape.

The Style Frame

Any shape can optionally specify characteristics that affect the way it is imaged, such as the size of the pen or the fill pattern to be used. These characteristics are specified by the values of slots in a style frame associated with the shape. If the value of the style frame is `nil`, the view system draws

the shape using default values for these drawing characteristics. See “Style Frame” on page 12-31 for complete details.

Drawing Compatibility

The following new functionality has been added for the 2.0 release.

New Functions

The following functions have been added:

- `GetShapeInfo`—Returns a frame containing slots of interest for the shape. See page 12-63 for details.
- `DrawIntoBitmap`—Draws shapes into a bitmap in the same way that the `DrawShape` method draws shapes into a view. See page 12-51 for details.
- `MakeBitmap`—Returns a blank (white) bitmap shape of the specified size. See page 12-49 for details.
- `MungeBitmap`—Performs various destructive bitmap operations such as rotating or flipping the bitmap. See page 12-52 for details.
- `ViewIntoBitmap`—Provides a screen-capture capability, writing a portion of the specified view into the specified bitmap. See page 12-53 for details.

New Style Attribute Slots

Version 2.0 of Newton system software supports two new slots in the style frame; they are the `clipping` slot and the `transform` slot. See “Controlling Clipping” on page 12-18 and “Transforming a Shape” on page 12-19 for details.

Changes to Bitmaps

Previous versions of Newton system software treated bitmaps very statically: they were created only from compile-time data, and the operations one could perform on them were limited to drawing them.

Version 2.0 of Newton system software provides a more dynamic treatment of bitmaps. You can dynamically create and destroy them, draw into them, and perform operations on them such as rotating and flipping them. This more flexible treatment of bitmaps allows you to use them as offscreen buffers as well as for storage of documents such as fax pages. See “Bitmap Functions” on page 12-49 for details.

Changes to the HitShape Method

Previous versions of `HitShape` returned a non-`nil` value if a specified point lies within the boundaries of one or more shapes passed to it. Version 2.0 of the `HitShape` function now returns additional information. See “Hit-Testing Functions” on page 12-54 for details.

Changes to View Classes

The `icon` slot of a view of the `clPictureView` class can now contain a graphic shape, in addition to bitmap or picture objects. See “Picture View” on page 12-34 for details.

Using Drawing and Graphics

This section describes how to use drawing and graphics to perform specific tasks. See “Drawing Reference” beginning on page 12-31 for descriptions of the functions and methods discussed in this section.

How To Draw

Drawing on the Newton screen is a simple two-part process. You first create a shape object by calling one or more graphics functions, such as `MakeRect` (page 12-59), `MakeLine` (page 12-56), and so on. You then draw the shape object by passing any of the shapes returned by the shape-creation functions, or an array of such shapes optionally intermixed style frames to the `DrawShape` (page 12-64) method. If a style frame is included in the shape array, it applies to all subsequent shapes in the array, until overridden by another style frame.

In addition to the shape object, the `DrawShape` method accepts a **style frame** parameter. The style frame specifies certain characteristics to use when drawing the shape, such as pen size, pen pattern, fill pattern, transfer mode, and so on.

This system is versatile because it separates the shapes from the styles with which they are drawn. You can create a single shape and then easily draw it using different styles at different times.

`DrawShape` can also accept as its argument an array of shapes instead of just a single shape. Therefore, you can create a series of shapes and draw them all at once with a single call to the `DrawShape` method. Additional style frames can be included in the shape array to change the drawing style for the shapes that follow them. The section “Using Nested Arrays of Shapes” on page 12-16, discusses the use of arrays of shapes in more detail.

Responding to the ViewDrawScript Message

When the system draws the view, it sends a `ViewDrawScript` message (page 3-132) to the view. To perform your own drawing operations at this time, you must provide a `ViewDrawScript` method that calls the appropriate drawing functions.

The system also sends the `ViewDrawScript` message to the view whenever it is redrawn. Views may be redrawn as the result of a system notification or a user action.

If you want to redraw a view explicitly at any particular time, you need to send the `Dirty` message (page 3-83) to it. This message causes the system to add that view to the area of the screen that it updates in the next event loop cycle. To cause the update area to redraw before the next event loop cycle, you must send the `RefreshViews` (page 3-84)function after the `Dirty` message.

Drawing Immediately

If you want to draw in a view at times other than when the view is opened or redrawn automatically, you must execute drawing code outside of the `ViewDrawScript` method. For example, you might need to perform your own drawing operations immediately when the user taps in the view.

You can use the `DoDrawing` method (page 12-70) for this purpose. The `DoDrawing` method calls another drawing method that you supply as one of its arguments.

WARNING

Do not directly use `DrawShape` to draw shapes outside of your `ViewDrawScript`. Standard drawing in `ViewDrawScript` and `DoDrawing` automatically set up the drawing environment. If you use `DrawShape` without setting up the drawing environment, your application could accidentally draw on top of other applications, keyboards, or floaters. ◆

Using Nested Arrays of Shapes

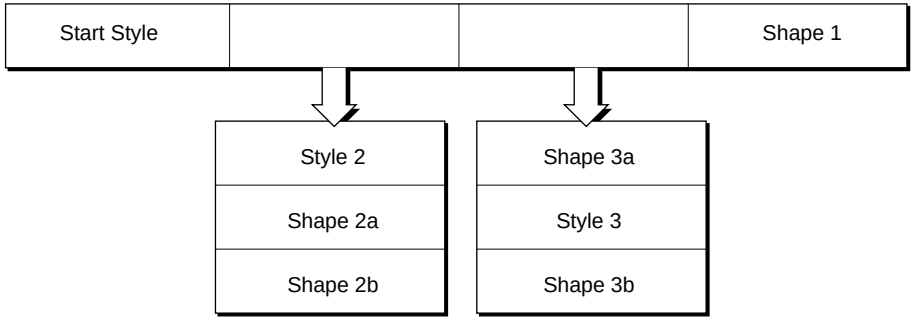
The `DrawShape` method can draw multiple shapes when passed an array of shapes as its argument. Style frames may be included in the shape array to change the drawing style used to image subsequent elements of the array. Each element of the array can itself be an array as well; this section refers to such an array as a **nested array**.

Styles are maintained on a per-array basis in nested arrays, and the *startStyle* parameter of `DrawShape` is always treated as though it were the first array element of the top most array. Therefore, compound shapes and multiple styles remain intact when nested arrays are combined into larger groupings.

When the `DrawShape` method processes a nested array, the shapes are drawn in ascending element order and drawing begins with the style of the parent array. Although the drawing style may change while processing the elements of an individual array, that style applies only to the elements of that particular array. Therefore, if an array happens to be an element of another array—that is, a nested array— style changes in the nested array affect the processing of its subsequent elements but the drawing style of the parent array is restored after the last element of the nested array is processed.

For example, you might nest arrays to create the hierarchy of shapes and styles depicted in Figure 12-9.

Figure 12-9 Example of nested shape arrays



If the nested shape array depicted in Figure 12-9 were passed to the `DrawShape` function, the results summarized in Table 12-1 would occur.

Table 12-1 Summary of drawing results

Shape	Style
2a	2
2b	2
3a	startStyle
3b	3
1	startStyle

The Transform Slot in Nested Shape Arrays

Within a single shape array, the `transform` slot is treated like a style frame: only one transform is active per array; if another transform is specified within the array, the previous transform is overridden. Within nested arrays, however, the `transform` slot is treated a little differently than most style slots. As the `DrawShape` method descends into nested arrays of shapes, changes to the `transform` slot are cumulative; the resulting transform is the net sum of all the transforms in the hierarchy. For example, if in Figure 12-9 *startStyle* has a transform of 10,10 and style3 has a transform 50,0 then shapes 2a, 2b, 1, 3a would be drawn offset by 10,10 but shape 3b would be drawn offset by 60,10.

Default Transfer Mode

The default transfer mode is actually a split state: bitmaps and text are drawn with a `modeOR` transfer mode, but other items (geometric shapes, pens, and fill patterns) are drawn with a `modeCOPY` transfer mode. However, when you actually specify a transfer mode (with a non-`nil` value in the `transferMode` slot of the style frame), all drawing uses the specified mode.

Transfer Modes At Print Time

Only a few transfer modes are supported for printing. Only `modeCOPY`, `modeOR`, and `modeBIC` may be used; other modes may produce unexpected results.

Note
Most problems occur when using PostScript printers, so you should test your code on LaserWriters as well as StyleWriters. ◆

Controlling Clipping

When the system draws a shape in a view for which the `vClipping` flag is set, it draws only the part that fits inside the view in which drawing takes place. Any parts of the shape that fall outside the boundaries of that view are not drawn, as if they have been cut off or clipped. The term **clipping** refers to this view system behavior; in common usage, the shape is said to have been "clipped to the destination view."

Note
Although the view system allows drawing outside the boundaries of a view for which the `vClipping` flag is not set, it does not guarantee that drawing outside the boundaries of the view will occur reliably. You need to make your destination view large enough to completely enclose the shapes you want to draw. You could also set the destination view's `vClipping` flag to clip drawing to the bounds of the destination view. Note also that an application base view that is a child of the root view always clips drawing to its boundaries. ◆

When no other clipping region is specified and `vClipping` is set, the boundaries of the destination view define the region outside of which drawing does not occur. This area is known as the **clipping region**. If you want to specify different clipping regions, you can use the style frame's `clipping` slot to do so. Because drawing is always clipped to the

boundaries of the destination view, regardless of any other clipping region you specify, you cannot use the `clipping` slot to force drawing outside the boundaries of a view.

If the style frame includes a `clipping` slot, the drawing of all shapes affected by this style frame is clipped according to the value of the `clipping` slot. If the value of the `clipping` slot is `nil` or if the `clipping` slot is not supplied, the clipping behavior of the destination view is used.

If the `clipping` slot contains a region shape, that region is used as the clipping boundary for drawing operations affected by this style frame. If the `clipping` slot contains an array of shapes or regions, the system passes the contents of the `clipping` slot to the `MakeRegion` (page 12-60) function to automatically create a new clipping region from the contents of this slot.

Note
Although putting an array of shapes in the `clipping` slot may seem convenient, it significantly increases the time required to process the style frame. For best performance from the view system, do not use this shortcut in style frames that are used repeatedly. ◆

Transforming a Shape

The `transform` slot changes the size or location of a shape without altering the shape itself. It accepts an array specifying an `(x,y)` coordinate pair or a pair of rectangles. The `[x,y]` coordinate arguments relocate a shape by specifying an offset from the origin of the destination view's coordinate plane. The rectangle arguments specify a mapping of the source and destination views that alters both the size and location (offset) of the source view when it is drawn in the destination view.

The rectangle arguments work the same way as the parameters to the `ScaleShape` (page 12-65) function (although transforms won't accept `nil` for the boundaries of the source rectangle): the size of the shape is changed proportionately according to the dimensions of the destination rectangle, and the coordinates of the destination rectangle can also be used to draw the shape in a new location.

The following code fragments demonstrate the use of offset coordinates and mapping rectangles as the value of the `transform` slot.

```
   transform: [30,50],// offset shapes by 30 h and 50 v
```

or

```
   transform:
[SetBounds(0,0,100,100),SetBounds(25,25,75,75)],
// half width and height, centered in relation to
// the original object(not the view) assuming that
// the first rect actually specified correct bounds
```

Using Drawing View Classes and Proto Templates

Five view classes and three proto templates, which you can use to create your own templates, are built into the system. The view classes include:

- `clPolygonView` (page 12-33)—displays polygons or ink, or accepts graphic or ink input.
- `PointsToArray` (page 12-73)—returns an array of data extracted from a polygon shape binary object.
- `ArrayToPoints` (page 12-74)—converts an array of points to a binary object of the class `'polygonShape` (as found in the `points` slot of a `clPolygonView`).
- `clPictureView` (page 12-34)—displays a bitmap or picture object shape.
- `clRemoteView` (page 12-35)—displays a scaled image of another view.

The proto templates include:

- `protoImageView` (page 12-35)—provides a view in which you can display magnify, scroll, and annotate images.
- `protoThumbnail` (page 12-45)—used in conjunction with a protoImageView. It displays a small copy of the image with a rectangle representing the location and panel in the image.

- `protoThumbnailFloater` (page 12-48)—provides a way to use a thumbnail, but also adjusts the thumbnail's size to reflect the aspect ratio of the image that it contains.

Displaying Graphics Shapes

Use the `clPolygonView` (page 12-33) class to display polygons or ink, or to accept graphic or ink input. The `clPolygonView` class includes these features:

- Shape recognition and editing, such as stretching of shapes from their vertices, view resizing, scrubbing, selection, copying to clipboard, duplicating, and other gestures, as controlled by the setting of the `viewFlags` slot.
- Snapping of new line endpoints to nearby vertices and midpoints of existing shapes.
- Automatic resizing to accommodate enlarged shapes (when the view is enclosed in a `clEditView`). This feature is controlled by the `vCalculateBounds` flag in the `viewFlags` slot.

Views of the `clPolygonView` class are supported only as children of views of the `clEditView` class. In other words, you can put a `clPolygonView` only inside a `clEditView`.

You don't need to create polygon views yourself if you are accepting user input inside a `clEditView`. You simply provide a `clEditView` and when the user draws in it, the view automatically creates polygon views to hold shapes.

Displaying Scaled Images of Other Views

Use the `clRemoteView` view class to display a scaled image of another view. This can be used to show a page preview of a full-page view in a smaller window, for example.

The view that you want to display inside of the remote view should be specified as the single child of the remote view. This child is always hidden, and is used internally by the remote view to construct the scaled image.

A clRemoteView should never have more than one view, the scaled view, otherwise the results are undefined and subject to change.

Here is an example of a view definition of the clRemoteView class:

```
myRemoteView := {...
   viewclass: clRemoteView,
   viewBounds: {left: 75, top: 203, right: 178,
               bottom: 322},
   viewFlags: vVisible+vReadOnly,
   viewFormat: nil,
   ViewSetupFormScript: func()
      begin
      // aView is the view to be scaled
      self.stepchildren := [aView];
      end,
...};
```

Translating Data Shapes

You can use the global functions PointsToArray (page 12-73) and ArrayToPoints (page 12-74) to translate points data between a polygon shape and a NewtonScript array.

Finding Points within a Shape

Use the HitShape function (page 12-54) to determine whether a pen event occurred within the boundaries of the shape. This operation is useful for implementing button-like behavior in any shape. Possible results returned by the HitShape function include:

```
nil   // nothing hit
true  // the primitive shape passed was hit
[2,5] // X marks the shape hit in the following array
      // shape := [s,s,[s,s,s,s,s,X,s],s,s]
```

You can retrieve the shape by using the value returned by the `HitShape` method as a path expression, as in the following code fragment.

```
result := HitShape(shape,x,y);
if result then // make sure non-nil
   begin
   if IsArray(result) then // its an array path
      thingHit := shape.(result);
   else
      thingHit := shape;// its a simple shape
   end
```

Although the expression `shape.(result)` may look unusual, it is perfectly legitimate NewtonScript; for further explanation of this syntax, see the "Array Accessor" discussion on page 2-10 of *The NewtonScript Programming Language*.

Using Bitmaps

You can dynamically create and destroy bitmaps, draw into them, and perform operations on them such as rotating, flipping, and sizing them. This flexible treatment of bitmaps allows you to use them as offscreen buffers as well as for storage of documents such as fax pages.

You can create and use bitmap images with the drawing bitmap functions. To create a bitmap you first allocate a bitmap that will contain the drawing with the `MakeBitmap` function (page 12-49). You then create a shape with a shape creation function (page 12-55). `DrawIntoBitmap` (page 12-51) then takes the drawing and draws it into the bitmap. The final step is to draw the bitmap on the Newton screen with the `DrawShape` function (page 12-64).

The following example shows how to draw a bitmap. It creates a bitmap by drawing a shape and then draws it onto the screen. This example then rotates the shape, scales it, and redraws it on the Newton.

```
bitmapWidth := 90;
bitmapHeight := 120;
```

```
vfBlack := 5;

// allocate a new bitmap
bitmap := MakeBitmap(bitmapWidth, bitmapHeight, nil);

// make a shape and draw it into the bitmap
shapes := MakeOval(0, 0, 50, 75);
DrawIntoBitmap(shapes, {fillPattern: blackFill}, bitmap);

// draw the bitmap
GetRoot():DrawShape(bitmap, {transform: [100, 100]});

// Rotation is a destructive operation:  it replaces the
// old bitmap with the new rotated bitmap.
MungeBitmap(bitmap, 'rotateRight, nil);

// translate and scale the bitmap
fromRect := SetBounds(0, 0, bitmapWidth, bitmapHeight);
toRight := 100 + floor(bitmapWidth * 1.25);
toBottom := 200 + floor(bitmapHeight * 1.25);

toRight := 100 + bitmapWidth * 5 div 4;
toBottom := 200 + bitmapHeight * 5 div 4;

toRect := SetBounds(100, 200, toRight, toBottom);

// draw the bitmap again
GetRoot():DrawShape(bitmap, {transform: [fromRect,
toRect]});
```

Making CopyBits Scale Its Output Bitmap

`CopyBits` (page 12-70) uses the bounds of the bitmap passed to it to scale the bitmap that it draws; so, by changing the bounds of the bitmap passed to `CopyBits`, you can cause this method to scale the bitmap it draws. If you

want to scale the output bitmap without changing the bounds of the original, call `ScaleShape` (page 12-65) on a clone of the original bitmap and pass the modified clone bitmap to the `CopyBits` method.

Storing Compressed Pictures and Bitmaps

NTK supports limited compression of pictures and bitmaps. If you store your package compressed (using the “optimize for space” setting), then all items in your package are compressed in small (approximately 1 K) pages, rather than object by object.

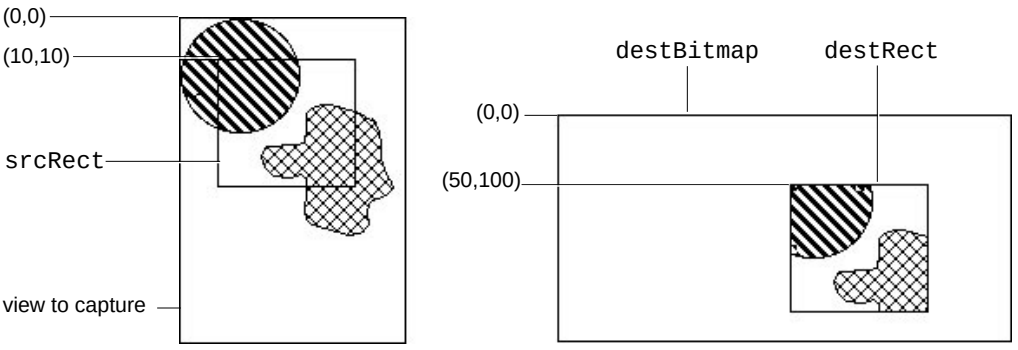
You can use the NTK compile-time function `GetNamedResource` to get a Macintosh PICT resource that can be drawn on the Newton in `clPictureViews`. PICT resources are generally smaller than bitmap frames because each bitmap within the PICT resource contains compressed bitmap data.

Note
This information applies to the Mac OS version of NTK, the Windows version differs; see the Newton Toolkit User’s Guide for details. ◆

Capturing a Portion of a View Into a Bitmap

Use the `ViewIntoBitmap` method (page 12-53) to capture a portion of a specified view into a specified bitmap. This function does not provide scaling capability, although scaling can be accomplished by passing the *destBitmap* bitmap returned by this method to the `DrawIntoBitmap` function as the value of its *shape* parameter. Figure 12-10 shows the relationships between the view to be captured, the source rectangle, the destination bitmap, and the destination rectangle.

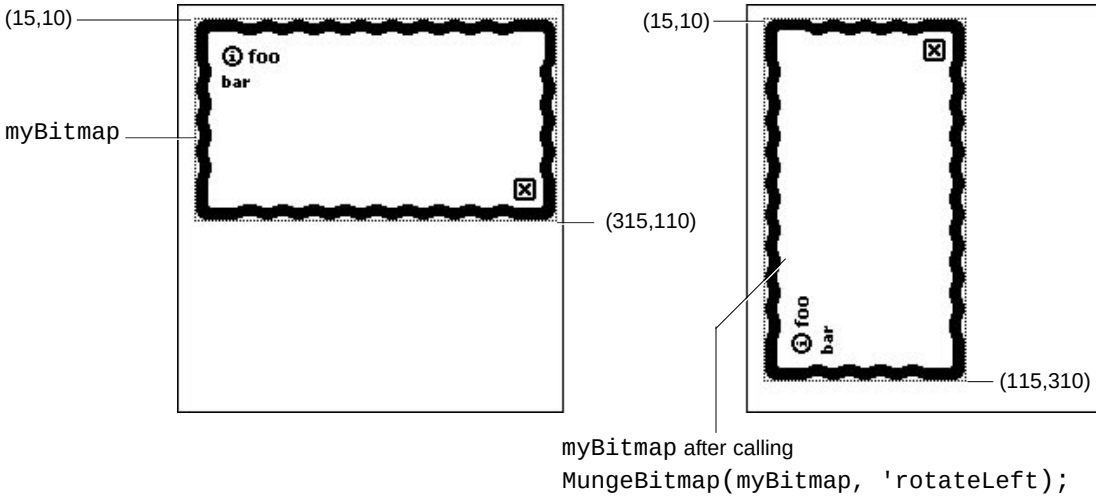
Figure 12-10 Example of `ViewIntoBitmap` method



Rotating or Flipping a Bitmap

Use the `MungeBitmap` function (page 12-52) to perform various bitmap operations such as rotating or flipping the bitmap. These operations are destructive to the bitmap passed as an argument to this function; the bitmap is modified in place and the modified bitmap shape is returned. Figure 12-11 illustrates how the `MungeBitmap` function works. See page 12-23 for a code example.

Figure 12-11 Example of `MungeBitmap` method



Importing Macintosh PICT Resources

The following information applies to the Mac OS version of NTK, the Windows version differs; see the Newton Toolkit User's Guide for details.

A Macintosh PICT resource can be imported into the Newton in two ways: as a bitmap or as a picture object. A Macintosh PICT resource is stored much more compactly on the Newton as a picture object; however, it may be slower to draw than a bitmap. The same Macintosh PICT resource may occupy much more space when imported as a bitmap, but may draw significantly faster. The method you should use depends on whether you want to optimize for space or speed.

A Macintosh PICT resource is imported as a bitmap by using the slot editor for the `icon` slot (an editor of the picture type). Alternatively, the resource can be imported as a picture object by using the `GetResource` or `GetNamedResource` compile-time functions available in NTK. In this case,

you must use an `AfterScript` slot to set the value of the `icon` slot to the picture object obtained by one of these resource functions.

Note
The constant `clIconView` can also be used to indicate a view of the `clPictureView` class (page 12-34). These two constants have identical values. ◆

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

```
aPicture := {...
   viewClass: clPictureView,
   viewBounds: {left:0, top:75, right:150, bottom:175},
   viewFlags: vVisible+vClickable,
   icon: myPicture,
...}
```

Drawing Non-Default Fonts

You can draw a font other than the default font by putting the font specifier style frame close to the text shape so that another style frame won't override it. Use either `DrawShape` (page 12-64) or `MakePict` (page 12-61).

There are several places where it might seem reasonable to put the style frame with the font specifier. `DrawShape` takes a style argument, so you could place it there:

```
:DrawShape(myText, {font: '{family: someFont, face: 0,
size: 9 }});
```

You can also embed a style frame in an array of shapes:

```
:DrawShape {font: ...}, myText, shape ], nil);
```

`MakePict` also could be used (this is from the Font Scaling example):

```
myText := MakePict([{penpattern: 0, (1)font: ...}, rect,
{(2)font: ...}, txtshape], {(3)font: ...});
```

You can also set the font in locations (1) or (2) with `MakePict`. In this case the font gets “encapsulated” into the PICT.

If the `{penpattern}` frame was not present in the picture shape, any of the above places should suffice to set the font.

PICT Swapping During Run-Time Operations

To set a default picture for a `clPictureView`, use NTK’s picture slot editor to set the `icon` slot of the `clPictureView`. You may select a PICT resource from any resource file that has been added to your project. The picture will be converted on the Macintosh from a type 1 or 2 PICT into a bitmap, and stored in your package at compile time. To change this picture at run time, you need to keep a reference to each alternate picture or bitmap. This is done using `DefConst` at compile time in a text file as follows:

```
OpenResFile(HOME & "Photos Of Ralph.rsrc");
// Here we convert a PICT 1 or PICT 2 into a BitMap.
// This is what NTK's picture slot editor does.
DefConst('kPictureAsBitMap,
         GetPictAsBits("Ralph", nil));

// Here the picture is assumed to be in PICT 1 format.
// If it is not, the picture will not draw and you may
// throw exceptions when attempting to draw the object.
DefConst('kPictureAsPict,
         GetNamedResource("PICT", "Ralph", 'picture));

// Verify this is a Format 1 PICT object!
if ExtractWord('kPictureAsPict, 10) <> 0x1101 then
   print("WARNING: Ralph is not a Format 1 PICT
resource!");
```

```
// This is one way to get the picture's bounds
information.
// You can also extract it from the picture's own bounds
// rectangle at either compile time or run-time, by using
// ExtractWord to construct each slot of a bounds frame.
DefConst('kPictureAsPictBounds,
         PictBounds("Ralph", 0, 0));

CloseResFile();
```

Notice there are two types of pictures: bitmaps (a frame with bits, bounds, and mask slots) and Format 1 PICTs (binary objects of class picture). `clPictureViews` (page 12-34) can draw both of these types of objects, so you just need to choose a format and use `SetValue` on the `icon` slot, as follows:

```
// SetValue(myView, 'icon, kPictureAsBitMap);
   SetValue(myView, 'icon, kPictureAsPict);
```

Optimizing Drawing Performance

You can use several methods to make drawing functions execute faster.

If you have a fairly static background picture, you can use a predefined PICT resource. Create the PICT in your favorite drawing program, and use the PICT as the background (`clIconView`). The graphics system also has a picture making function that enables you to create pictures that you can draw over and over again.

If you want to improve hit-testing of objects, use a larger view in combination with `ViewDrawScripts` or `ViewClickScripts` rather than using smaller views with individual `ViewClickScripts`. This is especially true of a view that consists of regular smaller views.

Drawing Reference

This section describes the protos, functions, and methods used by the drawing interface.

Data Structure

Drawing has the following data structure.

Style Frame

The style frame contains one or more of the slots listed here. If any single slot is not provided, the default value for that slot is used.

`transferMode` The drawing transfer mode for the pen (or for the text, if text is being drawn). Specify one of these standard constants: `modeCopy`, `modeOr`, `modeXor`, `modeBic`, `modeNotCopy`, `modeNotOr`, `modeNotXor`, `modeNotBic`. These constants are described in the section "viewTransferMode Constants" on page 3-71 in Chapter 3, "Views." The default transfer mode is a split state: bitmaps and text are drawn with a `modeOr` transfer mode, but other items (geometric shapes, pens, and fill patterns) are drawn with a `modeCopy` transfer mode. However, when you actually specify a transfer mode (by placing a non-`nil` value in the `transferMode` slot of the style frame), all drawing uses the specified mode.

`penSize` The size of the pen in pixels. You can specify a single integer to indicate a square pen of the specified size, or you can specify an array giving the pen width and height (for example, `[1, 2]`). This value is not used for drawing text. The minimum and default pen size is 1.

	However, no frame will be drawn for a shape if the pen pattern is set to `vfNone` (the default pen pattern is `vfBlack`)
`penPattern`	The pen pattern. You can specify the following patterns: `vfNone`, `vfWhite`, `vfLtGray`, `vfGray`, `vfDkGray`, and `vfBlack`. The default value is `vfBlack`. To use a custom pen pattern, store a binary object of class `'pattern` in this slot. An easy way to create such an object is to clone a binary string containing 16 Unicode hexadecimal digits, set the class of the clone to `'pattern` and store the result in this slot. For more information, see “Custom Fill and Frame Patterns” on page 3-29.
`fillPattern`	The fill pattern. You can specify the same patterns as for the pen. This value is not used for drawing text. The default value is `vfNone`. To use a custom fill pattern, store a binary object of class `'pattern` in this slot. For more information, see “Custom Fill and Frame Patterns” on page 3-29.
`font`	The font to be used for drawing text. The default is the font selected by the user in the Styles palette (stored in the `userFont` slot of the `userConfiguration` frame).
`justification`	The alignment of text in the rectangle specified for it. Specify one of the following symbols: `'left`, `'right`, `'center`. The default value is `'left`.
`clipping`	Specifies a clipping region to which all drawing will be clipped in addition to the default clipping. The value of this slot can be a primitive shape, a region or an array of shapes (from which a new clipping region is constructed automatically by the system). For more information see “Controlling Clipping” on page 12-18.
`transform`	Used to offset or scale the shape. The value of this slot is an array that can hold a coordinate pair or a pair of

source and destination rectangles. For more information see the section “Transforming a Shape” beginning on page 12-19.

View Classes

The following view classes are used to display objects in views.

Shape View (clPolygonView)

`clPolygonView`

Displays polygons, or ink, or accepts graphic or ink input.

These slots are of interest for a view of the `clPolygonView` class:

`viewBounds`	Set to the size of the view and the view location where you want it to appear.
`points`	If the view contains a polygon shape, this slot contains a binary data structure of the type `'polygonShape`, which holds the polygon data. See below for a detailed description of this binary data structure.
`ink`	If the view contains ink, this slot contains a binary data structure of the type `'ink`, which holds the ink data.
`viewFlags`	The default setting is `vVisible`. You will most likely want to set additional flags to control the recognition behavior of the view. For example, `vShapesAllowed`.
`viewFormat`	Optional. The default setting is `vfPen(2)`. The `vfPen` setting controls the thickness of polygon lines.

The `points` slot in a `clPolygonView` is a binary data structure with the class `'polygonShape`.

Picture View

`clPictureView`

The `clPictureView` view class is used to display a picture. A picture can be a bitmap, graphic shape, or picture object.

These slots are of interest for a view of the `clPictureView` class:

`icon`	A bitmap, graphic shape, or picture object to be displayed in the view. A bitmap is selected from a resource file by using the icon slot editor in NTK. A picture object is obtained from a resource file by using the `GetResource` or `GetNamedResource` compile-time functions in NTK.
`viewBounds`	Set to the size of the view and the location where you want it to appear.
`viewFlags`	The default setting is `vVisible`.
`viewFormat`	Optional. The default setting is `nil`.

A picture object is simply a binary object with the class `'picture`.

If the contents of the `icon` slot is a graphic shape, the style frame for drawing the shape in the view contains the single slot `transferMode`. The `transferMode` slot is set to the same value as the `viewTransferMode` slot of the view (if this slot exists), or to the default value `modeCopy` if there is no `viewTransferMode` slot in the view.

Your graphic shape can provide a different set of styles by including a style frame in the shape array. In this case, any `transferMode` slot in the style frame that you specify will override the `viewTransferMode` setting for the view.

Scaled View

viewClass:`clRemoteView`

The `clRemoteView` view class is used to display a scaled image of another view.

These slots are of interest for a view of the `clRemoteView` class:

`stepChildren`	Specify a single child view in this array. This child view will be scaled to fit inside the `clRemoteView`. Typically, you set this slot at run time in the `ViewSetupFormScript` method.
`viewBounds`	Set to the size of the view and the location where it to appear.
`viewFlags`	The default setting is `nil`.
`viewFormat`	Optional. The default setting is `nil`.

Graphics and Drawing Protos

This section describe the protos that work with graphics and drawing. The protos include:

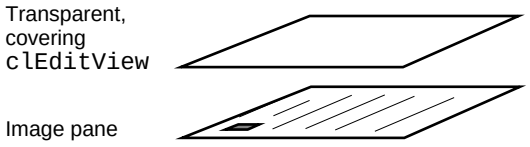
- `protoImageView`
- `protoThumbnail`
- `protoThumbnailFloater`

protoImageView

This proto provides a view in which you can display, magnify, scroll, and annotate images. However, it depends on the use of other protos to provide controls for magnifying, scrolling, and paging.

The structure of the `protoImageView` is shown in Figure 12-12.

Figure 12-12 `protoImageView` Structure



The annotations can be selected and modified when the image is shown at full size.

The image and annotations are clipped so that only the portion of their contents that fall within the bounds of their parent view is shown.

Annotations scroll along with the image.

In general, in this discussion, a reference to the “image” means both the image and the annotation, while the “image plane” refers only to the image. Also, references to the “pane” refer to the bounding box of the `protoImageView`, under the assumption that the image is larger than can be displayed in the box, so the `protoImageView` is a window, or pane, into the larger image. Finally, scaling frequently refers to both size and position of the pane in the image.

You may provide the following slots:

`Image`	This slot should contain a NewtonScript shape. It will be rendered by the image plane. This slot can be either a proto or parent inheritance. This slot is required if the image viewer is not opened with `OpenImage` or `ToggleImage`.
`Annotations`	This slot should either be `nil`, or should contain an array of views appropriate to be added as `viewChildren` to a `clEditView`. This slot can be either a proto or parent inheritance.

Note

This slot is referenced during view setup (see “Setup” on page 12-39 for details) and is not maintained afterwards; to retrieve user annotations, call the `GetAnnotations` method. ◆

`scalingInfo` This slot should either be `nil` or should contain a slot similar to that returned by `GetScalingInfo` (page 12-41). This slot can be provided by either proto or parent inheritance.

You can override the following slots:

`viewBounds` The default is `{top: 88, left: 0, right: 0, bottom: -24}`.

`viewJustify` The default setting is `vjParentFullH + vjParentFullV`.

`viewFlags` The default setting is `vVisible + vClipping + vClickable`.

`viewFormat` The default setting is `vfPen(1)`.

`zoomStops` An array specifying an ordered set of zoom stops, smallest to largest; used by the `ZoomBy` method. If this slot is not provided, it will be initialized to default set. Each item in the set should be either a number or a symbol. If a number, `zoomStops` specifies the fractional size to be displayed, where 1.0 is the size of the original image based on the resolution. If `zoomStops` is a symbol it may be `'fitInWindow`, `'fullSize`, `'fullResolution`, or `'twiceFullResolution`. The minimal default set is `['fullSize, 'twiceFullResolution]`. The symbol `'fullSize` should always be a member of the array.

`dragCorridor` An integer. When dragging the image, clinging to the closest axis when within a specific corridor smooths linear scrolling considerably. This specifies the distance from the closest axis the user must move the pen to break out of that corridor and scroll diagonally. The

default value is 7 (resulting in a 14-pixel corridor along both axis).

`grabbyHand` When appropriate, a picture is painted under the pen on `pendown` to indicate that the image can be dragged. This slot contains the appropriate shape to render. It should have top left = 0, 0. The picture is automatically centered under the pen.

Do not change the following slot:

`declareSelf` This slot is set by default to `'imagebase`. Do not change it.

The following additional slots and methods are used internally. They are listed here so that you don't inadvertently override them.

System slots:

`viewClass`, `declareSelf`, and `ViewSetupDoneScript`

Additional slots

`myImage`, `tempImage`, `tempAnnotes`, `tempScales`, `tempOpen`, `fXOffset`, `fYOffset`, `fMaxX`, `fMaxY`, `cHorMult`, `cVertMult`, `fAnnotateMode`, `handShape`, `usefulSizes`, `currentSize`, `fullSize`, `fZoomedTo`, `quiet`, `CalculateUsefulSizes`, `SetupZoomStops`, `SetupSizes`, `ZoomByDest`, `DoUndo`.

You can override the following methods:

penDown

`penDown(`*strokeUnit*`)`

Used to drag an image. Called by the image view's `ViewClickScript` to handle taps (except when in 'edit mode, see "SetAnnotationMode" on page 12-42). The default script drags the image. You can override the default

to handle the click. Keep in mind that it is not possible to override `ViewClickScript` as `protoImageView` is composed of multiple views, any one of which can be handling the tap.

strokeUnit Unit from the `ViewClickScript` method; contains information describing the interaction of the pen with the display.

ScalingInfoChanged

`ScalingInfoChanged(`*slot*`)`

Called whenever a frame returned by `GetScalingInfo` would change due to some programmatic action; for example, a call to `ZoomTo`, `ScrollBy`, and so on.

slot Value varies depending on the event causing the `GetScalingInfo` call:

`'zoom`	The magnification of the image changed.
`'scroll`	The image was scrolled.
`'dragging`	The image is being dragged by the pen.
`'dragDone`	The image is done being dragged by the pen.

You can invoke the following methods, which provide initialization support:

Setup

`Setup(`*image, annotations, scalingInfo*`)`

This method performs appropriate initialization to display the specified image. This method is typically used after the view is opened to let another image be displayed (for example, when switching pages in a fax). (Note that

the `protoImageView` `ViewSetupDoneScript` method calls `Setup` automatically.)

image	Contains a NewtonScript shape. It will be rendered by the image plane.
annotations	Is either `nil` or contains an array of views appropriate to be added as `viewChildren` to a `clEditView`.
scalingInfo	If specified, *scalingInfo* sets the image to the appropriate magnification and offset.

OpenImage

`OpenImage(`*image*`,` *annotations*`,` *scalingInfo*`)`

Opens and initializes the view displaying the image, annotations, and whatever scaling it was set to. If *scalingInfo* is `nil`, the view will not be changed. Otherwise, the image sets the scaling according to the specified scaling information. If the image is already open the imagery, annotations, and scaling (if specified) are set.

See `Setup` for the parameters.

ToggleImage

`ToggleImage(`*image*`,` *annotations*`,` *scalingInfo*`)`

Opens or closes the view and sets the image, annotations, and scaling information (if specified). If *scalingInfo* is `nil`, the image size does not change. If the image is already open, the image, annotations, and scaling information are set.

See `Setup` (page 12-39) for the parameters.

GetScalingInfo

```
GetScalingInfo()
```

This method takes no parameters. Returns a frame of scaling information. The returned `scalingInfo` frame has the following slots:

`offsetX`	The horizontal offset of the pane within the image (positive).
`offsetY`	The vertical offset of the pane within the image (positive).
`zoomedTo`	The symbol or number representing the current zoom.
`extent`	The bounding box of the image at the current scale.
`viewBox`	The (localbox) bounding box of the pane (never changes).

The following methods provide scrolling and annotation support.

HasAnnotations

```
HasAnnotations()
```

This method takes no parameters. Returns non-`nil` if the displayed image has annotations, `nil` otherwise.

GetAnnotations

```
GetAnnotations()
```

This method takes no parameters. It returns an array of views appropriate to become `clEditView` `viewChildren`. This array represents the current annotations that have been drawn on the `clEditView` annotation layer.

SetAnnotationMode

`SetAnnotationMode(`*theMode*`)`

Sets the annotation display behavior and the behavior when the pen is tapped.

theMode Specifies the mode as follows:

`'hide`	Annotations are not visible and a pen tap results in a drag
`'show`	Annotations are made visible, and a pen tap drags
`'edit`	Annotations are visible and editable.

Note

Due to system limitations, it is not possible to edit annotations at any magnification other than `'fullSize`. If you attempt to `SetAnnotateMode('edit)` while at any other magnification, an exception will be thrown. ◆

GetAnnotationMode

`GetAnnotationMode()`

This method takes no parameters. It returns the symbol representing the current annotation mode.

TargetChanged

`TargetChanged()`

This method takes no parameters. Called when any annotation is added or edited.

CanScroll

`CanScroll()`

This method takes no parameters. It returns a frame indicating the directions in which scrolling is possible. If scrolling is not possible `nil` is returned.

ScrollTo

`ScrollTo (`*x*`, `*y*`)`

Scrolls the image to the coordinates specified. This method returns a non-`nil` value if the image was moved, `nil` if it was not moved (either it was already there, or doing so would have moved the pane past the edge of the image). `ScrollTo` will not scroll the image away from the edge of the view.

x, *y*	The offset of the top-left corner of the pane from the top left corner of the image.

ScrollBy

`ScrollBy(`*deltaX*`, `*deltaY*`)`

Scrolls the image by the specified offset amount, where *deltaX* and *deltaY* indicate how far to move the pane within the image. This method returns a non-`nil` value if the image was moved, `nil` if it was not moved. `ScrollBy` will not scroll the image away from the edge of the view.

deltaX	The horizontal distance in which to scroll the image.
deltaY	The vertical distance in which to scroll the image.

The following methods provide magnification support.

ZoomBy

`ZoomBy(`*direction*`)`

Makes an image larger or smaller as specified by the sizes in the `zoomStops` array. If the current zoom is a number between a pair of stops, the image will be increased to the nearest stop in the direction specified (where a positive number enlarges the image).

For example, if `zoomStops` is:

```
['fitInWindow, 0.24, 0.5, 'fullSize, 2, 4,
'fullResolution, 'twiceFullResolution]
```

The current zoom is 0.35, `ZoomBy`(1) increases the image by 0.5 (that is, half size), `ZoomBy`(2) makes the image `'fullSize`, and so on. `ZoomBy` returns non-`nil` if the zooming was changed.

direction A number of discrete steps by which to zoom the image.

ZoomTo

`ZoomTo`(*imageSize*)

This method changes the size of the image.

imageSize An integer or symbol as described the `scalingInfo` slot on page 12-37.

CanZoomBy

`CanZoomBy`(*imageSize*)

Changes the size of the image as specified by the `zoomStops` array.

imageSize A number of discrete steps by which to zoom the image.

ZoomToBox

`ZoomToBox`(*boundsFrame*)

Resizes the image to the size specified with the *boundsFrame* parameter. Note that you don't need to specify the same aspect ratio as the original image; this method allows you to stretch the image in either dimension.

boundsFrame Specifies the size to which you want the image to resize.

protoThumbnail

This proto is designed to be used in conjunction with a `protoImageView`. It displays a small copy of the image (a “thumbnail” sketch), with a rectangle representing the location of the pane in the image.

In this discussion, the grey box refers to the rectangle representing the location of the pane in the image. Scaling frequently refers to both the size and position of the grey box in the thumbnail.

You may provide the following slots:

`ImageTarget` This slot should point to a view capable of responding to both the `GetScalingInfo` and the `ScrollTo` methods. If this slot is defined, the thumbnail proto does not need to provide the two methods.

`Image` If this slot is present when the image is opened, it is expected to contain a graphic shape or bitmap that will be used to render the background shape—the thumbnail sketch—in the view. If this slot is present, it must not be `nil`.

You can override the following slots to modify the appearance of the grey box or thumbnail:

`viewBounds` The default is `{top: 12, left: -50, right: -2, bottom: -23}`.

`viewJustify` The default setting is `vjParentRightH + vjParentFullV`.

`trackWhileScrolling` If non-`nil`, this slot causes intermediary calls to `ScrollTo` while the grey box is being dragged around the thumbnail. If `nil`, `ScrollTo` is called only when the pen is lifted.

The following additional slots and methods are used internally. They are listed so that you don’t inadvertently override them.

System Slots:

`viewClass`, `viewFlags`, `viewFormat`, `ViewClickScript`, `ViewSetupDoneScript`, `ViewDrawScript`

Additional Slots:

`tempImage`, `thumbnail`, `thumbnailBounds`, `greyBox`, `greyBounds`, `theShape`, `needToUpdate`, `RelocateGreyBox`

You can invoke the following methods:

Setup

`Setup(`*image)*

Prepares the thumbnail to show a new image created from `protoThumbnail`. The image is scaled and rendered into an internal bitmap image. This is useful for large images, as it reduces memory paging.

image The image to be scaled.

OpenThumbnail

`OpenThumbnail(`*image*`)`

Convenience routine to open thumbnails. If *image* is specified and an `image` slot is available, the parameter will take precedence. `OpenThumbnail` internally calls `Setup`.

image The image to display.

ToggleThumbnail

`ToggleThumbnail(`*image*`)`

If the image is open, it is closed. If the image is closed, `ToggleThumbnail` calls `OpenThumbnail`.

image The image to open or close.

Update

`Update()`

This method takes no parameters. It re-renders the thumbnail view, which can be fairly slow, as the grey box is rescaled. This slot is only necessary if the scaling information (the location or magnification) of the source image has changed; generally the standard `Dirty` call should suffice.

GetScalingInfo

```
GetScalingInfo()
```

This method takes no parameters. It must return a frame of scaling information like that returned by the `GetScalingInfo` method of `protoImageView`. The easiest way to do this is simply to call the `GetScalingInfo` method of an instance of a `protoImageView`.

PrepareToScroll

```
PrepareToScroll()
```

This method takes no parameters. It is called immediately before any scrolling is performed to allow you to perform any preparation necessary.

ScrollTo

```
ScrollTo(x, y)
```

Called to scroll the view if a pen down event causes scrolling (the usual case). Again, the easiest way to scroll is to call the `ScrollTo` method of an instance of a `protoImageView`. This method must be provided if the view can be clicked on.

x, *y* — The position of the pen in the thumbnail scaled to the `size` and `extent` slot in the frame returned by `GetScalingInfo`. For example, if the thumbnail is 10x10 and the extent is 100x100, a pen down at position 3,5 in the thumbnail will result in a call to `ScrollTo(30, 50)`.

DoneScrolling

```
DoneScrolling
```

This method is passed no parameters. It is called following the scroll operation to allow any necessary cleanup to be performed.

protoThumbnailFloater

This proto provides a convenient way to use a thumbnail. It follows the same basic conventions as the `protoThumbnail`, with the added benefit of being based on the `protoFloatNGo` proto so that it adjusts its size to reflect the aspect ratio of the image it contains. It will always be as large as possible without getting any larger in either dimension than the original `viewBounds`. Furthermore, it will adjust its bounds so that only the edges farthest away from the parent's closest edge will move; in other words, if the floater is dragged to the top-left, the bottom-right corner will move, while if it is at the bottom-right corner of the parent, only its top-left corner will change.

WARNING

This proto should not be parent full-justified, as this will break the code that adjusts its size. ◆

All of the slots are defined and used identically to the `protoThumbnail`, with the following additions that are used internally:

`maxW`, `maxH`, `ViewSetupFormScript`.

Functions and Methods

The following sections describe drawing functions and methods. This section contains the following topics:

- Functions to handle bitmaps
- Functions to handle hit testing
- Functions to handle creating shapes
- Functions that operate on shapes
- General utility functions

Bitmap Functions

This section describes the bitmap functions and methods.

MakeBitmap

`MakeBitmap (`*widthInPixels, heightInPixels, optionsFrame*`)`

Returns a blank (white) bitmap shape of the specified size. The origin of the bitmap returned is at (0,0); however, you can subsequently use the `OffsetShape` function to modify the returned bitmap's origin.

widthInPixels Width of the bitmap shape.

heightInPixels Height of the bitmap shape.

optionsFrame An optional frame specifying additional characteristics of the bitmap shape created by this method. It can contain any of the slots specified here. If this frame is not used, the value of the *optionsFrame* parameter must be `nil`.

`rowBytes` Specifies the number of bytes per row of the bitmap; use only for a data source that creates scan lines longer than the default value. An `exMakeBitmapBadArgs` exception is thrown if the value of `rowBytes` is not a multiple of 32 bits or is too narrow for the bitmap's width as specified by the *widthInPixels* parameter. When no other value is specified, this slot has the default value `BAND(`*widthInPixels*` + 31, -32) / 8`.

`resolution` Specifies high- or low-resolution images. Like a pen size, the value of the `resolution` slot may be an array or a single value. If this value is an array, the elements of the array specify the *x* and *y* dimensions of the pixels comprising the bitmap. If this slot stores a single value, it

specifies that the pixels are square, having equal values for their *x* and *y* dimensions. Applications that display or otherwise manipulate bitmap documents (for example, fax pages) need to use this slot to control scaling functionality. This slot's default value is `[72,72]` when no other value is specified.

`store` By specifying a store, the bitmap is created as a VBO (virtual binary object), which is a special extension available in the Dante Newton object system. To applications, VBOs appear to be NewtonScript binaries, but they are actually handled directly by the system, using automatic compression and decompression to allow these objects to be much larger that the available heap space. If you are going to create a bitmap, and you know that it will ultimately wind up in a soup on a particular store, you can increase the system efficiency by using this slot to specify the store on which to create the object.

If this slot is `nil`, the NewtonScript heap will be used, and the bitmap will not be a VBO. Developers should limit the use of the frames heap to small bitmaps only.

A throw occurs in the event the frames heap or store does not have enough space for the bitmap.

`companderName`

When a VBO (virtual binary object) is written to the store, the system uses a compander, or compression-decompression utility. This slot is a string that represents the name of the compander that is to be used when

writing or reading this bitmap from the store.

The default compander is `TPixelMapCompander`. This compander is efficient for monochrome images. During compression, the data is preprocessed by XORing scan lines. It is then passed on to the Lempel Ziv implementation contained in the ROM.

This slot allows developers to provide their own compander as `TCompressor` and `TDecompressor` protocols.

`companderData`

This slot is intended for optional arguments that would be passed to the compander. The default is `nil`.

DrawIntoBitmap

`DrawIntoBitmap` (*shape, styleFrame, destBitmap*)

Draws shapes into a bitmap in the same way that the `DrawShape` method (page 12-64) draws shapes into a view. Drawing is clipped to the boundaries of the destination bitmap.

shape Any of the shapes returned by the shape-creation functions, or an array of such shapes intermixed with optional style frames. If a style frame is included in the shape array, the style frame applies to all subsequent shapes in the array, until overridden by another style frame.

styleFrame A style frame as specified in the description of the `DrawShape` method beginning on page 12-64.

destBitmap The bitmap in which drawing takes place.

To perform offscreen buffering, your application's `ViewDrawScript` method can use the `DrawIntoBitmap` function to create a bitmap and then

draw that bitmap into the final onscreen view by sending the `DrawShape` message to the view.

MungeBitmap

`MungeBitmap (`*bitmap*, *operator*, *options*`)`

Performs various bitmap operations such as rotating or flipping the bitmap. These operations are destructive to the bitmap passed as an argument to this function; the bitmap is modified in place and the modified bitmap shape is returned.

bitmap A bitmap shape on which this function operates. For convenience, the bitmap shape is modified in place and the modified bitmap shape is returned in this slot.

operator A symbol specifying the bitmap modification to perform; it may have any of the following values:

`'flipHorizontal`
: Flips the image bitwise horizontally (mirror image).

`'flipVertical`
: Flips the image bitwise vertically (mirror image).

`'rotateLeft`
: Rotates the image 90 degrees left.

`'rotateRight`
: Rotates the image 90 degrees right.

`'rotate180`
: Rotates the image 180 degrees; unlike `'flipHorizontal`, the image is not mirrored.

The `'flip`*Direction* operators return a shape having the same dimensions as the source bitmap; no view bounds or other rectangles are changed.

The `'rotate`*Direction* operators, however, change the dimensions of the object; therefore, they change the

returned bitmap's bounds rectangle to reflect the new size and shape.

If the source bitmap has been offset, the coordinates of the upper-left corner of the returned object are the same as those of the source bitmap and the coordinates of the bottom-right corner of the returned bitmap are changed. Figure 12-10 on page 12-26 depicts the relationships between the coordinates of the source bitmap and those of the returned bitmap.

options The options frame contains slots for the support of future munge operations. Only one slot is supported at this time:

`callBack` A callback function provided by the application developer to allow the display of the progress of the three rotation operations to the user. The munge operations will call this function with an array argument, ranging from 0 to 100 inclusive, representing the completion percentage of the rotation operation.

ViewIntoBitmap

view`:ViewIntoBitmap(`*view*, *srcRect*, *destRect*, *destBitmap*`)`

Provides a screen-capture capability, writing a portion of the specified view into the specified bitmap. This function always returns nil. This function does not provide a scaling capability, although scaling can be accomplished by passing the *destBitmap* bitmap returned by this method to the `DrawIntoBitmap` function as the value of its *shape* parameter. See Figure 12-10 on page 12-26 for a graphical depiction of the relationships between the view to be captured, the source rectangle, the destination bitmap, and the destination rectangle.

view The view to be made into a bitmap.

srcRect The portion of the view that is to be captured, specified as a rectangle in the local coordinate system of the

source view. A value of `nil` causes this function to use the view bounds of the source view as the dimensions of the source rectangle. The size of the source rectangle is clipped to the intersection of *destRect* and the bounds of the destination bitmap.

Because *srcRect* expects local coordinates, you may need to call *myview*: `localBox()` to get correct coordinates of *srcRect* if `myView` is justified relative to other views.

destRect — Defines the bounds of the portion of the bitmap into which the image is drawn. A value of `nil` causes the view bounds of *srcRect* to be used as the default value of *destRect*. The bounds of *destRect* are clipped to stay within the bounds of the destination bitmap.

destBitmap — The bitmap shape into which the captured view image is written. You can use the `MakeBitmap` function to create this shape.

Hit-Testing Functions

The following functions allows you to determine whether a point or stroke lies within a specified shape.

HitShape

`HitShape(`*shape, x, y*`)`

Indicates whether the point described by the *x* and *y* coordinate parameters lies within the shape.

x — The x coordinate of the point to be tested, in local (view) coordinates.

y — The y coordinate of the point to be tested, in local (view) coordinates.

shape — A shape returned by one of the routines that creates shapes (such as `MakeRect`, `MakeOval`, `MakeRegion`,

`MakePolygon`, and so on.) You can specify an array of shapes for *shape*, and in this case, each shape in the array is hit-tested with the point. If a hit is found, the function returns immediately and subsequent shapes in the array are not tested.

When a single shape is passed to this function, it returns non-`nil` if the specified point lies within the boundaries of the shape and `nil` if the specified point does not lie within the boundaries of any shape passed to it. For unclosed polygons, the result of this function is undefined. When passed an array of shapes, this function returns an "array path" indicating the shape within which the point lies. The "array path" is an array in which each element represents an index into the nested array of shapes passed to `HitShape`.

PtInPicture

`PtInPicture(`*x*, *y*, *bitmap*`)`

x	The x-coordinate of the point to be tested, in local (view) coordinates
y	The y-coordinate of the point to be tested, in local (view) coordinates
bitmap	The bitmap object associated with the mask to be tested

Returns non-`nil` if the point described by the *x* and *y* coordinates lies within the mask associated with the specified bitmap object. If no mask is defined for the specified bitmap, this function tests whether the point lies within the bitmap itself. This function returns `nil` if the point is outside the test area.

Shape-Creation Functions

These global functions create shape objects which you can draw using the `DrawShape` method (described in the "Shape-Creation Functions" on page 12-55).

MakeLine

`MakeLine (`*x1*`, `*y1*`, `*x2*`, `*y2*`)`

Creates and returns the specified line shape.

x1	The x coordinate of the first point drawn.
y1	The y coordinate of the first point drawn.
x2	The x coordinate of the last point drawn.
y2	The y coordinate of the last point drawn.

MakeRect

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

Creates and returns the specified rectangle shape.

left	The x coordinate of the top-left corner of the rectangle.
top	The y coordinate of the top-left corner of the rectangle.
right	The x-coordinate of the bottom-right corner of the text's enclosing rectangle.
bottom	The y-coordinate of the bottom-right corner of the text's enclosing rectangle.

MakeRoundRect

`MakeRoundRect (`*left*`, `*top*`, `*right*`, `*bottom*`, `*diameter*`)`

Creates and returns a rounded rectangle shape (a rectangle having rounded corners).

left	The x coordinate of the top-left corner of the rectangle.
top	The y coordinate of the top-left corner of the rectangle.
right	The x-coordinate of the bottom-right corner of the text's enclosing rectangle.
bottom	The y-coordinate of the bottom-right corner of the text's enclosing rectangle.
diameter	The curvature of the rectangle corners, specified as if a circle of the specified diameter, in pixels, was placed in each of the rectangle's corners.

MakeOval

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

Creates and returns an oval shape. The oval is shaped to fit just inside the specified rectangle. If you specify a rectangle that is square, this method draws a circle.

left	The x coordinate of the top-left corner of the oval's enclosing rectangle.
top	The y coordinate of the top-left corner of the oval's enclosing rectangle.
right	The x-coordinate of the bottom-right corner of the text's enclosing rectangle.
bottom	The y-coordinate of the bottom-right corner of the text's enclosing rectangle.

MakeWedge

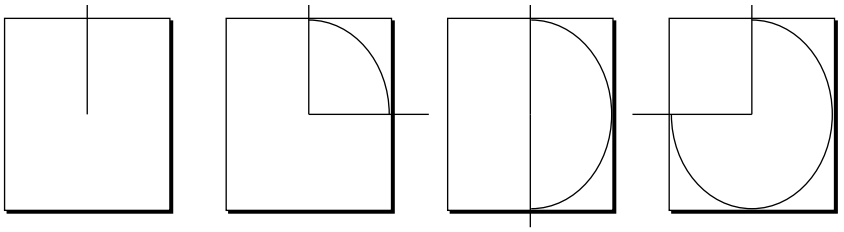
`MakeWedge` (*left*, *top*, *right*, *bottom*, *startAngle*, *arcAngle*)

Draws an arc as a part of an oval that fits just within the specified rectangle. If you draw the wedge with no fill, you see just the arc line. If you draw the shape with a visible fill pattern, you see a solid wedge shape.

left	The x coordinate of the top-left corner of the arc's enclosing rectangle.
top	The y coordinate of the top-left corner of the arc's enclosing rectangle.
right	The x-coordinate of the bottom-right corner of the text's enclosing rectangle.
bottom	The y-coordinate of the bottom-right corner of the text's enclosing rectangle.
startAngle	The angle at which the arc begins, in positive (clockwise) or negative (counterclockwise) degree values.
arcAngle	The angle through which the arc extends, in positive (clockwise) or negative (counterclockwise) degree values.

The angles are given in positive or negative degrees; a positive angle goes clockwise, while a negative angle goes counterclockwise. Zero degrees is at 12 o'clock high, 90 (or -270) is at 3 o'clock, 180 (or -180) is at 6 o'clock, and 270 (or -90) is at 9 o'clock. Other angles are measured relative to the enclosing rectangle: a line from the center of the rectangle through its top-right corner is 45 degrees, even if the rectangle isn't square; a line through the bottom-right corner is at 135 degrees, and so on.

Figure 12-13 Angles for arcs and wedges



MakePolygon

`MakePolygon (`*pointArray*`)`

Creates and returns the specified polygon graphic object.

pointArray An array of x and y coordinate pairs specifying the vertices of the polygon.

MakeShape

`MakeShape (`*object*`)`

Creates and returns a shape based on *object*. `MakeShape` may return a shape that is smaller in size than what you would get if you did the equivalent capture of a view into a bitmap with `ViewIntoBitmap.`

`Object` The following kinds of shapes are created, depending on what kind of object is passed in *object*:

- rectangle — You can pass in a bounds frame describing a rectangle. A bounds frame has the following slots: left, top, right, bottom. A rectangle shape is created and returned.
- points — You can pass in the value stored in the points slot in a view of class `clPolygonView`. This is a binary data

structure that has a class of `'polygonShape` and contains data describing a polygon shape.

A polygon shape is created and returned.

Note

This option is intended to create a shape from data you retrieve from a `clPolygonView`. However, you can manually create the points data structure by using the `ArrayToPoints` routine. ◆

bitmap	You can pass in a bitmap frame object. A bitmap shape is created and returned.You can use the compile-time function `GetPictAsBits` to create a bitmap from a `'PICT'` resource; for more information, see the "Using Resources" appendix in *Newton Toolkit User's Guide*.
picture	You can pass in a picture. A picture shape is created and returned.
view	You can pass in a view. A picture shape is created and returned.

MakeRegion

`MakeRegion` (*shapeArray*)

Creates and returns a region of arbitrary size, shape, and complexity. You define a region by defining its boundary with other shape-drawing functions. The boundary can be any set of lines and shapes (even including other regions) forming one or more closed loops. A region can be concave or convex, can consist of one connected area or many separate areas.

shapeArray	An array of shapes returned by any of the shape-making functions described in this section.

MakePict

`MakePict (`*shapeArray*`, `*styleFrame*`)`

Creates and returns a picture shape that is made by recording a sequence of drawing operations. This groups several drawn shapes into a single graphical entity which is easier, smaller, and faster to use in subsequent drawing operations than drawing each of the shapes individually each time.

This function works exactly like the `DrawShape` method except that the shapes are not drawn on the screen, but are instead drawn into a picture shape object that is returned.

shapeArray — An array of shapes to draw using the characteristics specified in *styleFrame*. The shapes can be any of the shapes returned by the shape-creation functions, and the shape array can include other style frames intermixed with the shapes. If a style frame is included in the shape *array*, it applies to all subsequent shapes in the array, until overridden by another style frame.

styleFrame — A frame having one or more of the slots described in the section "The Style Frame" beginning on page 12-11. If this frame is `nil`, the default values are used. If any single slot is not provided, the default value for that slot is used. See the `DrawShape` method on page 12-64 for a description of the style frame slots.

MakeText

`MakeText (`*`string`*`, `*`left`*`, `*`top`*`, `*`right`*`, `*`bottom`*`)`

Creates and returns a text shape drawn within the specified rectangle. The font used for the text is specified as the value of a slot in the style frame; see the description of the `DrawShape` method for additional information. `MakeText` can create only one line of text at a time.

string	The text string to be drawn
left	The x coordinate of the top-left corner of the text's enclosing rectangle.
top	The y coordinate of the top-left corner of the text's enclosing rectangle.
right	The x-coordinate of the bottom-right corner of the text's enclosing rectangle.
bottom	The y-coordinate of the bottom-right corner of the text's enclosing rectangle.

When drawn, the baseline of the text is placed at the bottom of the rectangle you specify as an argument to the `DrawShape` method. The text is clipped horizontally to the nearest letter boundary within the rectangle but it is not clipped vertically. The text is aligned to the left, right, or center of the rectangle you specify, as controlled by the `justification` slot in the style frame associated with the text shape. See the description of the `DrawShape` method on page 12-64 for a description of how values in the style frame affect drawing.

Shape Operations

These methods and global functions operate on shapes returned from the shape-creation functions described in the previous section.

You also can do hit-testing on shapes using the `HitShape` method. This method is described on page 12-54.

GetShapeInfo

`GetShapeInfo(`*shape*`)`

Returns a frame containing slots of interest for the shape.

shape Any of the shapes returned by the shape-creation functions, or an array of such shapes intermixed with optional style frames. If a style frame is included in the shape array, it applies to all subsequent shapes in the array, until overridden by another style frame.

For all shapes, this frame contains a bounds slot. For text shapes, it additionally contains a text slot; modifying this string will not affect the text shape. For bitmaps created using `MakeBitmap`, the frame will contain the following slots:

`bits` The binary object containing bitmap data.

`bitsBounds` The size of the bitmap itself, expressed as the boundaries of a rectangle having a (0,0) origin.

`bounds` The boundaries of the scaled and offset bitmap.

`depth` An integer expressing the number of bits per pixel. System software version 2.0 currently only supports the value `1`, but others are permitted as long as they are powers of 2. The maximum value allowed is `32`.

`resolution` An integer specifying the resolution of the bitmap, expressed in dots per inch. For example, the built-in fax viewer application uses this slot to store the resolution of the fax image.

`rowBytes` An integer specifying the number of bytes per horizontal row in the bitmap image

`scanOffset`	An integer specifying where bitmap image data begins, expressed as the number of bytes from the beginning of the bitmap.
`store`	The store on which a virtual binary object's soup resides. The value of this slot is `nil` for normal-size. See "MakeBitmap" on page 12-49 for storage details.

DrawShape

view`:DrawShape (`*shape*`,` *styleFrame*`)`

Draws the specified shape (or shapes) in the view using the characteristics specified in *styleFrame*.

shape	Any of the shapes returned by the shape-creation functions, or an array of such shapes intermixed with optional style frames. If a style frame is included in the shape array, it applies to all subsequent shapes in the array, until overridden by another style frame.
styleFrame	A frame having one or more of the slots listed in "Style Frame" on page 12-31. If this frame is `nil`, the default values are used. If any single slot is not provided, the default value for that slot is used.

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

```
myPattern := SetClass(Clone("\uAAAAAAAAAAAAAAAA"),'pattern);
```

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

Note that style frame values don't apply when drawing shapes that are pictures. They are drawn as is. When drawing bitmaps, only the `transferMode` slot is used; the other slots in the style frame don't apply.

OffsetShape

`OffsetShape (`*shape*`, `*deltaH*`, `*deltaV*`)`

Returns the shape with its bounds offset from the original bounds as specified.

shape	The shape to be offset.
deltaH	The horizontal amount by which the specified shape is to be offset from its original bounds.
deltaV	The vertical amount by which the specified shape is to be offset from its original bounds.

You can specify an array of shapes for *shape* in which each shape in the array will be offset. This function is destructive to the shape you pass it; that is, it modifies and returns that shape.

ScaleShape

`ScaleShape (`*shape*`, `*srcRect*`, `*dstRect*`)`

Enlarges or reduces one or more shapes from the size specified by the rectangle *srcRect* to the size specified by the rectangle *dstRect* and returns the scaled shape(s). This function is destructive to the *shape* argument; that is, it modifies and returns its value.

shape	A shape or array of shapes to be scaled.
srcRect	A view bounds frame defining a rectangle that encloses the shape at its original size. The frame has the slots `left`, `top`, `right`, `bottom`. If this frame is `nil`, the shape's original bounds are used as the source rectangle, effectively scaling the shape from its current size to the size of the destination rectangle.
dstRect	A view bounds frame defining a rectangle that encloses the shape at its modified size. The frame has the slots `left`, `top`, `right`, `bottom`.

Note
If the widths and heights of the source and destination rectangles are not proportionate, the returned shape is distorted to fit exactly within the destination rectangle, even if this means that the width and height of the shape are scaled unequally. ◆

ShapeBounds

`ShapeBounds (`*shape*`)`

shape	A shape or array of shapes

Returns a bounds frame describing the rectangle that encloses the shape. The bounds frame has the following slots: `left`, `top`, `right`, `bottom`. You can specify an array of shapes for *shape*, and in this case, this function returns the rectangle that encloses the entire group of shapes.

InvertRect

view:`InvertRect(`*left*`, `*top*`, `*right*`, `*bottom*`)`

Inverts the specified rectangle in the current view. It is important to send this message to a particular view so that clipping of the inversion display can be done properly.

left, *top*	Defines the left-top corner of the rectangle, relative to the local view.
right, *bottom*	Defines the right-bottom corner of the rectangle, relative to the local view.

InsetRect

`InsetRect(`*aBounds*`, `*aDH*`, `*aDV*`)`

Shrinks or expands the rectangle you specify with the *aBounds* frame: the left and right sides are moved in by the amount you specify in the *aDH* parameter; the top and bottom are moved toward the center by the amount you specify in the *aDV* parameter. If the value you pass in *ADH* or *ADV* is negative, the appropriate pair of sides is moved outward instead of inward. The effect is to alter the size by `2*aDH` horizontally and `2*aDV` vertically,

with the rectangle remaining centered in the same place in the coordinate pair.

aBounds	The bounds of the rectangle to alter.
aDH	The horizontal distance to move the left and right sides in toward or outward from the center of the rectangle.
aDV	The vertical distance to move the top and bottom sides in toward or outward from the center of the rectangle.

IsPtInRect

`IsPtInRect(`*anX*`, `*aY*`, `*aBounds*`)`

Checks to see if the point specified by *anX* and *aY* is in the bounds of the rectangle. Returns non-`nil` if the point (*anX*, *aY*) is inside *aBounds*, or else `nil`.

aBounds	The bounds of the rectangle to check.
anX	The horizontal distance to check.
aY	The vertical distance to check.

FitToBox

`FitToBox(`*sourceBox*`, `*boundingBox*`, `*justify*`)`

Makes a box fit into another box while maintaining the *source box*'s original aspect ratio and justifying that resulting box to *boundingBox*'s original aspect ratio, and justifying that resulting box to *boundingBox* according to the justify parameter. The result is a bounds rect.

sourceBox	The bounds rect you're trying to fit into *boundingBox*.
boundingBox	The area you have to display in, that is, the local box of a view.
justify	An integer encoded the same way as a viewJustify slot; only.

`vjCenterH`, `vjLeftH`, `vjRightH`, `vjCenterV`, `vjTopV`, and `vjBottomV` are supported.

MakeTextLines

`MakeTextLines(`*string*`, `*box*`, `*lineheight*`, `*font*`)`

Creates and returns a text shape drawn within the specified rectangle. The text shapes are made and wrapped in relation to the dimensions of the bounds frame specified by the value of the *box* parameter. Words are scanned in until the end of a line is reached, as determined by the width of *box*. The location of the next line is determined by the value of *lineheight*. Text shapes are made until either the string terminates or the limits of the box are reached. In the event that the first word on a line is longer than the width of the box, a partial word is made on the line.

string	The text string to be drawn.
box	The dimension of the bounds frame.
lineheight	The location of the next text line to be drawn.
font	The value of a slot in the style frame.

OffsetRect

`OffsetRect(`*rect*`, `*deltaX*`, `*deltaY*`)`

Returns a bounds frame which is *rect* moved to the right by *deltaX* and down by *deltaY*.

rect	The size of the rectangle and location where you want it to appear.
deltaX	How much to offset the horizontal coordinates in the frame.
deltaY	How much to offset the vertical coordinates in the frame.

SectRect

`SectRect(`*rect1*`, `*rect2*`)`

Returns a bounds frame that is the intersection of *rect1* and *rect2*. For example, if you pass *rect1* and *rect2*, you will get a result frame similar to `{ left: 15, top: 25, right: 100, bottom: 50 }`. If *rect1* and

rect2 do not intersect, an empty bounds frame is returned. Empty bounds can be one of the following:

```
      0 > rect1 := SetBounds(0, 0, 50, 50)
#44118F9  {left: 0, top: 0, right: 50, bottom: 50}
      0 > rect2 := SetBounds(100, 100, 150, 150)
#4411E71  {left: 100, top: 100, right: 150, bottom: 150}
      0 > sect := sectRect(rect1, rect2)
#4412419  {left: 0, top: 0, right: 0, bottom: 0}
```

rect1 and *rect2*	The size and location of the rectangles of which to find the intersection.

UnionRect

`UnionRect(`*unionRect*`, `*rect*`)`

Returns a bounds frame that is determined by the smallest rectangle that encloses both *unionRect* and *rect*. If *unionRect* is nil, a bounds with the same coordinates as *rect* is returned.

unionRect	The size and location of the rectangle of which to find.
rect	The size and location of the rectangle of which to find.

RectsOverlap

`RectsOverlap(`*aBounds1*`, `*aBounds2*`)`

Checks to see if there is an overlap between to specified rectangles. Returns non-`nil` if there is overlap between the two rectangles, or else `nil`.

aBounds1	The size and location of the first rectangle.
aBounds1	The size and location of the second rectangle.

Utility Functions

This section describes additional drawing functions and methods.

DoDrawing

view:DoDrawing(*drawMethodSym*, *parameters*)

Compile time only function that ensures that any drawing done by the *drawMethodSym* method does not overwrite other obscuring views (such as floating views that may be partially obscuring the view in which this method draws). Using the DoDrawing method is the preferred way to draw objects other than in a view's ViewDrawScript method.

DoDrawing sets the clipping according to the setting of the vClipping flag for the specified view, invokes the view's *drawMethodSym* method, and restores clipping to what it was before the *drawMethodSym* method was called. DoDrawing passes through the return value of the method called.

drawMethodSym — Quoted symbol specifying the method that performs drawing operations; for example, to indicate use of the DrawShape method, pass the symbol 'DrawShape as the value of this parameter.

parameters — An array of parameters to pass to the *drawMethodSym* method. Set the value of this argument to nil if the *drawMethodSym* method accepts no arguments.

Note

If the view's vClipping flag is not set, drawing is not clipped to the view's bounds but to the view bounds of the hierarchically closest parent view having its vClipping flag set. ◆

CopyBits

view:CopyBits(*picture*, *x*, *y*, *mode*)

Draws a bitmap in the specified location using the specified transfer mode.

picture — A reference to the bitmap object to be drawn. You can use the compile-time function GetPictAsBits to create a bitmap from a 'PICT' resource; for more

information, see the “Using Resources” appendix in *Newton Toolkit User's Guide*.

x — The x coordinate of the top-left corner of the bitmap

y — The y coordinate of the top-left corner of the bitmap

mode — One of the standard drawing transfer modes: `modeCopy`, `modeOr`, `modeXor`, `modeBic`, `modeNotCopy`, `modeNotOr`, `modeNotXor`, `modeNotBic`. If you pass `nil` for mode, the default, `modeCopy`, is used. These constants are described in the section “viewTransferMode Slot” in Chapter 3, “Views.”

Note

`CopyBits` uses the bitmap’s bounds slot to scale the bitmap. So, by changing the bounds of a bitmap (or more likely, a clone of a bitmap) you can achieve scaling. ◆

DrawXBitmap

`DrawXBitmap(`*bounds*`,`*picture*`,`*index*`,`*mode*`)`

Draws a single bitmap from the specified portion of a picture composed of horizontal rows of equal width bitmaps.

bounds — The size of the bitmap and the location in which it is to be drawn in the current view.

picture — A reference to the bitmap object to be drawn. You can use the compile-time function `GetPictAsBits` to create a bitmap from a `PICT` resource; for more information, see the “Using Resources” appendix in *Newton Toolkit User's Guide*.

index — The index in the bitmap resource of the particular bitmap that is to be drawn.

mode — One of the standard drawing transfer modes: `modeCopy`, `modeOr`, `modeXor`, `modeBic`,

`modeNotCopy`, `modeNotOr`, `modeNotXor`, `modeNotBic`. These constants are described in the section "viewTransferMode Slot" in Chapter 3, "Views."

The width of each row in the *picture* bitmap is assumed to be the width as specified in the *bounds* parameter; thus, if you specify a width of 20 pixels and an index of 2, the chunk of *picture* beginning at pixel 40 and extending horizontally through pixel 59 will be extracted and drawn.

IsPrimShape

`IsPrimShape(`*shape*`)`

Returns non-`nil` if the object passed in is a primitive shape (not an array of shapes) that can be passed to `DrawShape`. Because a shape can have many different internal structures, this procedure is the only way to reliably verify that an object is a shape. If the object is not a shape, this function returns `nil`.

shape Any shape returned by one of the shape-creation functions.

Note:
This object fails on an array of shapes—it returns non-`nil` only if the object is a single shape. ◆

view:LockScreen

view`:LockScreen(`*lock*`)`

Prevents the screen from updating or reverses the effect of a previous call to the `LockScreen` method.

lock A Boolean value; when set to `true`, the screen is locked and no updates can occur. To unlock the screen, call the `LockScreen` method again with *lock* set to `nil`. When the screen in unlocked, it forces an immediate screen update.

Normally, all drawing occurs in offscreen memory and the system periodically updates the screen bits from the offscreen memory. This function

prevents the copying of bits to update the screen. This allows you to make drawing calls, or erase and redraw things without an accompanying flicker if the screen would happen to update during your drawing sequence. When you are finished drawing, call `LockScreen(nil)` to unlock the screen and force an immediate update.

Here's how you would typically use LockScreen:

```
...
:lockscreen(TRUE);
:dodrawing(myDrawFnSym, nil)
:lockscreen(nil);
...
```

Note

The system automatically calls `LockScreen` before sending a view the `ViewDrawScript` message, and unlocks the screen afterwards. Therefore, you don't have to call `LockScreen` in your `ViewDrawScript` method, but only when you want to draw at some other time. ◆

The following two global functions allow you to convert the points data in a polygon shape between the binary data structure in the shape view and an array.

PointsToArray

`PointsToArray(`*polygonShape*`)`

Returns an array of data extracted from a polygon shape binary object.

polygonShape — A binary object of the class `'polygonShape` (from the `points` slot of a `clPolygonView`).

The first element in the returned array is an integer identifying the shape type (see the description of `clPolygonView` above). The second element in the array is the number of points. Beginning with the third array element, the remainder of the array consists of coordinate pairs describing the points. The third element contains the x coordinate of the first polygon point and the

fourth element contains the y coordinate, and so on. Coordinates are relative to the top left corner (0, 0) of the `clPolygonView`.

Here is an example of an array returned from a rectangle shape:

```
[11, 5, 0, 0, 0, 29, 40, 29, 40, 0, 0, 0]
```

The first element, 11, describes the array as a rectangle; the second element, 5, indicates that there are five points in the shape; and the remaining elements describe the five points—(0, 0), (0, 29), (29, 40), (40, 0), and (0, 0).

ArrayToPoints

`ArrayToPoints(`*pointsArray*`)`

Converts an array of points to a binary object of the class `'polygonShape` (as found in the `points` slot of a `clPolygonView`). The binary object is returned.

pointsArray	An array of points in the same format as returned by `PointsToArray`.

Summary of Drawing

Protos

protoImageView

```
aProtoImageView := {
_proto: ProtoImageView,
Image : shape,
Annotations : array,
scalingInfo : frame,
viewBounds : boundsFrame,
viewJustify: justificationFlags,
```

```
viewFormat : formatFlags,
zoomStops : array,
dragCorridor : integer,
grabbyHand : shape,
penDown : function, // drags image
ScalingInfoChanged : function, // called when scaling
                                  changes
Setup : function, // initializes the image
OpenImage : function, // opens image
ToggleImage: function, // closes image
GetScalingInfo : function, // returns scaling information
HasAnnotations : function, // returns annotation information
GetAnnotations : function, // returns an array of views
SetAnnotationMode : function, // sets display behavior
GetAnnotationMode : function, // returns a symbol
TargetChanged : function, // called when annotation is
                             changed
CanScroll : function, // returns scrolling information
ScrollTo : function, // scrolls an image
ScrollBy : function, // scrolls an image
ZoomBy : function, // makes an image larger or smaller
ZoomTo : function, // changes the size of the image
CanZoomBy : function, // changes the size of the image
ZoomToBox : function, // resizes the image
...
}
```

protoThumbnail

```
protoThumbnail : = {
_proto: protoThumbnail,
ImageTarget : view,
Image : shape or bitmap,
viewBounds : boundsFrame,
```

```
viewJustify : justificationFlags,
trackWhileScrolling : integer, // tracks the grey box
Setup : function, // prepares thumbnail
OpenThumbnail : function, // opens thumbnail
ToggleThumbnail : function, // opens or closes thumbnail
Update : function, // renders thumbnail view
GetScalingInfo : function, // returns scaling information
PrepareToScroll : function, // prepares for scrolling
ScrollTo : function, // scrolls a view
DoneScrolling : function, // cleans up a scroll operation
...
}
```

protoThumbnailPointer

```
protoThumbnailPointer : = {
_proto: protoThumbnailPointer,
ImageTarget : view,
Image : shape,
viewBounds : boundsFrame,
viewJustify : justificationFlags,
trackWhileScrolling : integer, // tracks the grey box
...
}
```

Data Structure

Style Frame

Functions and Methods

View Classes
clPolygonView
clPictureView
*viewClass:*clRemoteView

Bitmap Functions
MakeBitmap (*widthInPixels, heightInPixels, optionsFrame*)
DrawIntoBitmap (*shape, styleFrame, destBitmap*)
MungeBitmap (*bitmap, operator, options*)
view:ViewIntoBitmap(*view, srcRect, destRect, destBitmap*)

Hit-Testing Functions
HitShape(*shape, x, y*)
PtInPicture(*x, y, bitmap*)

Shape-Creation Functions
MakeLine (*x1, y1, x2, y2*)
MakeRect (*left, top, right, bottom*)
MakeRoundRect (*left, top, right, bottom, diameter*)
MakeOval (*left, top, right, bottom*)
MakeWedge (*left, top, right, bottom, startAngle, arcAngle*)
MakePolygon (*pointArray*)
MakeShape (*object*)
MakeRegion (*shapeArray*)
MakePict (*shapeArray, styleFrame*)
MakeText (*string, left, top, right, bottom*)

Shape Operations Functions

`GetShapeInfo(`*shape*`)`
view:`DrawShape (`*shape*`, `*styleFrame*`)`
`OffsetShape (`*shape*`, `*deltaH*`, `*deltaV*`)`
`ScaleShape (`*shape*`, `*srcRect*`, `*dstRect*`)`
`ShapeBounds (`*shape*`)`
`InvertRect(`*left*`, `*top*`, `*right*`, `*bottom*`)`
`InsetRect(`*aBounds*`, `*aDH*`, `*aDV*`)`
`IsPtInRect(`*anX*`, `*aY*`, `*aBounds*`)`
`FitToBox(`*sourceBox*`, `*boundingBox*`, `*justify*`)`
`MakeTextLines(`*string*`, `*box*`, `*lineheight*`, `*font*`)`
`OffsetRect(`*rect*`, `*deltaX*`, `*deltaY*`)`
`SectRect(`*rect1*`, `*rect2*`)`
`UnionRect(`*unionRect*`, `*rect*`)`
`RectsOverlap(`*aBounds1*, *aBounds2*`)`

Utility Functions

view:`DoDrawing(`*drawMethodSym*`,`*parameters*`)`
view:`CopyBits(`*picture*`, `*x*`, `*y*`, `*mode*`)`
`DrawXBitmap(`*bounds*`,`*picture*`,`*index*`,`*mode*`)`
view:`LockScreen(`*lock*`)`
`IsPrimShape(`*shape*`)`
`PointsToArray(`*polygonShape*`)`
`ArrayToPoints(`*pointsArray*`)`

CHAPTER 13

Sound

This chapter describes how to use sound in your application and how to manipulate Newton sound frames to produce pitch shifting and other effects.

You should read this chapter if you are attempting to use sound in an application.

This chapter provides an introduction to sound and related items, describing:

- sounds, sound channels, and sound frames, as well as a description of the new methods and functions added for 2.0
- specific tasks such as creating a sound frame, playing a sound, and manipulating sound frames
- methods, functions, and protos that operate on sound

About Newton Sound

This section provides detailed conceptual information on sound functions and methods. Specifically, it covers the following:

- overview of sound and the sound channel
- sounds related to user events
- a brief description of the sound frame
- new functions, methods, and messages added for NPG System Software 2.0 as well as extensions to sound code

Newton devices play only sampled sounds; sound synthesis is not supported. However, a number of built-in sounds are supplied in the Newton ROM that you can use in your application. You can also use the Newton Toolkit (NTK) to create custom sounds from Macintosh-style sound resource files.

You can easily associate any sound with a system event or play sound on demand. The system allows you to play sound synchronously or asynchronously.

All operations on sound frames are created by sending messages to a sound channel that encapsulates the sound frame and the methods that operate on it. Sound channels can play back sampled sound from any point within the data. Additionally, playback can be paused at any point in the sample data and later resumed from that point.

Sound channels have the following characteristics:

- There is no visual representation of a sound to the user.
- Sound channels must explicitly be created and destroyed.

As a result, the creation and disposal of sound channels follow this model:

- To create a sound channel, you send the `Open` message to a sound channel frame.
- To dispose of the sound channel, you send the `Close` message to it.

Event-related Sounds

Views can play sounds to accompany various events. For example, the system plays certain sounds to accompany user actions such as opening the Extras Drawer, scrolling the Notepad, and so forth.

Sounds in ROM

The system provides a number of sounds in ROM that are played to accompany various events. See “Resources” on page 13-26 for complete details.

Sounds for Predefined Events

All views recognize a small set of predefined slot names that specify sounds to accompany certain system events. To accompany one of these events with a ROM-based sound, store the name of the ROM-based sound in the appropriate view slot.

The following predefined slots can be included in views to play event-related sounds:

`showSound`	This sound is played when the view is shown.
`hideSound`	This sound is played when the view is hidden.
`scrollUpSound`	This sound is played when the view receives a `ViewScrollUpScript` message.
`scrollDownSound`	This sound is played when the view receives a `ViewScrollDownScript` message.

For example, to play a sound in ROM when the view opens, place its name in the view’s `showSound` slot.

In fact, all ROM_*soundName* constants are pointers to Newton sound frames stored in ROM. Instead of using one of these constants; however, you can store a Newton sound frame in a slot, causing the sound stored in that frame to play in accompaniment to the event associated with that slot. The next section describes the format of a Newton sound frame.

Sound Structures

Sound has two structures: a sound frame and a sound result frame. A sound frame stores binary data and some additional information used internally by the system. A sound result frame returns information to the sound frame when the sound channel stops or pauses. Like any other frame, a sound frame and sound result frame cannot be greater than 32KB in size. See pages 13-15 to 13-17, for a complete list of slots for both types of frames.

Compatibility

Sound frames have been extended so that those in version 1.*x* can be played without modification by devices based on version 2.0 of the Newton ROM. Not all 2.0 sound frames can be played by older Newton devices.

Two new functions have been added: `PlaySoundAtVolume` and `PlaySoundIrregardless`. `PlaySoundAtVolume` plays a sound specified by the sound frame at a specific volume level. `PlaySoundIrregardless` plays a sound no matter what the user's settings are.

For compatibility purposes, version 2.0 sound retains the `PlaySound` and `PlaySoundSync` global functions. However, all other sound-related operations must be performed by sending messages to an instantiated sound channel. See 13-22 to 13-23 for details.

Using Sound

This section describes how to use sound to perform specific tasks. See *Newton Toolkit User's Guide* for descriptions of the functions and methods discussed in this section.

Creating Sound Frames Procedurally

To create a sound frame, you usually need to create a copy of the sound frame you wish to modify. Because you cannot modify sound frames in ROM, you must copy the sound frame in order to modify the binary sample data.

Cloning the original version of a sound frame you want to modify also allows you to reset values to their original state and provides a means of recovering the original sound frame easily if an operation fails.

Cloning Sound Frames

You can use the `Clone` function to make a modifiable copy of the sound frame by simply passing the frame or its reference to `Clone` and saving the result in a variable, as in the following example:

```
clonedSound := clone(ROM_simpleBeep);
```

This technique is an extremely efficient means of creating a modifiable sound frame, because the copy created is a shallow clone; that is, the cloned frame `clonedSound` does not actually store a copy of the `ROM_simpleBeep` binary data. Instead, the `clonedSound` frame stores a pointer to the ROM data in its `samples` slot. Thus, the `clonedSound` frame is fairly lightweight in terms of overhead in the NewtonScript heap.

Creating and Using Custom Sound Frames

The following information applies to the Macintosh OS version of NTK, the Windows version differs; see the *Newton Toolkit User's Guide* for details.

The compile-time functions `GetSound` and `GetSound11` allow you to use the Newton Toolkit to create Newton sound frames from Macintosh `'snd '` resource data. This section summarizes the main steps required to create custom sound frames from Macintosh `'snd '` resources in NTK; for a complete discussion of this material, see the *Newton Toolkit User's Guide*.

Take the following steps to add a custom sound to your application.

1. Include the sound resource file in your application's NTK project.
2. In your application, create an evaluate slot to reference the sound frame through a compile-time variable.
3. In your Project Data file,
   - Open the sound resource file with `OpenResFile` or `OpenResFileX`.
   - If using `OpenResFileX`, store the file reference it returns.
   - Use the functions `GetSound11` or `GetSound` to obtain the sound frame.
   - Use a compile-time variable to store the sound frame returned by `GetSound` or `GetSound11`.
   - Use the function `CloseResFile` or `CloseResFileX`, as appropriate, to close the sound resource file; if you use the `CloseResFileX` function, you need to pass as its argument the saved file reference originally returned by `OpenResFileX`.
4. In your application,
   - Set the value of the evaluate slot to the name of the compile-time variable that stores the sound frame.
   - Pass the name of the evaluate slot as the argument to the `PlaySound` (page 13-21) or `PlaySoundSync` (page 13-21) function. These run-time functions play sound from anywhere in your code.

Playing Sound

There are two ways to play sounds in Newton system software. The first is to use the global sound functions `PlaySound` (page 13-21), `PlaySoundSync` (page 13-21), `PlaySoundAtVolume` (page 13-22) or `PlaySoundIrregardless` (page 13-23). The other way is to instantiate a sound playback channel and send messages to it. Each approach has benefits and drawbacks. Using the global functions is the simplest and most efficient approach, but it offers less control than sending messages to a sound channel.

Sound channels are appropriate for applications that require greater control over playback, such as one that allows pausing playback and sound completion. Sound channels are also useful for games, which might require that many sounds be available on short notice or that multiple sounds be played at the same time.

The `PlaySound` global function is best used with basic sounds: system beeps, alert sounds, notification sounds, and user interface sound effects. When you pass a sound frame to the `PlaySound` function, the system plays the sound. You cannot pause, query the sound channel for information, or specify a callback routine. On the other hand, you do not need to instantiate or delete a sound channel.

Using the PlaySound Function

To play a sound frame, pass it to the `PlaySound` global function, as in the following example:

```
PlaySound(ROM_funbeep);
```

In addition to the benefits of simplicity and efficiency, this global function provides compatibility for applications written to previous NTK versions of sound.

Using A Sound Channel to Play Sound

Using a sound channel to play a sound is accomplished by creating a sound channel and sending the `Start` (page 13-18) message to it.

Creating a Sound Channel For Playback

You create a sound channel by sending it the `Open` function.

The code that creates a sound channel for playback might look like the following example:

```
mySndChn := {_proto:protoSoundChannel};
mySndChn : Open();
```

Deleting the Sound Channel

When finished with the sound channel, you need to dispose of it by sending the `Close` (page 13-17) message to it. Most applications can dispose of the sound channel as soon as playback is completed; the callback function associated with a sound frame is an appropriate way to send the `Close` message to the channel.

Note
The system sound channel is never automatically disposed of even if the sound channel frame is garbage collected. You must send the `Close` message to the channel to dispose of the system sound channel. ◆

Playing Sound on Demand

You can use any of the global functions to play sound on demand. For example, you might want to play a sound when the user taps a button, or when a lengthy operation is complete. Sounds can be played synchronously or asynchronously, as described in the following section.

Synchronous and Asynchronous Sound

When a sound is played synchronously, the system waits for it to finish playing before beginning another task; when a sound is played asynchronously the playback can be interrupted because the system does not

wait for the sound to finish playing before beginning another task (such as playing a different sound).

When playback must be allowed to complete, use the `PlaySoundSync`, `PlaySoundAtVolume`, or `PlaySoundIrregardless` function to guarantee uninterrupted playback. Synchronous playback is generally preferred unless the sound is so long as to be tedious or the application requires a high degree of responsiveness to the user. The processor can do nothing else until it completes synchronous playback.

For applications that must respond instantly to user input, use the `PlaySound` function; your application can then call the sound chip before it finishes playing the current sound. This type of playback is useful for an application providing verbal instructions to a user who may want to cancel them at any point. Asynchronous sound can also be used to create interesting effects, such as playing a sample repeatedly to create a stuttering or echo-like effect.

Both approaches have benefits and drawbacks: synchronous playback can tie up the processor when it's inconvenient to do so; on the other hand, asynchronous playback is never guaranteed to complete. Your use of synchronous or asynchronous sound playback depends on your application's needs.

Hearing Is Believing

The following code example demonstrates the difference between asynchronous playback and synchronous playback. To hear the demonstration of the two types of sound playback, type following code example into the Inspector as it is shown here, select all of these lines, and press Enter.

```
print ("Synchronous sound demo");
call func()
begin
for i := 0 to 20 do PlaySoundSync(ROM_simplebeep);
end with();

print ("Async sound demo");
call func()
```

```
begin
for i := 0 to 20 do PlaySound(ROM_simplebeep);
end with();
```

The synchronous sound playback example plays the `rom_simplebeep` sound twenty times; the sound plays its entire length each time. Twenty repetitions may seem a bit laborious until you hear how quickly the same calls are made in asynchronous mode.

Note that the asynchronous version is able to call the sound chip fast enough that the sound does not have enough time to finish playing; as a result, part of the playback is clipped off with each new call to the `PlaySound` function. In fact, it's likely that you won't hear twenty sounds in the asynchronous playback demo; the calls come faster than the Newton sound chip is able to respond—which brings up an interesting point about the Newton sound chip.

About the Sound Chip

The Newton sound chip requires about 50 milliseconds to load a sound and begin playing it. It also requires about 50 milliseconds to clear its registers and ready itself for the next call after playback completes. Although most applications are not affected by this timing information, it is included for those developers that are interested, along with the following caveat: not to rely on the ramp-up and ramp-down times specified here because they may change in future Newton devices. ◆

Generating Telephone Dialing Tones

Applications can use the `Dial` (page 13-19) view method and the `RawDial` (page 13-21) global function to generate telephone dialing tones from NewtonScript. It is strongly recommended that you use these functions rather than attempting to generate dialing tones yourself. These functions produce dialing tones that meet the standards for all countries in which Newton devices are available, sparing the application developer the effort of dealing with widely varying telephone standards.

When using the `Dial` method you'll note that it returns immediately and the dialing begins a moment later. Even though the dialing tones are not generated until after the method returns, its dialing behavior is "synchronous" in that it cannot be interrupted; the reason dialing does not occur until after the method returns is that the dialing behavior is implemented as a deferred action.

If you need to perform other actions while generating dialing tones, such as posting status messages as various parts of the phone number are dialed, you can use the global function `RawDial` to dial asynchronously. The `RawDial` function accepts the same arguments as the `Dial` method; however, it dials asynchronously.

Note that both dialing functions map alphanumeric characters to the dialing tones that a standard telephone keypad produces for these characters. Standard telephone keypads do not implement the letters Q and Z; the `Dial` method and `RawDial` function map these letters to the tone for the digit 1. Pound (#) and asterisk (`*`) characters are mapped to the same tones that a standard telephone keypad provides for these characters.

Certain phone systems, such as those used for PBX and military applications, also generate special tones (DTMF dialing tones) for the letters A–D. Because the Newton ROM does not generate these special tones, its dialing functions map the characters `A`, `B`, `C` and `D` to the tones they generate on a standard telephone keypad.

Advanced Sound Techniques

This section describes advanced techniques for manipulating the sound frame or its playback. The topics discussed include pitch shifting and manipulating sample data to produce altered sounds.

Pitch Shifting

In general, you can set the value of a sound frame's `samplingRate` slot to any float value less than that specified by the `kFloat22kRate` constant. However, this usually results in poor sound quality. What usually works best

is to take an 11kHz sound and play it at some higher rate. Of course, 22kHz sound resources cannot be played at any higher sampling rate.

You can experiment with pitch shifting by playing sounds in the Inspector using either the `PlaySound` or `PlaySoundSync` functions. You can use any of the ROM sounds or you can use your own custom sounds. The following example describes how to shift a sound's pitch by altering the value of the sound frame's `samplingRate` slot. Remember when setting this slot that `samplingRate` must be a value of type `float`.

```
// keep a copy of original for future use
origSound := clone(ROM_simpleBeep);

// make a copy to modify
mySound := Clone(origSound);

// play the original sound
PlaySoundSync(mySound);

// play at half original pitch
mySound.samplingRate := origSound.samplingRate/2;
PlaySoundSync(mySound);

// note how easily we can return to normal pitch
mySound.samplingRate := origSound.samplingRate

// play at twice speed
mySound.samplingRate := origSound.samplingRate*2;
PlaySoundSync(mySound);
```

By using the output from a control view to alter the value of the sound frame's `samplingRate` slot, you can allow the user to interactively modify

the pitch of playback. The following example code changes the value of the `samplingRate` slot according to the setting of a `protoSlider` view:

```
theSlider.changedSlider := func()begin
if viewValue = maxValue then
   mySound.samplingRate := originalRate
   else mySound.samplingRate := (viewValue*1.01);
PlaySound(mySound);
end
```

For an example that uses output from a view based on the `protoKeypad` prototype, see the Newton DTS sample code on this topic.

Manipulating Sample Data

This section describes how to use the utility functions `ExtractByte` and `StuffByte` to manipulate individual bytes in sound sample data. Because of performance considerations, you'll only want to manipulate sample data on the Newton when it's absolutely necessary—even simple operations like the example here can take a long time to perform on a relatively small sound sample.

The following example, taken from the Newton DTS sample *SoundTricks*, extracts bytes from the end of the sample data and adds them to its beginning, thus reassembling the samples in reverse order to create a "backwards" sound.

▲ **WARNING**

The destination object into which you stuff bytes using the `StuffByte` function must be large enough to hold all the data you place in it or you'll overwrite other items in memory, corrupting the NewtonScript heap. ▲

```
// backwardSound is a slot in the app's base view
// if it's nil then create the backward sound
if (not backWardSound) then
```

```
begin
// get a frame to work with
   backwardSound := deepclone(ROM_funbeep);
// a var to store the modified sample data
   local sampleHolder := Clone(backwardsound.samples);
   local theSize := Length(sampleHolder) -1 ;
// Copy bytes from one end of the binary object
// to the other.
   for i := 0 to theSize do
      StuffByte(backwardSound.samples,i,
               ExtractByte(sampleHolder,theSize-i));
end;
```

The code in this example executes in the application's `ViewSetupFormScript` (page 3-127) method; as a result, the *SoundTricks* application takes an unacceptably long time to open. A better solution is to provide the backwards sound as a resource that can be played just like any other sound; there are a number of Macintosh sound editors available that you might use to create such a resource.

Reference

This section describes the data structures, the `protoSoundChannel`, ROM sounds, and sound functions in Newton system software.

Data Structures

Sound has two data structures: the sound frame and the sound result frame. Each structure is described in the following sections.

Sound Frame

The sound frame contains one or more of the slots listed here. If any single slot is not provided, the default value for that slot is used.

Slot Descriptions

`sndFrameType` Specifies the format of this sound frame. Currently, Newton sound frames always have the symbol `'simpleSound` in this slot; future Newton devices may store other values here. Required.

`samples` A frame of class `'samples` containing the binary sound data. The sound data must have been sampled at 11khz or 22kHz. Required.

`samplingRate` A floating-point or integer value specifying the rate at which the sample data is to be played back. The constants `kFloat11kRate` and `kFloat22kRate` can be used to specify standard rates of 11 kHz and 22kHz, respectively. If this slot is not provided, the default value of 22kHz is used.

Slot Descriptions

`compressionType`
: Currently, the value of this slot is always `kNone`, indicating no compression.

`dataType`
: Integer. Size of samples in bits. Optional. If present, it must be 1 (`k8Bit`). If missing, `k8Bit` is assumed.

`start`
: Integer. Index of first sample to begin play. Optional. If missing, 0 is assumed.

`count`
: Integer. Number of samples to play. If missing, `Length(samples) - start` is assumed.

`loops`
: Integer. Number of times to repeat the sound. For example, setting loops to 3 means play the sound a total of four times. Optional. If missing, 0 is assumed.

Callback

`Callback(`*state*`, `*result*`)`

Invoked when the sound frame completes.

state
: State is one of the following:

  0 = `kSoundCompleted`

  1 = `kSoundAborted`

  2 = `kSoundPaused`

result
: Result is an error code, if any.

Sound Result Frame Format

A sound result frame will have the following slots:

Slot Descriptions

`sound`	Reference to the `soundFrame` that was paused, stopped, or completed.
`index`	Index of the sample where the sound was paused or stopped. This number will be between `soundFrame.start` and (*soundFrame.start* + *soundFrame.count*).

Protos

Sound uses one proto: `protoSoundChannel`.

protoSoundChannel

The `protoSoundChannel` template provides methods that implement pause, playback, and callback of sounds. It also provides query methods that return whether the sound is running or paused.

Open

`Open()`

Opens the sound channel. This method throws an `|evt.ex.fr|` exception if an error occurs; otherwise, it returns `nil`.

Close

`Close()`

Closes the sound channel. This method throws `|evt.ex.fr|` exception if an error occurs; otherwise, it returns `nil`.

Note
You must call `Close`; otherwise, the sound channel is not disposed of in garbage collection. ◆

Schedule

`Schedule(`*soundFrameRef*`)`

Queues `soundFrame` for play. This method throws an `|evt.ex.fr|` exception if an error occurs; otherwise, it returns `nil`. As each sound completes, the sound channel sends the message `callback` slot to the `soundFrame` (if defined).

soundFrameRef The sound frame to be played. See "Sound Frame" on page 13-16 for a list of slots and a description of the `callback` function.

Start

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

Starts the sound channel. The channel begins playing sound frames in the order they were scheduled (see the previous description). This method throws an`|evt.ex.fr|` exception if an error occurs; otherwise it returns `nil`.

async A boolean value of `true` or *`nil`*. If *async* is `nil`, the call does not return until the entire play queue is empty (all scheduled sounds have completed).

Stop

soundChannel`:Stop()`

Stops the sound channel. The channel stops all scheduled sound frames, including the currently playing one, if any. Throws an `|evt.ex.fr|` exception if an error occurs. Returns a sound result frame (see page 13-17) indicating which sound frame was stopped, or `nil` if no sound was currently playing. All scheduled sound frames complete (via the `callback` function) with state 1 `(kSoundAborted)`.

Pause

soundChannel`:Pause()`

Temporarily halts the current playback process in the specified sound channel. If the sound channel is stopped when this message is sent, the message starts the channel, pausing its operation at the beginning of the sound data. If the sound channel is paused, the message resumes playback of the sound.

IsPaused

soundChannel`:IsPaused()`

Returns `true` if the specified sound channel is paused; otherwise, returns `nil`.

IsActive

`IsActive()`

Returns `true` if the channel is active (playing or paused); otherwise, returns `nil`.

Functions and Methods

The functions and methods described in this section play sounds, generate telephone dialing tones, and allow you to get and set the playback volume.

Dial

rootview`:Dial(`*numberString*`,` *where*`)`

Dials the specified telephone number synchronously as a deferred action, using the speaker or modem as specified. To dial asynchronously, use the global `RawDial` function.

This function always returns `true`.

numberString A string specifying the number to dial. Acceptable values for this string include only the digits `0`-`9`, the

alphanumeric characters A-Z, and the special characters *(asterisk), # (pound), - (dash) and, (comma). This function maps alphabetic characters to the tones that a standard telephone keypad generates for these characters. The letters Q and Z, which are not present on a standard telephone keypad, are mapped to the digit 1. Note that the letters A - D do not generate the specialized DTMF dialing tones used by some phone systems; these letters are mapped to the tones that they would produce on a standard telephone keypad. The dash (-) character inserts a delay of 50 milliseconds when dialing and the comma (,) character inserts a delay of 500 milliseconds when dialing.

where A symbol, either `'speaker` or `'modem` specifying whether to dial through the speaker or modem, respectively.

Because `Dial()` is a method of the root view, you'll want to send the message to the root view by prefixing the method call with

```
GetRoot():
```

For example, to use the `Dial` function, you would write code that looks like the following example:

```
GetRoot():Dial(555-1212, 'modem)
```

GetVolume

```
GetVolume()
```

Returns the current volume setting for sounds. This will be an integer from `0-4`.

PlaySound

`PlaySound(`*soundFrameRef*`)`

Plays a sound defined by the specified sound frame. The sound is played asynchronously; that is, this function returns immediately and the sound is played as a background process. This function always returns `true`.

soundFrameRef The sound frame to be played. See "Sound Frame" on page 13-15 for a list of slots.

PlaySoundSync

`PlaySoundSync(`*soundFrameRef*`)`

Plays a sound defined by the specified sound frame (see `PlaySound`). The sound is played synchronously; that is, the Newton stops everything it's doing, plays the sound, and then this function returns. This function always returns `true`.

soundFrameRef The sound frame to be played. See "Sound Frame" on page 13-15 for a list of slots.

RawDial

`RawDial(`*numberString, where*`)`

Dials the specified telephone number asynchronously, using the speaker or modem as specified.

This function always returns `true`.

numberString A string specifying the number to dial. Acceptable values for this string include only the digits `0-9`, the alphanumeric characters `A-Z`, and the special characters `*`(asterisk), `#` (pound), `-` (dash) and `,` (comma). This function maps alphabetic characters to the tones that a standard telephone keypad generates for these characters. The letters Q and Z, which are not present on a standard telephone keypad, are mapped to the digit 1. Note that the letters A - D do not generate the specialized DTMF dialtones used by some phone

systems; these letters are mapped to the tones that they would produce on a standard telephone keypad. The dash (-) character inserts a delay of 50 milliseconds when dialing and the comma (,) character inserts a delay of 500 milliseconds when dialing.

where A symbol, either `'speaker` or `'modem`, specifying whether to dial through the speaker or modem, respectively.

SetVolume

`SetVolume(`*volume*`)`

Sets the output level for all sounds. The default level is `4`, which is the highest volume level. This function always returns `nil`.

volume An integer value from `0` to `4` specifying the level at which sound is to be played. The value `0` turns sound output off completely and the value `4` specifies the highest available sound output level.

PlaySoundAtVolume

`PlaySoundAtVolume(`*soundFrameRef*`, `*volume*`)`

Plays a sound defined by the specified sound frame (see `PlaySound` on page 13-21). The sound is played asynchronously; that is, this function returns immediately and the sound is played as a background process. This function always returns `true`. The sound sets the volume before playing and restores it when it is complete.

soundFrameRef The sound frame to be played. See “Sound Frame” on page 13-15 for a list of slots.

volume An integer value from `0` to `4` specifying the level at which sound is to be played. The value `0` turns sound output off completely and the value `4` specifies the highest available sound output level. If volume is `nil`, the current sound volume is used.

PlaySoundIrregardless

`PlaySoundIrregardless(`*soundFrameRef*`)`

Plays a sound, regardless of the user sound preference settings (action and pen sound effects). The sound is played asynchronously; that is, this function returns immediately and the sound is played as a background process. This function always returns `true`.

soundFrameRef The sound frame to be played. See “Sound Frame” on page 13-15 for a list of slots.

PlaySoundIrregardlessAtVolume

`PlaySoundIrregardlessAtVolume(`*soundFrameRef*`, `*volume*`)`

Plays a sound at the specified volume, regardless of the user sound preference settings (action and pen sound effects), and restores the sound when it completes. The sound is played asynchronously; that is, this function returns immediately and the sound is played as a background process. This function always returns `true`.

soundFrameRef The sound frame to be played. See “Sound Frame” on page 13-15 for a list of slots.

volume The volume at which to play the sound.

PlaySoundEffect

`PlaySoundEffect(`*soundFrameRef*`, `*volume*`, `*type*`)`

Plays the sound at the specified volume, if user preferences allow the sound, and restores the sound volume when it completes. ROM code should use this function instead of `PlaySound`.

volume The volume at which to play the sound.

soundFrameRef The sound frame to be played. See “Sound Frame” on page 13-15 for a list of slots.

type Can be one of `'pen`, `'alarm`, or `'action`.

Clicker

```
Clicker()
```

Plays various “click” sounds. Use this for pen sounds instead of `PlaySound`.

Resources

This section describes sound resources.

The system provides a number of sounds in ROM that are played to accompany various events; these sounds are referenced by the following constants.

`ROM_alarmwakeup`
: The sound played when the Newton powers on automatically to display an alarm.

`ROM_bootsound`
: The sound played when the Newton is powered on.

`ROM_click`
: The sound played when the user taps on items such as buttons and close boxes.

`ROM_crumple`
: The first sound played when deleting an item from the Notepad; it accompanies an animated simulation of the note being wadded into a ball.

`ROM_dialtones`
: An array of 12 sounds that correspond to dialtones on a touchtone phone. Dialing tones 1-10 have a 1-to-1 correspondence with the array. Dialing tone # is set to 11 and * is set to 12.

`ROM_drawerclose`
: The sound played as the Extras Drawer closes.

`ROM_draweropen`
: The sound played as the Extras Drawer opens.

`ROM_flip`
: The sound played when turning pages in a Book Maker book.

`ROM_funbeep`
: The Trill sound in the user preferences Sound panel.

`ROM_hilitesound`
: The sound played to indicate to the user that the

	Newton device is in highlight mode, rather than inking mode. This sound plays when the user presses the stylus against the screen continuously; it is accompanied by the display of the highlighting mark.
`ROM_plinkbeep`	The Xylo sound in the user preferences Sound panel.
`ROM_simplebeep`	The Bell sound in the user preferences Sound panel.
`ROM_wakeupbeep`	The sound played when the Newton is powered on.
`ROM_plunk`	The second sound played when deleting an item from the Notepad; it depicts the sound of the crumpled note hitting the Trash.
`ROM_poof`	The sound played when an item is scrubbed; it accompanies the animated cloud that depicts the item “going up in smoke.”

Summary of Sound

Data Structures

Sound Frame

Sound Result Frame Format

Protos

protoSoundChannel

```
aProtoSoundChannel := {
_proto: protoSoundChannel,
Open : function, // opens sound channel
Close :  function, // closes sound channel
```

```
Schedule : function, // queues sound for play
Start : function, // starts sound channel
Stop : function, // stops sound channel
Pause : function, // halts playback
IsPaused : function, // checks if sound channel is paused
IsActive : function, // checks if  sound channel is active
...
}
```

Functions and Methods

rootview:Dial(*numberString*,*where*)
GetVolume()
PlaySound(*soundFrameRef*)
PlaySoundSync(*soundFrameRef*)
RawDial(*number*, *where*)
SetVolume(*volume*)
PlaySoundAtVolume(*soundFrameRef*, *volume*)
PlaySoundIrregardless(*soundFrameRef*)
PlaySoundIrregardlessAtVolume(*soundFrameRef*, *volume*)
PlaySoundEffect(*soundFrameRef*, *volume*, *type*)
Clicker()

Resources

ROM_alarmwakeup
ROM_bootsound
ROM_click
ROM_crumple
ROM_dialtones
ROM_drawerclose
ROM_draweropen
ROM_flip
ROM_funbeep

```
ROM_hilitesound
ROM_plinkbeep
ROM_simplebeep
ROM_wakeupbeep
ROM_plunk
ROM_poof
```

Find

This chapter shows you how your application can find text, dates or your own data types in application data items. If you want to provide users with the ability to search your application's data, you should become familiar with the Find service and the concepts discussed in this chapter.

Before reading this chapter, you should be familiar with the concept of the target of an action, as explained in Chapter 15, "Filing." You should also understand the use of views to image data, as explained in Chapter 3, "Views." If your application stores data as soup entries, you should understand the contents of Chapter 11, "Data Storage and Retrieval."

This chapter is divided into four main parts: an introduction, a conceptual section, a practical section and a reference section.

The first part, "Introduction," describes the core user interface to the Find service as well as variations and optional features you may elect to support.

The conceptual portion of the chapter, "About The Find Service,"consists of two sections:

- "Programmer's Overview" provides a code-level overview of the Find service and introduces important conceptual material.
- "Compatibility Information" describes differences between the current version of the Find service and previous ones.

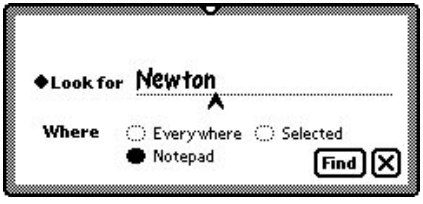
The practical portion of the chapter, “Using The Find Service,”provides code examples describing how to implement support for this service in your application. This section assumes familiarity with the conceptual material presented earlier in the chapter.

The last portion of this chapter, “Find Reference,”provides complete descriptions of all data structures, functions and methods used by the Find service.

Introduction

The Find service searches for occurrences of data items specified by the user. The user interacts with the Find service by means of a Find slip that may be supplied by the system, customized by the application developer or supplied entirely by the developer. Figure 14-1 illustrates the default Find slip that the system supplies.

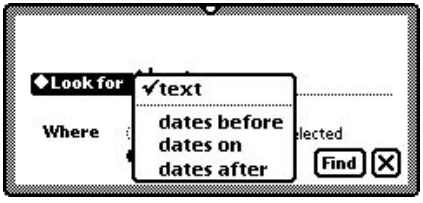
Figure 14-1 The Find slip



The default Find slip contains an input line used to specify a search string, several buttons used to specify the scope of the search and a pop-up menu used to specify the kind of search to perform. The default Find slip allows the user to specify whether the search string is a text item or a date by means of a selection in the popup menu shown in Figure 14-2. You can also

implement specialized searches for other kinds of data; these searches are described in more detail later in this section.

Figure 14-2 Specifying text or date searchesin the Find slip



Text searches are case-insensitive and find string beginnings only. That is, a search for the string “smith” may return the items “Smith” and “smithers,” but not “blacksmith.” Date searches find items dated before, after or on the date specified by the search string.

From the application developer’s perspective, text finds and date finds are nearly identical. The only significant difference between these two kinds of searches is the kind of test an item must pass to be included in the result of the search.

The system-supplied Find slip always contains the Everywhere and Selected buttons. If the currently-active application supports the Find service, it is represented by a button in this slip as well.

Searching for data in the frontmost application only is called a **local find** operation. Figure 14-1 on page 14-2 depicts a local find in the Notepad application.

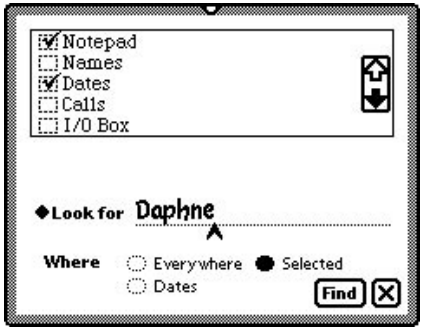
The Everywhere and Selected buttons are used to specify that the system perform searches in applications other than the currently active one. Applications must register with Find service in order to participate in such searches.

Tapping the Everywhere button specifies that the system is to conduct searches in all currently-available applications registered with the Find service. This kind of search is called a **global find**. Applications need not be open to participate in a global find.

A global find is similar to a series of local find operations initiated by the system. When the user requests a global find, the system executes a local find in each application registered with the Find service.

Tapping the Selected button causes a checklist to appear at the top of the Find slip. Included in the list are all currently-available applications that are registered with the Find service. The user can tap items in the list to place a checkmark next to those applications in which the system is to conduct a local find. This kind of search is called a **selected find**.The slip in Figure 14-3 depicts a search for the string `"Daphne"` in the Notepad and Dates applications.

Figure 14-3 Searching specified applications



In addition to these system-supplied variations on the Find slip, you can modify or completely replace the Find slip when your application is frontmost. Typically, you would do this in order to provide a customized user interface for specialized searches. Find slip customizations are discussed in detail later in this chapter.

After using the Find slip to specify the parameters of the search, the user initiates the search by tapping the Find button; alternatively, the user can cancel the search by tapping the close box to dismiss the Find slip.

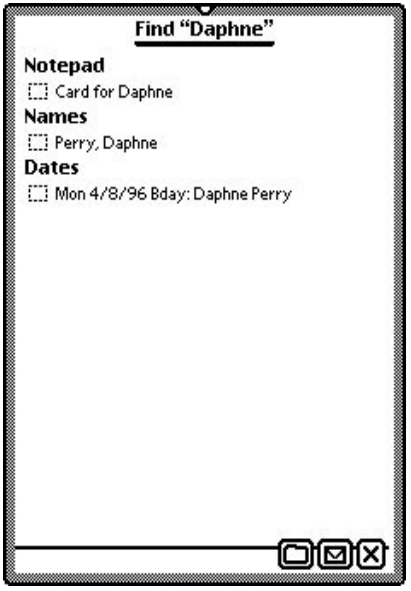
While the search executes, the system provides user feedback regarding its progress. Figure 14-4 depicts a typical progress slip.

Figure 14-4 Progress slip

The progress slip displayed by the Find service includes descriptive title text and an icon, as well as an animated barber-pole gauge, message text describing the progress of the search and a button that allows the user to cancel the search in progress.

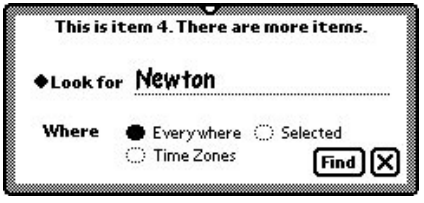
When the search is completed, the Find service displays an overview list of items that match the search criteria. Figure 14-5 depicts the Find overview as it might appear after searching all applications for the string “Boston”.

Figure 14-5 The Find overview



The user can tap on items in the Find overview to display them. As items are displayed, a status message at the top of the Find slip indicates which item is displayed and whether there are more results to display. Figure 14-6 depicts this status message.

Figure 14-6 Find status message



When more than one item is found, the status message indicates that there are more items to display.

Between uses, the Find service stores the setting of the Look For popup menu. The next time this service is used, it reopens in the most recently used find mode. Note that in order to conserve memory, the list of found items is not saved between uses of the Find service.

About The Find Service

This section provides additional information regarding the Find service.

Programmer's Overview

When the user taps the Find button, the system invokes your application's search method. The only significant difference between a date find and a text find is that a different search method locates the items that are returned. To support text searches, you must supply a `Find` method. To support searching by date, you must supply a `DateFind` method. You can conduct your own specialized searches by supplying an `AdditionalFind` method. You can support any of these search methods independently of one another; for example, you can choose to implement the `Find` method without implementing the `DateFind` method.

The search method scans your application's data and returns a frame containing the results of the search. The complement of slots in the result frame varies according to the finder proto on which it is based. A finder proto is a data structure that supplies most of the slots and methods required to conduct a particular kind of search; the characteristics of a particular finder proto are defined by the kind of data it supports. The system-supplied `ROM_SoupFinder` proto supports searching soup data. Your search method appends the result frame to the system-supplied `results` array. Global and selected finds may append more than one frame to this array as multiple applications complete their searches.

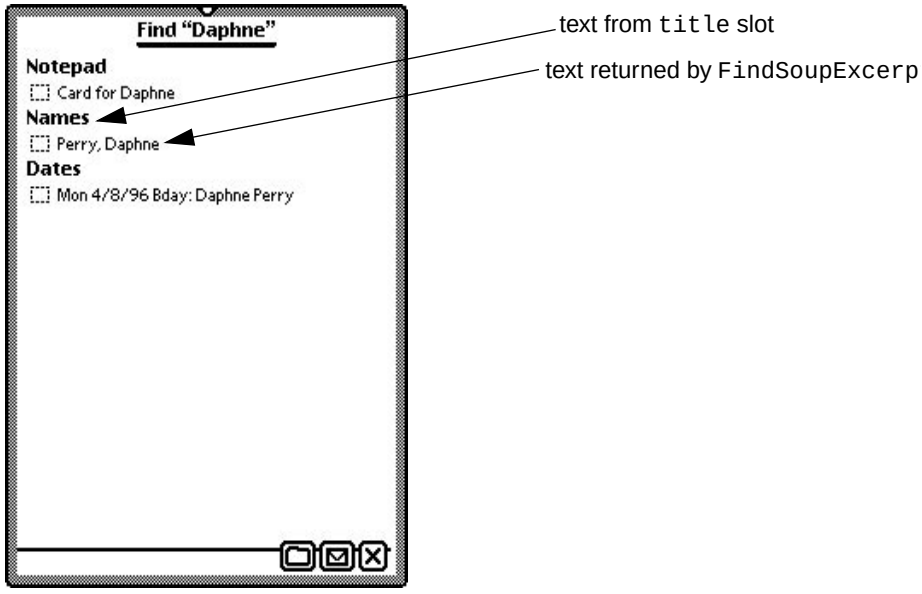
While the search progresses, the system provides user feedback regarding its progress automatically.

When the search method completes, the system displays to the user an overview list of the items that were found. For global or selected finds, each application in which items were found is identified by a heading, and the items that were found are listed under the heading. The application name that appears in this heading is supplied by a `title` slot that each application provides in its base view.

When the user taps scroll buttons to scroll through this list of found items, the system keeps track of the user's place in the array of found items. The strings identifying items in the overview are extracted from the `results` array by a `FindSoupExcerpt` method that you supply.

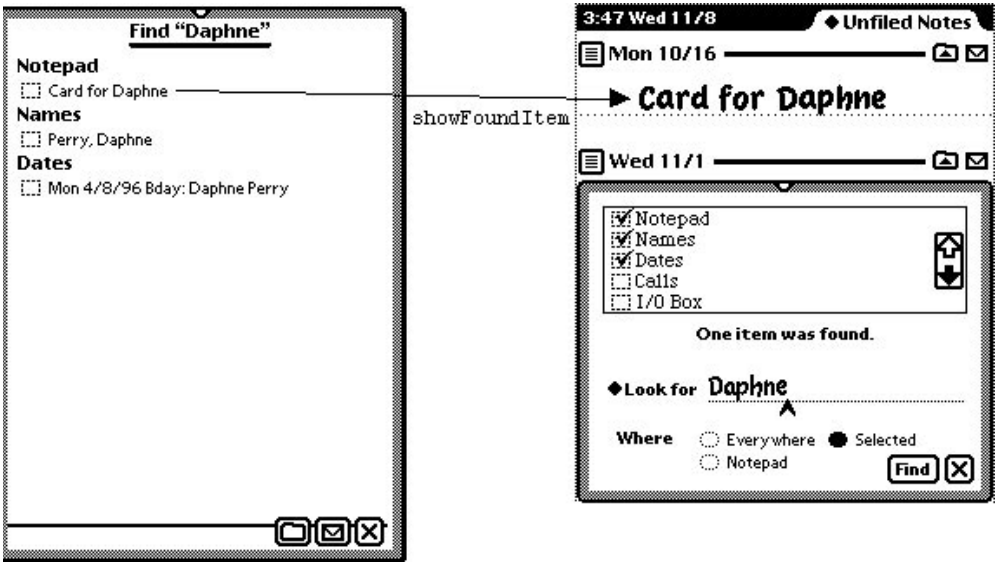
Figure 14-7 depicts the strings from the `title` slot and the `FindSoupExcerpt` method as they are used in a typical Find overview.

Figure 14-7 Identifying strings in the Find overview



When the user taps on an item in the overview, the system sends a `ShowFoundItem` message to the view specified by the `owner` slot of that item's element in the `results` array. In the template for the `owner` view, you must define a `ShowFoundItem` method that locates the item in your application's data and performs any actions necessary to display it, including scrolling or highlighting the item as appropriate. Although the interface to the `ShowFoundItem` method varies according to the finder proto on which your result frame is based, the very same method can often be used to display the results of both text and date searches.

Figure 14-8 The ShowFoundItem method displays an item from the overview



When the user files, deletes, or moves items in the overview of found items, your application is notified via the soup change notification mechanism. For a complete description of this mechanism, see the section "Soup Change Notification" in Chapter 11, "Data Storage and Retrieval."

About Global Finds and Selected Finds

When the user taps the Find button, the system invokes local find methods in the appropriate applications. For a local find, only the frontmost application is messaged. For a global find, all applications registered with the Find service are messaged. Selected finds message a user-specified subset of all applications registered for global finds. In terms of the messages sent, global finds and selected finds are very similar to local finds; however, there are some differences in these operations that your application needs to address.

The most important difference between local finds and other kinds of find operations is that when the system invokes your search method as part of a global or selected find, your application may not be open. Therefore, you need to make sure the application soup is available before actually searching for data.

Another consideration is that the value of the `scope` parameter passed to your search method changes according to the scope of the search. The significance of this value is completely implementation-specific; that is, you can choose to ignore it or you can modify your application's actions based on whether the value of this parameter is `'localFind` or `'globalFind`. The system passes the `'globalFind` symbol when it invokes your search method for a global find or a selected find. The `'localFind` symbol is passed for local find operations.

Compatibility Information

The current version of the Find service opens in the text or date find mode last used. The Find slip in versions of system software prior to 2.0 always opened in text find mode, regardless of the mode last used.

Find now performs "date equal" finds, and returns control to the user more quickly than previous versions did. The Find slip no longer displays to the user the total number of items in the search result; that is, instead of displaying user feedback such as "This is item 24. There are 36 items" the Find slip displays "This is item 24. There are (no) more items."

Do not modify any system data structures directly to register or unregister with the Find service. Instead, use the `RegFindApps` and `UnRegFindApps` functions provided for this purpose. Applications running on older Newton devices can use the `kRegFindAppsFunc` and `kUnregFindAppsFunc` functions provided by NTK for this purpose.

The `SetStatus` method is obsolete; use the `SetMessage` method instead.

Using The Find Service

This section describes how to implement support for Find in your application. It presumes understanding of the conceptual and introductory material in previous sections.

To add application support for the Find service, you need to

- create in your application's base view the `appName` slot containing the user-visible name of your application.
- choose a finder proto on which to base the result frame that your application's search method returns.
- create in the view referenced by the owner slot of your result frame the `title` slot containing the user-visible name of your application.
- supply at least one search method (`Find`, `DateFind` or `AdditionalFind`).
- append your search method's result frame to the system-supplied `results` array.
- supply a `FindSoupExcerpt` method that extracts strings from soup entries for display in the Find overview. This method is required only if you use the `ROM_SoupFinder` proto; if you do not use the `ROM_SoupFinder` proto you will need to display the overview yourself.
- supply a `ShowFoundItem` method that displays an individual element of the result frame that your search method returns.

Optionally, you may also

- register and unregister for participation in global and selected finds.
- add customized searches to the Look For popup menu by supplying an `AdditionalFind` method.
- add your own view to the Find slip by supplying in your application's base view a `FindSlipAdditions` method that returns a view template.

- suppress the display of the Find slip's input line by including a non-`nil` `dontIncludeInputLine` slot in the view template that your `FindSlipAdditions` method returns.
- completely replace the system-supplied Find slip with one of your own by supplying a `CustomFind` method in your application's base view.

The sections immediately following describe these tasks in detail.

Creating The Title Slot

A string that is your application's user-visible name must be available in a text slot called `title`. You need to create this slot in the view referenced by the `owner` slot of the result frame returned by your search method. Commonly, the `owner` slot references the application's base view and the `title` slot resides in this view.

The Find service uses this string in the list of application names displayed for selected finds as well as in the overview of found items.

Creating the appName Slot

Your application's base view must contain an `appName` text slot. This slot holds a string that is your application's user-visible name. This slot is used to name the Find slip radio button that represents your application when it is frontmost. It is also used by other system services to obtatin a user-visible name for your application.

Choosing a Finder Proto

The finder proto is used to construct the result frame returned by your search method. It is strongly recommended that you store your application's data in soups and use the `ROM_SoupFinder` proto to support the Find service.

If you prefer not to use this proto, you can implement your own finder proto. For suggestions regarding slots and methods that your custom finder proto may provide, see "Suggested Custom Finder Proto Slots and Methods" beginning on page 14-42.

Implementing Search Methods

Although the implementation of a search method is for the most part application-specific, some general requirements apply to all search methods. This section describes these requirements and advises you of potential problems your search methods may need to handle.

Your search method needs to be able to search for data in internal RAM as well as in RAM or ROM on a PCMCIA card. Using union soups to store your data makes it easier to solve this problem. Union soups allow you to treat multiple soups as a single soup, regardless of physical location.

As your search method proceeds through the application data, it must test each item against user-specified criteria and collect items meeting the test criteria in a result frame. The exact content of this frame is described in the section "Returning the Results of the Search" beginning on page 14-19.

Your search method needs to call the SetMessage method to provide a status string that the system displays to the user while the search is in progress. Each of the code examples included later in this section provide an example of how to do this.

When the search is completed, the search method must append the result frame to the system-supplied `results` array. This task is described in detail in the section "Returning the Results of the Search" beginning on page 14-19.

If your application registers for participation in global finds, your search methods may be invoked when your application is not open. Thus, your search methods must not rely on your application being open when they are invoked.

Using the StandardFind Method To Search For Text

You can use the system-supplied `StandardFind` method to search for text in soup-based application data. This function is described in detail in the reference section of this chapter on page 14-33.

To use `StandardFind` to implement text searches, you simply call it in the body of your application's `Find` method, as illustrated in the following code example.

Find

```
// this code resides in a slot named Find
// in your app's base view
// MyApplication.Find :
func(what, results, scope, statusView)
begin
   // kMySoupName is a constant containing a string
   // that is the name of your app's data soup
   // make sure the soup exists
   local mySoup
   if (mySoup:=GetUnionSoup(kMySoupName))
   // search for entry text beginning with what
   :StandardFind(what, mySoupName, results,
               statusView, nil);
end;
```

Using Your Own Text-Searching Method With ROM_SoupFinder

It is strongly suggested that you use the `StandardFind` method to implement your `Find` method if possible. The following code example illustrates the kinds of tasks this method performs. This example uses the `ROM_SoupFinder` proto to search for text in soup-based application data.

```
// MyApplication.Find :
func(what, results, scope, statusView)
begin
   // called by the Newton when the user chooses Find
   // this routine MUST be named Find
   local myResult; // for internal use

   // report status to the user
   // note use of GetAppName and unicode ellipsis
   if statusView then
   statusView:SetMessage("Searching in " &
                        GetAppName(kAppSymbol) &
                        $\u2026);
   // presume our soup def is registered
```

```
   // however app may be closed so get our own soup
   local mySoup:= GetUnionSoupAlways("My Soup");
   // make sure a member soup exists so query wont fail
   mySoup:GetMember(GetDefaultStore());

   // retrieve all entries with strings that contain what
   local cursor := mySoup: Query({text: what});

   // append result frame to sys-supplied results array
    if cursor:Entry() then begin
      myResult := {
         _proto: ROM_SoupFinder,
         owner: self,
         title:"My Application",
         findType: 'text,
         findWords: [what],
         cursor: cursor,
                     };
      AddArraySlot(results, myResult);
    end;

// it's not necessary to return myResult separately
// but you can do so if you want to use it elsewhere
   return myResult;
end;
```

Implementing the DateFind Method

Date-based searches have a lot in common with their text-based counterparts; in fact the only significant difference between these two operations is the search criteria. Rather than matching a text string, your `DateFind` method tests items against a specified time value and accepts or rejects them according to the criteria specified by the value of the `filter` parameter. This

parameter specifies whether to include in the search results items dated on, after or before a specified date.

You can simplify the implementation of date-based searches by time-stamping your application's data when it is stored. If you store application data as frames that hold the time they were created or modified in a particular slot, your `DateFind` method can simply test the value of this slot to accept or reject the entry.

DateFind Method Using ROM_SoupFinder

The sample code immediately following shows the implementation of a typical `DateFind` method using the `ROM_SoupFinder` proto.

```
// MyApplication.DateFind :
func(findTime, findType, results, scope, statusView)
begin
   // local vars for internal use
   local myEndTestFn ;
   local cursor ;
   local ourResult;
   local myEntry;
   local querySpec;

   // report status to the user
   if statusView then
      statusView:SetMessage("Searching in MyApp…");

   // get the soup
   local mySoup := RegUnionSoup(kAppSymbol, kSoupDef);

// findType is 'dateBefore, 'dateOn or 'dateAfter
// customize querySpec according to findType
   querySpec := {
            // indexPath could be any time slot/path
```

Find

```
            // this ex. assumes entries have a
            // timeStamp slot for sorting on 'timeStamp
            indexpath: 'timeStamp,
            };
// build the querySpec
   if findType = 'dateBefore then begin

   // because entries are sorted on timeStamp,
   // we start at first entry and we're done when
   // we find a date that is later than findTime
     querySpec.beginKey := 0;
     DateTest := func(item) item.timeStamp < findTime;
// we don't need an end test for 'dateAfter searches
// so we only build this part of cursor
// for a 'dateBefore search
   querySpec.endTest := func(item)
      item.timeStamp >= findTime;
      end;
else begin
   // in a dateAfter search we start at findTime
   // we don't need endTest because cursor stops
   // when it has seen all remaining entries

   querySpec.beginKey := findTime;
   DateTest := func(item) item.timeStamp > findTime;
   end;

// fill in the valid test according to kind of search
querySpec.validTest := DateTest;

// get the cursor
cursor := mySoup:Query(querySpec);
```

Find

```
//set up the items frame and add it to the results array
   if cursor:Entry() then begin
      ourResult := { _proto: ROM_SoupFinder,
                     owner: self,
                     title: "My Application",
                     findType: findType,
                     findWords: findTime,
                     cursor: cursor };
      AddArraySlot(results, ourResult);
      end;
// returns slot added to array or nil if "if" is false

// it's unnecessary to return ourResult separately
// but you can do so if you want to use it elsewhere
return ourResult;
end;
```

Returning the Results of the Search

The format of the result frame is determined by the kind of finder proto used to create it, rather than by the kind of information it holds; that is, the `Find` and `DateFind` methods return the same result frame if they are based on the same finder proto. For a complete description of the slots your result frame must contain, see the section "Result Frames" beginning on page 14-27.

After constructing the result frame, your search method needs to append it to the system-supplied `results` array. Each element in this array is a slot that contains a frame; the frame contains an array of items found in the search. You need to use the global function `AddArraySlot` to append to the `results` array a slot containing your result frame.

The following example shows a line of code that would be placed at the end of your application's `Find` method to store the results of the search. In this code fragment, the `results` parameter contains the array that was originally passed to the `Find` method, and the `myResult` parameter is the frame containing the items found in the search. This call to `AddArraySlot`

places the frame containing the results of the search at the end of the `results` array, in a slot named `myResult`.

```
AddArraySlot(results, myResult);
```

The `results` array is cleared when the Find slip closes.

Implementing the FindSoupExcerpt Method

If you use the `ROM_SoupFinder` proto to construct your search method's result frame, your application needs to supply a `FindSoupExcerpt` method which extracts the name of an item from the result frame and returns it to the system as a string. The string this method returns represents the item in the Find overview.

Your result frame can hold additional data for your `FindSoupExcerpt` method's use. For example, you could save the date associated with each item found and use this information to build more descriptive titles for items displayed in the overview from a date find.

The following example shows the implementation of a typical `FindSoupExcerpt` method.

```
// myApplication.FindSoupExcerpt:
func(entry, resultFrame)
begin
// if this isn't a date find, pass message to parent view
   if resultFrame.findType = 'dateBefore or
      resultFrame.findType = 'dateOn or
      resultFrame.findType = 'dateAfter
   then begin
         return entry.nameString;
        end
   else
   return GetRoot():FindSoupExcerpt(entry, resultFrame);
end
```

For a complete description of the `FindSoupExcerpt` method, see the section "Developer-Defined Methods For Find Overview Support" beginning on page 14-40.

Implementing The ShowFoundItem Method

The implementation of your `ShowFoundItem` method is dependent on the finder proto you used to construct your result frame. This section describes an example `ShowFoundItem` method suitable for use with the `ROM_SoupFinder` proto.

If you've based your result frame on the `ROM_SoupFinder` proto, the `ShowFoundItem` method is passed two arguments: the soup entry that the user tapped in the Find overview, and the frame that your application added to the `results` array. The system expects your application's `ShowFoundItem` method to look like the following example.

```
ShowFoundItem: func(myEntry, myCursor) begin . . . end
```

In the body of this method, you need to do whatever is necessary to display the soup entry `myEntry` from the cursor `myCursor`. Typically you'll send a `Close` message to the overview and open the appropriate view to display `myEntry`. The following code fragment shows the implementation of a typical `ShowFoundItem` method.

```
// for use with ROM_SoupFinder proto
// myApplication.ShowFoundItem:
func(entry, cursor)
begin // close my overview if its open
   if currentView = myOverview then begin
      myOverView:Close();
      myDisplayView:Open();
      // open view that displays the entry
   end;
   // scroll, highlight, etc. as necessary
   // to display the entry
```

```
   myDisplayView:DisplayEntry(entry, cursor);
end
```

Your application is always open when the `ShowFoundItem` messages is sent to it. For example, this message is sent when the user scrolls between found items from within the Find slip. The system also invokes this method when the user taps on an item in the Find overview. In this case, the system opens your application if necessary before sending the `ShowFoundItem` message to it.

Reporting Progress to the User

It is strongly recommended that your status messages be consistent with those that the built-in Newton applications display while a Find operation is in progress; for example, you can use a message such as “Searching in *appName*.”

Your search method uses the `SetMessage` method to display a string in the Find slip. The argument to the `statusContext` parameter of your search method is the frame to which you send the `SetMessage` message. The `SetMessage` method accepts as its argument the string to display while the search is in progress. Figure 14-9 depicts a typical status message from the Find service.

Figure 14-9 Typical progress message

The following code fragment shows how to use the `statusContext` parameter to display a progress message to the user.

```
if statusContext then
   statusContext:SetMessage("Searching in appName.");
```

The *appName* replaceable in this example would be replaced with the name of your application. There are several ways to obtain this string; they are listed here in order of preference:

- You can retrieve this string from the `appName` slot in your application's base view.
- You can retrieve this string from the `title` slot in your result frame or its owner.
- You can retrieve this string by calling `GetAppName(kAppSymbol);`

Global and Selected Find Registration

Use the `RegFindApps` function to register your application with the Find service; its counterpart, the `UnRegFindApps` function, reverses the effects of the `RegFindApps` function. You can call these functions from your application's `InstallScript` and `RemoveScript` methods.

Applications registered with the Find service participate in global finds; they also participate in selected finds when specified by the user.

Registration Unnecessary for Local Finds

You do not need to register with the Find service to support local finds. Global and local find support uses the same mechanism, which relies on the `Find` and `ShowFoundItem` methods that your application supplies. A global find is simply a series of local finds initiated by the system in applications that have registered for participation in global finds.

Customizing the Built-In Find Slip

The frontmost application can add views to the system-supplied find slip, such as buttons that enable the user to conduct specialized finds. Implementation of this feature requires modifications to application code.

Adding Your Own View To The Find Slip

To add your own view to the built-in Find slip, add a `FindSlipAdditions` method to your application's base view. This method must return a view template having a `clView` object as its base view. A view based on the template that this method returns is added automatically to the top portion of the built-in Find slip whenever your application is frontmost and the radio button specifying the front-most application is selected. For a complete description of the `FindSlipAdditions` method, see the section "Developer-Defined Methods" beginning on page 14-37.

Keep in mind that the user may need to scroll between found items while the Find slip is displayed; therefore, when customizing or replacing this slip, avoid making it so large that it obscures the display of the found items.

The example Find slip shown in Figure 14-10 includes a button that selects an application named My App. When this button is filled, the system adds to the Find slip a custom view containing an input line and a popup menu that specify price comparison criteria for the custom find method that this hypothetical application supplies.

Figure 14-10 Additional clView in Find slip

Performing Specialized Searches

To perform a specialized search—typically, one that accepts user input from the view you have added to the system-supplied Find slip—you can implement an `AdditionalFind` method in your application's base view.

Your `AdditionalFind` method may make use of the string provided by the system-supplied input line; the system passes this string in the `systemFindText` parameter to the `AdditionalFind` method. Or, you can suppress the display of this input line and rely strictly on input provided by the view you add to the Find slip. Suppression of the system-supplied input line is described in section "Suppressing The Input Line" beginning on page 14-26.

Your customized user input view is passed to the `AdditionalFind` method in the `myView` parameter. Your `AdditionalFind` method must extract from this view the additional user input it provides for use in a specialized search.

Your AdditionalFind method must append the results of the search to the array that the system passes in the findResults parameter to this method.

To display the results of the search,your AdditionalFind method calls the ShowFound method using code similar to the following example. The value of the *statusContext* parameter is passed to you by the system when it calls your AdditionalFind method.

```
statusContext:ShowFound('findFrontMost);
```

If only one item is found, the ShowFind method displays it; if more than one item is found, the ShowFound method displays the overview of items for the application specified by its argument; your AdditionalFind method must pass the symbol 'findFrontMost as the argument to the ShowFound method.

For a complete description of the AdditionalFind method, see the section "Developer-Defined Methods" beginning on page 14-37.

Suppressing The Input Line

The application can also suppress the display of the input line normally present in this slip. You may want to suppress the display of this view when providing your own input view for a specialized search method.

To remove the system's input line from the Find slip, include a dontIncludeInputLine slot in the view that your FindSlipAdditions method returns. The system modifies the height of the Find slip dynamically according to this slot's value. When the value of this slot is not nil, the display of the system-supplied input line is suppressed.

Replacing the Built-In Find Slip

Applications can also completely replace the system-supplied Find slip with a customized version. To implement a custom find slip that is opened when your application is frontmost, include a method named CustomFind in your application's base view. Your CustomFind method must open your own customized Find slip and do anything else that's appropriate. To actually

perform a text or date search from this slip, send the `FindIt` message to the root view; for example

```
GetRoot():FindIt(findText, scope, findType, selectedApps);
```

The `FindIt` method provides all of the system-supplied Find features, including a progress gauge and an overview of the search results. For a complete description of this method, see the section “Find Reference” beginning on page 14-27.

Find Reference

This section describes functions, methods and data structures used by the Find service.

System-Supplied Finder Protos

This section describes the system-supplied `ROM_SoupFinder` proto.

ROM_SoupFinder

`ROM_SoupFinder`

This system-supplied prototype supports result frames that hold soup-based data.

Slot descriptions

`selected` An array of currently-selected items. The format of this array is not documented. To determine the number of selected items, pass this array to the `Length` function.

Result Frames

This section describes required slots in result frames based on the `ROM_SoupFinder` system prototype. Because the implementation of the

result frame required by your own custom finder proto is entirely up to you, it is not discussed here. For more information about custom finder protos, see

Result Frame Based On ROM_SoupFinder Proto

If your application stores its data in soups, you need to base your search method's result frame on the `ROM_SoupFinder` proto. Your result frame needs to contain the slots described here. You can also add your own slots to this frame; they are ignored by the Find service.

`_proto`	Must have the value `ROM_SoupFinder;`
`owner`	Set to a view that can receive the `ShowFoundItem` message (usually your application's base view).
`cursor`	The cursor returned by your search method's query
`title`	A string that is your application's user-visble name. The system uses this string in the Find overview to separate matches found in each application when conducting global or selected finds. You can omit this slot if the frame referenced by `owner` has a `title` slot.
`findType`	Specifies whether the search is for text or date. The value of this slot is always one of the symbols `'text`, `'dateBefore`,`'dateOn` or `'dateAfter`.
`findWords`	Specifies the text to be matched or the date to be compared.

The following example shows a typical result frame based on the `ROM_SoupFinder` proto.

```
myResult := // for Find (text-based)
  {_proto: ROM_SoupFinder,
  // must use this value
  owner:self,
  // view that gets the ShowFoundItem message,
  // usually your app's base view
  cursor:myCursor,
```

```
   // cursor that iterates over entries returned
   // by search method's query
   // findType is a symbol:
   // either 'text,'dateBefore, 'dateOn or 'dateAfter
   findType:'text
   // indicates that this is a text find
   findWords:[search string]
   // array of words to match
   };
```

The slots in the frame returned by a date find are the same as those in the return frame for a text find; the only difference lies in the values stored in the `findWords` slot. For a date find, this slot contains an integer representing a date against which items are compared; in the case of a text find, this slot contains the text string to be matched. The following code example depicts the result frame returned by a date find.

```
myResult := // for DateFind
   {_proto: ROM_SoupFinder, // must use this value
   owner:self,
   // view that gets the ShowFoundItem message,
   // usually your app's base view
   cursor:myCursor,
   // cursor that iterates over entries returned
   // by search method's query
   // findType is a symbol:
   // either 'text, 'dateBefore, 'dateOn or 'dateAfter
   findType:'dateAfter
   // collect items after the specified date
   findTime:value // date to compare, as an integer
   };
```

System-Supplied Find Functions and Methods

Your application can use these system-supplied methods to support the Find service.

RegFindApps

`RegFindApps(`*appSymbol*`)`

Registers an application for global finds; that is, after the `RegFindApps` function executes, the Find service sends messages to the `GetRoot().(`*appSymbol*`)` view.

appSymbol The application symbol for the application that you want to register for global finds.

Note

To ensure your application's compatibility with future versions of Newton system software, use this function to register for global and selected finds. Applications running on older Newton devices can use the `kRegFindAppsFunc` function provided by NTK for this purpose. ▲

UnRegFindApps

`UnRegFindApps(`*appSymbol*`)`

Unregisters an application for global finds; that is, after the `UnRegFindApps` function executes, the system no longer sends `Find` messages to the view `GetRoot().(`*appSymbol*`)` when the user taps the All button in the Find slip.

appSymbol The application symbol for the application that you want to unregister for global finds.

Note
To ensure your application's compatibility with future versions of Newton system software, use this function to unregister for global and selected finds. Applications running on older Newton devices can use the `kUnregFindAppsFunc` function provided by NTK for this purpose. ▲

FindIt

rootView`:FindIt(`*findText*`,` *scope*`,` *findType*`,` *selectedApps*`);`

Performs text or date finds as specified by its arguments, invoking the specified applications' `AdditionalFind` methods as necessary while reporting progress to the user.

rootView — The view system's base view, as returned by the `GetRoot` function.

findText — The string for which this method searches; for date finds this value is a string representation of the pertinent date.

scope — A symbol specifying where to search for the *findText* data. The value of the *scope* parameter must be one of the following symbols:

`'findFrontMost` — Search the frontmost application's data.

`'findSelected` — Search data belonging to the applications specified in the Find slip. (The user can use checkboxes in the Find slip to specify a subset of all available applications that are currently registered with the Find service.)

`'findGlobal` — Search all applications' data.

findType The kind of search to perform. The value of the *findType* parameter must be one of the following symbols:

`'Text`	Search for text matching the *findText* string
`'DatesBefore`	Search for items having dates before that specified by the *findText* string.
`'DatesOn`	Search for items having dates matching that specified by the *findText* string.
`'DatesAfter`	Search for items having dates after that specified by the *findText* string.

selectedApps An array of application symbols specifying which applications' data to search. Pass `nil` for this argument when the value of the *scope* parameter is not the `'findSelected` symbol.

The application can also include an optional `SetMessage` method used to report status in a customized Find slip that replaces the system-supplied Find slip.

SetMessage

statusContext`:SetMessage(`*msg*`);`

Supply this optional method in your application's base view to accept strings from the system as the Find operation progresses.The strings passed to the SetMessage method are the status messages that appear in the system's Find slip; for example, "This is item 1; there are more items." Your SetMessage method performs any tasks necessary to display these messages to the user.

statusContext See the description of the `Find` method on page 14-37.

msg A message string passed by the system.

StandardFind

view:StandardFind (*what*, *soupName*, *results*, *statusContext*, *indexPath*)

Uses the ROM_SoupFinder proto to search for strings beginning with the specified text.

what	See the description of the Find method on page 14-37.
soupName	a string that is the name of your application's data soup. StandardFind uses this name to call GetUnionSoup for you.
results	See the description of the Find method on page 14-37.
statusContext	See the description of the Find method on page 14-37.
indexPath	the index path used in the query that this function makes against your application's soup data. Pass nil for this value if you don't want to sort the entries in the cursor on this value. For more information, see the section "Query Specification Frame," in Chapter 11, "Data Storage and Retrieval."

To ensure that you send the StandardFind message to the correct view, prefix the call with a colon (:), as in the following example.

```
:StandardFind (what, soupName, results, statusContext, indexPath);
```

IMPORTANT

Although this method is supplied by the root view, do not call self:StandardFind, or GetRoot():StandardFind, as you cannot be absolutely certain that the context of this call will always evaluate to the root view. ▲

ShowFound

statusContext:ShowFound(*whichApp*);

Displays a message, a single found item or an overview of found items as appropriate for the number of items found by the specified application.

statusContext	The view to which SetMessage messages are sent. The system passes the Find slip view as the value of the *statusContext* parameter to the AdditionalFind method.
whichApp	Must always be the symbol 'findFrontMost when the ShowFound method is invoked from within your AdditionalFind method.

ReSync

soupFinder:ReSync()

Returns the finder to its normal initial state and resets the cursor to the first entry in the set of found items. Generally, you should use this method to reset a soup finder, rather than using the Reset method, which only resets the soup finder's cursor.

Note

Do not override this method. ◆

Reset

soupFinder:Reset()

Resets the finder's cursor to the first found entry. This method performs none of the housekeeping tasks that the ReSync method does. In most cases the ReSync method is the more appropriate choice for resetting a soup finder.

Note

Do not override this method. ◆

Count

soupFinder`:Count()`

Returns the total number of found items, expressed as an integer value.

Note

Do not override this method. ◆

ZeroOneOrMore

soupFinder`:ZeroOneOrMore()`

Returns `0` if no entries were found, `1` if one entry was found or another number if more than one entry was found.

Note

Do not override this method. ◆

ShowEntry

soupFinder`:ShowEntry(`*entry*`)`

Causes the finding application to display the specified entry, opening the application if necessary. This method does not close the find overview.

entry The soup entry to display.

Note

Do not override this method. ◆

SelectItem

soupFinder`:SelectItem(`*item*`)`

Marks the specified item as selected.

Your soup finder can replace this method with a slot containing the value `nil` to suppress the display of checkboxes in the Find overview.

entry The found item to mark as selected.

IsSelected

soupFinder:`IsSelected(`*item*`)`

Returns `True` if the specified item is selected in the Find overview.

item The found item to test.

Note
Do not override this method. ◆

ForEachSelected

soupFinder:`ForEachSelected(`*cbFn*`)`

Passes each of the currently-selected items to the *cbFn* function.

cbFn A function object you supply. This function must accept one argument that is a soup entry.

Note
Do not override this method. ◆

FileAndMove

soupFinder:`FileAndMove(`*labelsChanged*`,` *newLabel*`,` *storeChanged*`,` *newStore*`)`

Files and / or moves the selected items.

newLabel The new value for the `label` slot when the *labelsChanged* parameter has the value `True`. This value is undefined when the value of the *labelsChanged* parameter is not `True`.

newStore The new store when the *storeChanged* parameter has the value `True`. This value is undefined when the value of the *storeChanged* parameter is not `True`.

You can override this method to perform additional application-specific tasks; however, it is suggested that your version of this method call the inherited method to actually file or move items. Note that the `FileAndMove`

message may be sent when no items are selected; thus, your override method must check whether any items are selected before doing any work.

Delete

soupFinder`:Delete()`

Deletes all currently-selected items from writeable stores

If you override this method, items can still be deleted and the crumple effect still happens, even if your override method does not call the inherited method.

GetTarget

soupFinder`:GetTarget()`

Returns a frame containing an array of targets used to route multiple entries from the Find overview.

Developer-Defined Methods

Your application conducts searches in response to messages that it receives from the system. You must supply a search method for each message that your application supports.

Find

`Find (`*what*`, `*results*`, `*scope*`, `*statusContext*`)`

The return value of this method is ignored. This method appends to the array passed in the *results* argument a frame containing instances of the specified string beginning.

The system supplies a global function, `StandardFind`, that you can use to implement your application's `Find` method for soup-based text data. If

you want to support date finds, you must implement your application's `DateFind` method yourself.

what contains the user-specified string for which Find is to search your application's data.

results an array of slots passed to your Find method by the system; your `Find` method appends a slot, `myResult`, to this array. The exact content of the `myResult` slot depends on the kind of finder proto used to create the frame returned by your search method.If you used the `ROM_SoupFinder` proto, the frame contains a cursor. If a global find is in progress, the `results` array may contain slots created by other applications' search methods.

scope indicates whether the search is local or global, allowing you to handle these two cases differently if you prefer. The value of this argument is always one of the symbols `'localFind` or `'globalFind`.

statusContext a frame to which you send the message `SetMessage`. The `SetMessage` function accepts as its sole argument a string to display to the user while the search is in progress.

DateFind

`DateFind(` *findTime*`,` *compareHow*`,` *results*`,` *scope*`,` *statusContext*`)`

Appends to the array passed in the *results* argument a frame containing entries meeting the specified date comparison criteria. The return value of this method is ignored.

findTime specifies the date selected by the user. The date is represented as an integer that is the number of minutes passed since midnight, January 1, 1904.

compareHow specifies whether the user chose to find items before, on or after the date specified by the value of the *findTime*

parameter. The value of the *compareHow* parameter is always one of the symbols `'dateBefore`, `'dateOn` or `'dateAfter`.

results See the description of the `Find` method on page 14-37.

scope See the description of the `Find` method on page 14-37.

statusContext See the description of the `Find` method on page 14-37.

AdditionalFind

`AdditionalFind(`*statusContext*`,` *findResults*`,` *myView*`,` *systemFindText*`)`

Performs a customized search, appends the search results to the *findResults* array and displays the results.

statusContext The view to which this method sends SetMessage messages to report progress to the user. The system supplies this value when it invokes your `AdditionalFind` method.

findResults The array to which this method appends the results of its search. The system supplies this value when it invokes your `AdditionalFind` method.

myView The view hierarchy added to the Find slip by your `FindSlipAdditions` method, including user input reflected as values stored by the children of the `clView` at the root of this hierarchy. Your `AdditionalFind` method must from the child views the user input values it requires to perform its specialized search.

systemFindText The value on the system-supplied Find slip input line.

To display the results of the search,use code similar to the following example.

```
statusContext:ShowFound('findFrontMost);
```

If only one item is found, the `ShowFind` method displays it; if more than one item is found, the `ShowFound` method displays the overview of items for the application specified by its argument; your `AdditionalFind` method must

pass the symbol `'findFrontMost` as the argument to the `ShowFound` method.

Developer-Defined Methods For Find Overview Support

The messages described in this section are sent to your application when the user taps buttons in the Find overview. You need to implement these methods yourself if your result frame is based on your own custom finder proto. The `ROM_SoupFinder` proto supports these messages for you, manipulating soup entries returned by the cursor.

FindSoupExcerpt

`FindSoupExcerpt(`*entry*`,`*resultFrame*`);`

Extracts the name of a specified item from the result frame and returns it as a string. The system displays this string to identify the item in the Find overview.

ownerView	The view specified by the `owner` slot in the result frame returned by the search method. For more information, see the section "Result Frames" beginning on page 14-27.
entry	the soup entry whose title is needed
resultFrame	the frame your application added to the results array

ShowFoundItem

`ShowFoundItem (`*myEntry*`, `*myCursor*`)// for ROM_SoupFinder`

Required. Locates the specified item in your application's data and displays it, performing any scrolling or highlighting that is appropriate. A typical `ShowFoundItem` method may need to

- open a view appropriate for displaying the target
- set the cursor or the `target` slot to reference the target
- scroll the contents of the display view to make the target visible

- highlight the target in the display view
- invoke the `SetMessage` method to identify the target in the Find slip

Application-Defined Methods For Find Slip Customization

This section describes developer-defined methods for adding views to the Find slip, suppressing the display of this slip's input line, and defining customized search methods. To completely replace the system-supplied Find slip, use the `CustomFind` method.

FindSlipAdditions

`FindSlipAdditions ()`

Optional method defined in your application's base view. This function must return a view template having a `clView` object as its base view. The view hierarchy defined by this template is added to the system-supplied Find slip when your application is frontmost and its radio button in the Find slip is enabled.

The following code fragment illustrates a typical `FindSlipAdditions` method;

```
myApp.FindSlipAdditions := func()
   begin
      // myFindSlipAdditions is a layout file
      // in your app's NTK project
      GetLayout("myFindSlipAdditions");
   end;
```

CustomFind

`CustomFind()`

This application-defined method opens your own customized Find slip and does anything else required to implement a customized search and display its results.

Suggested Custom Finder Proto Slots and Methods

Most applications do not need a custom finder proto; however, this section discusses methods and slots that may be useful when creating a custom finder proto or when specializing one of the existing finder protos. Keep in mind that the implementation of your custom finder proto is entirely up to you. Depending on your design goals, you may use all or none of the slots and methods mentioned here.

Suggested slots

`_proto`	the system-supplied finder proto on which your custom proto is based.
`currentItem`	As the user scrolls through the list of found items displayed in the Find overview, the system uses this slot to store an index to the currently-selected item.
`selectItem`	Provide a `nil`-value slot having this name to suppress the display of checkboxes next to your application's items in the Find overview.

ReSync

`ReSync()`

This application-defined method resets the current item to be the first item in the array of found items. If your proto is based on `ROM_SoupFinder`, this method should reset the cursor as well.

Call this method when disposing of the Find overview or when the user changes items in the Find overview, thus making it necessary to update and redisplay the overview. For example, you need to call this method when moving or deleting an item from the overview. You can also use this method to recover from errors encountered when the attempt to display an item fails, such as when advancing the cursor to an item returns `nil` or the `'deleted` symbol.

Delete

```
Delete ()
```

This application-defined method deletes from your application's data the selected items in the Find overview.

ShowOrdinalItem

```
ShowOrdinalItem (ordinal)
```

This application-defined method shows an entry specified by ordinal value; it can be used to scroll items in the Find overview. For example, to scroll to the next item in the overview, you could increment or decrement the `currentItem` index appropriately, call the appropriate cursor function to set the current item to the new index, and then redisplay the overview.

ordinal — One of the symbols `'first`, `'prev`, or `'next`, that is used to call the appropriate cursor method to retrieve the specified entry.

For more information on cursor methods see Chapter 11, "Data Storage and Retrieval."

AddFoundItems

```
AddFoundItems(foundItem, offset, maxCount)
```

This method builds the result frame that is returned by your search method. How you obtain the data stored in this frame depends on the type of finder proto you use. When using the `ROM_SoupFinder` proto, you'll get data from a cursor.

foundItem — A frame containing the items found. It should have the following slots:

```
{  // typical foundItem frame
   // we suggest you dont access
   //data directly
   _proto: someApplicationDataItem,
   // your custom finder proto
```

```
        finder: someFinderProto,
        // index of this item
        itemIndex:n
        // string identifying this item
        // in Find overview
        title:"the item name"
    }
```

offset	The number of preexisting items to skip over before adding the new one
maxCount	Maximum number of items that may be added to the result frame

Summary

This section summarizes the Find service.

System-Supplied Finder Protos

```
ROM_SoupFinder := { // for finding soup entries
selected: [...], // array of selected items
MakeFoundItem: function, // do not call or override
AddFoundItems: function, // do not call or override
ReSync: function, // resets soupFinder; do not override
Reset: function, // resets cursor only; do not override
Count: function, // returns # found items; do not override
ZeroOneOrMore: function, // returns # items; don't override
ShowEntry: function, // displays entry; do not override
SelectItem: function, // marks item as selected
IsSelected: function, // returns Boolean; dont override
ForEachSelected: function, // your callback; dont override
FileAndMove: function, // files/moves selected items
```

```
Delete: function, // deletes all selected items
GetTarget: function, // returns frame used by routing
...}
```

Application-Defined Data Structures

Find Result Frames

Text Search Using ROM_SoupFinder Proto

```
myFindResult := // text search using ROM_SoupFinder proto
   {_proto: ROM_SoupFinder, // must use this value
   owner:self, // view that gets ShowFoundItem message
               // usually your app's base view
   title: "My Application",// displayed in Find overview
                        // usually inherited from owner
   cursor:myCursor,// returned by search method's query
   findType:'text// must use this value for text search
   findWords:[search string] // array of words to match
   };
```

Date Search Using ROM_SoupFinder Proto

```
myDateFindResult := // date search using ROM_SoupFinder
   {_proto: ROM_SoupFinder, // must use this value
   owner:self, // view that gets ShowFoundItem message
               // usually your app's base view
   title: "My Application",// displayed in Find overview
                        // usually inherited from owner
   cursor:myCursor,// returned by search method's query
   findType:'dateAfter// or 'dateBefore or 'dateOn
   findWords:value // date to compare, as an integer
   };
```

Functions and Methods

RegFindApps(*appSymbol*)
UnRegFindApps(*appSymbol*)
rootView:FindIt(*findText*, *scope*, *findType*, *selectedApps*)
statusContext:SetMessage(*msg*)
view:StandardFind (*what*,*soupName*,*results*,*statusContext*,*indexPath*)
statusContext:ShowFound(*whichApp*)

Application-Defined Methods for Find Support

appBase:Find (*what*, *results*, *scope*, *statusContext*)
appBase:DateFind(*findTime*, *compareHow*, *results*, *scope*, *statusContext*)
appBase:AdditionalFind(*statusContext*, *findResults*, *myView*, *systemFindText*)

Application-Defined Methods For Find Overview Support

targetView:FindSoupExcerpt(*entry*,*resultFrame*);
targetView:ShowFoundItem (*myEntry*, *myCursor*)// for ROM_SoupFinder

Application-Defined Methods For Find Slip Customization

appBase:FindSlipAdditions ()
appBase:CustomFind()

Application-Defined Custom Finder Proto Methods

This section summarizes suggested methods only; the actual set of methods your finder proto must supply is defined by your own appliction-specific criteria.

customFinder:ReSync()
customFinder:Delete ()
customFinder:ShowOrdinalItem (*ordinal*)
customFinder:AddFoundItems(*foundItem*, *offset*, *maxCount*)

Find

Find

Filing

The Filing service allows the user to associate data items with folders displayed by the user interface. The user can create, edit and delete folders at will. This chapter describes how to support filing in your application.

Before reading this chapter, you should understand the use of views to image data, as explained in Chapter 3, “Views.” If your application stores data as soup entries, you should understand the contents of Chapter 11, “Data Storage and Retrieval.”

This chapter is divided into three main parts: a conceptual section, a practical section and a reference section.

The conceptual portion of the chapter, “About Filing,”describes the user interface to the Filing service, provides a code-level overview of this service and introduces important conceptual material. This section also describes differences between the current version of the Filing service and previous ones.

The practical portion of the chapter, “Using The Filing Service,”provides code examples describing how to implement support for this service in your application. This section assumes familiarity with the conceptual material presented earlier in the chapter.

The last portion of this chapter, “Filing Reference,”provides complete descriptions of all data structures, functions and methods used by the Filing service.

About Filing

The Filing service enables the user to associate data items with tags that represent folders in the user interface. In most cases, the filed items are soup entries that reside in their respective soups, rather than in any sort of folder or directory structure. Filing an item displayed on the screen simply associates its corresponding soup entry with the tag that represents a particular folder. Soup entries hold this tag in their `labels` slot.

The currently-displayed application data item is the item to be filed. This item is referred to as the **target** of the filing action. Your application must provide a **target view** that can manipulate the target; often, this is the same view that images the target’s data.

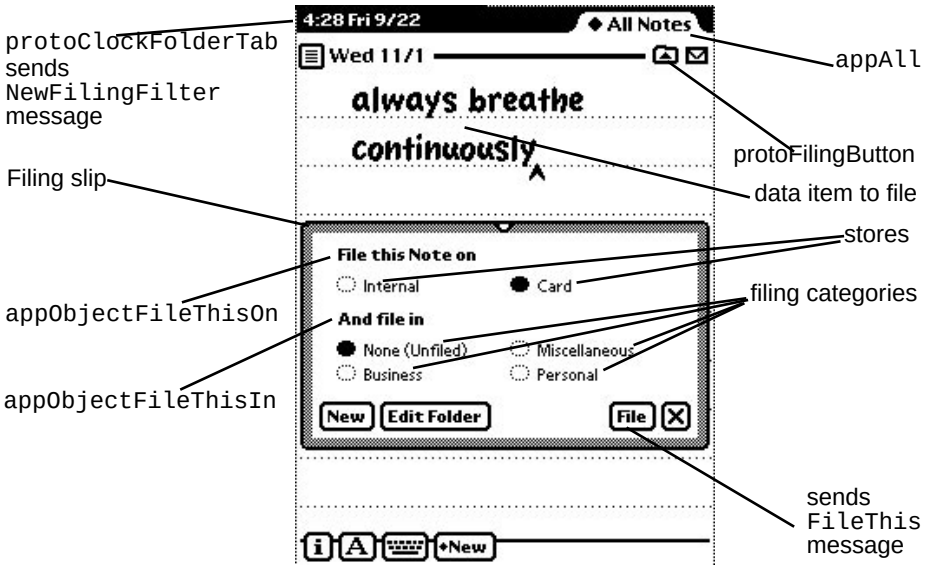
Although the application base view is often an appropriate target view, it may not be under all circumstances. For example, you might want the target view to be a floating window when one is present and the application’s base view at all other times. Applications that display more than one data item at a time, such as the built-in Notes application, may need to specify which of several equal child views is actually the target. You can override the system-supplied `GetTargetInfo` method as necessary to vary the target and target view according to circumstances.

If your targeting needs are less elaborate, you can use the default `GetTargetInfo` method supplied by the system; to use this method, your application base view must supply `target` and `targetView` slots. You are responsible for updating the values of these slots whenever the target changes; that is, whenever the data item on display changes.

To file the target, the user taps on the view provided by the `protoFilingButton` system prototype, which looks like a button with a

picture of a file folder on it. When the user taps the button, it displays the filing slip shown in Figure 15-1.

Figure 15-1 User interface to Filing service



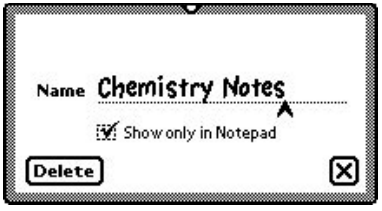
The filing slip displays a set of categories in which the target can be filed. These categories include all folders available to the application that displayed the filing slip, as well as the Unfiled category.

The filing slip also provides buttons for creating new folders and editing the names of existing ones, as well as a close box that dismisses the filing slip. The system allows the user to create a maximum of twelve local folders and twelve global folders. You can use the `RegFolderChanged` global function to execute your own callback function when the user adds, removes or edits folders. The companion function `UnRegFolderChanged` unregisters your callback function.

Folders may be designated as visible only from a specified application; the folders are said to be local to that application. Any folder can also be a global folder that is visible from all applications. Folders are designated as local or global by the user when they are created. The system does not permit the creation of local and global folders having the same name.

You need to create in your application's base view an `appName` slot that contains a string that is the user-visible name of your application. This string is used in various user messages that Filing and other system services display. For example, this string is used to complete the text that is displayed when the user edits the name of a local folder. Figure 15-2 depicts the text displayed when the user edits the name of a folder that is local to the Notepad application.

Figure 15-2 Creating a local folder



When the user taps the folder tab or the filing button, the system determines whether to display global or local folders by testing the values of optional slots that you can supply in the application's base view. You can set the value of the `localFoldersOnly` slot to `True` to cause the filing slip and folder tab views to display only the current application's local folders. You can set the value of the `globalFoldersOnly` slot to `True` to cause the filing slip and folder tab views to display only global folders. When these slots are both `nil` or missing, the filing slip and folder tab display global folders and the current application's local folders.

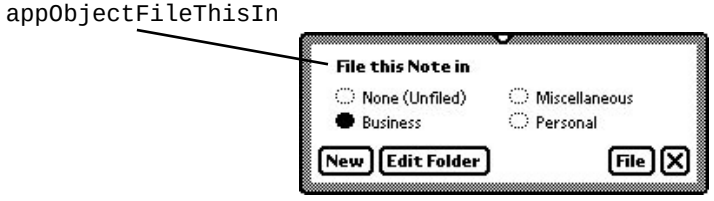
IMPORTANT ▲
The `localFoldersOnly` and `globalFoldersOnly` must not both hold non-`nil` values at the same time. ▲

Your target view can provide an optional `doCardRouting` slot to control the display of the buttons that specify the store on which the current data item is to be filed. When an external store is available and the value of the `doCardRouting` slot is `True`, the Filing slip includes the buttons that represent available stores.

You must supply the full text of the string that labels this group of store buttons. This string is held in an `appObjectFileThisOn` slot that you provide. Similarly, you must supply the full text of the string labelling the group of buttons that represent filing categories. This string is held in an `appObjectFileThisIn` slot that you provide. Figure 15-1 shows where the filing slip displays these strings.

When no external store is available or the value of the `doCardRouting` slot is `nil`, the system displays the simplified version of the filing slip shown in Figure 15-3.

Figure 15-3 Filing slip without external store

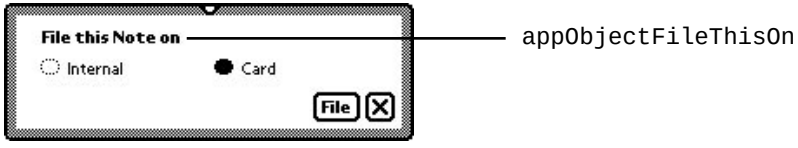


This simplified version of the filing slip does not include the buttons that allow the user to choose a store. Note that the string labelling the group of buttons representing filing categories differs slightly in this version of the filing slip. This string is provided by an `appObjectFileThisIn` slot that your application's base view supplies.

Regarless of any other options you may have implemented, the Filing slip always opens with the current filing category selected; for example, the `'business` folder is selected in Figure 15-3. If you include a non-`nil` `dontStartWithFolder` slot in your target view, the filing slip opens with no default folder selected. This feature is intended for use when you cannot necessarily determine a useful default filing category, such as when the target view is an overview that displays the contents of multiple folders.

When the value of the `doCardRouting` slot is the `'onlyCardRouting` symbol, the filing slip does not include the filing category buttons but allows the user to move the target between available stores without changing its filing category. Figure 15-3 shows the filing slip when an external store is available and the value of the target view's `doCardRouting` slot is the `'onlyCardRouting` symbol.

Figure 15-4 Filing Slip for `'onlyCardRouting`



When the user taps the File button, the system

- invokes the `GetTargetInfo` method to discover the target and the target view.
- sends the `FileThis` message to the target view.

Your target view must supply a `FileThis` method that performs any tasks necessary to file the target, such as:

- moving its soup entry to a different store
- redrawing the current view
- setting the target's `labels` slot to its new value

- performing any additional tasks that are appropriate.

The user can display items in various filing categories by tapping on the file folder tab view. Your application must provide this view, which is based on either of the `protoNewFolderTab` or `protoClockFolderTab` system prototypes.

Both of these folder tab views rely on an `appObjectUnfiled` slot that you provide in your application's base view. This slot contains the full text of the string "Unfiled *items*" in which *items* is the plural form of the target your application manipulates; for example, "Unfiled Notes."

The `protoClockFolderTab` proto is a variation on `protoNewFolderTab` that includes an icon the user can tap to display the current time. Your application can use `protoClockFolderTab` in place of `protoNewFolderTab` to include the clock feature. Both of these protos allow you to customize the action they take when the user taps on them. In addition, the protoNewFolderTab allows you to customize the text it displays to the left of the folder tab text.

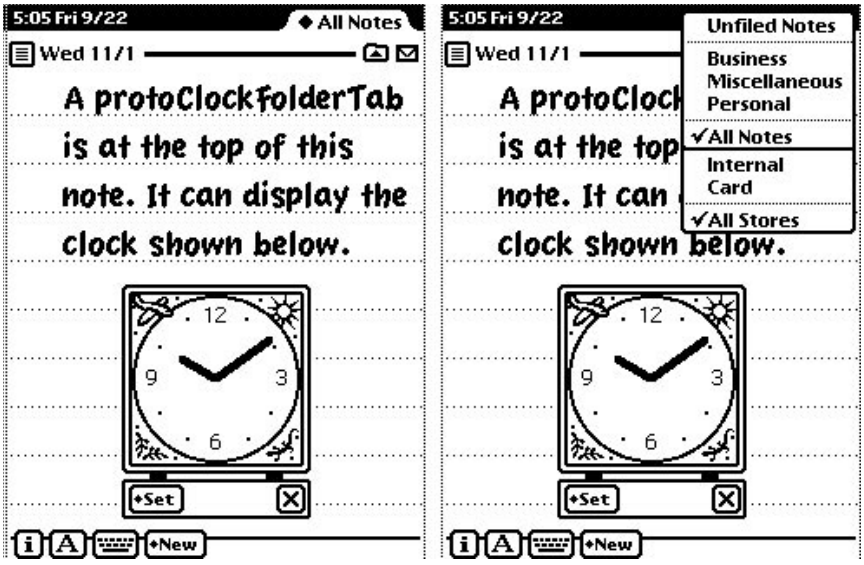
You can override the `protoClockFolderTab` view's display of the `protoSetClock` view by providing a `titleClickScript` method. This method is invoked when the user taps on the time displayed at the left side of the `protoClockFolderTab` view. Your `titleClickScript` method must accept no arguments.

You can also provide a `titleClickScript` method for views based on `protoNewFolderTab`; again, this method accepts no arguments. The `protoNewFolderTab` view displays a text string in the place that `protoClockFolderTab` displays the time.

Tapping the folder tab displays a popup list from which the user can choose a filing category. Your application must filter the display of filed items according to the category selected in this list; hence, the value retrieved from this list is referred to as the filing filter. A checkmark appears next to the currently selected filing filter; the user can tap an item in the list to select a new filing filter. In addition to selecting a filing filter in this popup, the user can specify whether to display items on the internal store, the external store or both; that is, the user can specify a stores filter in addition to a labels filter.

Figure 15-5 shows the folder tab popup list in a view based on the `protoClockFolderTab` proto.

Figure 15-5 Choosing a filing filter



To display items according to the user's choice of store, your target view must supply a `storesFilter` slot. When the target view has a `storesFilter` slot and more than one store is available, the folder tab views allow the user to specify a store in addition to a folder from which data items are retrieved for display. For example, the user might choose to display only entries in the `'business` folder on the internal store.

When the user chooses any filter from this popup menu, the system updates the `storesFilter` or `labelsFilter` slot and sends the target view a `NewFilingFilter` message. The argument passed to this method by the system tells you what changed—the stores filter or the labels filter—but not its new value.

You must supply a `NewFilingFilter` method that examines the `storesFilter` or `labelsFilter` slot and queries your application's soups appropriately. If the value of the `labelsFilter` slot is `nil`, your `NewFilingFilter` method must display all target items. Similarly, if the value of the target view's `storesFilter` slot is `nil`, your `NewFilingFilter` method must display items on all available stores.

Your `NewFilingFilter` method must also perform any other actions that are necessary to display the appropriate data, such as redrawing views affected by the new filter value.

As an adjunct to the `NewFilingFilter` message, you can also use the `RegFolderChanged` function to register your own callback functions for execution when the user adds, removes or edits folder names.

Filing Compatibility Information

Version 2.0 of the Newton operating system supports earlier versions of the Filing interface completely—no code modifications are required for older filing code to continue working under the version 2.0 operating system. However, it is strongly suggested that you update your application to the version 2.0 filing interface to take advantage of new features and remain compatible with future versions of the Newton operating system. This section provides version 2.0 compatibility information for applications that use earlier versions of the Filing interface.

Users can now create folders that are visible only from a specified application; the folders are said to be local to that application. Folders created using previous versions of the filing interface are visible to all applications when first read on a 2.0-based system. Applications can now filter the display of items according to the store on which they reside and according to whether they reside in local or global folders.

The symbols that represent folders are no longer tied to the strings that represent them to the user, as they were in previous versions of the Newton operating system. This new scheme allows you to use the same folder symbol everywhere for a particular concept, such as a business, while

varying the user-visible string representing that folder; for example the user-visible could be localized for various languages.

Applications can now route items directly to a specified store from the filing slip. In addition, registration for notification of changes to folder names has been simplified.

The `protoFolderTab` proto is replaced by the `protoNewFolderTab` and `protoClockFolderTab` protos.

The `protoFilingButton` proto now supplies its own borders. You do not need to enclose the filing button in another view to produce a border around the button.

The `folderChanged` and `filingChanged` methods are obsolete. Both are replaced by the `FileThis` method and the folder change registry. If you define a `FileThis` method, the system does not send `folderChanged` and `filingChanged` messages to your application. Instead of supplying a `folderChanged` method, your application should register a callback function with the folder-change notification mechanism to perform tasks when the user adds, deletes or edits folders.

The `filterChanged` method is obsolete. It has been replaced by the `NewFilingFilter` method.

The new slots `appObjectFileThisIn` and `appObjectFileThisOn` support localization of your application's Filing messages into languages having masculine and feminine nouns.

The `DefaultFolderChanged` function is obsolete. Do not use this function.

The `target` and `targetView` slots are superseded by your override of the `GetTargetInfo` method. If you do not override the system-supplied `GetTargetInfo` method, you must include these slots in your application's base view.

Registration for notification of changes to folder names has been simplified. Use the new functions `RegFolderChanged` and UnRegFolderChanged to register for folder-change notification.

Using The Filing Service

You need to take the following steps to support filing in your application.

- Add a `labels` slot to your application's soup entries.
- Create in your application's base view the slots `appName`,`appAll`, `appObjectFileThisIn`, `appObjectFileThisOn` and `appObjectUnfiled`.
- Supply a filing target—it is recommended that you override the `GetTargetInfo` method; however, if you do not, your application base view must supply `target` and `targetView` slots for use by the default method. You are responsible for keeping the values of these slots current.
- Create a `labelsFilter` slot in your application's target view.
- Implement the `FileThis` and `NewFilingFilter` methods.
- Add a filing button view and a folder tab view to your application.
- register a callback function with the folder-change notification mechanism.

Optionally, you can

- create a `doCardRouting` slot in your application's base view.
- create a `dontStartWithFolder` slot in your target view.
- create a `storesFilter` slot in your application's target view.
- implement support for local folders only or global folders only.
- customize the title text in your `protoNewFolderTab` or `protoClockFolderTab` view.
- customize the action that your `protoClockFolderTab` view takes when the user taps on the time displayed as its title text.

The sections immediately following describe these tasks in detail.

Creating the Labels Slot

Each of your application's soup entries must contain a `labels` slot. It is recommended that you make this slot part of the default soup entry created by your soup entry creation method. (For more information, see the section "Shared Frame Map" in Chapter 11, "Data Storage and Retrieval.")

When the user files the entry, the system stores a value in this slot. Setting the value of the `labels` slot is really the only "filing" that is done—the entry still resides in the soup, but your `FileThis` method provides for the user the illusion that the data has been put in a folder. The system sets the value of the target's `labels` slot for you when the user files the target.

The `labels` slot can store either a symbol or the value `nil`. If the value stored in this slot is `nil`, your `FileThis` method must treat the item as unfiled. If a symbol is stored in this slot, your `FileThis` method must test the value of this slot to determine whether the entry should be displayed and then redraw the display appropriately. Similarly, your `NewFilingFilter` method tests the value of this slot to determine whether to display the item when the filing filter changes.

Creating the appName Slot

You must create in your application's base view an `appName` slot containing a string that is the user-visible name of your application.

Creating the appAll Slot

You must create in your application's base view an `appAll` slot containing a string of the form

`All` *Items*

where *Items* is the plural for the items to be filed, such as cards, notes, and so on. For example, if the user taps the folder tab view in the built in Notes application, the last item in the popup list is "All Notes."

The following code fragment defines a typical `appAll` slot.

```
appAll: "All Notes"
```

Creating the appObjectFileThisIn Slot

You must define the `appObjectFileThisIn` slot in your application's base view. This slot holds the full text of the message to be displayed to the user when filing a single item; for example,

`"File this widget in"`

This string is shown at the top of the Filing slip pictured in Figure 15-1 on page 15-3.

Creating the appObjectFileThisOn Slot

You must define the `appObjectFileThisOn` slot in your application's base view. This slot holds the full text of the string labelling the group of buttons that represent stores in the Filing slip; for example,

`"File this` *item* `on"`

where item is the singular case of the target your application files, such as a card, a note and so on.

For an example of this string, see Figure 15-4 on page 15-6.

Creating the appObjectUnfiled Slot

You must define an `appObjectUnfiled` slot in your application's base view. This slot holds a string of the form

`Unfiled` *Items*

where *Items* is the plural case of the items to be filed, such as cards, notes, and so on. For example, if the user taps the folder tab view in the built in Notes application, the first item in the popup list is "Unfiled Notes."

The following code fragment defines a typical `appObjectUnfiled` slot.

```
appObjectUnfiled: "Unfiled Notes"
```

Specifying the Target

The `GetTargetInfo` method identifies the current target and target view to the system. Depending on your needs, you can use the default version of this method or override it.

The default version of this method relies on values obtained from `target` and `targetView` slots that your application base view provides. You are responsible for updating these slots whenever the filing target changes.

Creating the Target Slot

The `target` slot contains the soup entry with which the user is working, such as the current card or note to be filed. If there is no active item, this slot must have the value `nil`.

Your application must update the value of the `target` slot every time the user views a new item. Because the selection of a new item is an application-specific detail, it is difficult to recommend a means of updating this slot that will be appropriate for every application; however, it is common to update the value of this slot from the `viewClickScript` of the active view.

Creating the TargetView Slot

The `targetView` slot contains the view that receives messages from the Filing service and can manipulate the target. The application's base view is usually an appropriate value for this slot.

Overriding the GetTargetInfo Method

You can implement your own `GetTargetInfo` method if the default version supplied by the system is not suitable for your application's needs. For example, if your application images data items in floating windows or displays more than one data item at at time, you'll probably need to supply a `GetTargetInfo` method that can return an appropriate target and target view in those situations.

To override this method, create in your application base view a slot named `GetTargetInfo` and implement this method as specified in its description on page 15-26.

Sending the GetTargetInfo Message To a Different View

Normally the `GetTargetInfo` message is sent to the application's base view; however, such behavior may not be appropriate for applications having more than one "active" view. For example, the built-in Notepad application can display multiple active notes.

To specify that the `GetTargetInfo` message be sent to a view other than the application base view, your appliation's base view must provide a `GetActiveView` method that returns the view in which the item to be routed resides. The `GetTargetInfo` message is sent to the view specified by the return result of the `GetActiveView` method.

Creating the labelsFilter slot

Your application's target view must supply a `labelsFilter` slot. The system sets the value of this slot for you. The `labelsFilter` slot stores a value indicating the current filing filter (selected from the popup menu in the folder tab view.) This slot can store either a symbol or the value `nil`.

Your `NewFilingFilter` method must update the display of your application's data according to the value of the `labelsFilter` slot.

Creating the storesFilter slot

Your application's target view must supply a `storesFilter` slot. The system sets the value of this slot for you. The `storesFilter` slot stores a value indicating the current store filter (selected from the popup menu in the folder tab view.) This slot can store either a symbol or the value `nil`.

Your `NewFilingFilter` method must update the display of your application's data according to the value of the `storesFilter` slot.

Adding the Filing Button

You need to take the following steps to add the `protoFilingButton` view to your application.

- In NTK, sketch the filing button using the `protoFilingButton` proto and declare it to the application's base view.
- Set appropriate values for the button's `viewBounds` slot.
- Supply the filing button's `buttonClickScript` method. Your `buttonClickScript` method needs to call the inherited `buttonClickScript` method after it's performed the appropriate application-specific tasks.

Adding the Folder Tab View

Your application must display a folder tab in its base view. This is the view that displays the currently selected filing category, as shown in Figure 15-1 on page 15-3. This view is based on the `protoNewFolderTab` or `protoClockFolderTab` system proto. (The `protoClockFolderTab` view is pictured in Figure 15-8 on page 15-25.)

Adding the folder tab view to your application is easy. In NTK, sketch the folder tab in at the top of your application's base view using the `protoNewFolderTab` proto and declare your folder tab view to the application's base view.

Customizing Folder Tab Views

Take the following steps to display your own string as the title text in a `protoNewFolderTab` view:

1. Set the value of a `text` slot in a `title` view that is declared to your `protoNewFolderTab` view
2. Send a `RedoText` message to the `title` view.

For example,

```
SetValue(titleView, 'text, "The text string");
titleView:RedoText();
```

Note
Do not create a `title` slot in the `protoNewFolderTab` view. ◆

Defining a TitleClickScript Method

The folder tab view's `TitleClickScript` method is invoked when the user taps on the title text in a `protoNewFolderTab` view or the time displayed as title text in a `protoClockFolderTab` view. The default `TitleClickScript` method provided for `protoNewFolderTab` views does nothing. The default `TitleClickScript` method of `protoClockFolderTab` views displays a `protoSetClock` view.

You can provide your own `TitleClickScript` method to customize the action your folder tab views take when the user taps on them.

Implementing the FileThis Method

When the user taps the File button in the filing slip, the system sends the `FileThis` message to the target view. Your `FileThis` method must perform any actions necessary to file the target and redraw the current display appropriately.

For example, if your application is displaying an overview list of unfiled items when it receives this message, your `FileThis` method needs to redraw the list without the newly-filed item in it, providing the user-interface illusion that the item has been moved.

Your `FileThis` method must also handle the case in which the user re-files an item in the category under which it already resides. In this case, the appropriate response is to do nothing; unnecessarily redrawing views that have not changed makes the screen appear to flicker or flash. Because the value of the target's `labels` slot does not change unless you change it, you can test this slot's current value to determine whether the new value is different.

The arguments to the `FileThis` method supply all the information necessary to file a soup entry, including the item to file (the target), the category under which it is to be filed (the value to which you set the target's `labels` slot) and the store on which it is to be filed.

If the value of the `labelsChanged` parameter to the `FileThis` method is `True`, your `FileThis` method must use the value of the `newLabels` parameter to update the value of the target's `labels` slot. However, if the value of the `labelsChanged` parameter is `nil`, the value of the `newLabels` parameter is undefined—don't use it!

Similarly, if the value of the `storeChanged` parameter is `True`, your `FileThis` method must move the target to the new store. However, if the value of the `storeChanged` parameter is `nil`, the value of the `destStore` parameter is undefined—don't use it!

The following code example shows the implementation of a typical `FileThis` method. Remember to call `EntryChangeXmit` from this method so that your changes to filed entries are saved!

```
FileThis: // example code - your mileage may vary
func(target, labelsChanged, newLabels, storeChanged, destStore)
   begin
      local update;
      if labelsChanged AND target.labels <> newLabels then
      begin
         target.labels := newLabels;
         EntryChangeXmit(target, kAppSymbol);
      end // labelsChanged
      if storeChanged and EntryStore(target) <> destStore then
         begin
            update := true;
            // move the entry to the new store
            // make sure you handle locked stores too
         end; //storeChanged
      else // we didnt get a valid entry, notify user & bail out
```

```
      :Notify( kNotifyAlert, title,
         LocObj("There is nothing to file.",
         'myApp.nothingToFile));
   if update then // transmit a soup change notification
      local data := {oldSoup: EntrySoup(target), entry: target};
      :XmitSoupChange(destSoup, kAppSymbol, 'entryMoved, data);
end; // FileThis
```

Implementing the NewFilingFilter Method

When the user changes the current filing filter in the folder tab view, the system calls your application's `NewFilingFilter` method. You need to define this method in your application's base view. Your `NewFilingFilter` method must update the query that retrieves items matching the current filing category and perform any other actions that are appropriate, such as redrawing views affected by the change in filing filter.

The symbol passsed as the sole argument to your `NewFilingFilter` method specifies which of the `storesFilter` or `labelsFilter` slots changed in value. This argument does not specify the slot's new value, however. Your `NewFilingFilter` method must use the current value of the specified slot to retrieve those soup entries that fall into the new filing category.

Using The Folder-change Notification Service

You can use the `RegFolderChanged` global function to register callback functions to be executed when the user adds, removes or edit folders. For example,

```
myCallback := func(oldFolder, newFolder);
begin
   // retag entries
end;
RegFolderChanged(kMyCallbackID1, myCallback);
```

The `UnRegFolderChanged` function removes a specified callback from use by the folder change mechanism; for example,

```
UnRegFolderChanged(kMyCallbackID1);
```

Creating the doCardRouting slot

If you want to support routing items to stores from the filing slip, you need to create a `doCardRouting` slot in your application's base view. When an external store is available and the value of this slot is non-`nil`, the filing slip displays buttons allowing the user to route the target to a specified destination store. If this slot has a non-`nil` value but no external store is available, these "card-routing" buttons are not displayed.

Supporting Local or Global Folders Only

If for some reason you need to suppress the display of either local or global folders in the filing slip and the folder tab views, you can do so by setting the values of optional `localFoldersOnly` and `globalFoldersOnly` slots that you supply in your application's base view. Note that this is intended to be an application design decision that is made once, rather than a user preference that can change.

When the `localFoldersOnly` slot holds the value `True`, the filing slip and folder tab views do not display the names of global folders. When the `globalFoldersOnly` slot holds the value `True`, the filing slip and folder tab views do not display the names of local folders.

IMPORTANT ▲
The `localFoldersOnly` and `globalFoldersOnly` must not both hold non-`nil` values at the same time. ▲

Interface to User-Visible Folder Names

The symbols that represent folders are not tied to the strings that represent them to the user. As a result, you can use the same folder symbol everywhere for a particular concept, such as a business, while varying the user-visible

string representing that folder; for example the user-visible string could be localized for various languages.

You can use the `GetFolderStr` function to retrieve the user-visible string associated with a folder symbol.

Filing Reference

This section describes data structures, system prototypes, functions and methods that your application can use to support the Filing service.

Target Information Frame

The frame returned by the `GetTargetInfo` method. The root view supplies a default version of this method that returns frames used by the Filing service. The built-in applications override this method to return their own target information frames. You can override this method to return your own information frame as well. In addition to any slots that you supply, the frame your override method returns must contain the slots described here.

`target`	The data item that is the object of the operation in progress; that is, the item to be filed, routed or otherwise manipulated. The default version of the `GetTargetInfo` methodretrieves this value from a `target` slot that your application's base view provides and maintains.
`targetView`	The view to which the system service sends messages; for example, the view to which Filing sends the `FileThis` message. The default version of the `GetTargetInfo` method retrieves this value from a `targetApp` slot that your application's base view provides and maintains.

`targetStore` If `target` is a soup entry, then this slot holds the store on which the entry resides. The default version of the `GetTargetInfo` method returns `nil` for this value.

Filing Protos

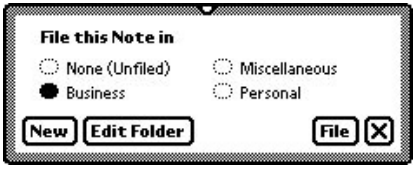
This section describes system-supplied button and folder-tab prototypes that support the Filing service.

protoFilingButton

This proto is used to include the filing button in a view. When the user taps the filing button, the system displays a slip containing a list of categories under which the user can choose to file the currently selected item. When the user taps the File button, the system sends the `FileThis` message to your application.

Here is an example of the filing slip:

Figure 15-6 The filing slip



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

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

The `protoFilingButton` is used in conjunction with the `protoNewFolderTab` to implement filing for an application.

Slot descriptions

`viewBounds`	Set to the size and location where you want the filing button to appear. We recommend that you put the filing button with other buttons on a status bar, if you have one.
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV + vjSiblingLeftH`.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(2) + vfRound(4)`.

protoNewFolderTab

The `protoNewFolderTab` is used in conjunction with the `protoFilingButton` to implement filing for an application. The `protoNewFolderTab` view should be placed at the top of your application view. When the user taps the title on the tab, this proto displays a pop-up list of folders. A check mark appears next to the currently selected folder in this list; the user can tap an item in the list to select a new folder to be displayed. When a new folder is selected, the system sends the `NewFilingFilter` message to your application and collapses the list, displaying the currently selected folder name on the file folder tab. The developer can supply an optional icon displayed at the left of the show bar; when the user taps this icon this proto displays a developer-supplied message to the user.

Figure 15-7 illustrates the folder tab, the pop-up list, an optional icon and a typical user message.

Figure 15-7 protoNewFolderTab

Slot descriptions

`RedoText` This method resets the title text according to the value of a `text` slot your folder tab view supplies and redraws the view.

`TitleClickScript`
Optional. This application-defined method is invoked when the user taps on the title text to the left of the folder tab. The default version of this method does nothing.

IMPORTANT ▲
The `protoNewFolderTab` prototype is not yet available from the Dante Platforms file as of the date of this printing. To access this proto, use the `@669` magic pointer directly. ▲

protoClockFolderTab

The `protoClockFolderTab` is used in conjunction with the `protoFilingButton` to implement filing for an application. The `protoClockFolderTab` view should be placed at the top of your application view. When the user taps the title on the tab, this proto displays a pop-up list of folders. A check mark appears next to the currently selected folder in this list; the user can tap an item in the list to select a new folder to be displayed. When a new folder is selected, the system sends the `NewFilingFilter` message to your application and collapses the list, displaying the currently selected folder name on the file folder tab. When the user taps on the time displayed at the left side of the proto, an analog clock view is displayed.

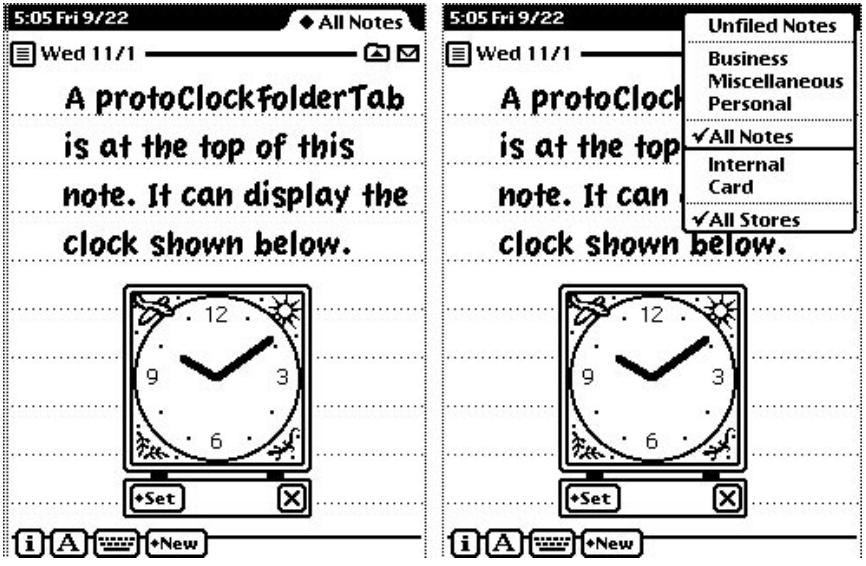
Figure 15-8 shows a `protoClockFolderTab` view as it appears to the user when the popup list of filing categories is collapsed and when it is displayed. This illustration also depicts the `protoSetClock` view that is displayed when the user taps the time displayed in the upper-left corner of the `protoClockFolderTab` view.

Slot descriptions

`RedoText` This method resets the title text according to the value of a `text` slot your folder tab view supplies and redraws the view.

`TitleClickScript`
Optional. This application-defined method is invoked when the user taps on the title text to the left of the folder tab. The default version of this method does nothing.

Figure 15-8 Selecting a filing category in the protoClockFolderTab popup list



IMPORTANT ▲
The `protoClockFolderTab` prototype is not yet available from the Dante Platforms file as of the date of this printing. To access this proto, use the `@670` magic pointer directly. ▲

Filing Methods

This section describes the functions provided by the Filing service.

GetTargetInfo

view`:GetTargetInfo(`*targetType*`)`

Returns a target information frame required by system services such as Filing and Routing. The frame that this method returns specifies the item that is the object of the action (such as the item to file or route), the view to which the system service sends messages (usually your application's base view) and, when necessary, the store onto which the target item is to be moved. The default version of this method is provided by the root view; the built-in applications override this method. You can override this method to provide additional information in your target info frame or to define additional values for the *reason* parameter. If you override this method, your override method must call the inherited version of the `GetTargetInfo` method.

targetType Specifies the operation for which the target information is required. The default method recognizes the symbols `'filing` and `'routing` as valid values for this parameter.

For descriptions of the slots in the target information frame that this method returns, see the section "Target Information Frame" beginning on page 15-21.

To specify that the Filing service send the `GetTargetInfo` message to a view other than the application base view, you must define a `GetActiveView` method in your application's base view.

MoveTarget

root:MoveTarget (*target*, *destStore*)

Moves or copies the specified target to the specified store. If the target is an entry in a read-only soup, its data is copied rather than moved to the destination soup; that is, the original entry is not deleted from the source soup.

The default version of this method moves a soup entry to the same-named soup on the specified store; it is used by the system-supplied filing service. You can override this method to move data other than soup entries. Your override method should call the default method to move soup entries.

target The target data to be moved. If this argument is not a soup entry, the default `MoveTarget` method does nothing; thus, your override method can call the default `MoveTarget` method to handle soup entries.

destStore The store to which this method moves the target data, expressed as an index into the stores array; for example,

```
GetStores()[0]; // the internal store
```

root A reference to the root view, as returned by the `GetRoot` function.

DefaultFolderChanged

`DefaultFolderChanged(`*soupName*`, `*oldFolder*`, `*newFolder*`)`

Updates the specified folder label for all entries in the specified soup and reflects the change in the folder tab view if the current filing category is the one that has changed.

soupName A string that specifies the name of the soup associated with the filing category that was renamed or deleted.

oldFolder A symbol that specifies the folder (label) that was changed.

newFolder A symbol specifying the new name of the folder that was changed. The value of this parameter is `nil` for a folder that has been deleted.

UpdateFilter

`UpdateFilter(`*oldFolder*`, `*newFolder*`)`

Updates the text on the folder tab if the user has renamed the current filing category.

oldFolder A symbol that specifies the folder (label) that was changed.

newFolder A symbol specifying the new name of the folder that was changed. The value of this parameter is `nil` for a folder that has been deleted.

RegFolderChanged

`RegFolderChanged(`*callbackID*`,`*callBackFn*`)`

Registers a callback function to execute when the user adds, removes or edits a folder. The return value of this method is unspecified; do not rely on it.

callbackID Unique symbol identifying the *callBackFn* function to the folder change mechanism. Because this symbol must be unique among all symbols registered with the folder change registry, your application's `appSymbol` or some

variation on it is normally used as this parameter's value.

callBackFn A closure that is called when a folder changes.The function must be of the form
`func(`*oldFolder*`,` *newFolder*`);`
Its parameters are:

oldFolder A string that is the name of the folder that changed.

newFolder A string that is the new name of the folder specified by the *oldFolder* parameter. The value of this parameter is `nil` if the *oldFolder* folder was deleted.

The value returned by the *callBackFn* function is ignored.

UnRegFolderChanged

`UnRegFolderChanged(`*callbackID*`)`

Unregisters the specified callback function from the folder-change notification mechanism. The value returned by this function is unspecified; do not rely on it.

callbackID A unique symbol identifying the closure to be unregistered. This symbol was passed to the `RegFolderChanged` function to register this callback function with the folder-change notification mechanism. Normally, the value of this parameter is the application symbol or some variation on it.

GetFolderStr

`GetFolderStr(`*folderSym*`)`

Returns the user-visible string associated with the specified symbol. Returns nil for symbols not associated with a folder. Returns the string “Unfiled” when passed `nil` as its argument.

folderSym The symbol for which this function returns a folder name string.

RemoveAppFolders

`RemoveAppFolders(`*appSym*`);`

Removes all folders local to the specified application. Any folder used by an application other than the specified application is untouched. Items filed in the removed folders will subsequently be considered unfiled; however, no change notification message is broadcast because the change presumably affects only the caller of this function. Unless your application makes use of global folders, you normally call this function from your appliaction’s `DeletionScript` method. (The `DeletionScript` method is invoked when the application package is scrubbed from the Extras drawer; for more information, see the description of this method in *The Newton Toolkit User’s Guide*.)

GetFolderList

`GetFolderList(`*appSymbol*`, `*localOnly*`)`

Returns an array of symbols representing the folders available for use by the specified application; the symbols are ordered according to an alphabetic sort of the folder strings associated with them. The *localOnly* and *nonSysOnly* parameters can be used to specify whether this function includes global folders in its result.

appSymbol Symbol identifying the application for which this function returns local folders.

localOnly Set to `True` to specify that this function not return the symbols of global folders.

Developer-Defined Filing Methods

The methods described here must be provided by the developer to support the Filing service.

FileThis

targetView:`FileThis`(*item*, *labelsChanged*, *newLabels*, *storeChanged*, *newStore*)

This developer-supplied method must do everything required to file the current data item; this message is sent to the view specified by the `GetTargetInfo` method when an unfiled item is filed or when a previously filed item is moved to another folder.

item	The item to be filed, as specified by your application's `GetTargetInfo` method.
labelsChanged	This value is `True` if the value of the target's filing category changed. For all other values of this parameter, the value of the *newLabels* parameter is undefined.
newLabels	The new label symbol only when the value of the *labelsChanged* parameter is `true`; otherwise, this value is undefined.
storeChanged	This value is `True` if the store specified for filing the target has changed. For all other values of this parameter, the value of the *newLabels* parameter is undefined.
newStore	The new store only when the value of the *storeChanged* parameter is `true`; otherwise, this value is undefined.

IMPORTANT ▲

If you support `FileThis` you are responsible for performing all tasks necessary to file the entry. That is, you must change the value of its `labels` slot and move the entry to the new store as necessary. The Filing service does not handle these changes for your when you supply a `FileThis` method. Note that the 1.x filing interface still works if you do not supply your own `FileThis` method; however, future versions of the Newton operating system may not support the older interface. ▲

NewFilingFilter

targetView`:NewFilingFilter(`*newFilterPath*`)`

The system sends the `NewFilingFilter` message to the target view when the user picks a new category of items in a folder tab. This developer-supplied method must perform any actions necessary to display items in the filing category specified by the `labelsFilter` and `storesFilter` slots. Typically, this method queries the application's soups for items in the new filing category and then redraws views affected by the change in the filing filter.

The value of the *newFilterPath* parameter specifies which of the `storesFilter` or `labelsFilter` slots changed, but does not provide the new value of the specified slot. Your implementation of this method must test the value of the appropriate slot for use in the construction of a query spec.

This method replaces the `FilterChanged` method. If the `NewFilingFilter` method is defined, the `FilterChanged` message is not sent at all. If the `NewFilingFilter` method is not defined, the `FilterChanged` message is sent to the target view. The system uses proto and parent inheritance to find your implementation of the `NewFilingFilter` method.

newFilterPath The filter path that changed, as specified by either of the `'storesChanged` or `'labelsChanged` symbols.

GetActiveView

`GetActiveView()`

This developer-supplied function must return the view to which the system sends `GetTargetInfo` messages. This method is intended for use in applicatins that may display more than a single active view at the same time. For example, the built in Notepad application can display multiple active notes at the same time.

Summary

This section summarizes the Filing service.

Filing Protos

```
protoFilingButton := { ... }
protoNewFolderTab := { ... }
protoClockFolderTab := { ... }
```

Target Information Frame

This frame is returned by the `GetTargetInfo` method.

```
{target: item,// the item to file or route
targetView: view, // filing messages are sent to this view
targetStore: store, // store on which target resides
mySlot: myData// optional - your additional slots
...}
```

Filing Functions and Methods

view:GetTargetInfo(*targetType*)
root:MoveTarget (*target*, *destStore*)
DefaultFolderChanged(*soupName*, *oldFolder*, *newFolder*)
UpdateFilter(*oldFolder*, *newFolder*)
RegFolderChanged(*callbackID*,*callBackFn*)
UnRegFolderChanged(*callbackID*)
GetFolderStr(*folderSym*)
RemoveAppFolders(*appSym*)
GetFolderList(*appSymbol*, *localOnly*)

Developer-Supplied Filing Functions and Methods

targetView:NewFilingFilter(*newFilter*)
targetView:FileThis (*item*,*l abelsChanged*, *newLabels*, *storeChanged*, *newStore*)
baseView:GetActiveView()

Additional System Services

This chapter shows you how your application can use some of the other system services that Newton devices provide. Topics include:

- Providing the capability to undo and redo user actions
- Using idler objects to perform periodic operations
- Using the soup change notification service to broadcast or respond to soup-change messages.
- Using the Notifications and Alarms service to display user messages and execute callback functions at specified times.
- Using progress indicators to provide user feedback
- Using the power registry to execute callback functions when the Newton is powered on or off
- Using the login screen to execute callback functions
- Using a custom help book to provide online help to the user

If you are developing an application that provides any of these objects or services you should become familiar with the material discussed in this chapter.

This chapter is divided into four main parts: an introduction, a conceptual section, a practical section and a reference section.

The first main section, “Introduction To Additional System Services,” provides brief descriptions of the topics and services listed on page 16-1.

The next main section, “About Additional System Services,” provides conceptual information not included in the introductory or practical sections. Where applicable, these sections also provide compatibility information for older Newton devices. Note that this chapter does not provide an “About” section for every topic it discusses.

- “About Undo” describes the mechanism that the system provides for undoing and redoing the user’s most recent action. This section also provides compatibility information regarding previous versions of the Undo mechanism.
- “About Alarms” briefly describes the functions in the alarm library, provides conceptual information regarding the use of alarms and provides compatibility information for using alarms on older Newton devices.
- “About Powering On and Off” describes the cooperative model that Newton devices use to turn power on and off. This section also provides compatibility information regarding previous versions of power-management functions.

The practical section, “Using Additional System Services,”consists of several sections that present code examples describing the use of the services and objects listed on page 16-1. These sections assume familiarity with the conceptual material presented earlier in the chapter.

The last portion of this chapter, “System Services Reference,”provides complete descriptions of all data structures, functions and methods provided by the services and objects described in this chapter.

Introduction To Additional System Services

This section introduces the system services described in this chapter.

Undo

You can register callback functions that the system uses to reverse the actions of functions and methods in your application when the user taps the Undo button. A second tap on the Undo button undoes the actions of the callback function, in effect redoing the user's original action.

Idler Objects

An idler object sends a message to your view periodically to execute the `viewIdleScript` method that you provide for that view. You can perform periodic tasks from any view for which you have installed an idler object.

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

Soup Change Notification

The soup change notification service registers your application to receive messages when data in specified soups change in some way. You can also use this service to broadcast your own change messages when your application makes changes to soup data. For a complete description of this service and its use, see Chapter 11, "Data Storage and Retrieval."

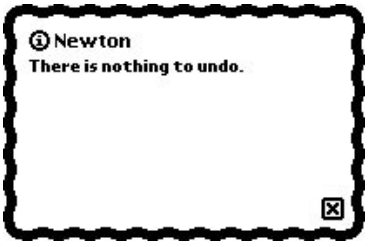
User Notifications and Alarms

The Notification service enables you to display messages to the user at will. In addition to displaying a user alert, the Alarms service provides applications with a way to perform actions at specified times.

User Alerts

The view method `Notify` displays a user alert slip similar to the one shown in Figure 16-1. The last four alert messages are saved by the system, and the user can scroll through them by tapping the universal scroll arrows while the alert view is open. Also, the user can tap on the circled “i” to display the date and time of the message.

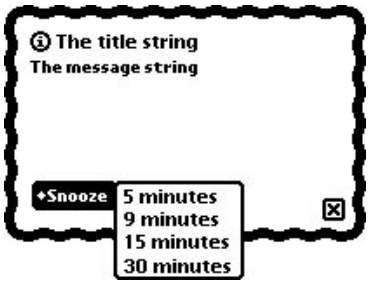
Figure 16-1 User alert



User Alarms

The Alarms service can be used to display an alert slip and perform actions at a specified time. If the Newton is asleep at the time the alarm is to execute, the alarm powers up the Newton and executes the alarm. The user can temporarily dismiss the alert for a specified time period by tapping on the Snooze button included in the alarm slip, as shown in Figure 16-2.

Figure 16-2 Alarm slip with Snooze button



The “Using Alarms” section describes how to set alarms, how to remove alarms and how to obtain system information about currently-scheduled alarms.

Progress Indicators

This section describes the automatic busy cursor and the status slip, which you can use to provide feedback to the user during lengthy operations.

Automatic Busy Cursor

When the system is temporarily unavailable for user input, it displays the busy cursor shown in Figure 16-3. Your application need not do anything extra to support the busy cursor; the system displays it automatically.

Figure 16-3 Busy cursor

Notify Icon

The notify icon is a small blinking star that you can display at the top of the screen to remind the user that an operation is in progress. You can use the notify icon to alert the user in a way that can handle callbacks without using as much screen space as a normal progress slip would require.

Status Slips With Progress Indicators

For complex operations requiring more user feedback than the automatic busy cursor provides, you can use the `DoProgress` function to call the function that implements that operation. The `DoProgress` function displays a status slip that provides graphical progress indicators and informative messages. Figure 16-4 depicts a typical progress slip displayed by the `DoProgress` global function.

Figure 16-4 Progress slip with barber pole gauge

The status slip that this function displays is based on the `protoStatusTemplate` system prototype. You can also use this system prototype to provide your own status slips.

Status slips based on `protoStatusTemplate` may contain any of the following optional items:

- Title text; for example, the name of the application or the operation in progress.

- Message text; for example, “Searching in *ApplicationName*...”, “Connecting to modem…” and so on.
- A bar gauge that displays relative completeness of the current operation as a shaded portion of the entire gauge. Alternatively, the slip can display a simple barber pole gauge. A barber pole gauge animates a set of diagonal stripes while the operation progresses but does not indicate how much of the operation has been completed.

Power Registry

The Power Registry implements a cooperative model for powering the Newton on and off. When power is turned on or off, this service

- notifies registered applications of the impending change in power status
- executes registered callback functions

The Power Registry model is based on the registration of callback functions to be executed when the Newton is powered on or off. You can use the `RegPowerOn` function to register a callback function for execution when the Newton is powered on. The `UnRegPowerOn` function reverses the action of the `RegPowerOn` function; that is, it removes the specified function from the registry of functions called when the Newton is powered on.

Similarly, you can use the `RegPowerOff` function to register a callback function for execution when the Newton is powered off. The `UnRegPowerOff` function removes the specified function from the registry of functions called when the Newton is powered off.

Note that “power-off”callback functions can delay an impending shutdown when necessary. For example, your callback function might delay an impending power off operation until it can successfully tear down a communications endpoint.

Login Screen

Password-protected Newton devices display the login screen when the Newton is powered on. Once the user enters the correct password into the

login screen's keypad, the system dismisses this screen and executes any callback functions registered with it.

The login screen provides a user interface for callback functions executed when the Newton is powered on. If password protection is not enabled at the time the Newton device is powered on, the system does not display the login screen.

Online Help

Your application can provide a `protoInfoButton` view that displays customized online help to the user. Help can also be displayed from the Intelligent Assistant. For a complete description of how to create a help book and integrate it with your application, see version 2.0 of the *Book Maker User's Guide*. For a description of the `protoInfoButton` proto, see Chapter 7, "Controls and Other Protos." For information about displaying online help from the Assistant, see Chapter 17, "Intelligent Assistant."

About Additional System Services

This section provides additional information regarding the system services described in this chapter.

About Undo

This section describes the mechanism that the system provides for undoing and redoing the user's most recent action.

From within each function or method that must support Undo, your application registers a function object that can reverse the actions of the function or method. This function object is called an **undo action**.

The `AddUndoAction` method ties an undo action to a specific view. If that view is no longer open when the user taps Undo, the action does not take

place. Because an undo should generally cause a visible change, it is often desirable to tie undo action to views.

When it is not desirable or feasible to tie an undo action to an open view, you can use the `AddUndoCall` or `AddUndoSend` functions to execute the undo action unconditionally. For example, view methods usually use the `AddUndoAction` view method to post their undo action; however, if the view will not be open when the user taps Undo, you may need to use the `AddUndoCall` or `AddUndoSend` functions to post the undo action. If your action relies on the view being open or some other condition, it must test that condition explicitly.

Undo Compatibility Information

The user interface standards for version 2.0 of the Newton operating system call for the user's second tap on the Undo button to provide a Redo rather than a second level of Undo. Your undo action must create an undo action for itself to implement this interface. For more information, see the section "Using Undo Actions" beginning on page 16-16.

The global function `AddUndoAction` is obsolete, having been replaced by the view method of the same name that provides similar functionality. Existing code that uses the global function can be converted by simply prefixing a colon to the message selector. For example,

```
AddUndoAction(....)
```

becomes

```
:AddUndoAction(...)
```

About Alarms

This section briefly describes the functions in the alarm library. For detailed descriptions and programming information, see the section "Notification and Alarms Reference" beginning on page 16-34. Before actually writing code that uses this library, it's recommended that you also take a few minutes to

try out the Inspector-based examples in the following section, “Working With Alarms From the Inspector.”

You can use either of the `AddAlarm` or `AddAlarmInSeconds` functions to schedule a new alarm. You can also use these functions to substitute a new alarm for one that is currently scheduled but has not yet executed.

The `AddAlarm` function creates each new alarm as a soup entry (thus, alarms persist across warm reboots) and returns the **alarm key** that uniquely identifies the new alarm. This alarm key is a string that you provide as one of the arguments to the function that creates the alarm; it includes your developer signature as a suffix. For example, a typical alarm key might be the string `"lunch:myApp:NewtDTS"` or something similar. Note that only the first 24 characters of an alarm key are significant.

IMPORTANT ▲
Do not manipulate the alarm soup or its entries directly. Use only the interface described in this chapter to manipulate alarms. ▲

Alarm keys are used to manipulate individual alarms. The `GetAppAlarmKeys` function returns the keys for all alarms installed by a specified application. The `GetAlarm` function returns the alarm frame associated with a specific alarm key; similarly, the `RemoveAlarm` function deletes the alarm frame associated with a specific alarm key. The `RemoveAppAlarms` function deletes all of the alarms having a specified key; for example, you could use this function to delete all alarms having keys that end with the suffix `":myApp:NewtDTS"`.

After an alarm executes, it is made available for garbage collection; thus, alarms that have already executed may no longer exist. For this reason, it is unwise to store references to alarms or try to manipulate them by any means other than the interface that the system provides.

An easy way to become familiar with these functions is to experiment with the Inspector-based examples in the “Using Alarms” section.

Courteous Use of Alarms

Your application needs to schedule and use alarms in a way that does not hamper the activities of other applications residing on the user's Newton. While limiting your applications to a single alarm might be too restrictive, scheduling a daily wake-up alarm for the next year by creating 365 different alarms would use up a lot of the internal store. Remember that each alarm you schedule uses space in the internal store. You need to exercise reasonable judgment when creating multiple alarms.

Similarly, your alarm actions should be brief, so they don't interfere with other alarms. If you need to do something time-consuming or repetitive, use a deferred action or set up a `viewIdleScript`.

Alarms Compatibility

This section highlights differences between version 2.0 of the alarms service and previous versions.

Rescheduling Unacknowledged Alarms

If the user is away from the Newton when the alarm goes off and the Newton goes back to sleep before the user returns, the user won't know the alarm went off until powering up the Newton again. Newton system software version 2.0 provides a Persistent Alarms user preference that solves this problem. When this preference is enabled, alarms continue to reschedule themselves until the user acknowledges them. To conserve battery power, the length of time between reappearances of a particular alarm increases in proportion to the number of times it has gone unacknowledged.

When your application is running on system software prior to version 2.0, the best way to handle this problem is to have your callback function continue scheduling alarms, within reasonable limits, until the user acknowledges one. For example, it might reschedule the alarm for one minute later. When that alarm's callback executes, it can reschedule the alarm for 5 minutes later, then perhaps another one in 10 minutes, and so on. This works because the alarms will wake the Newton; however, you need to choose an appropriate time at which to stop rescheduling alarms or you'll

drain the Newton's battery. Consider, for example, the scenario in which the user is on vacation and won't be back for a week.

Common Problems With Alarms

This section describes common problems encountered with use of the alarm library.

Problems With Alarm Callback Functions

Alarms are kept in a soup; thus, they persist across restarts and installation or removal of packages. This means that the arguments you pass to your callback function are also stored in a soup; hence, these arguments are also copied deeply. (See the description of the `DeepClone` method on page 20-9, for more information.) Therefore, you must avoid indirectly referencing large objects lest they unnecessarily inflate the size of your entries in the alarm soup.

A classic example of accidentally referencing a large object is in dynamically creating the function object that is executed with the alarm. Function objects may contain references to the lexical environment and the current receiver (`self`) at the time they are created. For example, function objects created from a view method reference the root view through their parent chain. If you pass such a function object to the `AddAlarm` function, the system attempts to copy the entire root view into the alarm soup. One way to minimize the size of your callback function object is to pass as your callback a one-line function that invokes a method in your application package to actually do the work.

In any case, debugging your callback function will be difficult because any exceptions it raises will be caught and ignored by the alarm mechanism. Thus, you need to debug your callback functions thoroughly before passing them to the `AddAlarm` function.

Note that your application may not be open when its alarm executes. In fact, your application may not even be installed. Your code needs to handle these cases appropriately. The following code fragment shows how to test for the presence of an application before sending a message to it.

```
if GetRoot().(kAppSymbol) then
   GetRoot().(kAppSymbol):DoSomething()
else
   GetRoot():Notify(...)
```

If your alarms aren't useful when your application isn't installed, you should consider simply removing them, using the `RemoveAlarm` or `RemoveAppAlarms` functions, in your application's `removeScript` method. There is no point in wasting space with useless alarms that display `Notify` messages such as "Sorry, this alarm can't execute because the Fwiblob application isn't installed."

Alarm Functions in InstallScript or RemoveScript Methods

The system-supplied alarm functions cannot be installed on the Newton device from an `installScript` or `removeScript` method. In other words, if the alarm functionality is not already present in the Newton and you use one of the five alarm functions in an `installScript` or `removeScript` method, it may not succeed. This situation can cause the Newton to hang, requiring a reset. Note that a reset will cause your `installScript` method to run again. In such a situation the user may need to cold-boot the Newton to finally end the cycle, which will erase all existing alarms, soups, user prefs, and so on.

You can avoid this problem entirely by not using the alarm functions during your `installScript` or `removeScript` methods. Alternately, your `installScript` or `removeScript` methods can spawn deferred actions to execute alarm-related code.

Alarms and Sound

The Alarm panel in user preferences controls the volume of alarm sounds. Do not change preferences behind the user's back.

About Powering On and Off

This section describes the cooperative model that Newton devices use to turn power on and off.

When the Newton device is powered on, the system passes to your “power on” callback function a symbol indicating the reason it was called, allowing it to condition its actions according to the reason the system was powered on. For example, you might perform one set of actions when the user presses the power switch and another set of actions when the device is powered on by the execution of an alarm.

The `'user` symbol indicates that the user pressed the power switch. The `'emergencyPowerOn` symbol is passed any time the Newton device is powered up after an emergency power-off. An emergency power-off is any shutdown in which one or more power-off scripts did not execute. The `'serialgpi` symbol indicates the presence of +5 volts on the serial port general purpose input pin (pin 7). The `'alarm` symbol indicates that the power-on was caused by the execution of an alarm. The `'cardlock` symbol indicates that a PCMCIA card was inserted or removed.

Similarly, when the Newton is powered off, the system passes to your “shutdown” callback function a symbol indicating the reason it was called. This symbol, passed as the value of your callback’s *why* parameter, allows it to condition its actions according to the way the system was powered off. The `'user` symbol indicates that the user initiated the shutdown. The `'idle` symbol indicates the system initiated shutdown after the Newton was left idle for the period of time specified by the user’s Sleep preferences. The `'because` symbol indicates that the Newton powered off for some other unspecified reason.

The system also passes to your callback function a symbol indicating the current status of the shutdown operation. This symbol is passed as the value of your callback’s *what* parameter. The value of this parameter is used as the basis for the cooperative shutdown process. Your callback function must return a value indicating whether to continue the power-off sequence or delay it. This value is passed to all registered shutdown functions, allowing

each of them to indicate whether they are ready to shut down or need time to complete a task.

The `'okToPowerOff` symbol indicates that the system has received a request to shut down. In response to the `'okToPowerOff` symbol, your callback can return the value `True` to specify that shutdown may continue, or it can return the value `nil` to cancel the shutdown process. Note that an`'okToPowerOff` symbol does not guarantee that shutdown will occur—another callback function may cancel the power-off sequence.

The `'powerOff` symbol indicates that shutdown is imminent. If the callback function must first perform an operation asynchronously, such as the disposal of a communications endpoint, it can return the `'holdYourHorses` symbol to delay shutdown. After completing the task for which you delayed shutdown, you must call the `PowerOffResume` function as soon as possible to resume the power-off sequence.

Returning the value `nil` in response to the `'powerOff` symbol allows the power-off sequence to continue. Your callback function must return the value `nil` in response to any symbols other than those described here.

These symbols are summarized inTable 16-3 on page 16-53 and Table 16-4 on page 16-55.

Power Compatibility Information

Applications can now register callback functions to be executed when the Newton powers on or off. All of the functions that provide this service are new.

The `BatteryLevel` function is obsolete. It has been replaced by the `BatteryStatus` function.

About the Notify Icon

To report progress to the user, most applications display a status slip based on `protoStatusTemplate`. Normally, this slip includes a close box that you can use to hide the status slip and add an action to the notify icon. The action shows the status slip again.

Note
Status views that use the `DoProgress` function are an exception to this rule. Do not include a close box in these views. ◆

The notify icon maintains a list of these actions. When the user taps the notify icon, a popup menu of actions appears. Choosing an item from the menu invokes that action and removes it. You can also remove an action by calling the `KillAction` method—for example, if your in-progress task completes while the `protoStatusTemplate` view is hidden, you should close the status view and remove the action from the notify icon's menu.

Using Additional System Services

This section describes how to support the system services discussed in this chapter.

Using Undo Actions

The following code example shows how to provide undo capability in a view. Imagine that you have a view that uses cards. That view has a particular method, `DeleteCard`, that it uses to delete a card. Within the `DeleteCard` method, you call the `AddUndoAction` function, passing as its arguments the name of the card that was deleted and a different method that will add the card (thereby undoing the delete operation). Your call to *view*:`AddUndoAction` would look like this:

```
DeleteCard: func(theCard
            begin
               // statements that delete a card

               // call AddCard as an undo action
               :AddUndoAction ('AddCard, [theCard])
            end,
```

You also need to supply the `AddCard` method, which would look similar to the following example. Note that it too provides an undo action—one that calls the original action, thereby completing the Undo/Redo cycle.

```
AddCard: func(theCard)
         begin
            // statements that add a card
            . . .
            // call DeleteCard as an undo action
            :AddUndoAction ('DeleteCard, [theCard])
         end,
```

Avoiding Undo-Related "Bad Package" Errors

The `AddUndoAction` method saves the current context (`self`) so that later it can send the "redo" message (the argument to the *methodName* parameter of the `AddUndoAction` method) to the appropriate place. As a result, it is possible that references to your application can stay in the system inside the Undo implementation's data structures after the application has been removed. These references can cause -10401 (bad package) errors when the user taps Undo after ejecting the card on which the application resides. Using the `EnsureInternal` function on all parameters passed to the `AddUndoAction` function does not remedy this problem.

You can use the `ClearUndoStacks` function to clean up dangling references to `self` that are left behind by the `AddUndoAction` method. The `ClearUndoStacks` function is generally called from the `viewQuitScript` method of the application base view. You can call this function elsewhere as necessary but you must do so very cautiously to avoid damaging other applications' undo actions.

Note

The `ClearUndoStacks` function deletes all pending undo actions—including those posted by other applications. Use this function cautiously. ◆

Using Idler Objects

This section describes how to install an idler object for a specified view. An idler object sends `viewIdleScript` messages to the view periodically.

Note that an idler object cannot guarantee its `viewIdleScript` message to be sent at precisely the interval you specify. The `viewIdleScript` message cannot be sent until an executing method returns; thus the idler's message may be delayed if a method is executing when the interval expires.

Using an idler object is straightforward; you need to

- Send the `SetUpIdle` message to the view that is to receive the `viewIdleScript` message
- Implement the `viewIdleScript` method that is to be executed.

Note

Do not install idler objects having idle time intervals of less than 100 milliseconds. ◆

Using Soup Change Notification

For a complete description of this service and its use, see Chapter 11, "Data Storage and Retrieval."

Using Alarms and Notifications

This section describes the use of functions, methods and system protos that provide alarms and notifications.

Using the Notify Method to Display User Alerts

Note that the string arguments to the `Notify` method must be wrapped in calls to the `EnsureInternal` global function to avoid invalid references that may occur when the user removes a PCMCIA card. The following code fragment illustrates the correct way to call the `Notify` method.

```
GetRoot():Notify(kNotifyAlert, EnsureInternal("LlamaCalc"),
                EnsureInternal("You've run out of Llamas!"));
```

Working With Alarms From the Inspector

This section shows you how to use the Alarms service by providing Inspector-based experience with the most important alarms functions.

The first thing to try is setting an alarm. The following code fragment sets an alarm to go off one minute from the time you execute the call to the `AddAlarm` function in the Inspector.

```
AddAlarm("foo:NewtDTS",Time()+1,["foo","bar"],nil,nil);
```

The Inspector returns the alarm key associated with the new alarm.

```
#440D559  "foo:NewtDTS"
```

Within a minute, a slip similar to the one shown in Figure 16-2 should appear on the Newton screen. Remember that the `Time` function returns the current time in minutes; thus, if you execute the above code at 2:42:45, then the `Time` function will return 2:42 and the alarm will be set for 2:43, which is only 15 seconds later. To set alarms at more precise intervals, you can use the `AddAlarmInSeconds` function instead of the `AddAlarm` function.

The first argument to the `AddAlarm` function is the alarm key, which is a string used to uniquely identify the alarm for future reference. Use your developer signature as a suffix to generate unique alarm keys just as you would use it to generate unique package names. The second argument is the time at which the alarm is to execute, specified in minutes since midnight January 1, 1904.

The third argument is an array of arguments to pass to the `AlarmUser` function when the alarm goes off. The alarm library calls this function to display the alarm notification slip; the slip contains a snooze button that the user can tap to re-execute the alarm after a specified interval in minutes.) The last two arguments specify a callback function and its arguments. If a callback function is supplied, the system calls it when the alarm executes; this example doesn't use a callback function.

If you do not want to use the alarm slip with snooze button that `AlarmUser` displays, you can pass a different array as the third argument to the `AddAlarm` function. If you pass a three-element array as this argument, the `AddAlarm` function calls the `Notify` function instead of the `AlarmUser` function. The first element of this three-element array is the *level* argument to the Notify function; this argument allows you to specify whether a slip is displayed at all as well as several other variations; for details, see the description of the Notify function in the Utilities chapter of this book. In the following example, a standard notification level is specified by the system-supplied `kNotifyAlert` constant. The remaining two arguments are the title and message passed to the `Notify` function. Thus, to use `Notify` rather than `AlarmUser`, the call to `AddAlarm` would look like the following example.

```
AddAlarm("foo:NewtDTS",Time()+1,[kNotifyAlert,"foo","bar"],nil,nil);
```

The next example demonstrates the `GetAlarm` and `RemoveAlarm` functions. First, use the `AddAlarm` function to set an alarm for two minutes in the future; this should give you enough time to execute the rest of the code example before the alarm is executed.

```
AddAlarm("foo:NewtDTS",Time()+2,[kNotifyAlert,"foo","bar"],nil,nil);
```

Now you can use the `GetAlarm` function to retrieve information about the alarm you just created. This function relieves your application of having to redundantly store alarm information; for example, you need not keep track of the time for which an alarm is set because it is already made available by the `GetAlarm` function.

```
GetAlarm("foo:NewtDTS");
```

The frame returned by the `GetAlarm` function contains all the information you originally passed to the `AddAlarm` function when creating this alarm.

IMPORTANT ▲

Do not modify the frame that the `GetAlarm` function returns. ▲

```
#44116D9    {key: "foo:NewtDTS",
            Time: 46813104,
            notifyArgs: [3, "foo", "bar"],
            callBackFn: NIL,
            callBackParams: NIL,
            …}
```

Note that this frame is a soup entry. Because alarms are kept in a soup, they must observe the restrictions discussed in the section “Common Problems With Alarms” beginning on page 16-12.

The next example uses the `RemoveAlarm` function to remove the alarm. If you executed this code soon enough, the alarm will not have gone off, and will still be available for you to remove. If you did not execute this code soon enough, you may need to create another alarm with which to try out this function. If you want your application’s alarms to execute only when your application is installed, you’ll need to call this function in your application’s `removeScript` method.

```
RemoveAlarm("foo:NewtDTS");
```

The `RemoveAlarm` function returns an unspecified non-`nil` value to indicate it successfully removed the alarm. Currently, it returns the same thing as the `GetAlarm` function, but you must not rely on this fact.

```
#44116D9    {key: "foo:NewtDTS",
            Time: 46813104,
            notifyArgs: [3, "foo", "bar"],
            callBackFn: NIL,
            callBackParams: NIL,
            …}
```

The final example uses the `GetAlarm` function again to confirm that the alarm was indeed removed.

```
GetAlarm("foo:NewtDTS");
```

This call returns `nil`, indicating that no alarm having the specified key was found.

```
#2          NIL
```

The next Inspector example demonstrates the `GetAppAlarmKeys` and `RemoveAppAlarms` functions.

This example uses a loop to set up five different alarms, incrementing the value of the local variable `i` in each iteration to generate a different key for each alarm. If the value of `i` didn't change, the loop would set a single alarm in the first iteration and replace it on subsequent iterations. The `call func()... with` syntax is required to execute a loop from the top level of the Inspector.

```
call func()
begin
local i;
for i := 1 to 5 do begin
   local alarmTime := Time() + i;
   local key := i & "foo:NewtDTS";
   local notifyArgs :=
      [kNotifyAlert,HourMinute(alarmTime),"Alarm#" && i];
   AddAlarm (key,alarmTime,notifyArgs,nil,nil);
   end;
end with ();
```

The next code fragment uses the `GetAppAlarmKeys` function to return the keys for all the alarms set in the previous example. This code passes to `GetAppAlarmKeys` the common suffix shared by all of the alarm keys created in the previous example. The returned keys are sorted in execution order, with the key representing the first alarm to execute occupying the first position in the array.

```
GetAppAlarmKeys (":NewtDTS");
#440B249
["2foo:NewtDTS","3foo:NewtDTS","4foo:NewtDTS","5foo:NewtDTS"]
```

Note that only four alarm keys were returned; this is because the first alarm went off before the `GetAppAlarmKeys` function executed. Although the `GetAppAlarmKeys` function returns only the keys for alarms that have not yet executed, you need to be aware that an alarm may execute the moment after a call to the `GetAppAlarmKeys` function returns; your code must deal with this possibility appropriately rather than assuming that any key returned by this function represents a valid alarm frame.

The final example uses the `RemoveAppAlarms` function to remove all the remaining alarms installed in this example. If your application's alarms can't execute meaningfully when the application is not installed, you can remove them by calling the `RemoveAppAlarms` function from your application's `removeScript` method. The single argument to the `RemoveAppAlarms` function is an alarm key suffix (just as for the `GetAppAlarmKeys` function).

```
RemoveAppAlarms(":NewtDTS");
#C        3
```

The return value of this function is an integer specifying the number of alarms it actually removed. Because another of the alarms scheduled in the example executed before the call to `RemoveAppAlarms` was made, this function returned a value indicating that it removed only three alarms.

For more examples, including a demonstration of the use of callback functions with the alarm library, see the "False Alarm" Newton DTS code sample.

Using Progress Indicators

This section describes how to use the automatic busy cursor, the `DoProgress` method and `protoStatusTemplate` views. For information on using the `SetMessage` method provided by the Find slip, see the section "Reporting Progress to the User," in Chapter 14, "Find."

Using the Automatic Busy Cursor

Your application need not do anything extra to support the busy cursor; it is displayed automatically when the system is temporarily unavailable for user input.

Using the Notify Icon

To add an action to the notify icon and display it, call the `AddAction` method as in the following example.

```
myFunc := func()
          begin
             getroot():sysbeep();
          end;

theAct:= getroot().notifyicon:AddAction("Beep", myFunc, nil);
```

You need to save the result that the `AddAction` method returns. Pass this object to the `KillAction` method to remove the action from the notify icon's list of actions. For example,

```
getRoot().notifyIcon:KillAction(theAct);
```

Using the DoProgress Function

To provide user feedback during a lengthy operation, you can use the `DoProgress` function to display a status view and call the function that implements that operation. The DoProgress function is suitable only for tasks that complete synchronously. To report the progress of asynchronous work, you must display your own `protoStatusTemplate` view and update it yourself, as described in "Using protoStatusTemplate Views" beginning on page 16-26.

The `DoProgress` function acccepts as its arguments a symbol specifying the kind of progress indicator to display (a thermometer gauge or a barber pole), an options frame that allows you to customize the progress-reporting view

further, and a function that actually performs the operation on which you are reporting progress.

You must not allow the user to close the progress slip without cancelling the progress-reporting operation—if the `DoProgress` method sends status messages to a non-existent view, the Newton will hang. You must hide the close button normally provided by the `DoProgress` method. You can do this by including in the options frame that you pass to `DoProgress` a `closebox` slot having the value `nil`, as shown in the following code fragment.

```
local myOpts := {closebox:nil,
                 icon: kMyIcon,
                 statusText: kAppName,
                 gauge: 10,
                 titleText:"One moment, please…"}
```

The function you pass to `DoProgress` must accept as its sole argument the view that displays `SetStatus` strings to the user. For example, to report status while performing the `myFunc` function, pass it as the value of the *workFunc* parameter to the `DoProgress` function, as illustrated in the code fragment immediately following.

```
// options and data are accessible from workFunc because
// the workFunc "closes over" this context
myOpts := { wFnData:"Confabulating",
            closebox:nil,
            icon: kMyIcon,
            statusText: kAppName,
            gauge: 10,
            titleText:"One moment, please."};

myFunc := func (contextView)
   begin
      for x := 1 to 10 do
         begin
```

```
            myOpts.titleText:= :SomeWork(myOpts.wFnData);
            myOpts.gauge := x * 10,
            try
               contextView:SetStatus('vGauge,myOpts);
            onexception |evt.ex.cancel| do
               // do your cleanup here
               ReThrow();
            onexception |evt.ex| do
               //handle anything else
               ReThrow();
         end; // for loop
   end; // workFunc
DoProgress('vGauge, myOpts, myFunc);
```

The `myFunc` function's argument is the view that displays `SetStatus` strings to the user. Note that `myFunc` is structured in a way that permits it to call the progres slip's `SetStatus` method at regular intervals, passing values used to update the progress slip's gauge and message string.

When the user taps the Stop button in the progress view, the `SetStatus` method throws an `evt.ex.cancel 666` exception. You can catch this exception and use it to perform any housekeeping that may be necessary when your work function is not allowed to complete. If no such cleanup is necessary, you can ignore the exception. In either case, however, you must rethrow the exception or `DoProgress` will never return and the Newton will appear to hang. Thus, the call to `SetStatus` is wrapped in an exception-handling block.

Using protoStatusTemplate Views

You need to take the following steps to use a `protoStatusTemplate` view for reporting progress to the user:

- Specify the components of the status slip and their initial values. You can use one of the system-supplied templates or you can create your own template.

- Open the status slip. There are three parts to this process:
  - Instantiate your status view by passing its template to the `BuildContext` function.
  - Initialize the status slip by passing the template's setup frame to the status slip's `ViewSet` method.
  - Invoke the status slip's `Open` method to display the slip.
- Perform the operation on which you are reporting progress. You can implement this operation as a work function that the `DoProgress` method calls or you can just call your own function that periodically updates the status slip. If you use the `DoProgress` method, you must hide your status slip's close box to prevent the user from closing the slip without cancelling the operation; otherwise, serious system problems can occur.
- To update the status slip as the operation progresses, invoke the status view's `UpdateIndicator` or `ViewSet` method as necessary. If you use the `DoProgress` method, you can update the view from within the work function that the `DoProgress` method calls.

Defining Status View and Component View Templates

You can create a status slip view from any of the system-supplied templates `vStatus`, `vGauge`, `vBarber` or `vStatusTitle` or you can create your own template for this view.

Your status view template must have a `_proto` slot that holds the value `protoStatusTemplate`. It may also include

- methods that must be available to the status view or its children.
- any other values needed to set up the status view. You can store these as a frame in an optional `initialSetup` slot. If this slot is present, the view system will use its contents to initialize the status view automatically when it is instantiated.
- your own component view templates and their setup frames.

The component view template must contain the following slots:

`height` Height of the status slip, (not the component view) expressed in pixels. Making this value part of the

component view template allows status slips to resize themselves based on the height specified by their component views.

`name` — Symbol representing this template to the `ViewSet` method. For example, the name of your component template is `'vMyBarber`, this slot should hold the `'vMyBarber` symbol.

`kids` — Array of frames specifying the view templates used to instantiate the components of this view.

For example, the following status template defines a custom barber gauge based on the `protoBarberGauge` supplied by the system. The `myStatusTemplate` template defined in this example includes a `myInitialSetup` frame that is passed to the `ViewSet` method to initialize this view before opening it.

```
myStatusTemplate := {
   _proto: protoStatusTemplate,
   // custom self-animating barber pole
   // pass 'vMyBarber.myInitialSetup to statusView:ViewSet(...)
   vMyBarber:
      {
      height:105,
      name:'vMyBarber,
      kids:
         [
            protoStatusText,
            {  _proto: protoStatusBarber,
               viewIdleScript: func()
                  begin
                  // if hidden, don't bother with updating barber
                     if Visible(self) then
                     // animate barber pole
```

```
                  base:UpdateIndicator({name: 'vMyBarber,
                                        values: {barber: nil}});
               // return number of ticks to wait
               // before returning to viewIdleScript
               300;
            end,
         viewSetupDoneScript: func()
            begin
               inherited:?viewSetupDoneScript();
               :SetupIdle(1);// kick off idle script
            end,
      },
      {  _proto: protoStatusButton,
         text: "Stop",
         // default is statusView:CancelRequest()...
         // buttonClickScript: func() ...
      },
   ],
initialSetup: // used to initialize view automatically
      {
      name: 'vMyBarber,
      appSymbol: kAppSymbol,
      values:
         {
         icon: kDummyActionIcon,
         statusText: "Computing IsHalting…",
         closeBox: func() base:Hide(),
         },
      },
}
```

Opening the Status Slip

To instantiate a status slip view from its template, pass the template to the `BuildContext` method, as shown in the following code fragment.

```
statusView := BuildContext(myStatusTemplate);
```

Next, you need to initialize this view. If your status view template provided an `initialSetup` slot containing a setup frame, the system uses the frame in this slot to perform this initialization automatically.

If your status template does not provide an `initialSetup` slot, you need to set some initial values for the status view from within its `viewSetupDoneScript` method. Pass to the `ViewSet` method a setup frame as described on page 16-46.)

```
statusView:ViewSet(myInitialSetup);
```

Once the view has been initialized, send the `Open` message to display it, as in the following code fragment.

```
statusView:Open();
```

Reporting Progress

Once your status slip is open you can perform the task on which you report progress. You should structure the task such that it alternates between performing some work and updating the status slip.

You can use the status slip's `UpdateIndicator` method to update the gauge view only, as in the following example.

```
statusView:UpdateIndicator({values:{gauge: 50}});
```

To update other items in the status slip, you need to use the `ViewSet` method. Although you can use the `ViewSet` method to initialize all of the status slip's components, you need not always pass all of these values. Once the status slip is open you need only pass to this method the values that are to be changed.

```
statusView:ViewSet({
            name: 'vGauge,
            values: {titleText: "Text at the bottom",
                     gauge: 30} // 30% filled
                  });
```

Because the `ViewSet` method rebuilds all of the status slip's child views and sends the `viewUpdateFormScript` message to all of these views, you'll obtain better performance by calling `UpdateIndicator` rather than `ViewSet` when you just need to update the gauge view. The `UpdateIndicator` method sends the `viewUpdateFormScript` message only to those views having an `indicator` slot that holds a non-`nil` value.

System Services Reference

This section describes functions, methods and data structures that support the system services described in this chapter. Items are grouped according to the system service they support; for example, all of the functions, methods and data structures pertaining to the Find service are described in the Find Reference. Each main section includes subsections that separate system-supplied items from those which the application developer must provide.

Undo Reference

This section describes functions and methods that your application can use to provide Undo/Redo behavior.

AddUndoCall

`AddUndoCall(`*callBackFn*`, `*argArray*`)`

Registers a function object to be called unconditionally when the user taps Undo. The return value of this function is unspecified—do not rely on it.

callBackFn	A function objtect that performs the undo operation.
argArray	Array of arguments to pass to the function object specified by the function parameter

AddUndoSend

`AddUndoSend(`*receiver*`, `*message*`, `*argArray*`)`

Registers a message and arguments to be sent to a specified receiver unconditionally when the user taps Undo. The return value of this function is unspecified—do not rely on it.

receiver	Frame to which the specified message is sent
message	Symbol that is the message to send
argArray	Array of arguments to pass with the message

AddUndoAction

view`:AddUndoAction(`*methodName*`, `*argArray*`)`

Registers with the system an undo action for the specified view.

methodName	A symbol (it must be preceded by a single quotation mark) that is the name of the method to be called when the user taps the Undo button. This method must always return `True`.
argArray	An array of parameters to be passed to the method specified by the *methodName* parameter.

ClearUndoStacks

`ClearUndoStacks()`

This function removes all pending undo actions from the system, including those destined for other applications; hence, this function is best used sparingly and with caution. It is recommended that applications call this method from their `viewQuitScript` method, and only if they have previously called the `AddUndoAction` function.

IMPORTANT
Do not call this function from the application's `removeScript` method. ▲

Idler Reference

This section describes functions and methods you can use to perform periodic tasks.

SetupIdle

view`:SetupIdle(`*milliseconds*`)`

Installs or changes an idler object for the specified view. (An idler object calls the specified view's `viewIdleScript` method periodically.) The `SetupIdle` method always returns `nil`.

milliseconds The number of milliseconds to wait before calling the `viewIdleScript` method for the first time. After the first time, the view's `viewIdleScript` method returns an integer which is the delay until this method is next called.

You can call the `SetupIdle` method at any time to reset the idle time immediately.

To remove the idle routine from the view, call this method again, passing `0` as the value of the *milliseconds* parameter. You can also remove the idler by returning `nil` from the view's `viewIdleScript` method.

Note

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

Do not install idlers that use extremely short intervals (less than 100 milliseconds). ◆

Notification and Alarms Reference

This section describes in detail the alarm and notification functions discussed in this chapter.

Notify

view`:Notify(`*level*`,` *headerStr*`,` *messageStr*`)`

Uses the system notification facility to display a message or otherwise notify the user.

level Specifies the notification level to use and can be one of the following constants:

`kNotifyLog`
: The notice is only entered into the notification log; the user is not alerted.

`kNotifyMessage`
: The user is alerted by a blinking notice icon that a message is pending. Tapping the icon causes pending messages to be displayed in an alert view.

`kNotifyAlert`
: The notice is immediately displayed to

the user in an alert view and the system beep is played.

`kNotifyQAlert`
The notice is immediately displayed to the user in an alert view.

headerStr A string that is displayed as a header on the notice. Usually this is the name of your application or a major component of it.

messageStr A string that is the message to the user.

▲ **WARNING**
String arguments to the `Notify` method must be wrapped in calls to the `EnsureInternal` global function to avoid invalid references that may occur when the user removes a PCMCIA card. ▲

AddAlarm

`AddAlarm(`*alarmKey*`,` *timeSpec*`,` *argsArray*`,` *cbFn*`,` *cbParms*`)`

Registers an alarm to execute at a specified time and returns its alarm key. When the alarm executes, it wakes the Newton if necessary, and displays a specified notification message. You can take additional action by specifying a callback function and its arguments.

alarmKey A string that uniquely identifies the alarm. Note that only the first 24 characters of an alarm key are significant. Use your developer signature or application symbol as a suffix to ensure the uniqueness of this string; for example, `"wakeUp2:llamaApp:NewtDTS"` specifies the `wakeUp2` alarm set by the `llamaApp` application from the developer `NewtDTS`. If an alarm having the specified key already exists, this function removes it and replaces it with the new alarm.

timeSpec The time the alarm is to execute, specified as either an integer or a date frame. If specified as an integer, the value represents the alarm time in minutes since

midnight, January 1, 1904 (similar to the encoding of the value returned by the `Time` function). To specify as a date frame, use the value returned by the `Date` global function.

argsArray — An array of either two or three arguments passed to the function that actually displays the notification slip to the user. Two-element arrays [*title* , *message*] are passed to the `AlarmUser` function when the alarm goes off. See the description of the `AlarmUser` function for details. Three-element arrays [*level*, *title* , *message*]are passed to the `Notify` function. See the the description of the `Notify` function for details.

If the value of *argsArray* is `nil`, the alarm does not call the `Notify` or `AlarmUser` functions when it executes.

cbFn — A function object to be executed when your alarm goes off. Passing `nil` as the value of this argument specifies that no function object is to be executed.

cbParms — Arguments to be passed to *cbFn*. Pass `nil` for this argument if no callback function is being used.

AddAlarmInSeconds

`AddAlarmInSeconds(`*alarmKey*`,` *timeSpec*`,` *argsArray*`,` *cbFn*`,` *cbParms*`)`

Registers an alarm to execute at a specified time and returns its alarm key. This function is the same as the `AddAlarm` function except that it allows you to specify the alarm's execution time more precisely. See the description of the `AddAlarm` function for additional information.

alarmKey — See the description of the `AddAlarm` function.

timeSpec — The time the alarm is to execute, specified as either an integer or a date frame. If specified as an integer, the value represents the alarm time in seconds since midnight, January 1, 1993 (similar to the encoding of the value returned by the `TimeInSeconds` function). To

	specify this value as a date frame, use the value returned by the `Date` global function.
argsArray	See the description of the `AddAlarm` function.
cbFn	See the description of the `AddAlarm` function.
cbParms	See the description of the `AddAlarm` function.

AlarmUser

`AlarmUser(`*title*`, `*message*`)`

Plays an alarm sound and displays a notification slip with a snooze button; this notification slip is illustrated in Figure 16-2.

Normally, the `AlarmUser` function is called by the `AddAlarm` function rather than the application. The `AlarmUser` function respects the user's settings for the alarm sound and volume when executing the alarm. The return value of this function is unspecified; do not rely on it.

title	The string that is the title of the notification slip that this function displays
message	The string that is the body text of the notification slip that this function displays

RemoveAlarm

`RemoveAlarm(`*alarmKey*`)`

The `RemoveAlarm` function unschedules an alarm that has not yet executed. This function returns `nil` if it is unable to find an alarm having the specified key. If the alarm is found and removed, this function returns an unspecified non-`nil` value. If you want your application's alarms to execute only when your application is installed, you'll need to call this function in your application's `removeScript` method.

alarmKey	A string that uniquely identifies the alarm; it is passed to the `AddAlarm` function when the alarm is created. See the description of the `AddAlarm` function for more information.

GetAlarm

`GetAlarm(`*alarmKey*`)`

Returns a frame containing information about the alarm associated with the specified key; this frame and its contents must not be modified.

IMPORTANT

Do not modify the frame that this function returns. ▲

alarmKey A string that uniquely identifies the alarm; it is passed to the `AddAlarm` function when the alarm is created. See the description of the `AddAlarm` function for more information.

The alarm frame returned by this function contains the slots described immediately following; do not rely on the values of any other (undocumented) slots that you may find in this frame.

`key` The alarm key. For more information, see the description of the `AddAlarm` function.

`time` The time at which the alarm is to execute, expressed as the number of minutes since midnight, January 1, 1904.

`notifyArgs` Array of three arguments (or `nil`) to be passed to the `Notify` function when this alarm executes.

`callBackFn` Function object specifying a callback function to be executed with this alarm (or `nil`).

`callBackParams` Array of arguments to this alarm's callback function (or `nil`).

GetAppAlarmKeys

`GetAppAlarmKeys(`*alarmKeySuffix*`)`

Returns an array of all alarm key strings having the specified suffix; if the alarm keys are implemented according to Newton DTS recommendations, this array contains all alarm keys associated with the application using the specified suffix. The returned keys are sorted in execution order, with the key representing the first alarm to execute occupying the first position in the array.

alarmKeySuffix A string used as the suffix in all alarm keys created by a particular application; for example `":NewtDTS"` or `":AlarmSample1:NewtDTS"`

RemoveAppAlarms

`RemoveAppAlarms(`*alarmKeySuffix*`)`

Removes all alarms having key strings ending in the specified suffix; if the alarm keys are implemented according to Newton DTS recommendations, this function can be used to remove all alarms created by a particular application. This function returns an integer value specifying the number of alarms that it removed. If your application's alarms can't execute meaningfully when the application is not installed, you need to remove them by calling this function from the application's `removeScript` method.

alarmKeySuffix A string used as a suffix in all alarm keys created by a particular application; for example `":NewtDTS"` or `":AlarmSample1:NewtDTS"`

Progress-Reporting Reference

This section describes the protos and methods used for progress reporting.

DoProgress

`DoProgress(`*kind*`, `*options*`, `*workFunc*`)`

The `DoProgress` method displays a status slip, calls the function object you pass as one of its arguments, and returns a value indicating how the slip was dismissed. The slip can optionally include a title, message text, and an animated bar gauge or barber-pole progress indicator. This method returns the `'cancelled` symbol when the user cancels the operation; otherwise, this method returns the value `nil`.

kind The kind of gauge view component to display in the status slip. The `'vGauge` symbol specifies that a horizontal progress gauge is to be displayed. The `'vBarber` symbol specifies that a barber-pole gauge is

to be displayed. The value `nil` specifies that no gauge is to be displayed.

options A frame specifying optional characteristics of the progress slip. This frame contains the following slots:

`closebox`	Required. You must place the value nil in this slot to hide the close box normally provided by the status slip.
`gauge`	Required when the *kind* parameter has the `'vGauge` value. An integer specifying the percentage of the operation that has been completed.
`barber`	Required when the *kind* parameter has the `'vBarber` value. The value `True` specifies that the barber-pole gauge is to be animated when the *workFunc* function calls the `SetStatus` method.
`icon`	Optional. A bitmap icon displayed in the upper-left corner of the status slip. Typically it identifies the operation, such as Find, or it identifies the application displaying the progress slip.
`statusText`	Optional. A string displayed at the top of the status slip. It displays the name of the operation in progress or the name of the application that displays the slip. If the slip displays an optional icon, the `statusText` string is displayed to the right of it.
`titleText`	Optional. A string displayed at the bottom of the status slip. This string can be used to provide additional information regarding the operation's progress.

workFunc A function object accepting as its sole argument the view that is the status slip.This function object performs the operation on which `DoProgress` reports status. As

the operation proceeds, this function updates the progress slip's gauge and title text periodically by calling the SetStatus method of the object passed as its argument. For example, the following code fragment does some work and updates the progress gauge and title text with each iteration of the loop.

```
local myOpts := {closebox:nil,
                 icon: kMyIcon,
                 statusText: kAppName,
                 gauge: 10,}
                 titleText:"One moment, please…"}

workFunc := func (contextView) begin
   for x := 1 to 10 do begin
      myOpts.gauge := :SomeWork();
      contextView:SetStatus('vGauge,myOpts);
   end; // for loop
end; // workFunc
```

The following variation displays a barber-pole gauge instead of a progress gauge; the only difference is the substitution of the barber slot for the gauge slot in the frame passed as the second argument to the SetStatus method.

```
func (contextView) begin
   for x := 1 to 10 do begin
      local busyStr := :SomeWork();
      contextView:SetStatus('vGauge,
                     { titleText:busyStr,
                     barber: True}
         end; // loop
      end; // workFunc
```

The parameters to the SetStatus method are the same as the first two parameters to the DoProgress

function. Any slots specified in options passed to the `SetStatus` method override the original slot values passed to the `DoProgress` function; those that are not specified remain as originally passed to the `DoProgress` function.

contextView The view that is the status slip containing the gauge, text and icon displayed by the `DoProgress` method.

SetStatus

contextView`:SetStatus(`*kind*`,` *options*`)`

Updates the status view provided by the `DoProgress` method. The `SetStatus` method must be called from within the work function passed as an argument to the `DoProgress` method. For details, see the description of the *workFunc* parameter to the `DoProgress` method, beginning on page 16-39.

kind See the description of the `DoProgress` method, beginning on page 16-39.

options See the description of the `DoProgress` method, beginning on page 16-39.

protoStatusTemplate

The `protoStatusTemplate` is a configurable status view used to report the progress of lengthy operations to the user. You can use this proto to create views containing animated graphical elements and status messages similar to those used by the built in applications and the system itself.

The `protoStatusTemplate` view, shown in Figure 16-5, is a container view based on `protoFloater` that itself supplies a `protoStatusIcon` view and a `protoStatusCloseBox` view as its view children. The system supplies several special child protos that are used to add graphical elements to this basic container view, which declares itself as the base of this view hierarchy. These child protos are described in the section “Status View Components,” immediately following.

The `protoStatusTemplate` view provides a single method, `ViewSet`, that you can use to initialize or update the set of child views displayed by a `protoStatusTemplate` view.

Figure 16-5 protoStatusTemplate

Slot Description

`initialSetup`	A frame specifying initial values for configuring the status slip and its components. For a complete description this frame, see the description of the `ViewSet` method.

Status View Components

Figure 16-6 illustrates the system-supplied protos used to add view components to a `protoStatusTemplate` slip.

Figure 16-6 Status View Components

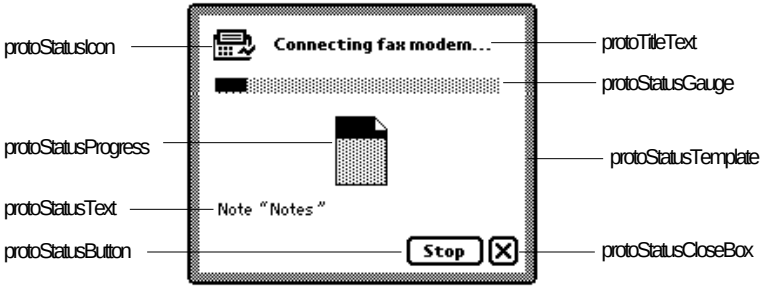


Table 16-1 names the slot that each component checks to update its screen display. To update a particular proto, use the slot specified in the Name Slot column to construct a values frame that you can pass as the `values` argument to the `ViewSet` or `UpdateIndicator` methods.

Table 16-1 Status View Components

Proto	Name Slot	Description
`protoStatusIcon`	`icon`	An icon
`protoStatusText`	`statusText`	Text
`protoTitleText`	`titleText`	Text
`protoStatusProgress`	`progress`	A thumbnail gauge
`protoStatusGauge`	`gauge`	A horizontal gauge
`protoStatusButton`	`primary`	A button
`protoStatusCloseBox`	`closeBox`	A close box
`protoStatusBarber`	`barber`	A horizontal animated barber pole

The value that you supply in the slot specified by Table 16-1 must follow the rules described here for each proto:

`protoStatusIcon`

The value in the `icon` slot must be a bitmap frame, as returned from `GetPictAsBits` function. Store this value in the `icon` slot of your `values` frame.

`protoStatusText`

The value in the `statusText` slot must be a string. Store this value in the `statusText` slot of your `values` frame.

`protoTitleText`

The value in the `titleText` slot must be a string. Store this value in the `titleText` slot of your `values` frame.

`protoStatusProgress`

The value in the `progress` slot must be either a single integer (for example, 50) that reflects the current value of the gauge, or an array of integers giving the current value, minimum and maximum (for example, [50, 0, 100]). By default, the minimum value is 0 and the maximum value is 100. Store this value in the `progress` slot of your `values` frame.

`protoStatusGauge`

The value in the `gauge` slot must be either a single integer (for example, 50) that reflects the current value of the gauge, or an array of integers giving the current value, minimum and maximum (for example, [50, 0, 100]). By default, the minimum value is 0 and the maximum value is 100. Store this value in the `gauge` slot of your `values` frame.

`protoStatusBarber`

Always set the value of the `barber` slot to `True`.

`protoStatusButton`

The value in the `primary` slot must be either a string used as the button text, or a frame with a `text` slot (the button's text) and a `script` slot (the button's `ButtonClickScript` method). If only the string is set, then the default `ButtonClickScript` method calls the application's base view's `CancelRequest` method.

If you specify `nil`, or if you specify a frame and its `text` slot is `nil`, the button is not drawn.

Also, if you include a `ShiftItem` method that returns another view, the button will "adjust" its view if the view returned by `ShiftItem` is not visible.

Store this value in the `primary` slot of your `values` frame.

`protoStatusCloseBox`

The value in the `closeBox` slot must be either `nil` or the close box's `ButtonClickScript` method. If `nil`, then the close box is not drawn. Note that the default behavior is `base:Close`. When not using `DoProgress`, you can override this method to send the `Hide` message and substitute a notify icon for the status template view. For more information, see the sections "Notify Icon" beginning on page 16-6 and "Using the Notify Icon" beginning on page 16-24.

Store this value in the `closeBox` slot of your `values` frame.

ViewSet

statusView`:ViewSet(`*setup*`)`

Initializes or updates status view components and values as specified by the *setup* frame. When this message is sent to a closed status view it sets up the status view's children as specified and must be followed by the Open message to display the view. When this message is sent to an open status view, it redraws the view hierarchy in addition to setting up the view children.

When using this method to initialize the status view—in other words, the first time you invoke this method, before actually opening the status view—you must supply all values that the status view requires, including those specifying the components of the view (such as a `vGauge` indicator) and any values that are appropriate (such as the indicator's position). Once the status view is open, you need only pass those values you need to update, such as

the position of the vGauge indicator—values that are not changed remain in effect.

setup A frame specifying the set of view templates and other values used to instantiate or update the status view. This frame can contain the following slots:

- `appSymbol` The application symbol of the owner application.
- `name` A symbol specifying the termplate that provides one or more components of the status view, such as a gauge, title text, message text, an icon and so on. This symbol can be one of the system supplied values `'vGauge`, `'vBarber`, `'vStatus`, `'vStatusTitle` or a symbol representing your own template. If you provide your own template, be sure to declare all of its component views to the `protoStatusTemplate` view itself.
- `values` A frame containing the values to be set or updated in the view component specified by the `name` slot. This frame may contain slots that supply text, an icon and other configurable elements of the view component specified. This frame must contain a slot named for the value of the `name` slot in the view component to be set or updated. The value of that slot is the value you want the view component to have. Table 16-1 summarizes the `name` slots for each of the system-supplied view components.

  For example, Table 16-1 states that the value of a `protoStatusGauge` view is held in its `progress` slot. Thus, your `values` frame needs to contain a `gauge` slot that contains the current value of the `protoStatusGauge` gauge.

UpdateIndicator

statusView:UpdateIndicator(*setup*)

Updates values and redraws views having an `indicator` slot that holds a non-`nil` value. Use this method only on views that are already open; it sends a `viewUpdateFormScript` message to them. Of the system-supplied component views, this method affects `protoStatusGauge`, `protoStatusProgress` and `protoStatusBarber` views only.

setup — See the description of the `ViewSet` method on page 16-46. You need only include in this frame the values that have changed, rather than the entire setup frame you would pass to the `ViewSet` method.

CancelRequest

statusView:CancelRequest()

This optional method provides an opportunity for you to perform any housekeeping that may be necessary when the user cancels an operation in progress. Typically, you would include in your status slip a Stop button based on `protoStatusButton`. The default `buttonClickScript` method of a `protoStatusButton` view sends the `CancelRequest` message to your application's base view.

Notify Icon Reference

These functions allow you to add and remove actions from the notify icon's list.

AddAction

notifyIcon:AddAction(*title, cbFn, args*)

Registers the specified callback function with the notify icon, adds a text item to the notify icon's menu and returns an object representing the callback function that was added. Your application should save this object to pass to the `KillAction` method.

If no actions were present when the `AddAction` method is called, the notify icon appears. If the menu is displayed when this method is called, its behavior is undefined. (Currently this function closes the menu but you must not rely on this behavior.)

notifyIcon	The notify icon view. You can get a reference to this view by using code similar to the following example. `local icon := GetRoot().notifyIcon`
title	String that appears in the notify icon's popup menu.
cbFn	Function object to be executed when the user chooses the *title* item from the notify icon's menu.
args	Array of arguments to the *cbFn* function. Pass `nil` for this value if the *cbFn* function accepts no arguments.

KillAction

notifyIcon`:KillAction(`*obj*`)`

Removes an action from the NotifyIcon menu. If the action removed is the last action, the NotifyIcon disappears. If the menu is displayed when this method is called, its behavior is undefined.

notifyIcon	The notify icon view. You can get a reference to this view by using code similar to the following example. `local icon := GetRoot().notifyIcon`
obj	Saved object returned when this action was added by the `AddAction` method.

Power Registry Reference

This section describes functions that provide power-management information and functions you can use to register callback functions to be executed when the Newton device is powered on or off.

BatteryCount

`BatteryCount()`

Returns the count of installed battery packs. Battery 0 is always the primary cell pack. Battery 1 is always the backup battery.

BatteryStatus

`BatteryStatus(`*which*`)`

which	An integer identifying the battery for which to return status information. The value 0 specifies the primary battery pack. The value 1 specifies the backup battery.

This function returns a status frame for the specified battery. The status frame contains the following slots:

`batteryType`	Contains one of the following symbols, or an integer: `'alkaline` Battery is standard alkaline `'nicd` Battery is nickel-cadmium `'nimh` Battery is nickel-metal hydride `'lithium` Battery is lithium
`batteryVoltage`	A real number giving the current battery voltage.
`batteryCapacity`	An integer, indicating the percentage of a full charge that the battery contains.
`batteryLow`	An integer, indicating the percentage of a full charge at which the “low battery” warning should be triggered by the system.
`batteryDead`	An integer, indicating the percentage of a full charge at which the “dead battery” warning should be triggered and the unit should be shut down by the system.
`acPower`	Contains a symbol (`'yes` or `'no`) indicating whether or not the unit has AC power applied. Note that this does not imply that the battery is charging. See `chargeState` to determine that.

`acVoltage`	A real number giving the AC voltage being supplied by an AC adapter, or `nil` if AC power is not applied.

`chargeState`	Contains one of the following symbols, or an integer:
	`'discharging`
	The battery is not charging
	`'trickleCharging`
	The battery is trickle-charging
	`'fastCharging`
	The battery is fast-charging
	`'fullyCharged`
	The battery is fully charged

`chargeRate`	An integer giving the number of minutes until the battery is charged or discharged, depending on `chargeState`.
`chargeCurrent`	A real number indicating the current, in milliamps, that is being supplied to charge the battery, if it is charging. If the battery is discharging, this is the current supplied from the battery to the system.
`ambientTemp`	A real number indicating the ambient temperature in degrees Celsius.
`batteryTemp`	A real number indicating the battery temperature in degrees Celsius.

A `nil` value for a slot means the battery driver has no information. The slots containing symbol values (`batteryType`, `chargeState`, `acPower`) may contain integers if the battery driver returned something other than the values listed here.

RegPowerOff

`RegPowerOff(`*callbackID*`,`*callBackFn*`)`

Registers a function object to be executed when the Newton powers off. The arguments passed by the system to your callback function indicate the reason for the shutdown operation and its current state. Your callback must respond to all cases and must return a value indicating to the system whether to proceed with shutdown.

The value returned by the `RegPowerOff` function is unspecified and may change in the future; do not rely on values returned by this function.

callbackID A unique symbol identifying the function object to be registered; normally, the value of this parameter is the application symbol or some variation on it.

callBackFn The function object to be executed when the Newton powers off. This function object accepts two arguments and must be of the form

```
func(what,why) begin... end;
```

This function object must return a value indicating whether to continue the power-off sequence or delay it. When responding to the `'okToPowerOff` symbol, the value `True` specifies that shutdown may continue and the value `nil` cancels the shutdown process. Returning the value `'holdYourHorses` delays the impending shutdown until you call the `PowerOffResume` function.

what The state of the shutdown sequence, as indicated by the `'okToPowerOff` and `'powerOff` symbols. Table 16-4 summarizes the meanings of these symbols.

why The reason for the shutdown operation, as indicated by one of the symbols `'user`, `'idle`, or `'because`.Table 16-3

summarizes the meanings of these symbols.

Table 16-2 Values for *what* parameter to `RegPowerOff` function

Argument	Meaning	Possible Response	Meaning
`'okToPowerOff`	shutdown requested	`nil`	cancel shutdown
`'okToPowerOff`	shutdown requested	`True`	continue shutting down
`'powerOff`	shutdown imminent	`'holdYourHorses`	delay shutdown until `PowerOffResume` is called
`'powerOff`	shutdown imminent	`nil`	continue shutting down
any other value	unspecified	nil	N/A

Table 16-3 Values for *why* parameter to `RegPowerOff` function

Argument	Meaning
`'user`	User cycled power switch
`'idle`	Going to sleep
`'because`	Unspecified

For more information, see “About Powering On and Off” beginning on page 16-14.

UnRegPowerOff

`UnRegPowerOff(`*callBackID*`)`

Unregisters the specified callback function from the power-off notification mechanism. The value returned by this function is unspecified; do not rely on it.

callbackID A unique symbol identifying the function object to be unregistered. This symbol was passed to the `RegPowerOff` function to register this callback function with the power-off notification mechanism. Normally, the value of this parameter is the application symbol or some variation on it.

PowerOffResume

`PowerOffResume(`*callbackID*`)`

This function is used to resume a final power-off sequence which you have temporarily delayed. For details, see the description of the `RegPowerOff` function beginning on page 16-51. The value returned by the `PowerOffResume` function is unspecified and may change in the future; do not rely on values returned by this function.

callbackID A unique symbol identifying the power-off handler that delayed the power-off sequence. This symbol was passed to the `RegPowerOff` function to register the handler with the power-off notification mechanism. Normally, the value of this parameter is the application symbol or some variation on it.

RegPowerOn

`RegPowerOn(`*callbackID*`,` *callBackFn*`)`

Registers a function object to be executed when the Newton powers on. The arguments passed by the system to your callback function indicate the reason the Newton device was powered on.

The value returned by the RegPowerOn function is unspecified and may change in the future; do not rely on values returned by this function.

callbackID A unique symbol identifying the function object to be registered; normally, the value of this parameter is the application symbol or some variation on it.

callBackFn The function object to be executed when the Newton powers on. This function object accepts a single argument and must be of the form

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

why The reason the Newton device was powered on, as indicated by one of the symbols 'user, 'emergencyPowerOn, 'serialgpi, 'alarm, or 'cardlock. Table 16-4 summarizes the meanings of these symbols. For more information, see "About Powering On and Off" beginning on page 16-14.

Table 16-4 Values for *why* parameter to RegPowerOn function

Symbol	Meaning
'user	User cycled power switch
'emergencyPowerOn	Last shutdown did not complete correctly
'serialgpi	+5 volts on serial port GPI Pin (pin 7)
'alarm	Power-on caused by alarm
'cardlock	Card inserted or removed

UnRegPowerOn

`UnRegPowerOn(`*callbackID*`)`

Unregisters the specified callback function from the power-on notification mechanism. The value returned by this function is unspecified; do not rely on it.

callbackID A unique symbol identifying the function object to be unregistered. This symbol was passed to the `RegPowerOn` function to register this callback function with the power-on notification mechanism. Normally, the value of this parameter is the application symbol or some variation on it.

Login Screen Reference

This section describes methods provided by the login screen.

RegLogin

loginScreen`:RegLogin(`*callbackID, callBackFn*`)`

Registers a function object to be executed when the user gets past the login screen—either by entering the correct password or because no password is in use. For tasks involving human interface, use of the sleep screen is usually more appropriate than using a power-on script. The value returned by the `RegLogin` method is unspecified and may change in the future; do not rely on values returned by this function.

For related information, see the description of the `RegPowerOn` function beginning on page 16-54.

loginScreen The view that is displayed just after the splash screen when the Newton is powered on. You can use code similar to the following fragment to obtain a reference to the login screen.

```
local login := GetRoot().loginScreen;
```

callbackID A unique symbol identifying the function object to be registered; normally, the value of this parameter is the application symbol or some variation on it.

callBackFn The function object to be executed when the Newton powers on. This function object accepts no arguments and must be of the form

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

UnRegLogin

loginScreen:UnRegLogin(*callbackID*)

Unregisters the specified callback function from the login notification mechanism. The value returned by this function is unspecified; do not rely on it.

loginScreen The view that is displayed just after the splash screen when the Newton is powered on. You can use code similar to the following fragment to obtain a reference to the login screen.

```
local login := GetRoot().loginScreen;
```

callbackID A unique symbol identifying the function object to be unregistered. This symbol was passed to the RegPowerOn function to register this callback function with the power-on notification mechanism. Normally, the value of this parameter is the application symbol or some variation on it.

Online Help Reference

Please see the following sources for complete information about online user help.

- For information about creating and using application help (help books) see version 2.0 of the *Book Maker User's Guide*.

- For information about opening help books from `protoInfoButton` views, see the description of this proto in Chapter 7, “Controls and Other Protos.”
- For information about using the Assistant to display help, see Chapter 17, “Intelligent Assistant.”

Summary of Additional System Services

This section summarizes the services documented in this chapter.

Undo

`AddUndoCall(`*callBackFn*`, `*argArray*`)`
`AddUndoSend(`*receiver*`, `*message*`, `*argArray*`)`
view`:AddUndoAction(methodName, `*argArray*`)`
`ClearUndoStacks()`

Idlers

view`:SetupIdle(`*milliseconds*`)`

Notification and Alarms

```
AddAlarm(alarmKey, time, argsArray, callBackFn, callBackParams)
AlarmUser(title, message)
RemoveAlarm(alarmKey)
GetAlarm(alarmKey)
GetAppAlarmKeys(alarmKeySuffix)
RemoveAppAlarms(alarmKeySuffix)
view:Notify(level, headerStr, messageStr)
```

Reporting Progress

```
contextView:SetMessage(msgText)// for user msgs in find slip
DoProgress(kind, options, workFunc) // for synchronous tasks
notifyIcon:AddAction(title, cbFn, args)
notifyIcon:KillAction(obj)
```

protoStatusTemplate

```
aProtoStatusTemplate := { // report asynch task progress
_proto: protoStatusTemplate,
viewBounds : boundsFrame,
initialSetup: frame, // initial setup frame
ViewSet: function, // update the view
UpdateIndicator: function, // change one dialog item
CancelRequest: function, // called on cancel
...
}
```

Power Registry

`BatteryCount()`
`BatteryStatus(`*which*`)`
`RegPowerOff(`*callbackID*`,`*callBackFn*`)`
`UnRegPowerOff(`*callBackID*`)`
`PowerOffResume(`*appSymbol*`)`
`RegPowerOn(`*callbackID*`,`*callBackFn*`)`
`UnRegPowerOn(`*callbackID*`)`

Login Screen

loginScreen`:RegLogin(appSymbol,callBackFn)` 115
loginScreen`:UnRegLogin(`*`callbackID`*`)` 116

Online Help

Creating and using: see *Book Maker User's Guide*

Opening from `protoInfoButton` **views:** see Chapter 7, "Controls and Other Protos."

Displaying from Assistant: see Chapter 17, "Intelligent Assistant."

Soup-Change Notification

See Chapter 11, "Data Storage and Retrieval."

Intelligent Assistant

The Intelligent Assistant is a system service that attempts to complete actions specified by the user's written input. The Assistant can be thought of as an alternate interface to Newton applications and services.

The Assistant can complete a number of built-in tasks using the built-in applications and you can program it to execute any task that your application performs. You can also display your application's online help from the Assistant.

This chapter describes how to make application behaviors and online help available from the Assistant. If you want to provide a textual interface to your application or its online help, you should become familiar with the Assistant and the concepts discussed in this chapter.

Before reading this chapter, you should be familiar with the concept of the target of an action, as explained in Chapter 15, "Filing." You should be familiar with the behavior or service that your application provides through the Assistant before reading this chapter. Familiarity with lexical dictionaries is helpful as well, though not essential. Lexical dictionaries are described in Chapter 10, "Recognition."

This chapter is divided into four main parts: an introduction, a conceptual section, a practical section and a reference section.

The introductory section, “Introduction to the Assistant,” describes the behavior of the Assistant’s user interface in a variety of scenarios. Your application’s Assistant services should behave similarly.

The conceptual portion of the chapter, “About the Assistant,”consists of the following sections.

- “Programmer’s Overview” provides a code-level overview of the Assistant and introduces important conceptual material.
- “About Matching” provides detailed information about the process of matching text input to objects representing actions and targets
- “About the Signature and Preconditions Slots” describes how these slots specify the objects required to complete a task.
- “About The Task Frame” describes the information that the Assistant returns after parsing a user input string.
- “Resolving Matching Conflicts” describes the mechanism by which the Assistant resolves conflicts that occur when elements of the user input string match more than one action or target object.
- “About Routing Items From The Assistant” describes how to specify a routing target from the Assistant.
- “Compatibility Information” describes differences between the current version of the Assistant and previous ones.

The practical portion of the chapter, “Using the Assistant,”provides code examples showing how to make application behaviors available through the Assistant. This section assumes familiarity with the conceptual material presented earlier in the chapter.

The last portion of this chapter, “Assistant Reference,”provides complete descriptions of all data structures, functions and methods used by the Assistant.

Introduction to the Assistant

This section describes the Assistant's behavior in a variety of user scenarios.

When the user invokes the Assistant, the system passes the current text selection to it. If no text is selected, the system passes to the Assistant the contents of a buffer that contains the most recent text input.

The Assistant then attempts to match words in the input string with templates and dictionaries that classify the words as actions, targets, or unknown entities.

Depending on the amount of information that parsing the input string provides, the Assistant may attempt to complete a task or it may prompt the user to supply additional information.

Input Strings

The Assistant is pre-programmed with a list of words used by the built-in applications. In addition to using these words, you can program the Assistant to associate words of your own with tasks your application performs. You can associate multiple verbs with a single action; for example, the Assistant maps the words "phone", "call" and "dial" to the same task.The user cannot add words to the Assistant's vocabulary.

The Assistant uses some of the same dictionaries as the recognition system when attempting to classify items in the input string. For example, it uses the system's built-in lexical dictionaries to classify strings having a certain format as phone numbers.

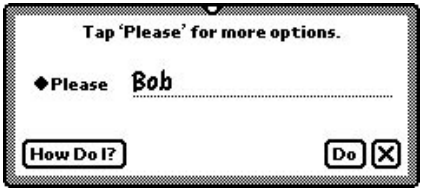
The ordering of words in the input phrase is not significant— for example, the input phrase "Bob fax" produces the same result as the phrase "Fax Bob." This syntax-independent architecture allows easier localization of applications for international audiences.

The input string passed to the Assistant must not contain more than fifteen words.

No Verb in Input String

If the Assistant cannot determine the user's intended action, it displays a string at the top of the Assist slip instructing the user to tap the Please menu for more options. The Please menu allows the user to specify an action when the Assistant cannot determine one. For example, using the string "Bob" as partial input, the Assistant can perform a number of actions: it can find Bob, fax Bob, call Bob, schedule a meeting with Bob and so on. However, this input string does not indicate which of these actions is the user's actual intent. Figure 17-1 shows the Assist slip as it would appear if the string "Bob" were the only input provided to the Assistant.

Figure 17-1 Assist slip



When prompted in this manner, the user must provide additional information on the Assist slip's input line or choose an action from the Please menu. The top portion of the Please menu includes each of the actions currently registered with the Assistant. When you register an action with the Assistant, it automatically appears in this menu. The Please menu is shown in Figure 17-2.

Figure 17-2 The Please menu

The built-in tasks that the Assistant can complete include calling, faxing, finding, mailing, printing, remembering (adding Todo items), scheduling (adding meetings), getting time information from the WorldClock and displaying online help. Note that the top portion of this menu displays only one word for each action the Assistant can perform. For example, the word “call” appears here but the synonyms “ring” and “phone” do not. Recently-used synonyms may appear in the bottom half of the slip, however.

To allow the user to repeat recent actions easily, the bottom portion of the Please menu displays the eight strings most recently passed to the Assistant. Thus, the string “ring Maurice” appears here, even though the action of placing a phone call is represented only by the verb “call” in the top portion of the menu.

When the input string includes a verb that the Assistant can use, the Please slip is not displayed at all. Instead, the Assistant displays a task slip or simply completes the action specified by the input string.

When the user taps the Do button in the Assist slip, the system attempts to match words or phrases from the input string with templates that define actions and targets to the Assistant.

Ambiguous or Missing Information

If the input string specifies an action but does not provide enough information to complete the action, the Assistant tries to fill in as much of the required information as possible; however the user may still need to resolve ambiguities or provide additional information from within a task slip that the Assistant displays for this purpose.

For example, if the input string is “fax bob,” the Assistant can query the Names soup for information such as Bob’s name and fax number.However, the user may still need to correct the task slip if the Assistant chooses the wrong Bob from the Names soup, or the user may need to write Bob’s fax number on the slip if the number was not found.

The Task Slip

In addition to providing a means of correcting missing or ambiguous input, the task slip also provides the user with one last chance to confirm or dismiss execution of the task before the Assistant actually takes action. It’s especially important to provide this opportunity to confirm, modify or cancel the task if executing it will change the user’s current context (open other applications), modify the user’s data or inconvenience the user in some way.

Figure 17-3 Sample IA task slip

About the Assistant

This section provides additional conceptual information about the Assistant, including a programmer's overview.

Programmer's Overview

An application makes a behavior (such as dialing the phone) available through the Assistant by registering with the system a **task template** that defines that behavior to the Assistant. The task template

- specifies the behavior as a named task in its `primary_action` slot.
- specifies additional templates required to complete the task.
- defines slots, data and methods required to complete the task.
- specifies input strings that invoke the task (such as "phone", "call", "dial" and so on.)
- defines a view template for displaying a slip to the user.

Your application's `InstallScript` method calls the `RegTaskTemplate` function to register your task template and its supporting templates with the Assistant. Your application's `RemoveScript` method calls the `UnRegTaskTemplate` function to unregister your templates when your package is removed.

The task template relies on other templates that define actions and potential targets of those actions. An action, such as "call" or "pay", is defined by an **action template.** The target of the action, such as "Bob" or "Apple", is defined by a **target template**.

When an input string is passed to the Assistant, it attempts to match words and phrases with strings in the `lexicon` slot of currently-registered templates. For example, if the lexicon slot for a particular template holds the words "call," "dial," and "phone," this template is matched when the input string contains any of the words "call" "dial" or "phone."

When a word or phrase in the input string matches a word or phrase in a template's `lexicon` slot, the Assitant creates a frame from that template. Frames created from action templates are called **action frames** and frames created from target templates are called **target frames.**

If the Assistant can match an action frame with the `primary_act` slot in a task template, it uses that template to create a **task frame**. The task frame holds all of the information defined in the task template as well as several additional slots added by the Assistant when parsing the input string.

If the Assistant cannot match an action frame with any of the task templates currently registered, it displays the Assist slip and prompts the user to specify an action.

If the Assistant matches more than one action frame with a currently registered task template's `primary_act` slot, it resolves the conflict by determining whether the additional frames required by the task template are present. The conflict-resolution process is described in the section "Resolving Matching Conflicts" beginning on page 17-15

Once the Assistant has created a task frame, it calls a `postparse` method that you must supply. The `postparse` method is where the Assistant hands control to your application. Your `postparse` method must perform any

actions required to complete the primary task. For example, this method may create a task slip and display it to the user, or it can perform the task, or it can look for additional information in slots that the Assistant returns in the task frame.

The `postparse` method commonly acts as a dispatching method that invokes the subtasks comprising the primary action; however, it can also be used to retrieve information that the Assistant was unable to find on its own. For example, if the user input phrase is “Call to Bob” and the word “Bob” does not match any entries in the Names soup, the target of the action “call” is still unknown to the Assistant. However, because the word “to” appears in the call action template’s lexicon, your `postparse` method can find the word that appears after “to” in the original phrase and try to use it as the target of the action. Because this word, “Bob,” does not currently match an existing entry in the soup, the `postparse` method uses this information to fill out a task slip instead of calling Bob. After the user supplies the information missing from the task slip, the `postparse` method can perform the primary action that the task frame specifies, perhaps adding Bob’s name and fax number to the Names soup in the process.

IMPORTANT

Once your `postparse` method is invoked, it is in complete control of Newton. Your `postparse` method must include any code necessary to complete the primary action, such as code required to display a task slip, validate input, and handle errors. ▲

About Matching

The process of extracting words or phrases from user input and matching them to templates registered with the Assistant is carried out by the `ParseUtter` function. This section describes the matching process in more detail and discusses strategies for dealing with unmatched words, partially-matched phrases and words that match multiple templates.

When parsing the input string, the Assistant matches entire words only. For example if the word `"telephone"` appears in a template’s `lexicon` slot,

that template is not matched when the word "phone" appears in the input string.

The Assistant's matching process is case-insensitive; thus, if the word "phone" appears in a template's lexicon slot, that template is matched when the word "Phone" appears in the input string.

If absolutely nothing in the input string is matched, the return result of the ParseUtter function is unspecified. However, if any word in the input string is matched, the ParseUtter function returns a frame containing objects created from the matched words and additional information about matched words. The matching process and the result frame returned by the ParseUtter function are described in the section "Programmer's Overview" beginning on page 17-7.

When none of the words in the input string matches an action template, the Assistant may use the information it did match to try to determine a likely action. For example, when the user enters the phrase "buzz 555-1234", the Assistant does not match the word "buzz" to an action template but it can identify "555-1234" as having the format of a telephone number. Based on that information, the Assistant creates a task frame from the built-in call_act template and displays a call slip to prompt the user for additional information.

The Assistant ignores words that do not appear in any template's lexicon, thereby allowing the user to enjoy more natural interaction with Newton. Rather than limiting the user to terse commands, the Assistant extracts meaningful words from phrases such as "Make a phone call to Bob at work" and ignores the others.

Unmatched words appear in the noisewords slot of the frame that the ParseUtter function returns. Your postparse method may be able to use the contents of this array to determine further information about the user's intended action. For example, if there are no entries for Bob in the Names soup, the word "bob" is not matched and is returned as an element of the noisewords array. The word "to" is also likely to show up in this array. Because words are added to this array in the order they were parsed, your postparse method can extract the word following "to" from the noisewords array and attempt to use it as a target. The recommended action

in this sort of situation is to fill in a task slip with this information and display it to the user for confirmation.

When a word appears in the lexicon of more than one template it can cause the Assistant to match the wrong template. For example, two games might both register action templates having the word “play”in their lexicon, or you might attempt to register a template that duplicates a word in the lexicon of one of the system-supplied templates. A strategy for resolving these kinds of conflicts is described in “Resolving Matching Conflicts” beginning on page 17-15.

About the Signature and Preconditions Slots

Your task template must define two slots, called `signature` and `preconditions`, which store arrays of template names and arrays of symbols, respectively.

The `signature` slot specifies the templates that create frames required to complete the primary action. For example, to send a fax the Assistant needs to have frames representing the action of faxing, a fax number, the name of the person to whom the fax is sent and the time the fax is to be sent. The following code fragment specifies by name the templates required to create these frames.

```
signature: [fax_action, fax_number, who_obj, when_obj],
```

When the Assistant parses the input phrase “fax Bob” it matches the word “fax” to an element in the `lexicon` slot of the `fax_action` template, causing it to create an action frame from that template.

All of the words representing elements of the `signature` array do not necessarily need to appear in the input string; for example, the Assistant might supply the `fax_number` target frame by finding Bob’s fax number in the Names soup and matching it to the `fax_number` template to create that target frame.

However, each of elements of the `signature` array must be matched one way or another in order to complete the task. This is a key point: if any frame represented in the `signature` array is not present (that is, if its lexicon was

not matched) the Assistant does not attempt to execute the primary_act and your `postparse` method is not called.

The `preconditions` slot holds an array of symbols specifying the names of slots that the Assistant creates in the task frame. These slots hold frames created from the templates in the `signature` array. For example, the `preconditions` array in the following code fragment specifies that the task frame must contain `action`, `number`, `recipient` and `when` slots.

```
preconditions: ['action, 'number, 'recipient, 'when],
```

Each element of the `preconditions` array is related to the corresponding element of the `signature` array; specifically, an element of the `preconditions` array specifies the name of the slot that is to hold the frame created from the template in the corresponding element of the `signature` array. For example, when the word “fax” is matched to the `fax_action` template that is the first element of the `signature` array, the action frame created is stored in an `action` slot named for the symbol that is the first element of the `preconditions` array. Similarly, the target frame created by matching a fax number with the `fax_number` template occupying the second element of the `signature` array is stored in a `number` slot named for the symbol that is the second element of the `preconditions` array, and so on.

The Assistant creates a task frame slot only when a template is matched. For example, if the `when_obj` is not matched, the `when` slot is not created.

About The Task Frame

When your `postparse` method is called, the the `ParseUtter` function will have added slots to the task frame. You can use the information in these slots for your own purposes. For example, your `postparse` method can extract additional information for use in filling out a task slip.

This section describes the `entries`, `raw`, `phrases` and `noisewords` slots that the `ParseUtter` function adds to the task frame. These slots are only added when they are required to hold information; for example, if no soup

entries are matched in the parsing process, the `entries` slot is not added to the task frame.

About the Entries Slot

If a template matches a soup entry, an alias to that entry is returned in the `entries` slot of the result frame that the `ParseUtter` function returns.

About the Raw Slot

The result frame returned by the `ParseUtter` function contains a `raw` slot that holds an array. Each element of the array is itself a simple array, which lends great flexibility to this data structure. The `raw` array may sometimes include as one of its elements the original unparsed user input string.

When the user enters the phrase “fax Bob,” the `raw` array looks like the following example.

```
raw: [[{isa: {value: "action"},
         Lexicon: [["fax"]],
         value: "fax"}],
      [{isa: {isa: {}},
        value: "bob"}]],
```

This example illustrates the complex nested data structures that are commonly found in this array. Because the templates supplied for application developers are themselves derived from other templates, it is not unusual to for nested frames and arrays to appear as the elements of the raw array.

When necessary, you can examine elements of this array for objects or information the Assistant did not find when parsing the input string initially. For example, phone and fax numbers appear in the value slot after an input string has been parsed, or you can examine lexicon arrays and isa slots to determine how strings were parsed.

Fortunately, you’ll almost never need to navigate the raw array programmatically. It is returned mainly for debugging purposes. The useful information in this array is returned to you in other task frame slots.

About the Phrases Slot

The result frame returned by the `ParseUtter` function contains a `phrases` slot that holds an array of strings. Each word in the input phrase that matches a template is returned as an element of the `phrases` array.

For example the following code fragment depicts the `phrases` array that might be returned after parsing the user phrase “call Bob.”

```
// input string is "call Bob"
phrases: ["call", "Bob"],
```

Elements of the `phrases` array can store more than a single word. For example, if the parser searches the Names soup for an object of the type `who_obj` having the value `"Bob Anderson"`, that element of the phrases array stores the entire string `"Bob Anderson"`, as the following code fragment shows.

```
// input string is "call bob anderson"
phrases: ["bob anderson", "call"],
```

Note that strings may appear in the `phrases` array in a different order than they did in the original input to the Assistant.

Parsing for Special-Format Objects

In all of the examples shown thus far, the Assistant stored the type of the phrase in the raw array and stored the phrase itself in the phrase array. Some objects, however, can only be parsed for correctly by using a lexical dictionary describing their format. These objects include formatted numbers such as phone numbers, currency values, dates, and times.

When the Assistant uses a lexical dictionary to parse an object, it stores the parsed phrase as a single string in the `value` slot. For example, if the user had entered the phrase “Call Bob at 343-4594” the Assistant would store the phone number in the `value` slot, as shown in the following example.

```
raw: [[call_act]
      [lex:{isa: phone_obj,
```

```
                    lexicon: ["phone number"]},
                  value: "343-4594"}]
        ],
```

Note that the `value` slot is only created when the Assistant parses a special-format string.

Resolving Matching Conflicts

Of course the best way to resolve a template-matching conflict is to avoid one by ensuring that the names of your templates are unique. You can do so easily by appending your developer signature to the names of templates and slots you define. However, template-matching conflicts may still arise despite the best of intentions. This section describes the means by which the Assistant attempts to resolve such conflicts.

When a verb matches more than one action template, the Assistant must choose the appropriate action template. For example, imagine that two games, Chess and Checkers, are currently loaded in the Extras drawer. If both games specify the word “play” in their action template’s lexicon, the Assistant cannot open the correct application by simply matching this verb; the correct game to open must be determined by some other means. For example, the code fragment below shows the template for an action, `play_act`, that might conceivably be defined by both games.

```
play_act := { value: "play action",
              isa: 'dyna_user_action,
            lexicon: [ "play"  ]
            };
```

The task templates for each of these games might look like the following example. Note that both templates define `play_act` as the primary action.

```
chessTemplate := { value:  "Chess Template",
                       isa:  'task_template,
              primary_act:  play_act,
            preconditions:  ['generic_action,'action, 'game],
                signature:  ['dyna_user_action, play_act, chess_obj],
                postparse:  func()
                            begin
                              print("we made it, chess!");
                            end,
                    score:  nil},

checkersTemplate := { value:  "Checkers Template",
                        isa:  'task_template,
             primary_act:  play_act,
           preconditions:  ['generic_action, 'action, 'game],
              signature:  ['dyna_user_action, play_act, chkrs_obj],
               postparse:  func()
                           begin
                             print("we made it, checkers!");
                           end,
                   score:  nil},
```

If you specify the `'dyna_user_action` symbol as the first element of your task template's `signature` slot, any target (noun) that distinguishes one template from another enables the Assistant to select the correct template. For example, the two target templates in the following code fragment represent the games Chess and Checkers, respectively.

```
chess_obj := { value: "chess obj",
                  isa: 'dyna_user_obj,
              lexicon: [ "chess" ]
              };
```

```
chkrs_obj := {  value: "checkers obj",
                    isa: 'dyna_user_obj,
             lexicon: [ "checkers" ]
            };
```

The fact that the lexicons in these templates do not match allows the Assistant to resolve the ambiguity. When the string “play chess” is passed to the Assistant, the word “play” is matched to the `play_act` template and the Assistant creates an action frame from that template. Similarly, the Assistant matches the word “chess” with the `chess_obj` template’s lexicon and creates a target frame from that template.

Note that the `play_act` action frame can be matched to the `primary_act` of either the `chess_Template` or the `checkers_Template`. Because the signatures of both of these templates specify the `dyna_user_action` and `play_act` frame types as their first two elements, the conflict is not resolved until the third element of one of these `signature` arrays is matched.

The `chess_obj` target frame matches the third element of the signature for the `chess_template`. It does not match any elements of the signature in the `checkers_Template`. Thus, matching the word “chess” resolves the conflict because the only `signature` array that has been matched completely is that of the `chess_Template`.

About Routing Items From The Assistant

When routing an item from the Assistant—for example, when filing, faxing, printing or mailing the item— the Assistant sends a `GetTargetInfo` message to your application. The root view supplies a `GetTargetInfo` method that returns information such as the item to be routed and the view that is able to manipulate it. This method relies on `target` and `targetView` slots supplied by your application’s base view. You can define your own `GetTargetInfo` method if you need to supply different target information. For more information, see the section “Specifying the Target” beginning on page 15-14 in Chapter 15, “Filing.”

Normally the `GetTargetInfo` message is sent to the application's base view; however, such behavior may not be appropriate for applications having more than one "active" view. For example, the built-in Notepad application can display multiple active notes.

To specify that the `GetTargetInfo` message be sent to a view other than the application base view, your appliation's base view must provide a `GetActiveView` method that returns the view in which the item to be routed resides. The `GetTargetInfo` message is sent to the view specified by the return result of the `GetActiveView` method.

For detailed information on supporting routing in your application, see Chapter 2, "Routing Interface," in *Newton Programmer's Guide 2.0: Communications.*

Compatibility Information

The Assistant now matches entire words only, instead of allowing partial matches.

The Assistant no longer upcases the words that it returns.

The Assistant now adds the eight most recently parsed phrases to the bottom of the Please pop-up menu. Phrases do not need to be interpreted successfully by the Assistant to be included in this list.

The result frame returned by the `ParseUtter` function contains a new slot named `entries`, which stores aliases to soup entries that were matched when parsing the input string.

Using the Assistant

This section describes how to make an application behavior available through the Assistant, as well as how to display online help from the

Assistant. This material presumes understanding of the conceptual material provided in previous section.

Making Behavior Available From the Assistant

You need to take the following steps to make an application behavior available through the Assistant.

- Create an action template for your primary action. If necessary, create additional action templates that define subtasks of the primary action.
- Create zero or more target templates. (Some actions require no target; others may use system-supplied target templates.)
- Implement your `postparse` method.
- Create a task template.
- Register and unregister your task template with the Assistant at the appropriate times.

The sections immediately following describe these tasks in detail.

It is recommended that you begin by defining the action and target templates required to complete the primary action. After doing so, you will have a better idea of the tasks your `postparse` method will need to handle. After creating all the necessary templates and writing a suitable `postparse` method, defining the task template itself is likely to be a trivial task.

Defining Action and Target Templates

Action templates and target templates are simply frames that contain a specified set of slots and values.You need to define templates for all of the actions and targets required to complete a task. One of the action templates must define the primary action.

Take the following steps to define an action template or a target template.

- Define a frame containing the `value`, `isa` and `lexicon` slots.
- Assign the frame to a slot or variable that is the name of the template.

- Place in the `value` slot a string that identifies the action or target that this template defines.
- If the template defines an action, place the symbol `'dyna_user_action` it its `isa` slot. If the template defines a target, place the symbol `'dyna_user_obj` in its `isa` slot.
- Define the words or phrases this template is to match as an array of strings in its `lexicon` slot. If you place the name of a system-supplied template in your template's `isa` slot, your template inherits its lexicon from the system-supplied template; however, you should be aware that `isa` slot values other than the symbols `'dyna_user_action` and `'dyna_user_obj` may interfere with the system's ability to match your template successfully. For more information, see the section "Defining Your Own Frame Types To The Assistant," immediately following.

Sample Action Template

The following code fragment defines an action template called `cbPayAction` that might be used by a home banking application.

```
cbPayAction := {
   value: "Pay Action", // name of action
   isa: 'dyna_user_action, // must use this value
   lexicon: ["pay", "paid"] // words to match
};
```

Sample Target Template

The following code fragment defines a target template called `cbPayee` that might be used by a home banking application.

```
cbPayee := {
   value: "Who Object",// name of target
   isa: 'dyna_user_obj, // must use this value
   lexicon: ["to", "Bob"]// words to match
   };
```

Defining Your Own Frame Types To The Assistant

The conflict-resolution mechanism relies on the use of system-supplied dynamic user object templates. You can define your own symbol for your template's `isa` slot as long as it ultimately refers to a template having one of the symbols `'dyna_user_action` or `'dyna_user_obj` as the value of its `isa` slot. For example, you can define a `my_action` template that is a `'dyna_user_action` , as shown in the following code fragment.

```
my_action := {
  value: "my action", // name of this action
  isa: 'dyna_user_action, // must use this value
  lexicon: ["jump", "hop"] // words matching this action
};
```

Based on the definition above, you can derive a `my_other_action` template that holds the value `'my_action` in its `isa` slot, as in the following example.

```
my_other_action := {
  value: "my other action", // name of this action
  isa: 'my_action, // subclass of 'dyna_user_action
  lexicon: ["leap", "lunge"] // words matching this action
};
```

You can take a similar approach to define your own target object types by placing the `'dyna_user_obj` symbol in your target template's `isa` slot. For example, you can define a template `my_target` that is a `dyna_user_obj` and use its symbol in this slot also, as in the following code fragment.

```
my_target := {
  value: "my target", // name of this target
  isa: 'dyna_user_obj, // must use this value
};
```

Based on the definition above, you can derive another target template from `my_target`, and store the value `'my_target` in its `isa` slot, as in the following example.

```
my_other_target := {
  value: "my other target", // name of this target
  isa: 'my_target, // subclass of 'dyna_user_obj
};
```

Implementing the Postparse Method

Your `postparse` method implements the behavior that your application provides through the Assistant. This method resides in the `postparse` slot of your task template. It is called after all of the templates in the `signature` slot have been matched.

Your `postparse` method must provide any behavior required to complete the specified task, such as obtaining additional information from the `ParseUtter` result frame as necessary and handling error conditions.

Sample Postparse Method

The following code fragment is an example of a `postparse` method that tests for the existence of the `value` slot and accesses its content.

```
postParse: func()
begin
  local index, thePhraseType, thePhraseText;

  for index := 0 to Length(raw)-1 do
  begin
   if ClassOf(raw[index]) = 'Array then
   begin
     thePhraseType := raw[index][0];
     thePhraseText := phrases[index];
   end;
   else
```

```
        if ClassOf(raw[index]) = 'lex then
        begin
          thePhraseType := raw[index][0].isa;
          if hasSlot(raw[index][0], 'value) then
           thePhraseText := raw[index][0].value;
          else
           thePhraseText := phrases[index];
        end;
        // Display phrase text in task slip
       end;
    end;
```

Defining the Task Template

Your task template defines a primary action and its supporting data structures.

Take the following steps to define a task template.

- Define a frame containing the `value`, `isa`, `primary_act`, `postParse`, `signature`, `preconditions` and `score` slots. Subsequent bullet items in this section describe the contents of these slots.
- Assign the frame to a variable that is the name of the task template.
- Place in the `value` slot a string that identifies the task that this template defines.
- Place the `'task_template` symbol in the `isa` slot.
- Place the name of the slot or variable defining your primary action in the `primary_act` slot.
- Place the name of the slot or variable defining your `postparse` method in the `postparse` slot.
- Place in the `signature` slot an array of the names of all action and target templates required to complete this task.
- Place in the `preconditions` slot an array of symbols specifying the names of slots the Assistant must create to hold frames built from the

templates specified in the `signature` array. The `preconditions` array and the `signature` array must have the same number of elements. Furthermore, the symbols in the `preconditions` array must appear in the same ordinal position as their counterparts in the `signature` array; that is, the first element of the `preconditions` array is related to the first element of the `signature` array, the second element of the `preconditions` array is related to the second element of the `signature` array, and so on. For more information, see the section "About the Signature and Preconditions Slots" beginning on page 17-11.

- Place the value `nil` in the score slot. This slot is used internally by the Assistant.

Sample Task Template

The following code fragment is an example of a task template. This template might be used to implement an action in a home banking application.

```
payTemplate := {
   value: "Pay Template", // name of this template
   isa: 'task_template, // you must use this value
   primary_act: cbPayAction, // primary action
   postparse: checkbook.postParseScript,
                        // postparse method
   signature: [cbPayAction, cbWho, cbAmount, cbWhen],
                        // required templates
   preconditions: ['action, 'name, 'amount, 'when],
                        // slot names
   score: nil     // for internal use
};
```

Registering and Unregistering the Task Template

To register your task template, call the `RegTaskTemplate()` function from your application's `InstallScript` method. When the task template is registered successfully, the value returned by `RegTaskTemplate` is a reference to the task frame in the NewtonScript heap. Assign the value

returned by this function to a local variable—you'll need it later to unregister the task template. If `RegTaskTemplate` returns a `nil` value, the template was not registered successfully.

To unregister your task template, call the `UnRegTaskTemplate` function, passing as its argument the result that was returned by `RegTaskTemplate` when you first registered the template. It is recommended that you call the `UnRegTaskTemplate` function from your application's `RemoveScript` method.

Displaying Online Help From the Assistant

Application help takes the form of a help book created with Newton Book Maker. You need to take the following steps to open a help book from the Assistant.

- In your application's base view, define a `viewHelpTopic` slot . The value of this slot is a string that is the name of a topic in the help book to be opened.
- Define an action template for opening the appropriate help book. The global functions `ShowManual` and `OpenHelpTo` open the system supplied help book. The `OpenHelpBook` and `OpenHelpBookTo` functions open a help book that you supply. The `ShowManual` function is described in Chapter 20, "Utility Functions."All of these functions are described in version 1.1 of the *Newton Book Maker User's Guide* .
- Define a task template that holds the name of your action template as the value of its `primary_act` slot.
- Register and unregister the task template at the appropriate times.

For information on defining, registering and unregistering templates, see the preceding section, "Making Behavior Available From the Assistant" beginning on page 17-19.

For information on displaying online help from an information button that your application provides, see the description of the `protoInfoButton` system prototype in this book. For information on creating your own online help, see the *Newton Book Maker User's Guide.*

Assistant Reference

This section describes slots, frames, templates, functions and methods used by the Assistant.

Action Template

The action template defines to the Assistant a frame representing a single action such as “call,” “pay,” or “remind.” The completion of a complex task may require the use of several action templates, each defining a discrete task that is completed as part of the primary task. The action template also stores a list of words or phrases, called the lexicon, that the Assistant uses to match this template with words or phrases from user input.

The Assistant provides several predefined action templates. They are summarized in the section “System-Supplied Action Templates” beginning on page 17-27. You use the system-supplied `dyna_user_action` template to define new actions to the Assistant.

Your action template must provide the following required slots.

`value` Required. The Assistant uses this slot only when using a lexical dictionary to parse a special-format string such as a phone number. You can use this slot to hold a comment string that indicates the name of this template.

`isa` Required. The value of this slot identifies the object type of the frame created from this template. You must store in this slot a symbol identifying this template as being an action that you defined (as opposed to one defined by the system.) The symbol `'dyna_user_action` is acceptable, as would be the symbol for any template derived from a template having the value `'dyna_user_action` in its `isa` slot. For more

information, see “Defining Your Own Frame Types To The Assistant” beginning on page 17-21.

`lexicon` Required. This slot holds an array of one or more words or phrases to match with this template. The Assist slip displays the first value in this array as an item in the Please... pop-up menu when this template is matched as the primary action.

System-Supplied Action Templates

`dyna_user_action`
Generic action template having no lexicon. All of your action templates must descend from this template to enable the Assistant to resolve verb-matching conflicts.

`call_act` Action template for dialing the telephone. This template’s lexicon includes the strings `"call"`, `"phone"`, `"ring"` and `"dial"`.

`find_act` Action template for invoking the Find service. This template’s lexicon includes the strings `"find"`, `"locate"`, `"search for"`, and `"look for"`.

`fax_act` Action template for faxing the target data item. This template’s lexicon includes the string`"fax"`.

`print_act` Action template for printing the target data item. This template’s lexicon includes the string`"print"`.

`about_act` Action template for displaying the About box. This template’s lexicon includes the string`"about newton"`.

`time_act` Action template for retrieving time values from the World Clock application. This template’s lexicon includes the strings `"time"`, `"time in"`,`"the time in"`, `"what time is it"`, `"what time is it in"`, `"the time in"`, `"what time"`, `"what is the time"` and `"what is the time in"`.

`remind_act`	Action template for creating To Do items. This template's lexicon includes the strings `"remember"`, `"remind"`, `"remind me"`, `"to do"`, `"todo"`, `"don't forget to"`, and `"don't let me forget to"`.
`mail_act`	Action template for sending electronic mail. This template's lexicon includes the strings `"mail"`, `"send"` and `"email"`.
`schedule_act`	Action template for scheduling meetings and events in the Dates application. This template's lexicon includes the string `"schedule"`.
`meet_act`	Action template for scheduling meetings and events in the Dates application. This template is based on the `schedule_act` template. Its lexicon includes the strings `"meet"`, `"meet me"`, `"see"`, and `"talk to"`.
`meal_act`	Action template for scheduling meals in the Dates application. Because meals are considered meetings (events with a beginning and ending time), this template is based on the `schedule_act` template.

Meals

These system-supplied action templates are used to schedule meals inthe built-in Dates application. All of these templates provide a string (such as `"breakfast"`) that is used as the default title of the meeting. These templates also define a `usualTime` slot that provides a default value for the starting time of the meal. These templates are based on the `meal_act` template.

`breakfast_act`	Action template for scheduling breakfast in the Dates application. Its lexicon includes the string

	"breakfast". The default starting time for this meeting is 7:00 am.
brunch_act	Action template for scheduling brunch in the Dates application. Its lexicon includes the string "brunch". The default starting time for this meeting is 10:00 am.
lunch_act	Action template for scheduling lunch in the Dates application. Its lexicon includes the string "lunch". The default starting time for this meeting is 12:00 pm (noon).
dinner_act	Action template for scheduling dinner in the Dates application. Its lexicon includes the string "dinner". The default starting time for this meeting is 7:00 pm.

Special Events

The templates described here define special_event_act frames. These action frames define methods used to schedule events in the Dates application. With the exception of the holiday template, all of these action frames schedule events that recur annually on a specified date. The event that the holiday template schedules does not repeat because holidays do not necessarily fall on the same date each year. The special_event_act template is derived from the schedule_act template.

birthday	Action template for scheduling an annual repeating birthday event in the Dates application and adding this information to an appropriate Names soup entry if one exists. Its lexicon includes the strings "birthday", "bday" and "b-day".
anniversary	Action template for scheduling an annual repeating anniversary event in the Dates application and adding this information to an appropriate Names soup entry if

	one exists. Its lexicon includes the string `"anniversary"`.
`holiday`	Action template for scheduling a non-repeating holiday event in the Dates application. Its lexicon includes the string `"holiday"`

Developer-Supplied Action Templates

You must supply the action template specified by the value of your task template's `primary_act` slot.

You must also supply any additional action templates specified by the `signature` slot of the task template.

Target Template

The target template defines to the Assistant a frame representing the target of an action; targets are generally people, places or things. The target template also stores a list of words or phrases, called the lexicon, that the Assistant uses to match the template with words or phrases from user input.

The Assistant provides several predefined target templates. They are summarized in the section "System-Supplied Target Templates" beginning on page 17-31. You need to use the `dyna_user_obj` template to define new targets to the Assistant.

Your target template must provide the following required slots.

`value`	Required. Currently unused, but required for compatibility with future versions of the Assistant. You can put a comment string indicating the name of the template in this slot.
`isa`	Required. The value of this slot identifies the object type of the frame created from this template. You must store in this slot a symbol identifying this template as being a target that you defined (as opposed to one defined by the system.) The symbol `'dyna_user_obj` is

	acceptable, as would be the symbol for any template derived from a template having the value `'dyna_user_obj` in its `isa` slot. For more information, see “Defining Your Own Frame Types To The Assistant” beginning on page 17-21.
`lexicon`	Required unless your template is derived from a system-supplied template, in which case your template can use the system-supplied lexicon. This slot holds an array of one or more words or phrases to match with this template.

System-Supplied Target Templates

The Assistant provides the predefined target templates described here. You use the `dyna_user_obj` template to define new targets to the Assistant.

Places

The following system-supplied templates define `where_obj` templates. No lexicons are associated with these templates because the Assistant uses lexical dictionaries to match them. The `where_obj` template is derived from the `user_obj` template.

```
address, city, region, country, postal_code,
phone,parsed_phone, phone_tag, faxPhone, homePhone,
workPhone, carPhone, mobilePhone, beeper, places,
company, city, county, state, country, town and province
```

Note that in addition to the items you would expect to be treated as places (such as postal codes and the names of cities, states and provinces), the Assistant treats phone numbers as places.

Times

The templates described here define `when_obj` frames. The `when_obj` template is derived from the `parsed_number` template.

`time, date`

User Object Template

The system-supplied user object template provides the basis for the Assistant's conflict-resolution mechanism. This section describes system-supplied templates for persons, groups, titles and custom targets, all of which are based on the user object template.You must use the `dyna_user_obj` template to define new targets to the Assistant.

For more information, see "Resolving Matching Conflicts" beginning on page 17-15.

`dyna_user_obj`	Generic target template having no lexicon. All of your target templates must descend from this template to enable the Assistant to resolve conflicts. This template is derived from the system-supplied user object template.
`who_obj`	Abstract target template having no lexicon, descended from the system-supplied user object template. Do not base your templates on the `who_obj` template. Instead base your target templates on the dyna_user_obj template.
`what_obj`	Abstract target template having no lexicon, descended from the system-supplied user object template. Do not base your templates on the `what_obj` template. Instead base your target templates on the dyna_user_obj template.
`where_obj`	Abstract target template having no lexicon, descended from the system-supplied user object template. Do not base your templates on the `where_obj` template. Instead base your target templates on the `dyna_user_obj` template.

People

The system-supplied `person` template defines a `who_obj` frame. The `title`, `affiliate`, `custom` and `group` templates are based on the `person`

template. These templates have no lexicons associated with them because the Assistant uses lexical dictionaries to match them.

`person` Target template for frames representing an individual person. You can base your own templates representing individual persons on this template.

`title` Target template for frames representing the title of an individual person, such as “Manager”, “Owner”, and so on. You can base your own templates representing titles of individual persons on this template.

`affiliate` Target template for frames representing a person affiliated with an individual, such as their friend, co-worker, and so on. You can use this template to creae your own templates representing affiliated persons.

`group` Target template for frames representing groups of people, such as “writers”, “engineers” , and so on. You can base your own templates representing groups of people on this template.

`custom` Target template for frames representing customized categories of persons, such as those taller than a specified height. You can base your own customized categories of individual persons on this template.

Miscellaneous Templates

This section describes the salutationPrefix template, which is derived from the system-supplied `parser_obj` template.

`salutationPrefix`

Action template for creating `parser_obj` frames. This template’s lexicon includes the strings "dear", "to", "attention", "attn", "attn." and "hey".

Developer-Supplied Target Templates

You must supply any required target template not supplied by the system. Required target templates are specified by the task template's `signature` slot.

Task Template

The task template defines an application behavior to the Assistant. A behavior consists of an action, such as "call," "pay," or "remind," that is generally directed at a target, such as "Bob" or "Apple." An action and its target are defined by an action template and a target template, respectively.

All task templates must define the following required slots.

`isa` Required. This slot identifies the object type of the frame created from this template. Task templates must store only the value `'task_template` in this slot. You cannot use the symbol for another template derived from this one instead.

`primary_act` Required. This slot stores the name of the action template that defines an application behavior to the Assistant. The action template that this slot identifies may itself require the use of additional action templates and target templates.

`preconditions` Required. This slot stores an array of symbols specifying the names of slots that the Assistant creates to store action frames and target frames. The `preconditions` array must have the same number of elements as the `signature` array because the Assistant uses these two arrays in parallel. For more details, see the section "About the Signature and Preconditions Slots" beginning on page 17-11.

`signature` Required. This slot holds an array of frame types that may be stored in the slots specified by the `preconditions` array. The `signature` array must

	hold the frame type of at least one action frame and zero or more target frames. The `signature` array must have the same number of elements as the `preconditions` array because the Assistant uses these two arrays in parallel. For more details, see the section "About the Signature and Preconditions Slots" beginning on page 17-11.
`postparse`	Required. The method to be invoked after the Assistant parses the user input. Frequently, the task template's primary action is actually invoked by the `postparse` method—for example, if the user asks to "fax Bob" and Newton cannot do so until the Assistant has retrieved Bob's fax number, the primary action of sending the fax would correctly be invoked after the `parseUtter` function returns the task frame. Another common use for the `postparse` method is to display a task slip view that provides the user with an opportunity to confirm, modify or dismiss the primary action before it is executed.
`taskslip`	Optional. This slot holds a view template associated with the task template. Commonly this view is a task slip that displays information about the primary action for confirmation or editing by the user.
`score`	Used internally by the Assistant. Place the value `nil` in this slot.

Developer-Supplied Task Template

You must always supply a task template, which defines the application behavior made available through the Assistant.

Help Topic Slot

Your application's base view can supply a `viewHelpTopic` slot that the Assistant uses to open a help book to the appropriate topic.

```
viewHelpTopic // slot specifying your app's help topic
```

System-Supplied Assistant Functions

This section describes functions provided for you by the Assistant.

RegTaskTemplate

`RegTaskTemplate(`*theTemplate*`)`

Registers a specified task template with the Assistant.

theTemplate The template to register.

UnRegTaskTemplate

`UnRegTaskTemplate(`*theTemplate*`)`

Unregisters a specified task template with the Assistant.

theTemplate The template to unregister.

ParseUtter

`ParseUtter(`*inputString*`);`

This function takes the following actions and calls your `postparse` method.

- parses the input string passed as its argument
- matches words and phrases in the input string to templates currently registered with the Assistant
- creates action frames and target frames from the matched templates
- creates a task frame based on matching an action frame to a task template
- creates slots holding action frames and target frames in the task frame

- as necessary, creates the `raw`, `origphrase`, `phrases`, `noisewords`, `entries` and `value` slots in the task frame.

See the section "Programmer's Overview" beginning on page 17-7 for a detailed description of these tasks.

As necessary, this function creates the following slots in the task frame.

`raw` An array of phrase types and values used internally by the Assistant. Each element of the array is itself a simple array, which lends great flexibility to this data structure. The `raw` array may sometimes include as one of its elements the original unparsed user input string. For more information, see "About the Raw Slot" beginning on page 17-13.

`origphrase` Holds the original user input phrase as a simple array of strings. Each element of this array is a single word from the user input phrase, and the words appear in the array in the order in which they appeared in the user input phrase.

`phrases` A simple array of strings derived from the *inputString* string. Each element of this array is a string that matched a template currently registered with the Assistant. These elements may be phrases themselves; under certain conditions, for example, the full name of the fax recipient ("Bob Dobbs") may be stored as a single element in this array. For more information, see "About the Phrases Slot" beginning on page 17-14.

`noisewords` An array of strings derived from the *inputString* string.Each element of this array is a string that did not match any template currently registered with the Assistant. Because the parser breaks unmatched phrases on word delimiters such as spaces, tabs, and return

characters, the elements of this array are always single words.

`entries` — Aliases to soup entries that were matched. Your `postparse` method can use these aliases to retrieve matched soup entries instead of querying for them. Do not access this slot directly; instead, use the `GetMatchedEntries` function to retrieve these entries. For more information about entry aliases, see Chapter 11, “Data Storage and Retrieval.”

`value` — An optional slot that holds formatted strings such as phone numbers, currency values and dates. The Assistant typically uses the `value` slot to return the results of a parse conducted using a lexical dictionary. Your `postparse` method can use the `value` slot for this purpose as well. An example describing the use of the `value` slot appears in the section “Parsing for Special-Format Objects” beginning on page 17-14.

GetMatchedEntries

`GetMatchedEntries (`*which*`, `*entries*`);`

Returns an array of entry aliases to soup entries that were matched by the Asssistant.

which — Symbol specifying a subset of entries to return. Acceptable values are any of the `'person`, `'places` or `'allEntries` symbols.

entries — The `entries` slot from the result frame returned by the Assistant.

Typically you would call this function from your postParse method, passing to it the `entries` slot of the result frame as in the following code fragment.

```
local candidates := GetMatchedEntries('allEntries, self.entries);
```

Developer-Supplied Assistant Functions and Methods

This section describes functions, methods and templates that you must supply.

PostParse

taskFrame`:PostParse();`

Your `PostParse` method must do anything necessary to perform the action specified by the frame in the `primary_act` slot of *taskFrame*, such as handling error conditions, extracting further information from the result frame returned by the `ParseUtter` function or displaying a task slip to the user. The Assistant calls your `PostParse` function after matching all of the templates specified by the task template.

taskFrame	The frame created by the Assistant from the task template.

Summary

This section summarizes the functions, methods and data structures used by the Assistant.

Templates

```
aTaskTemplate := { // summary of task template
   value: string, // name of task
   isa: 'task_template, // you must use this value
   primary_act: myAct, // name of primary action template
   PostParse: myMethod // you must supply this func obj
```

```
   signature: [t1, t2 … tn],// required templates
   preconditions: ['s1, 's2, … 'sn], // required slots
   score: nil     // required for internal use
   // you can add any additional slots you require
…};

anActionTemplate := { // summary of action template
   value: string , // name of action
   isa: 'dyna_user_action, // you must use this value
   lexicon: [str1, str2 … strN] // words/phrases to match
   // you can add any additional slots you require
};

aTargetTemplate := { // summary of target template
   value: "Who Object",// name of target
   isa: 'dyna_user_obj, // must use this value
   lexicon: ["to", "Bob"]// words to match
   // you can add any additional slots you require
   };
```

System-Supplied Action Templates

```
dyna_user_action, call_act, find_act, fax_act,
print_act, about_act, time_act, remind_act, mail_act,
schedule_act, meet_act, meal_act,
```

Meals

```
breakfast_act, brunch_act, lunch_act, dinner_act
```

Special Events

birthday, anniversary, holiday,

System-Supplied Target Templates

The system supplies templates for representing places, times, people, user objects and salutations.

Places

```
address, city, region, country, postal_code,
phone,parsed_phone, phone_tag, faxPhone, homePhone,
workPhone, carPhone, mobilePhone, beeper, places,
company, city, county, state, country, town and province
```

Times

```
time, date
```

User Objects

```
dyna_user_obj, who_obj, what_obj, where_obj,
```

People

```
person, title, affiliate, group, custom
```

Miscellaneous

```
salutationPrefix // “to”, “Dear”, “Hey”, and so on
```

Task Frame

After parsing a string presented as input to the Assistant, the `ParseUtter` function creates a task frame similar to the one shown here. Note that thes slots are created only as needed—a given task frame may not necessarily hold every slot shown here.

```
{raw: [[{…}, …],…],// frame types & values, internal use
origphrase:[str1, str2, … strN], // array of single words
phrases:[str1, str2, … strN], // matched words or phrases
```

```
noisewords:[str1, str2, … strN], // unmatched words/phrases
entries:[alias1, alias2, … aliasN], // matched soup entries
value: string , // internal use, parsed numbers as strings
score: aValue, // internal use only
… }
```

Assistant Functions and Methods

```
ParseUtter(inputString) // matches input to templates
tmpltRef := RegTaskTemplate(myTemplt)//register w/ Assistant
UnRegTaskTemplate(tmpltRef)// unregister task template
GetMatchedEntries(which, entries)// returns array of aliases
```

Developer-Supplied Functions and Methods

```
taskFrame:PostParse() // called after input is parsed
```

Application Base View Slots

```
viewHelpTopic // topic in help book
```

Built-In Applications and System Data

This chapter describes the interfaces to the built-in applications. It also describes the soup formats used by the built-in applications and the soup format of the System soup.

You should read this chapter if your application needs to interact with any of the built-in applications, or with the System soup variables. This chapter describes how to:

- add cards, layouts, and data to the Names application and access its soup
- interact with the Dates application and access its soups
- interact with the Time Zone application and access its soups
- interact with the To Do List application and access its soup
- interact with the Notes application and access its soup
- add auxiliary buttons
- correctly access user data stored in the system soup

IMPORTANT
Soup formats are subject to change. Applications that rely on soup formats risk incompatibility with future versions of Newton software. ▲

To avoid future compatibility problems with soup format changes, you should use the global functions `GetSysEntryData` (page 18-99) and `SetSysEntryData` (page 18-99) to get or change entries in any of the built-in soups. They allow you to get and set the values of slots in a soup entry.

Familiarity with Chapter 1, “Overview,” Chapter 5, “Stationery,” and Chapter 11, “Data Storage and Retrieval,” of this manual is particularly valuable in reading this chapter.

About the Built-In Applications and System Data

About the Names Application

The Names (Cardfile) application is built with the NewtApp framework and so uses data definitions (commonly called “data defs”) and data views. This architecture allows extensibility, the addition of new data views, card types, and card layout styles without altering the Names application itself. For more information on data defs and views, see chapters Chapter 4, “NewtApp Applications,” and “Chapter 3, “Views,”.

The Names application interface allows you to programmatically add complete cards, and add information to a cardfile entry. In addition several root view methods let you access information in a Names soup entry.

The name of the application is Names. That is what the user sees, but in programming the term Cardfile is used interchangeably and appears in the code.

Names Compatibility

All the Names methods variables, and constants are new in this version. The `'group`, `'owner`, and `'worksite` types are new. `'person` and `'company` include many new slots.

The Names application converts incoming soup entries that contain 1.x data into soup entries that conform to the 2.0 data format. Conversion includes adding particular slots to soup entries and, in some cases, removing slots. The data conversion occurs when the Newton receives 1.x entries by beaming, synchronizing using Newton Connection, restoring from a backup, or updating a 1.x card to 2.0.

A user can beam a card created in version 2.0 to a Newton running earlier version and it will read and edit the card, and *vice versa.*

In addition to changes in the programmatic interface, the 2.0 version has extensive changes in the user interface to accommodate the increased number of card types, layout styles, and data.

About the Dates Application

The Dates (Calendar) application interface consists of many methods (pages 18-49 through 18-74) of the calendar object. Always use these methods to access or modify Dates application data. Even though the soup format is documented for your information, do not directly modify Dates soup entries, except for any special slots that you might want to add and maintain yourself.

Two basic kinds of events can be scheduled with the Dates application. They are:

- A **meeting** is an entry for a specific time during the day. People can be invited and the meeting can be scheduled for a particular location. Note that meetings use two kinds of icons, one for regular meetings and a special icon for weekly meetings (meetings that repeat at the same time each week).
- An **event**, is an entry for a day, but not for a particular time during that day. Examples include a birthday, an anniversary, or a vacation. Events are

entered into the blank space at the top of the Dates application while in the Day view. Events use three kinds of icons: one for single-day events, one for multi-day events, and one for annual events (such as birthdays).

Meetings and events can repeat. That is, they can reoccur on one or more days in addition to the original meeting or event date.
The name of the application is Dates. That is what the user sees, but in programming the term calendar is used interchangeably and appears in the code.

Dates Compatibility

This section describes Dates features that are new, changed, and obsolete in system software version 2.0, compared to version 1.x.

- All the Dates methods, variables, and constants described in this chapter are new.
- The following slots are new in Dates soup meeting frames: `class`, `instanceNotesData`, `mtgIconType`, `mtgInvitees`, `mtgLocation`, and `version`. In addition, the `mtgText` slot now contains a rich string instead of a plain string.
- Dates soup notes frames are new (see the section "Notes Frames" beginning on page 18-79).
- The Dates application converts incoming soup entries that contain 1.x data into soup entries that conform to the 2.0 data format. Conversion includes adding particular slots to soup entries, and in some cases, removing slots. The data conversion occurs when the Newton receives 1.x entries by beaming, synchronizing using the Newton Connection Kit, or restoring from a backup.
- In addition to changes in the programmatic interface, the 2.0 version has extensive changes in the user interface.

In system software version 1.x, the Dates application allowed notes (text and graphics) to be written without an associated meeting marker. In system software 2.0, such notes, previously called annotations, cannot be written. Annotations imported via Newton Connection Kit from a 1.x system are still

visible and editable, however. As in version 1.x, these objects are stored in the `ROM_CalendarSoupName` soup.

In 1.x versions, all instances of a repeating meeting share the same set of notes. A repeating meeting is one that recurs at a periodic time interval. In 2.0, notes of a repeating meeting are local to each occurrence of the meeting. When a 1.x repeating meeting is converted to 2.0 format, all the notes are typically added to the meeting instance whose meeting slip the user opens first.

About the To Do List Application

The To Do List application is integrated with the Dates application. The user accesses the To Do List through a button in the Dates application. The user can choose to view Dates information, To Do List information, or a combination of the two (Day's Agenda).

The To Do List application programming interface (API) allows you to create To Do List items programmatically, check them off, obtain them according to date and other criteria and remove them by date and other crteria (See "To Do List Reference" on page 18-80).

To Do List Compatibility

Version 2.0 reads and converts all 1.x To Do List soups. It does not reproduce styling of text, because 2.0 doesn't support styling. It does not allow shapes and sketches in a task, so shapes and sketches are thrown away.

Because the internal structure of the 2.0 To Do List soup is completely different from that of the 1.x version, when you transmit a 2.0 soup to a 1.x system it creates a 1.x entry.

About Fax Soup Entries

If you want to use a received fax in your application, you can find it in the In/Out Box soup or set up a process to rout it to your application by means of the `PutAwayScript` message (See the section "PutAwayScript" of

Chapter 2, “Routing Interface,”in the *Newton Programmer’s Guide: Communications*,) or by means of the `AutoPutAway` message (see the section “Automatically Putting Away Items” in the same chapter).

The `PutAwayScript` message results from a user action. An application can register to handle putting away fax data by using the `RegAppClasses` function (see the section “RegAppClasses” in Chapter 2, Routing Interface in*Newton Programmer’s Guide: Communications*).

The `AutoPutAway` message requires no user action. The In Box checks for an `AutoPutAway` method in the base view of the application whose `appSymbol` slot matches that in the item. If the `AutoPutAway` method exists, then the In Box sends the `AutoPutAway` message to the application, passing the incoming item as a parameter.

In either case, the `body` slot from the In / Out Box entry is passed to the application. All the fax data that an application needs is embedded within the `body` slot (see “Using Fax Soup Entries” beginning on page 18-28).

You may also want to use the system prototypes that relate to viewing and manipulating images with your fax data. They are `protoImageView`, `protoThumbnail`, and `protoThumnailFloater` and are documented in the section “Graphics and Drawing Protos” beginning on page 12-35.

About the Time Zone Application

The Time Zone application lets the user access information about locations, which may come from the system, the user, or from your application. The user can browse the cities of the world for the time, and other travel information such as network access phone numbers. The user can define a **Home City**, an **Away City**, and other locations of interest (**emporia**); the information the user has specified is available to your application. When the user specifies that the Newton is in a new location, local information, such as net work phone numbers, is available to your application. See the chapter “Localizing Newton Applications” on page 19-1 for more information on localization.

Time Zone Comparability

The Time Zone application runs only on version 2.0, and is fully comparable with older versions back to 1.3.

About the Notes Application

Notes (or Notepad) is a simple NewtApp that allows the user to create new stationery. It contains a New Button, which the user employs to create new Stationery, to scroll up and down, to rout notes, to file them, and to scan an overview.

Developers can create and add their own stationery. Using and creating stationery is covered in Chapter 5 “Stationery” of this guide.

The name of the application is Notes or Notepad. That is what the user sees, but in programming the term paperroll is also used and appears in the code.

Notepad Compatibility

There are some anomalies in converting ink from 2.0 to earlier versions of the system. 2.0 ink text is converted to straight ink when viewed in 1.x versions. Paragraphs with mixed regular and ink text are converted so that the regular text loses any styles, for example in 1.x versions, it becomes 18-point user font. Any paragraph that contains ink text is reflowed in 1.x versions so that line layouts (breaks) are not the same as in the original. This means that the paragraph may grow taller. Ink works converted from 2.0 to 1.x straight ink appear in the size originally drawn, not in the 2.0 scaled size.

About Auxiliary Buttons

A set of functions allows you to add buttons to the status bars of the Notes and Names applications. Third-party applications can use this mechanism to allow themselves to be extended.

About System Data

The applications programing interface allows you to save preference or state information and to register with the system to receive notification of changes to the `userConfiguration` frame. Applications can also register a view to be added to the user Preferences roll.

Using the Interfaces

This section describes how to use the interfaces to each of the built-in applications and the system data.

Using the Names Application

This section describes

- adding a new data item to cards
- adding a new card layout style
- adding a new type of card to the Names application
- adding cards to the Names application
- using the Names soup

Adding a New Data Item

The Add button on the Names status bar allows the user to add new items of information to a card, such as a phone number for a person. There is a Custom choice on the Add picker (pop-up menu), through which the user can create special data items that contain a simple text field. However, you can programmatically add new choices to the picker by creating and registering new data views with the Names application.

The Add button creates its picker by looking at the registered stationary for the Names application. Any stationary will show up in the Add picker if it's

`type` slot was set to `'editor` when the card type of the current card matches the data class that the new view definition was registered for.

Names view definition contain a special slot called `infoFrame` that contains additional information required by the Names application. The `infoFrame` slot is a frame itself whose slots are described on page 18-35.

Note that in the Names application, and in other places where the areacode is needed (such as `userConfiguration`, `Setup`, and `Connection`), the phone number is not stored as a plain string but as a formatted string, so that the different parts of the phone number can be parsed without guesswork. For methods to find and display phone numbers (see “MakePhone” on page 20-122 through “ParsePhone” on page 20-124).

Here is an example of an `infoFrame` for a Names data view defining a view called “AddCar Info” that has two fields, Make and Model:

```
{checkPaths: '[carMake, carModel],
checkPrefix: '[true, [pathExpr: carInfo]],
stringData: nil,
format: "^?0Make: ^0\n||^?1Model: ^1||" }
```

It is not necessary to set `'stringData` to `nil`, but it is necessary to set it to non-nil values in cases such as phone numbers.
When chosen from the Add picker the first time, this view would first fill in the `carMake` and `carModel` slots in the soup entry with the user’s entries. If chosen again, this view would then create a `carInfo` array containing one frame for each additional data set. These frames would look like this:

`{carMake:` *make*`, carModel:` *model*`}`

When a view from the Add picker is instantiated, the system sets the view slot `selectedPath` to the path expression where data should be entered (or to `nil` if the data should be entered directly into the soup entry). For example, when chosen from the Add picker the first time, the view in this example would have its `selectedPath` slot set to `nil`, meaning the information should be put directly into the soup entry. When chosen from the picker the third time, the `selectedPath` array is set to `[pathExpr:`

`carInfo, 1]`, to indicate that the new car information should go into the second frame in the `carInfo` array.

Adding a New Card Layout Style

When the Card layout is selected in the Show picker, the Names application looks at the `cardType` slot of the current card to determine which kind of business card layout to use for that card. You can create new card layout data views and register them with the Names application. Card data views must have the `type` slot set to `'bizcard`, and must contain a `bizCardNum` slot and a `bizCardIcon` slot.

The `bizCardNum` slot contains an integer that corresponds to the value stored in the `cardType` slot of the card entries. The values 0-3 correspond to the business card layouts that are built into the system.

The `bizCardIcon` slot contains an icon representing the new layout, to be shown in the Card Style view, where the user can change the type of card layout to be used for a particular card.

Adding a New Type of Card

The New button on the Names status bar creates its picker by looking at the registered data defs for the Names application. All data defs whose `superSymbol` slot is set to `'Names` will show up in the New picker. When the user picks a choice, the `MakeNewEntry` routine defined for that data definition will be called.

Built-in choices on the New picker include Person, Company, and Group. You can create a new type of card for the Names application by supplying a new data definition.

Names data defs contain two special slots, `overviewIcon` and `viewsToDisplay`, which are described on page 18-35.

Adding a Card to the Names Application

The `AddCard method` (page 18-37) of the Names application adds a new card programmatically. To send the `AddCard` message to the `cardfile`

view, you'll need a reference to that view. You can reference the `cardfile` with this code:

```
GetRoot().cardfile;
```

So to send the `AddCard` message, you would use code like this:

```
GetRoot().cardfile:AddCard(arg1, arg2);
```

Using the Names Soup

To avoid future compatibility problems with soup format changes, you should use the global functions `GetSysEntryData` (page 18-99) and `SetSysEntryData` (page 18-99) to get or change entries in any of the built-in soups. They allow you to get and set the values of slots in a soup entry. If you don't use these functions to get and set entry slots in the built-in soups, your application may break under future versions of system software.

protoPersonaPopup

This proto is used for a picker that lets the user maintain and switch between different owner cards, or "**personae**." Here's an example:

The diamond appears only if there is more than one owner card; otherwise you see just a name without a diamond. Tapping on the name produces a picker showing the names of all owner cards registered in the cardfile of this Newton.

protoEmporiumPopup

This proto is used for a picker that lets the user maintain and switch between different information relevant to various places she may be in. Here's an example:

When the user chooses a different city, information like time zone, area code, and so on is changed to reflect the different location. Choosing "Other City" allows the user to choose a different city anywhere in the world.

Note that you can get the information on things like `owner` and `worksite` from the `UserConfiguration` frame. For example, you if need the dialing prefix, you should use.

```
GetUserConfig('dialingPrefix).
```

Using the Dates Application

This section describes

- adding meetings or events
- finding, moving, and deleting meetings or events
- getting and setting information for meetings or events
- creating and using a new meeting type
- controlling the Dates display
- miscellaneous operations
- using the Dates soups

The interface to the Dates application is provided by several methods of the calendar view. To send one of these messages to the `calendar` view, you'll need a reference to that view. You can reference the `calendar` view with this code:

```
GetRoot().calendar;
```

So, to send one of these messages, you would use code like this:

`GetRoot().calendar:AddEvent(`*mtgText*`,` *mtgStartDate*`,` *repeatPeriod*`,` *repeatInfo*`);`

Adding Meetings or Events

To add a new meeting programmatically, you use the `AddAppointment` method (described on page 18-49).

Here are some examples:

To schedule a one-hour lunch appointment:

```
GetRoot().calendar:AddAppointment("lunch with Ellen",
StringToDate("6/30/95 12:00pm"), 60, nil, nil)
```

To schedule a twice-a-week meeting on Mondays and Wednesdays:

```
GetRoot().calendar:AddAppointment("design meeting",
StringToDate("11/6/95 10:30am"), 60, 'weekly [1, 3])
```

To schedule a once-a-year party engagement at New Years Eve:

```
GetRoot().calendar:AddAppointmentNew Years' Eve",
StringToDate("12/31/96 9:00pm"), 120 'yearlyByWeek, nil)
```

To add a new event programmatically, you use the `AddEvent` method (described on page 18-51).

Here is are some examples:

To schedule an event:

```
GetRoot().calendar:AddEvent("buy flowers",
StringToDate("6/30/95"), nil, nil)
```

To schedule a birthday that repeats yearly:

```
GetRoot().calendar:AddEvent("George's birthday",
StringToDate("2/22/95"), 'yearly, nil)
```

To schedule Mother's Day:

```
GetRoot().calendar:AddEvent("Mother's Day",
StringToDate("5/14/95"), 'yearlyByWeek, nil)
```

Finding, Moving, and Deleting Meetings or Events

If you query the calendar soups to get the meeting frame, remember that the soup entries in the Repeat Meetings and Repeat notes soups represent series of repeating meetings and events, not individual occurrences. Calling `DeleteAppointment` (page 18-53) or `DeleteEvent` (page 18-55) on a repeating soup entry deletes the entire series. To delete a particular instance of a repeating meeting or repeating event, you need to call `DeleteAppointment` or `DeleteEvent`, respectively, with a meeting frame that represents a particular instance of the repeating series or with the title and date of the instance. `FindAppointment` (page 18-56) returns such a meeting frame, which has the following format where *datetime* is the time and date of the particular occurrence and *soupentry* is the appropriate soup entry.

```
{  viewStationery: 'RepeatingMeeting
   class: 'meeting,
   mtgStartDate: datetime,
   repeatTemplate: soupentry
}
```

Note that, for an instance of a repeating meeting, the `viewStationery` slot contains the value `'RepeatingMeeting` and for an instance of a repeating event, the slot contains `'CribNote`.

`FindExactlyOneAppointment` works just like `FindAppointment` except that if it finds more than one appointment it throws an exception.

You can find open time in the calendar for scheduling a meeting programmatically with the `FindNextMeeting` method (described on page 18-59).

Here is an example:

```
GetRoot().calendar:FindNextMeeting(StringToDate("11/1/95 2pm"))
```

To move an appointment programmatically you employ the `MoveAppointment` method (described on page 18-62).

Here is an example:

```
getroot().calendar:MoveAppointment("Job Review", stringtodate
("9/1/96 9:30pm"), stringtodate("9/8/96 4:00pm"), 60)
```

To move a meeting and give it a different duration:

```
GetRoot().calendar:MoveAppointment("lunch with Ellen",
StringToDate("6/30/95 12:00pm"), StringToDate
("6/30/95 11:30am"), 90)
```

`MoveOnlyOneAppointment` works just like `MoveAppointment` except that if it finds a non-exception instance that fits the criteria, it moves only that instance. `MoveAppointment` would move all the non-exceptions in that case.

To move a single occurrence of a repeating meeting:

```
GetRoot().calendar:MoveOnlyOneAppointment
("design meeting", StringToDate("11/13/95 10:30am"),
StringToDate("11/14/95 2:00pm"), nil)
```

Here are three similar examples of finding meetings:

To find all meetings in the month of June with the word "lunch" in the title or the notes:

```
GetRoot().calendar:FindAppointment(nil, '["lunch"],
[StringToDate("6/1/95 12:00am"), StringToDate("6/30/95
11:59pm")], nil, nil)
```

To find the unique meeting with the title "lunch with Ellen" in the month of June:

```
GetRoot().calendar:FindExactlyOneAppointment
("lunch with Ellen", nil, [StringToDate
("6/1/95 12:00am"), StringToDate("6/30/95 11:59pm")], nil)
```

To find the next meeting after 2 P. M. on November 1 1996:

```
GetRoot().calendar:FindNextMeeting(StringToDate ("11/1/96 2pm"))
```

Here are some examples of deleting meetings or events of different types:

To delete a meeting:

```
GetRoot().calendar:DeleteAppointment("lunch with Ellen",
StringToDate("6/30/95 11:30am"), true)
```

To delete an event:

```
GetRoot().calendar:DeleteEvent("buy flowers",
StringToDate("6/30/95"), true)
```

To delete one occurrence of a repeating meeting:

```
GetRoot().calendar:DeleteAppointment("design meeting",
StringToDate("11/20/95 10:30am"), true)
```

To delete a repeating meeting:

```
GetRoot().calendar:DeleteRepeatingEntry
("design meeting", StringToDate("11/6/95 10:30am"), true)
```

Getting and Setting Information for Meetings or Events

You can programmatically obtain the location, invitees, icon type and notes to a meeting by means of methods documented on pages 18-60 through 18-61.

You can programmatically set the location, invitees, icon type, and alarm for a meeting by means of methods documented beginning on page 18-68.

You can programmatically set the stop date of a meeting or event with the `SetRepeatingEntryStopDate` method (see page 18-73).

Here is an example, which prompts the user to stop watching TV after five days:

```
cal:=getroot().Caledar;
:startDate := StringToDate("7/14/96 7:00pm");
aMeeting:= cal: AddEvent("Watch TV", startDate daily,nil)
.
.
.
cal: SetRepeatingEntryStopDate("Watch TV", startDat
startDate+5*1440) // One day is 1440 minutes
```

To invite two people to the meeting, where one is in the Names soup and one isn't:

```
GetRoot().calendar:SetMeetingInvitees("design meeting",
StringToDate("11/6/95 10:30am") [{name: {first: "Jane", last:
"Smythe"}}, GetUnionSoup("Names"):query({type: 'words, words:
'["Bob", "Anderson"]}):entry()])
```

To get the list of invitees to a meeting:

```
GetRoot().calendar:GetMeetingInvitees("design meeting",
StringToDate("11/6/95 10:30am"))
```

To set the location of a meeting to a place not in the Names soup:

```
GetRoot().calendar:SetMeetingLocation("design meeting",
StringToDate("11/6/95 10:30am"), "Blue Room")
```

To get the location of a meeting:

```
GetRoot().calendar:GetMeetingLocation("design meeting",
StringToDate("11/6/95 10:30am"))
```

To set the notes of a meeting:

```
GetRoot().calendar:SetMeetingNotes("design meeting",
StringToDate("11/6/95 10:30am"), [{viewStationery:
'para, text:"Brainstorm on the killer app", viewBounds:
{left:10, top:7, right:110,bottom:20}}])
```

To get the notes of a meeting:

```
GetRoot().calendar:GetMeetingNotes("design meeting",
StringToDate("11/6/95 10:30am"))
```

To set a 15-minute advance notice alarm for a meeting:

```
GetRoot().calendar:SetEntryAlarm("design meeting",
StringToDate("11/6/95 10:30am"), 15)
```

To set the stop date of a repeating meeting:

```
GetRoot().calendar:SetRepeatingEntryStopDate("desig
meeting", StringToDate("11/6/95 10:30am")
StringToDate("12/20/95 10:30am"))
```

To set the icon type of an event so that it displays the `AnnualEvent` icon:

```
GetRoot().calendar:SetMeetingIconType("George's
birthday", StringToDate("2/22/95"), 'AnnualEvent)
```

To get the icon type of an event:

```
GetRoot().calendar:GetMeetingIconType("George's
birthday", StringToDate("2/22/95"))
```

Creating and Using a New Meeting Type

The following example code registers a new meeting type for a monthly meeting.

```
GetRoot().calendar:RegMeetingType(EnsureInternal
                        ('|MonthlyMeeting:MyApp|),
      {  item: "Monthly Meeting",
         icon: myIcon,
         smallicon: mySmallIcon,
         NewMeeting:
            func(date, parentBox)
               begin
               local cal := getroot().calendar;
               local appt := cal:AddAppointment("",
date, 60, 'monthly, nil);
               appt.meetingType :=
'|MonthlyMeeting:MyApp|;
               EntryChange(appt);
               appt;// the calendar will open the
                        default meeting slip
               end,
      });
```

To register a new meeting type for a monthly event with a private slip:

```
GetRoot().calendar:RegMeetingType(EnsureInternal
('|Monthly Event:MyApp|),
   {  item: "Monthly Event",
      icon: myIcon,
      smallicon: mySmallIcon,
      NewMeeting:
         func(date, parentBox)
            begin
            local cal := getroot().calendar;
            local appt := cal:AddEvent("", date,
            'monthly, nil);
            appt.meetingType := '|MonthlyEvent:MyApp|;
            EntryChange(appt);
```

```
            :OpenMeeting(appt, date, cal:GlobalBox());
            nil;// tells calendar not to open default
                  meeting slip
            end,
   OpenMeeting:
      func(meeting, date, parentBox)
         begin
         local cal := getroot().calendar;
         local slip :=
            BuildContext(
               {  _proto:  MP_protoFloater
                  viewFlags:
                  vClickable+vFloating+vApplication+
                  vClipping,
                  viewJustify:0,
                  viewBounds:SetBounds(parentBox.left+40,
                  parentBox.top+40, parentBox.right-40,
                  parentBox.bottom-100),
            declareSelf:'base,
         stepChildren:[
            {  viewStationery: 'para,
            text:    "test slip",
            viewBounds:RelBounds(10, 10, 50, 20),
            },
            {  _proto:MP_protoCancelButton,
            buttonClickScript: func()
// closes the view and tells calendar the view was closed
            AddDeferredSend(cal, 'RememberedClose,
            [base]),
         }
      ],
   });
```

```
cal:RememberedOpen(slip);// opens view and tells
                            calendar a view was opened
nil;// tells calendar not to open default meeting slip
end,
});
```

The methods `UnRegMeetingType` (page 18-74), `NewMeeting` (page 18-66), `OpenMeeting` (page 18-67), `RememberedOpen`(page 18-68), and `RememberedClose` (page 18-68) allow you to manipulate the result of users choosing a meeting type in the New picker.

You can display a picker of repeating meeting types by means of `protoRepeatPicker` (page 18-47) and `protoRepeatView` (page 18-47).

Controlling the Dates Display

There are two system variables you can set to control specific features of the Dates display.

The Dates variable `firstDayOfWeek` specifies what the first day of the week should be, for display purposes. It holds an integer value from 0 to 6, where 0 means Sunday, 1 means Monday, and so on. The default value is 0, which means by default all months show Sunday as the first day of the week.

Once this value has been set, all new views of the class `clMonthView`, and views that display meeting frequency, reflect the new value; but existing views must be closed and reopened to reflect the new value. This variable is a slot in the global `userConfiguration` frame (the Dates application checks there first), or in the locale bundle frame (Dates checks there second).

The Dates variable `useWeekNumber` controls display of a week number in the upper-left corner of the Dates view. If this slot is non-`nil`, the Dates application displays the week number there. The first week of the year is number 1 and the last week is number 52. This variable is a slot in the locale bundle frame.

To get and set the value of slots in the `userConfiguration` frame, use the functions `GetUserConfig` and `SetUserConfig`. To return the current locale bundle frame, use the global function `GetLocale`.

Miscellaneous Operations

You can programmatically display meetings in the Dates application, To Do items, or the agenda for a specified date by means of the `DisplayDate` method (discussed on page 18-55).

You can programmatically obtain an array of the currently selected and displayed dates by means of the `GetSelectedDates` method, page 18-62).

You can open the meeting slip for a specific meeting or event programmatically by means of the `OpenMeetingSlip` method (page 18-64).

You can add or delete an item in the Info picker in the base view of the calendar by means of the `RegInfoItem` (see page 18-65) and `UnRegInfoItem` (see page 18-73) methods.

Here is an example of how to add a command to the Info button for displaying the next day's To Do List:

```
GetRoot().calendar:RegInfoItem('|NextDayToDo:MyApp|,
         {  item: "Next Day's To Do",
            doAction: func()GetRoot().calendar:
            DisplayDate (GetRoot() .calendar:
            GetSelectedDates()[0]+(24*60), 'ToDoList)
})
```

Using the Dates Soup

The Dates application stores meeting information, notes, and To Do List items in the following soups:

Soup (name string)	Description
`ROM_CalendarSoupName` (“Calendar”)	Entries are meeting frames for non-repeating meetings.
`ROM_RepeatMeetingName` (“Repeat Meetings”)	Entries are meeting frames for repeating meetings and notes frames for notes associated with specific instances

of a repeating meeting. A single meeting frame entry describes all of the instances of a repeating meeting.

`ROM_CalendarNotesName` (“Calendar Notes”)
Entries are meeting frames for non-repeating events.

`ROM_RepeatNotesName` (“Repeat Notes”)
Entries are meeting frames for repeating events and notes frames for notes associated with specific instances of a repeating event.

The slot structure of meeting frames is described in the section “Meeting Frames” beginning on page 18-74.

The slot structure of notes frames is described in the section “Notes Frames” beginning on page 18-79.

Although the format of the various Dates soups is documented in this section, you should not directly change entries. It is best to use the methods supplied by the Dates application to get and set entries.

The To Do List uses a soup called “To Do List”. In this soup, each day has a single entry, with a `topics` slot containing an array element for each task.

A possible conflict arises when one entry exists in internal store, and one on a storage card. In that case, the To Do List merges them into one entry for display.

Using the To Do List Application

This section provides infirmation about the use of To Do List soups and methods.

Using the To Do List Soup

In this soup, each day has a single entry that includes a `topics` slot, which contains an array element for each task. All repeating tasks are saved in a single entry, with a `date` slot of 0.

If one day is represented by an entry on each of several stores of the soup—for example, if there was one entry in internal store and one on a storage

card—the To Do List merges the entries for display purposes, but not in internal store.

All repeating tasks are saved in a single entry, with a date slot of 0. (Again, there may actually be several entries, one per store, but the To Do List merges them for display.)

The To Do List uses views based on `ListView` (described in“Structured List Views” on page 8-18). The To Do LIst is limited to one level (by the ListView `maxLevel` slot).

WARNING

If you create multiple-level lists, the results are not guaranteed. ▲

You can use the `ListView` method `SetDone` (see page 8-80) to programmatically check off a To Do List item.

To avoid future compatibility problems with soup format changes, you should use the global functions `GetSysEntryData` (page 18-99) and `SetSysEntryData` (page 18-99) to get or change entries in any of the built-in soups. They allow you to get and set the values of slots in a soup entry. If you don’t use these functions to get and set entry slots in the built-in soups, your application may break under future versions of system software.

See“To Do List Soup Format” beginning on page 18-80) for detailed information about this soup.

Using To Do List Methods

The applications programming interface provides methods to create To Do items programmatically, retrieve them according to date, return To Do item shapes and task shapes, obtain the next task after a date, and to remove To Do items according to date (see “To Do List Reference” on page 18-80).

Here is an example of creating a repeating To Do task:

```
t1 := time();
t2 := t1+60*1440; // 1 day = 1440 minutes
rr := {mtgStartDate: t1,
```

```
        mtgStopDate: t2,
        repeatType: kWeekInMonth,
        mtgInfo: kThirdWeek + kTuesday};
GetRoot().calendar:GetToDo():CreateToDoItem (time(),
"test", nil, rr);
```

Using the Time Zone Application

The application interface provides utility functions to discover locations close to a specified location, to add a city to the cities soup, and to set the current location. Methods let you obtain the name of location soups, the union soup for a location and control cloning from ROM to RAM. You can also obtain a city, country, or Daylight Savings Time that matches a string (see page 18-85).

Adding a City to the City Soup

The code:

`GetRoot().worldclock:NewCity(newCity, workSite, makeHome)` adds the city specified by `newCity` to the cities soup. Always set `workSite` to `nil`. If `makeHome` is `true`, the current location is set to the location. The entry should have the following slots:

Slot description

name	Required. A string containing the name of the location.
longitude	Required. The longitude of the location. The formula for generating this value appears below.
latitude	Required. The latitude of the location.
gmt	Required. The offset in minutes from Greenwich Meant Time).
country	Required. A symbol representing the country in which the city is located.
areaCode	Optional. The area code of the location.
region	Optional. The region of the location. For cities in the US this should be the state. For cities in Canada this should be the province.
airport	Optional. The airport designation for the city.
ew2400	Optional. The local access number for connecting to eWorld at 2400 baud.
ew9600	Optional. The local access number for connecting to eWorld at 9600 baud.

Here is an example:

```
{name:"Portland",
longitude:354036540,
latitude:67929083,
gmt:  -28800,
country:'USA,
areaCode:503,
region:"OR",
airport:"PDX",
ew2400:2410496,
ew9600:2950337}
```

To calculate the latitude or longitude of a location use the following functions:

```
CalcLngLat := func(dgrs, min, sec, westOrSouth) begin
   local loc;
   loc := dgrs / 180 + min / (180 * 60) + secs / (180 *
                              60 * 60);
   loc := rinttol(loc * 0x10000000);
   if westOrSouth then
      loc := 0x20000000 - loc;
   loc;
end;
```

See “LatitudeToString” on page 20-24 and “LongitudeToString” on page 20-24, for documentation of methods that return a string representation of an encoded latitude or longitude value.

Setting the Current Locations

The code:

`GetRoot().worldclock:SetLocation(location)`
sets the current location in the `userConfiguration` frame to the value passed in the *location* parameter. This function also handles setting the `currentCountry` and `currentAreaCode`.

Using Fax Soup Entries

When an entry is submitted to the In / Out Box from a transport such as fax receive, or from an application, the fax is stored in the `body` slot of the In / Out Box soup entry. The In / Out Box stores the original application soup entry in a frame called `'body` within the In / Out Box soup entry, where a user can view it. Applications may be passed an In / Out Box soup entry as part of the putting-away process. For more information about how to handle items coming from the In / Out Box see the section “PutAwayScript” of Chapter 2, “Routing Interface,”*Newton Programmer’s Guide: Communications,*.

The `body` frame has the following structure.

Slot descriptions

`sender` A string for the sender's phone number.

`pages` An array of page data. Each page is a frame with the following slots:

- `image` Required. This slot contains a NewtonScript shape object. For details see "protoImageView" on page 12-35.
- `annotations` Required. The default is `nil`. When not `nil`, this slot contains an array of shapes or paragraphs as in the Notes application.
- `thumbnail` Optional. A shape placed here by the FaxViewer as a cached object. If you don't put one in, it will be added. (See the section "protoThumbnail" on page 12-45.)

`info` A frame with the following slots.

- `dataRate` The transmission rate of the fax in baud, default 2400.
- `endTime` The point at which the fax completed in the number of minutes passed since midnight, January 1, 1904, The default is 0.
- `pageCount` The number of pages in the fax. The default is 0.
- `pixelWidth` The default width for all pages in pixels, default 0.
- `pixelHeight` The default height for all pages in pixels, default 0.
- `resolution` The image in dots per inch (dpi). Like a pensize, the value of the resolution slot may be an array or a single value. If this value is an array, the two elements of the array specify the x and y values in dpi. If this slot holds a single value, the pixels

	are square and have the same value for x and y.
`startTime`	The point at which the fax started in the number of minutes passed since midnigh, January 1, 1904. The default is 0.

Warning
The `info` slot contains internal information and is subject to change. ◆

Using the Notes Application

This section describes the use of the soup format of the built-in Notes application and of system data. The application interface method specific to Notes, `MakeTextNote`, which you use to pragmatically write a note to the Notes application, is described on page 18-89

Using Notes Methods

The Notes application issues a warning when a note exceeds 8K (8192 bytes) and suggests the user should work with a different note or create a new one. Nothing, however, actually prevents the user from continuing to add data to a large note. Developers can change the limit trigger by setting the slot `maxEntrySizeBytes` (See page 18-90) in the paperroll base, for example

```
GetRoot().paperroll.maxEntrySizeBytes := 10000.
```

You can turn the limitation off completely by setting `maxEntrySizeBytes` to `nil`. You can also change the behavior at the limit by overriding the function `AppMaxSizeExceeded` (see page 18-89) also in the `paperroll` base. This function is passed one parameter, the entry, which is the soup entry for the offending note. Currently, this function opens the Notify slip, but developers can use it in whatever way they need. Since the size check is performed in `newtApp` code, this limit can be implemented for any descendent of `newtApp` that contains `maxEntrySizeBytes` and a `AppMaxSizeExceeded` slots.

Using the Notes Soup

The Notes soup holds individual entries for the different kinds of built-in stationery as follows:

- Note: The lined paper used for the Notes.
- Outline: Paper with automated outlining.
- Checklist: Outline paper with a box to check off completed items.

The reference section (“Notes Soup Format” beginning on page 18-90) provides detailed information on the data structures that support these stationeries.

You can see for yourself what the entries look like by creating the stationery you want with the data you want. In the listener, create a cursor and then look at its entries as with the following code.

```
entry := (cursor := Query(GetUnionSoup(“Notes”), {type:
                ‘index})):Entry()
```

You can look at specific data by referencing the slots for example:

```
entry.tally.griddata[0].
```

To look at additional entries, use `cursor:Next()`.

To avoid future compatibility problems with soup format changes, you should use the global functions `GetSysEntryData` (page 18-99) and `SetSysEntryData` (page 18-99) to make change entries in any of the built-in soups. They allow you to get and set the values of slots in a soup entry. If you don’t use these functions to get and set entry slots in the built-in soups, your application m ay break under future versions of system software.

Using Auxiliary Buttons

You can add buttons to the status bars or in other locations in the Notes and Names applications. Third-party applications may also use this mechanism to allow themselves to be extended.

`RegAuxButton` and `UnRegAuxButton` (See page 18-94) are the functions that add and remove a button from the auxiliary button registry; they are called by **button providers**. Button providers can ignore the descriptions of the other functions, which are called by **button hosts**.

Here is an example of registering a button:

```
RegAuxButton('|smileButton:PIEDTS|,
      {destApp: 'cardfile,
      _proto: protoTextButton,
      text: "Smile!",
      viewBounds: RelBounds(0,0,40,10),
      buttonClickScript: func() print("Cheese!")
      });
```

Here is an example of unregistering a button

```
UnRegAuxButton('|smileButton:PIEDTS|)
```

If your application is the backdrop application, this array contains both buttons specific to your application and any designated for the backdrop application. You do not need to write any special code to detect whether your application is the backdrop application.

You should be careful what assumptions you make about the environment where the button will appear. The buttons may not be on a `protoStatusBar` (see page 7-62) or have a `'base` slot available by inheritance, and the implementation details of the built-in applications may well change in the future. Remember to check your assumptions and catch exceptions.

Any application that adds buttons to another application should provide a preference that allows the user to enable or disable the display of the source application buttons in the destination application. The user may want to suppress buttons because the buttons from several source applications may be too many to fit in a single destination application. The user should be able to choose which should appear, and the preference should normally default to *not* registering the button.

Note that a button you register might not appear at all if too many buttons are already registered in the destination application, so you must ensure alternative access to your application through conventional means, for example the Extras drawer.

Also note that packages that install buttons into other applications may cause the system to display the card reinsertion warning if they are on a card that has been ejected. It is wise to advise or require users to install packages on the internal store if they are going to register buttons in other applications.

`GetAuxButtons`, along with `AddAuxButton` and `RemoveAuxButton` (see page 18-95) are for use by button hosts that are adding the buttons to their own status bars (or wherever is appropriate). You should call `GetAuxButtons` in your base view's `viewSetupChildrenScript` and merge the resulting array with the buttons that always appear on your status bar. You should probably also override the `viewBounds` or `viewJustify` slots of the buttons, to place them correctly.

Using System Data

The system stores user preferences and other system information in a soup. You should use the compile-time constant `ROM_SystemSoupName` to get a reference to the name of the system soup (“System”). The system soup initially contains only a single entry containing all user preference and system related settings in the slots in that entry.Each application that needs to save user preference or state information should create a single entry in the system soup to store its data.Each entry in the system soup must contain a slot named `tag` whose value is a string uniquely identifying the application to which the entry belongs. The system user preferences entry contains a `tag` slot value of `userConfiguration`. Other than the `tag` slot, each entry can contain any other slots needed to store application-specific data.

The system soup is indexed on the `tag` slot, allowing quick access to your application’s entry.

To avoid future compatibility problems with soup format changes, you should use the global functions `GetUserConfig` (page 18-101) and `SetUserConfig` (page 18-102) to get and set the values of slots in the system `userConfiguration` frame. If you don't use these functions to get and set slots in the `userConfiguration` frame, your application may break under future versions of system software.

Your application can also register with the system to receive notification of changes to the `userConfiguration` frame. To do this, use the functions `RegUserConfigChange` (page 18-102) and `UnRegUserConfigChange` (page 18-103).

The slots in the `userConfiguration` frame are documented in the section "User Configuration Frame" beginning on page 18-96.

Adding Preferences and Formulas Roll Items

The User Configuration interface allows applications to add items to the Preferences and Formulas rolls in the Extras Drawer.

▲ **WARNING**

These functions are not available on all platforms; they may be provided by the `Platforms` file in Newton Toolkit. ▲

This section describes a new system prototype and accompanying functions that allow you to add and remove your own views in the Preferences roll in the Extras Drawer. The `RegPrefs` function registers with the system a view template based on the `protoPrefsRollItem` system prototype.The `UnRegPrefs` function reverses the effects of the `RegPrefs` function (See pages 18-99 through 18-101).

protoPrefsRollItem

The `protoPrefsRollItem` system prototype describes a preferences view normally added to the system-supplied Preferences roll. Note that items added to the Preferences roll must specify system-wide preferences rather than application-specific ones. Application-specific preferences must be set in the application itself.

Names Reference

This section describes the routines provided to interface with the Names application, and the data structures used when interacting with the Names application.

Names Data Structures

Names Data Definition Frame

Names data definition frames contain the following special slots, in addition to the usual data definition slots.

Slot descriptions

`overviewIcon` A tiny version of the icon in the `icon` slot, to be used when displaying this kind of card in an overview. You should keep the icon smaller than 11 by 11 pixels; any taller looks awkward in the overview. `Nil` values are not allowed.

`viewsToDisplay` An array containing the names of data views registered for this data definition. These views are shown in the All Info view.

Names Info Frame.

Names data views contain a special slot called `infoFrame` that contains additional information required by the Names application.

Slot descriptions

`checkPaths` An array of paths to data collected by this layout.

`checkPrefix` An array of two path expressions. The first is the path for the first data set, or `true` if the first data set is part

	of the soup entry. The second is the path for subsequent data sets (or `nil` if no multiple data sets are allowed). If the second path is `nil`, the data view will appear in the Add picker until the user chooses it and adds data to it. At that point, no more data of that type can be added, so the item is removed from the Add picker.
`stringData`	Set to `true` if the data sets consist of single strings. A `nil` value means frames are created for each item rather than strings.
`format`	A string to be passed to `ParamStr` along with the data set. The string returned by `ParamStr` is used for display in the All Info view. Alternatively, you can use the method `FormatFunc` to return a string.

The following method is also included.

FormatFunc

`FormatFunc(`*path*`, `*Array*`)`

As an alternative to the `format` slot, you can define this method. It returns a string to be displayed in the All Info view to represent the data from this view.

pathArray	An array of the data in the data set, corresponding to the paths in the `checkPaths` slot.

Names Soup Format

This section describes the format of entries in the Names soup. Each entry consists of a frame with the following slots. All are optional.

Slot descriptions

`bcEmailNetwork` This slot has two parameters. *entry* is the entry in the Cardfile. *which* is the type (`'sprint` or `'concert`)

`bcEmailAddress` This slot has two parameters. *entry* is the entry in the Cardfile. *which* is the a string for the email address.

`bcCreditCards` This slot has two parameters. *entry* is the entry in the Cardfile. *which* is the an array of strings for the owner, card name, and number of the credit card.

`bcPhoneNumber` This slot has two parameters. *entry* is the entry in the Cardfile. *which* is the a string for the phone number.

Names Methods

The most important method for use with the Names placating is `Addcard`, which enables you to add a new card programmatically.

AddCard

cardfile`:AddCard(`*dataDefType*`,` *entryFrame*`)`

Creates a new card in the Names application.

dataDefType A symbol giving the data definition type for the new Names entry. The following symbols are allowed: `'person`, `'company`, `'group`, `'owner`, or `'worksite`.

entryFrame A frame containing the slots that you want the new Names entry to have. These slots are described below. If you specify an empty frame, a new card of the correct data definition type is created.

This function returns the newly-created entry, or `nil` if none was created. (The entry is not created if *dataDefType* is invalid).

The *entryFrame* can contain the following slots for an entry with a *dataDefType* of `'person`:

`cardType` This contains one of the following constants that describe the card layout: `kSquiggle(0),`

	`kPlain(1), kSeparate(2) and kCross(3)` (See "Names card layouts" on page 18-104).
`names`	`Affiliated names` *(company contacts, family members, etc.)*. The frame can contain strings for these slots:
	`honorific` A sting or rich string for an honorific title, e. g. Ms. or Dr.
	`first` A string or rich string for first name.
	`last` A string for a last name.
	`title` A string or rich string that identifies this names relationship to the main name.
`name`	This frame contains strings for the following slots:
	`honorific` A sting or rich string for an honorific title, e. g. Ms. or Dr.
	`first` A string or rich string for first name.
	`last` A string for a last name.
	`title` A string or rich string for job title.
`company`	A string or rich string for company name.
`companies`	This is an array of frames of type `{company: string, title: string}`. You can add more than one company to a person card.
`address`	A string or rich string for the first line of an address.
`address2`	A string or rich sting for the second line of an address.
`city`	A string for a city.
`region`	A string for region (a State in the US).
`postal_code`	A string for postal code (zip code in US).
`country`	A string naming the country. This slot should be set by calling `SetCountryClass(`*string*`)`.
`phones`	An array that contains strings of phone numbers.
`email`	A string for e-mail address.

`emailAddrs` A frame of additional e-mail addresses with the following slots:

- `email` A string.
- `emailpassword` For cards of type `'owner` only, a string.

`emailPassword` Only in owner cards. A string for the owner's e-mail password.

`pagers` An array of pager information frames. Each frame can have the following slots:

- `pagerNum` string for pager number
- `pagerPIN` string for pager PIN.

`bday` Contains an immediate value; the date the user entered for the birthday in the number of minutes passed since midnight. January 1 1904.

`bdayEvent` An alias to a calender meeting.

`anniversary` Contains an immediate value; the date the user entered for the anniversary, in the number of minutes passed since midnight. January 1 1904.

`anniversaryEvent` An alias to a calender meeting.

`notes` Specifies the notes to be set. This parameter must be an array of paragraph and polygon frames, or it can be a string, which replaces all existing notes. You can specify `nil` to clear the notes.

`sorton` A string (not a rich string because of sorting).

The *entryFrame* can contain the following slots for an entry with a *dataDefType* of `'company`:

`name` This frame contains strings for the following slots:

- `honorific` A sting or rich string for an honorific title, e. g. Ms. or Dr.
- `first` A string or rich string for first name.
- `last` A string for a last name.
- `title` A string or rich string for a job title.

`cardType`	This contains one of the following constants that describe the card layout: `kSquiggle(0), kPlain(1), kSeparate(2) and kCross(3)`see “Names card layouts” on page 18-104.
`company`	A string for company name.
`names`	Names of contacts person at the company. The frame can contain strings for these slots: `honorific` A sting or rich string for an honorific title, e. g. Ms. or Dr. `first` A string or rich string for first name. `last` A string for a last name. `affiliation` A string or rich string for company affiliation. `title` A string or rich string for a job title.
`address`	A string or rich string for the first line of the address.
`address2`	A string or rich string for the second line of the address.
`city`	A string containing the name of a city.
`region`	A string for region (State in US, Province in Canada).
`postal_code`	A string for postal code (zip code in US).
`country`	A string naming the country. This slot should be set by calling `SetCountryClass(`*string*`)`
`phones`	A array containing phone numbers as strings.
`email`	A string for an e-mail address.
`emailAddrs`	A frame of additional e-mail addresses with the following slots: `email` A string. `emailpassword` For cards of type `'owner` only, a string.
`notes`	Specifies the notes to be set. This parameter must be an array of paragraph and polygon frames, or it can be a string, which replaces all existing notes. You can specify `nil` to clear the notes.
`sorton`	A string (not a rich string because of sorting).

The *entryFrame* can contain the following slots for an entry with a *dataDefType* of 'group:

`cardType` This contains one of the following constants that describe the card layout: `kSquiggle(0), kPlain(1), kSeparate(2) and kCross(3)` see "Names card layouts" on page 18-104.

`group` A string for the group's name.

`members` An array containing namerefs representing the members of the group.

`notes` Specifies the notes to be set. This parameter must be an array of paragraph and polygon frames, or it can be a string, which replaces all existing notes. You can specify `nil` to clear the notes.

`sorton` A string (not a rich string because of sorting).

The *entryFrame* can contain the same slots for an entry with a *dataDefType* of 'owner as it does for an entry with the *dataDefType* of 'person. In addition, an 'owner entry contains a slot named owner, which contains a frame with these slots:

`spareTime` A string.

`bankAccounts` an array of frames corresponding to bank accounts, which contain the following slots:

`bankAcctNum`
A string which contains the account number.

`bankContactNum`
A phone number string for the bank account contact.

`creditCards` an array of frames corresponding to credit card accounts, which contain the following slots:

`creditCard`Name
A string for the credit card user's name.

`creditCardNum`
A string that for the account number.

`creditCardExpDate`
A string that contains the expiration date.

`creditCard`ContactNum
A phone number string for the credit card account contact.

`signature` The signature the User enters north signature sip. It is text and ink.

The *entryFrame* can contain the following slots for an entry with a *dataDefType* of `'worksite`:

`cardType` This contains one of the following constants that describe the card layout: `kSquiggle(0), kPlain(1), kSeparate(2) and kCross(3)` see "Names card layouts" on page 18-104.

`place` A string for the place name.

`dialingPrefix` A string for dialing prefix needed to get an outside line from this worksite.

`areaCode` A string for the area code of this worksite.

`printer` A string representing the printer the user has chosen from among network printers.

`mailAccess` An array of frames of form

```
{mailPhone: string, mailNetwork: 'concert baud: 1200}.
```

`connectionPhone` A phone number string to access e-mail at this site

`connectionNetwork`
A string that combines in order the network zone and machine name. The parameters are `currentSelection`, which returns the selected entity and `currentZone`, which returns the currently selected AppleTalk zone.

`cityAlias`	An alias to the closest city entry in the `worldData` soup.
`countrySymbol`	A symbol representing the country.
`country`	A string representing the country
`notes`	Specifies the notes to be set. This parameter must be an array of paragraph and polygon frames, or it can be a string, which replaces all existing notes. You can specify `nil` to clear the notes.
`sorton`	A string (not a rich string because of sorting).

AddCardData

`AddCardData(`*`entry`*`, `*`layoutSym`*`, `*`newData`*`)`

This method is the handiest way to add information to a Cardfile entry. It allows you to specify an entry and add a string or frame to it.

entry	The entry in the Cardfile soup where you want to add data.
layoutSym	A symbol giving the data definition type for the new Cardfile entry. The following symbols are allowed:

'person, 'company, 'group, 'owner, or 'worksite. The following values are recognized for these symbols:

'person	name, company, address, phone, email, pager, and persona. 'pager gives you a frame with slots 'pagerNum (a string) and 'pagerPIN (a string). 'personal gives you 'anniversary (a date) and 'bday (a date).
'owner	name, company, address, phone, email, pager, personal, signature, creditCard, and bankAccount.
'company	name, address, phone, and email.
'worksite	connection and emailInfo. 'EmailInfo gives you the frames that the bcemailNetwork method returns: 'mailPhone, 'baud, and 'mailNetwork.

newData The data you wish to add. It may be a string or a frame, depending on the type of data.

nil values are not supported.

newData must match the expected format of the data. For example, to add a new affiliate to a person card, use:

```
Getroot().cardfile:AddCardData with (personcard, 'name,
                        {first: "Test", last: "This"}).
```

To add a fax phone number to a company:

```
Getroot().cardfile:AddCardData(<companycard>, 'phone,
            SetClass(MakePhone({areacode: "617", phone:
            "555-1212"}), 'faxPhone))
```

ReplaceInkData

Cardfile:ReplaceInkData(*entry*, *layoutSym*, *oldString*, *checkPath*, *newString*)

This method replaces a specified ink string in a cardfile entry with a recognized string.

entry	The entry in the Cardfile soup where you want to add data.
layoutSym	A symbol giving the data definition type for the new Cardfile entry. Symbols are allowed as described on page 18-43.
oldString	The ink string to replace.
checkPath	The additional path where the data is found (See “Names Info Frame.” on page 18-35.)
newString	The recognized string to replace *oldString*.

AddLayout

Cardfile:AddLayout(*layout*)

Because the Show picker is not dynamic, you must call AddLayout for Names to use the layout. This method allows you to specify a view definition to appear in the Show picker.

layout	This parameter specifies the view definition that you want to appear in the Show picker. Passing a bad layout will cause and error.

RemoveLayout

Cardfile:RemoveLayout(*layoutSym*)

This method takes a symbol and removes the cardfile layout corresponding to that symbol.

layoutSym	The symbol of the cardfile or be removed.

bcEmailAddress

Cardfile:`bcEmailAddress(`*entry*`, `*which*`)`

This method takes a soup entry and e-mail type, and returns an array of frames with e-mail information. The `'email` slot in these frames is a string representing the email address. If the entry is an owner card, the frame may also contain an `'emailPassword` slot, which holds the password string. The method returns `nil` if no frames are found.

entry	The entry in the card file.
which	One of the types of e-mail from `GetGlobals().emailSystems.emailClasses`. If the this parameter is an array of symbols instead of a single symbol, the matches for both symbols are returned

bcEmailNetwork

Cardfile:`bcEmailNetwork(`*entry*`, `*type*`)`

This method takes a soup entry and the type of e-mail network, and returns an array of frames with the following slots:

Slot description	
mailNetwork	A network type symbol.
mailPhone	A string for a phone number.
baud	An integer, the baud rate.

The method returns `nil` if no frames are found.

entry	The entry in the card file.
type	The type of network (`'sprint` or `'concert`). If the this parameter is an array of symbols instead of a single symbol, the matches for all the symbols are returned.

bcCustomFields

Cardfile:bcCustomFields(*inEntry, inWhich*)

This method returns an array containing frames with custom field information. The slot 'label contains the label for the custom field; the slot 'value contains the value stored in the slot of that custom field. The method returns nil if no frames are found.

inEntry	The entry in the card file.
inWhich	The name of the custom slot. nil returns all custom fields. If the this parameter is an array of symbols instead of a single symbol, the matches for all the symbols are returned

Dates Reference

The Dates application uses the protos protoRepeatPicker and protoRepeatView. as well as many methods. See page 18-107 for a list of Dates error codes.

Dates Protos

The Dates application is based on two protos: protoRepeatPicker to display a menu of repeating types and protoRepeatViewto create a draggable view.

protoRepeatPicker

Based on protoLabelPicker (See page 6-32), protoRepeatPicker displays a picker of repeating meeting types. This proto assumes the existence of a declared child of an ancestor of the protoRepeatPicker proto named RepeatingView that derives from protoRepeatView. Picking Other from the choice of repeating meeting types opens the RepeatingView.

Slot descriptions

`selectedMeeting` Required. This slot must be of the template or inherited by the template. The slot's value is typically a meeting frame (see page 18-74.) `protoRepeatPicker` uses the `'mtgStartDate` slot of the meeting, and uses the `'repeatType`, `'mtgInfo`, or `'repeatTemplate` slots if they are present.

`originalMtgDate` Required. If the `selectedMeeting` value is a repeating Meeting soup entry, then this slot must contain the date of an instance of that repeating meeting; otherwise, the slot is ignored. The slot should be in the template or inherited by the template.

`viewBounds` The bounds of the system prototype.

`newMtgDate` Required. The value of this slot must be the value of the `mtgStartDate` slot of the `selectedMeeting` slot. The slot should be in the template or inherited by the template.

`protoRepeatPicker` provides its own `viewSetupFormScript` and `labelActionScript`. If you override these methods, be sure to call *inherited*`:viewSetupFormScript()` and *inherited*`:labelActionScript()`. In addition, `protoRepeatPicker` provides default `textSetup` and `pickerSetup` methods.

protoRepeatView

This is a `protoFloatNGo` system prototype that is displayed in its parent to create a draggable view. Its default justification is centered horizontally and flush with the top of the parent. See page 7-58 for a discussion of `protoFloatNGo` and its parents.The view is 204 pixels wide and 190 pixels high, so the parent should be at least that wide and high. A view that contains a `protoRepeatView` child should also have a `protoRepeatPicker`.

Slot descriptions

`GetRepeatSpec`	A method that takes no parameters and returns a `repeatTemplate` containing the repeat information (See "GetRepeatSpec" on page 18-62.)
`viewFlags`	Defaults to `vClickable+vFloating`.
`viewFormat`	Defaults to `vfFillWhite+vfFrameDragger+vfPen(7) +vfInset(1)+vfRound(5)`.
`viewJustify`	Defaults to `vjParentCenterH`.
`viewBounds`	Defaults to `RelBounds(0, 0, 204, 190)`.

See page 7-58 for documentation of `protoFloatNGo`.

Dates Methods

This section includes a description of the methods supplied by the Dates application. These methods belong to the calendar view.

AddAppointment

calendar`:AddAppointment(`*mtgText*`,` *mtgStartDate*`,` *mtgDuration*`,` *repeatPeriod*`,` *repeatInfo*`)`

Creates a meeting and adds it to the appropriate Dates soup, and updates the calendar display, if necessary.

mtgText	A string that is the meeting text.
mtgStartDate	An integer specifying the start date and time of the meeting, in the number of minutes passed since midnight, January 1, 1904. If the meeting repeats, this is the start date and time of its first occurrence.
mtgDuration	A positive integer specifying the duration of the meeting in minutes.

repeatPeriod Used to indicate a repeating meeting. Specify one of the following symbols, or `nil`:

`nil`	Meeting is not repeating.
`'daily`	Meeting repeats every day.
`'weekly`	Meeting repeats weekly on the same day.
`'biweekly`	Meeting repeats biweekly on the same day.
`'monthly`	Meeting repeats monthly on the same day.
`'monthlyByWeek`	Meeting repeats monthly by the week in the month.
`'yearly`	Meeting repeats yearly on the same day.
`'yearlyByWeek`	Meeting repeats yearly by the week in the month.

repeatInfo Used only if *repeatPeriod* is `'weekly`, `'monthlyByWeek`, or `'yearlyByWeek` to specify when the meeting repeats. If not used, this parameter must be set to `nil`.

If *repeatPeriod* is `'weekly`, then this parameter must be an array of one or more numbers between 0 and 6 (where 0 = Sunday, 1 = Monday, etc.) These numbers specify on which days of each week the meeting repeats. You can also specify `nil`, which means the meeting repeats on the same day in which it was originally scheduled.

If `repeatPeriod` is `'monthlyByWeek`, then this parameter must be an array of one or more numbers between 1 and 5 (1 is the first week of each month, 2 is the second week of each month, and so on, up to 5, which is the last week of each month). These numbers specify which weeks each month the meeting repeats. You can also specify `nil`, which means the meeting repeats on the same week in which it was originally scheduled.

If *repeatPeriod* is `'yearlyByWeek`, then usually you can specify `nil`, since the week in the month is predetermined by the date you pick for the first instance of the meeting. However, if the day falls during the fourth or fifth week of a month, it is not always possible to determine in exactly which week subsequent instances of the meeting will occur. In this case, you should specify an array containing the single number 4 or 5, to indicate the week. For example, November 1994 had only four Thursdays, so Thursday, November 24th 1994 could be interpreted as the fourth Thursday or as the last Thursday.

This method returns the soup entry added for the new meeting.

AddEvent

calendar`:AddEvent(`*mtgText, mtgStartDate, repeatPeriod, repeatInfo*`)`

Creates an event and adds it to the appropriate Dates soup, and updates the calendar display, if necessary.

mtgText A string that is the event text.

mtgStartDate An integer specifying the start date of the event, in the number of minutes passed since midnight, January 1, 1904. If the event repeats, this is the start date of its first occurrence. Note that events don't have a specific time during the day, so by convention, they must always be created at midnight. The Dates application expects this. Don't create events at other times.

`AddEvent` automatically sets *mtgStartDate* to midnight at the beginning of the day.

repeatPeriod Used to indicate a repeating event. Specify one of the following symbols, or `nil`:

`nil`	Event is not repeating.
`'daily`	Event repeats every day.
`'weekly`	Meeting repeats weekly on the same day.
`'biweekly`	Meeting repeats biweekly on the same day.
`'monthly`	Meeting repeats monthly on the same day.
`'monthlyByWeek`	Event repeats monthly by the week in the month.
`'yearly`	Meeting repeats yearly on the same day.
`'yearlyByWeek`	Meeting repeats yearly by the week in the month.

repeatInfo Used only if `repeatPeriod` is `'weekly`, `'monthlyByWeek`, or `'yearlyByWeek` to specify when the event repeats. If not used, this parameter must be set to `nil`.

If *repeatPeriod* is `'weekly`, then this parameter must be an array of one or more numbers between 0 and 6 (where 0 = Sunday, 1 = Monday, etc.) These numbers specify which days each week the event repeats. You can also specify `nil`, which means the event repeats on the same day in which it was originally scheduled.

If *repeatPeriod* is `'monthlyByWeek`, then this parameter must be an array of one or more numbers between 1 and 5 (1 is the first week of each month, 2 is the second week of each month, and so on, up to 5, which is the last week of each month). These numbers specify which weeks each month or year the event repeats. You can also specify `nil`, which means the event repeats on the same week in which it was originally scheduled.

If *repeatPeriod* is `'yearlyByWeek`, then usually you can specify `nil`, since the week in the month is predetermined by the date you pick for the first instance of the event. However, if the day falls during the fourth or fifth week of a month, it is not always possible to determine in exactly which week subsequent instances of the event will occur. In this case, you should specify an array containing the single number 4 or 5, to indicate the week.

This method returns the soup entry added for the new event.

Here is an example:

```
getroot().calendar:AddEvent("Mother's Day",
stringtodate("5/14/95 12:00am"), 'yearlyByWeek, nil)
```

DeleteAppointment

calendar`:DeleteAppointment(`*mtgTextOrFrame, mtgStartDate, deleteOneOnly*`)`

Finds the meeting(s) at the given date and time, with the given meeting text, and deletes them all. If an instance of a repeating meeting is found, only that single instance is deleted. If a meeting frame is passed as an parameter, the method ignores the other parameters and deletes that meeting frame.

This method also updates the Dates display, if necessary.

mtgTextOrFrame — A string that is either the meeting text of the meeting you want to delete or a meeting frame, which is typically the frame returned by *calendar*`:FindAppointment`.

mtgStartDate — An integer specifying the start date and time of the meeting, in the number of minutes passed since midnight, January 1, 1904. If *mtgTextOrFrame* is a meeting frame, the value of *mtgStartDate* is ignored.

deleteOneOnly — A Boolean value that specifies whether to delete just one or multiple meetings (if multiple meetings are found). If

you specify `nil`, and more than one meeting is found, all found meetings are deleted. If you specify `true`, and more than one meeting is found, then this method throws an exception. This method always returns `nil`.

DeleteRepeatingEntry

calendar`:DeleteRepeatingEntry(`*mtgTextOrFrame, mtgStartDate, deleteOneOnly*`)`

This method finds the repeating meeting(s) or event(s) at the given date and time, with the given meeting text and deletes them all. All instances of the repeating meeting / event are deleted, not just the instance at the given time and date. If a meeting frame is passed as an parameter, the method ignores the other parameters and deletes that meeting frame. This method also updates the Dates display, if necessary.

mtgTextOrFrame — A string that is either the meeting text of the meeting you want to delete or a meeting frame, which is typically the frame returned by *calendar*`:FindAppointment`.

mtgStartDate — An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method will find all events scheduled during the day of *mtgStartDate*.

deleteOneOnly — A Boolean value that specifies whether to delete just one or multiple meetings / events (if multiple meetings or events are found). If you specify `nil`, and more than one meeting or event is found, all found meetings / events are deleted. If you specify `true`, and more than one meeting or event is found, then this method throws an exception. This method always returns `nil`.

DeleteEvent

calendar:DeleteEvent(*mtgTextOrFrame, mtgStartDate, deleteOneOnly*)

Finds the event(s) on the given date and time, with the given text, and deletes them all. If an instance of a repeating event is found, only that single instance is deleted. If a meeting frame is passed as an parameter, the method ignores the other parameters and deletes that meeting frame. This method also updates the Dates display, if necessary.

mtgTextOrFrame — A string that is either the meeting text of the meeting you want to delete or a meeting frame, which is typically the frame returned by *calendar*:FindAppointment.

mtgStartDate — An integer specifying the start date and time of the event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method will find all events scheduled during the day of *mtgStartDate*.

deleteOneOnly — A Boolean value that specifies whether to delete just one or multiple events (if multiple events are found). If you specify nil, and more than one event is found, all found events are deleted. If you specify true, and more than one event is found, then this method throws an exception. This method always returns nil.

DisplayDate

calendar:DisplayDate(*date, format*)

Displays the Dates meetings, To Do List items, or agenda for the specified date. Executing this method is equivalent to the user tapping on that date in the month view. Note that this method is meant to be called only when the calendar is open.

date — An integer specifying a date, in the number of minutes passed since midnight, January 1, 1904.

format A symbol that specifies what is to be displayed, as follows:

`'Day`	The day view.
`'ToDoList`	The To Do list.
`'Agenda`	The day's agenda.
`nil`	The calendar continues showing the current view, after closing any overviews.

These views are equivalent to the similarly named views listed on the Show button picker in the Dates application.

This method always returns `nil`.

FindAppointment

calendar`:FindAppointment(`*mtgText, findWords, dateRange, type, maxNumberToFind*`)`

Finds and returns one or more meetings and / or events using specific words and / or a date range as search criteria.

This method returns an array of the resulting meetings and events. They are soup entries in the case of non-repeating meetings and events and instance frames in the case of repeating meetings and events, as discussed under "Finding, Moving, and Deleting Meetings or Events" on page 18-14. If no entries are found, `nil` is returned. The instance frames for repeating meetings and events do not have title slots because the title is stored in the `repeattemplate` of each instance to save space and to allow for shared titles in a repeating meeting.

mtgText A string that is the meeting text or event text of the item(s) you want to find. Specify `nil` to match all entries satisfying the other search criteria (*findWords*, *dateRange*, and *type*).

findWords An array of words or word beginnings (specified as strings) which are used as search criteria to find meetings or events. If the text of the meeting or event, or

of the notes, contains all of the words in this array, then it satisfies this criteria. The words in the array can be split between the meeting text and the meeting notes. The word search is not case sensitive, and it only searches word beginnings; it will not find a string that occurs inside a word. Specify `nil` to not use this search criteria.

dateRange A single time, an array of two times, or `nil`. A time is specified as the number of minutes passed since midnight, January 1, 1904.

If you specify a single start time, all meetings at that time and all events on the day containing that time will satisfy the search criteria. If you specify an array of two times, all meetings and events between the two times will satisfy the search criteria. Specify `nil` to not use this search criteria.

type Used to limit the found items to meetings, repeating meetings, events, or repeating events. Specify one of the following symbols, an array of these symbols (to include multiple types), or `nil`:

`nil`	This search criteria is not used.
`'Meeting`	Search for non-repeating meetings.
`'Event`	Search for non-repeating events.
`'RepeatingMeeting`	Search for repeating meetings.
`'RepeatingEvent`	Search for repeating events.

maxNumberToFind

An integer that specifies the maximum number of items to find. After this number of items is found, this method stops searching and returns the results. Specify `nil` to use the default of 50.

If you specify `nil` to get all meetings on a particular day, `FindAppointment` returns a frame, which describes the particular occurrence of the repeating meeting on that day. That frame has the dateand time of the particular occurrence plus a slot, `'repeatTemplate,` which references the soup entry.

For a non-repeating meeting, `FindAppointment` returns the soup entry itself.

FindExactlyOneAppointment

calendar`:FindExactlyOneAppointment(`*mtgText, findWords, dateRange, type*`)`

Finds and returns exactly one meeting or event using specific words and/or a date range as search criteria.

This method returns the resulting meetings and events. They are soup entries in the case of non-repeating meetings and events and instance frames in the case of repeating meetings and events, as discussed under “Finding, Moving, and Deleting Meetings or Events” on page 18-14. If no entry is found, or more than one is found, then this method throws an exception.

mtgText — A string that is the meeting text or event text of the item you want to find. Specify `nil` to match an entry satisfying the other search criteria (*findWords*, *dateRange*, and *type*).

findWords — An array of words or word beginnings (specified as strings) which are used as search criteria to find the meeting or event. If the meeting text, event text, or notes text contains all of the words in this array, then it satisfies these criteria. The words in the array can be split between the meeting text and the meeting notes. The word search is not case sensitive and it only searches words from their beginnings. It does not find a string that occurs inside a word. Specify `nil` to not use this search criteria.

dateRange A single time, an array of two times, or `nil`. A time is specified as the number of minutes passed since midnight, January 1, 1904.

If you specify a single start time, a meeting at that time or an event on the day containing that time will satisfy the search criteria. If you specify an array of two times, a meeting or event between the two times will satisfy the search criteria. Specify `nil` to not use this search criteria.

type Used to limit the found item to a meeting, repeating meeting, event, or repeating event. Specify one of the following symbols, an array of these symbols (to include multiple types), or `nil`:

`nil`	This search criteria is not used.
`'Meeting`	Search for a non-repeating meeting.
`'Event`	Search for a non-repeating event.
`'RepeatingMeeting`	Search for a repeating meeting.
`'RepeatingEvent`	Search for a repeating event.

FindNextMeeting

calendar`:FindNextMeeting(`*date)*

Finds the first meeting after the specified date and time. This method is useful for an application that wants to find open time in the calendar for scheduling a meeting.

date An integer specifying a date and time, in the number of minutes passed since midnight, January 1, 1904.

This method returns an array containing the meeting start date and its duration. If there is more than one meeting scheduled at that time, the duration of the longest one is returned. If there are no meetings scheduled after the specified date, `nil` is returned.

GetMeetingIconType

calendar:GetMeetingIconType(*mtgTextOFrame, mtgStartDate*)

Returns the type of icon used for a particular meeting or event. The following symbols can be returned: 'Event, 'Meeting, 'WeeklyMeeting, 'MultiDayEvent, or 'AnnualEvent.

mtgTextOrFrame A string that is either the meeting text of the meeting whose icon you want or a meeting frame, which is typically the frame returned by *calendar*:FindAppointment.

mtgStartDate An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method will find an event scheduled any time during the day of *mtgStartDate*.

If the meeting or event uses a custom meeting type defined by RegMeetingType, that type is not returned by GetMeetingIconType. GetMeetingIconType will return either 'Meeting or 'Event, depending on whether the custom meeting type is created using AddAppointment or AddEvent. You must look at the meetingType slot of the custom meeting or event to determine the unique custom meeting type defined by RegMeetingType.

GetMeetingInvitees

calendar:GetMeetingInvitees(*mtgText, mtgStartDate*)

Returns the list of invitees for a meeting, or returns nil if there are none.

mtgText A string that is the meeting text of the meeting for which you want to get the list of invitees.

mtgStartDate An integer specifying the start date and time of the meeting, in the number of minutes passed since midnight, January 1, 1904.

The list returned is an array of name reference frames; see the discussion of the proto protoListPicker on page 6-110 of Chapter 6, “Pickers, Pop-up Views, and Overviews.”)

GetMeetingLocation

calendar:GetMeetingLocation(*mtgText, mtgStartDate*)

Returns the location for a meeting, or returns `nil` if there is none.

mtgText	A string that is the meeting text of the meeting for which you want to get the location.
mtgStartDate	An integer specifying the start date and time of the meeting, in the number of minutes passed since midnight, January 1, 1904.

The meeting location is returned as a name reference frame as described under `GetMeetingInvitees`. For details see the discussion under “protoListPicker” beginning on page 6-110 of Chapter 6, “Pickers, Pop-up Views, and Overviews.”

GetMeetingNotes

calendar:GetMeetingNotes(*mtgText, mtgStartDate*)

Returns the notes for a meeting or event, or returns `nil` if there are none.

mtgText	A string that is the meeting text of the meeting or event for which you want to get the notes.
mtgStartDate	An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904.

The notes are returned as an array of paragraph and polygon views.

GetSelectedDates

calendar:GetSelectedDates()

Returns an array of the currently selected and displayed dates. This array always has at least one element. If the Dates application is closed, the method returns nil.
Dates are integers specifying a date and time, in the number of minutes passed since midnight, January 1, 1904. Note that the time for each date in the array is set to midnight at the beginning of the day.

GetRepeatSpec

GetRepeatSpec()

GetRepeatSpec is a function of protoRepeatView (See page 18-48.) It takes no parameters and returns a repeatTemplate containing the repeat information. In particular, the repeatTemplate returned has slots for 'mtgStartDate, 'repeatType, and 'mtgInfo.
If the selectedMeeting slot of protoRepeatPicker is a repeating meeting and if the repeat settings (in the protoRepeatPicker and protoRepeatView) have not changed, then GetRepeatSpec returns the repeatTemplate of selectedMeeting.
If selectedMeeting is a repeating meeting and the repeat settings have been changed to “Don't repeat”, GetRepeatSpec returns nil.

MoveAppointment

calendar:MoveAppointment(*mtgText*, *mtgStartDate*, *newStartDate*, *newDuration*)

Finds the unique meeting or event with the given text at the given date and time, and changes the start date and / or the meeting duration.

mtgText	A string that is the meeting text of the meeting or event which you want to move.
mtgStartDate	An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. Note that events don’t have a

specific time during the day, so this method will find an event scheduled any time during the day of *mtgStartDate*.

newStartDate An integer specifying the new start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. If `nil`, the start date and time remain unchanged.

newDuration A positive integer specifying the new duration of the meeting in minutes. If `nil`, the duration is unchanged. If the entry is an event, specify `nil`.

This method always returns `nil`.

If you specify a repeating meeting or event that is not an exception case, this method changes the start time and duration of all repeating instances of the meeting or event that are not exceptions. However, if *newStartDate* is not a day that matches the original repeating pattern, then this method changes *newStartDate* to the first day after the one specified that does match the repeating pattern.

For example, if the repeating meeting normally occurs on Tuesdays and Thursdays, and you specify a *newStartDate* that is on a Friday, this method will change the *newStartDate* to the following Tuesday.

If no meeting or event is found, or more than one is found, then this method throws an exception.

MoveOnlyOneAppointment

calendar`:MoveOnlyOneAppointment(`*mtgText*`,` *mtgStartDate*`,` *newStartDate*`,` *newDuration*`)`

Finds the unique meeting or event with the given text at the given date and time, and changes the start date and/or the meeting duration.

mtgText A string that is the meeting text of the meeting or event which you want to move.

mtgStartDate An integer specifying the start date and time of the meeting or event, in the number of minutes passed since

midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method will find an event scheduled any time during the day of *mtgStartDate*.

newStartDate An integer specifying the new start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. If `nil`, the start date and time remain unchanged.

newDuration A positive integer specifying the new duration of the meeting in minutes. If `nil`, the duration is unchanged. If the entry is an event, specify `nil`.

This method always returns `nil`.

If you specify a repeating meeting or event that is not an exception case, this method changes it to an exception case and applies the new start time and duration to the new exception.

If no meeting or event is found, or more than one is found, then this method throws an exception.**OpenMeetingSlip**

calendar`:OpenMeetingSlip(`*meetingEntry*`,` *date*`,` *openDefaultSlip*`)`

Opens the meeting slip for the specified meeting or event.

meetingEntry A meeting soup entry. If this is a repeating meeting or event, also specify the *date* parameter.

date Used only if *meetingEntry* is a repeating meeting or event. This parameter must be the date and time of a particular instance of the repeating meeting or event.

openDefaultSlip A Boolean. Set this value to `true` to cause the Dates application to open the default meeting slip for the meeting. Set this value to `nil` to cause the Dates application to send the `OpenMeeting` message to the frame registered for this meeting type, if there is one. (For more details on `OpenMeeting` See page 18-65.)

This method always returns `nil`.

RegInfoItem

calendar:RegInfoItem(*symbol*, *frame*)

Adds an item to the end of the Info button picker in the base view of the Dates application. This method always returns nil.

symbol — A unique symbol identifying the item.

frame — A frame containing two slots:

item	A string or a bitmap frame. This is the item to display in the picker.
DoAction	This method is called if the user picks this item. It takes no parameters. This method always returns nil.

Items added by this method are not persistent across a system restart.

To remove the item added by this method, call UnRegInfoItem.

RegMeetingType

calendar:RegMeetingType(*symbol*, *frame*)

Registers a new meeting type for the Dates application. The meeting type appears in the New button picker in the Dates application.

symbol — A unique symbol identifying the meeting type.

frame — A frame describing the new meeting type. The frame slots are described below.

This method always returns nil.

The slots in *frame* are as follows:

Slot description	
item	Required. A string that is the meeting type name to appear in the picker.
icon	Required. A bitmap frame containing a bitmap to appear next to the name in the picker. This bitmap is also displayed in the New pickers, the day view, a

	agenda view, and overview. The bitmap must be no larger than 24 pixels wide by 15 pixels high.
`newMeeting`	Required. Called if the user chooses this meeting type in the New picker (See "NewMeeting" on page 18-66.)
`smallicon`	Optional. A bitmap frame containing a bitmap to be displayed in the meeting slip. The bitmap must be no more than 12 pixels high.
`memory`	Optional. A symbol for the system storage location for previous meeting titles of this new meeting type. When the user makes a new meeting of this type, the Title picker in the meeting slip lists previous meeting titles of this meeting type as a convenience. If you don't provide this slot, the Dates application uses the storage location allocated for the underlying meeting type (that is, as a meeting).
`openMeeting`	Optional. Called if the user taps on the icon of a meeting or event whose `meetingType` slot matches the symbol under which this meeting type was registered (See "OpenMeeting" on page 18-67.)

NewMeeting

meetingtype`:NewMeeting (`*date*`, `*parentBox*`)`

This method is called if the user chooses this meeting type in the New picker. It is a slot of the frame register with `RegMeetingType`. It takes two parameters, *date* and *parentBox*. This method must create a meeting (or event) using the `AddAppointment` or `AddEvent` method and must set the `meetingType` slot to the symbol under which the meeting type was registered. You set the `meetingType` slot in the soup entry returned by `AddAppointment` or `AddEvent` and call `EntryChange` to save it.

date	This parameter is the current date displayed by the Dates application.

parentBox This parameter is the global `viewBounds` of the calendar base view.

If `NewMeeting` returns the new meeting frame, the Dates application performs the default action, which is to open the default meeting slip. If this method returns `nil`, the Dates application does nothing and this method should perform any actions necessary.

If this method opens its own meeting slip or other view, it should do so by using the calendar method `RememberedOpen`. That method opens the view and records it so that the Dates application can close the view if the Dates application is closed.

OpenMeeting

frame`:OpenMeeting(`*meeting*`,` *date*`,` *parentBox*`)`

This method is called if the user taps on the icon of a meeting or event whose `meetingType` slot matches the symbol under which this meeting type was registered. It is a slot of the frame registered with `RegMeetingType`.

meeting This parameter is the soup entry for the tapped item

date This parameter is the date and time of the meeting

parentBox This parameter is the global `viewBounds` of the calendar base view.

If `OpenMeeting` returns a non-`nil` value, the Dates application performs the default action, which is to open the default meeting slip. If this method returns `nil`, the Dates application does nothing and this method should perform any actions necessary.

If this method opens its own meeting slip or other view, it should do so by using the calendar method `RememberedOpen`. That method opens the view and records it so that the Dates application can close the view if the calendar is closed.

Meeting types added by `RegMeetingType` are not persistent across a system restart.

To remove the meeting type added by `RegMeetingType`, call `UnRegMeetingType`.

RememberedClose

calendar:RememberedClose(*view*)

Closes a view in the calendar that had been opened with RememberedOpen. If the view is closed without calling this method, the Dates application keeps a reference to the view until the calendar is closed, which wastes memory.

view The view to close.

This method always returns nil.

This method should be used to close views opened by RememberedOpen from within the *frame*.OpenMeeting method of RegMeetingType and from within the *frame*.DoAction method of RegInfoItem.

RememberedOpen

calendar:RememberedOpen(*view*)

Opens a view in the calendar and records it so that if the calendar is closed, that view is also closed.

view The view to open.

This method always returns nil.

This method should be used to open views from within the *frame*.OpenMeeting method of RegMeetingType and from within the *frame*.DoAction method of RegInfoItem.

Views opened by RememberedOpen should be closed by RememberedClose.

SetEntryAlarm

calendar:SetEntryAlarm(*mtgText*, *mtgStartDate*, *minutesOrDaysBefore*)

Sets an alarm on the meeting or event with the given text at the given date and time. If the meeting or event is an instance of a repeating meeting or event, the alarm is set for all instances of the repeating meeting or event.

mtgText A string that is the meeting text of the meeting or event for which you want to set the alarm time.

mtgStartDate An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method will find an event scheduled any time during the day of *mtgStartDate*.

minutesOrDaysBefore

A non-negative integer, which specifies how far in advance of the meeting or event the alarm should go off. This parameter can be 0 (zero) or a positive integer. Zero means the alarm goes off at the time of the meeting, or at 9 A.M on the day of the even. For a meeting, this integer should specify the number of minutes before *mtgStartDate* that you want the alarm to go off. For an event, this integer should specify a number of days before the event. Event alarms go off at 9 A.M.

You can specify `nil` to clear an alarm that is currently set.

This method always returns `nil`.

SetMeetingIconType

calendar`:SetMeetingIconType(`*mtgText*`,` *mtgStartDate*`,` *newIconType*`)`

Finds a particular meeting or event and sets its icon type. If the item found is an instance of a repeating meeting or event, the icon type is changed for all instances in that repeating series.

mtgText A string that is the meeting text of the meeting or event for which you want to set the icon type.

mtgStartDate An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method will find an

event scheduled any time during the day of *mtgStartDate*.

newIconType A symbol specifying the new icon type to set for the meeting or event. You can specify the following icon types: `'Event`, `'Meeting`, `'WeeklyMeeting`, `'MultiDayEvent`, or `'AnnualEvent`.

This method always returns `nil`.

If the new icon type is incompatible with the type of the meeting or event, then this method throws an exception. Table 18-1 shows the icon types and the meeting and event types with which they are compatible.

Table 18-1 Compatible icon and meeting/event types

Icon type	Compatible meeting/event types
`'Meeting`	Any type of meeting, but not events
`'WeeklyMeeting`	Only meetings that repeat on the same day each week
`'Event`	Any type of event
`'MultiDayEvent`	Only events that repeat every day
`'AnnualEvent`	Only events that repeat on the same day each year

SetMeetingInvitees

calendar`:SetMeetingInvitees(`*mtgText*`,` *mtgStartDate*`,` *invitees*`)`

Sets list of invitees for the meeting specified by *mtgText*, and *mtgStartDate*.

mtgText A string that is the meeting text of the meeting for which you want to set the list of invitees.

mtgStartDate An integer specifying the start date and time of the meeting, in the number of minutes passed since midnight, January 1, 1904.

invitees An array specifying the invitees to be set. The array can contain the following objects: name references (see the description below), Names soup entries, aliases to Names soup entries, or a frame containing first and last name strings (for write-in names). The last item, the frame, must have this structure:

`{name: {first:` *string*`, last:` *string*`}}`

Note that you can specify the empty string, or `nil`, or leave the slot out if the first or last name is missing.

To clear the list of invitees for the meeting, specify `nil` for the *invitees* parameter.

This method always returns `nil`.

If the specified meeting is a repeating meeting and is not an exception meeting, this method sets the list of invitees for all meetings in the repeating series. If the specified meeting is a repeating meeting exception, the list of invitees applies to that exception meeting only.

Use `MakeNameRef` to reference names as follows:

```
GetDataDefs('|nameRef.people|): MakeNameRef
```

SetMeetingLocation

calendar`:SetMeetingLocation(`*mtgText*`,` *mtgStartDate*`,` *location*`)`

Sets the location for the meeting specified by *mtgText* and *mtgStartDate*.

mtgText A string that is the meeting text of the meeting for which you want to set the location.

mtgStartDate An integer specifying the start date and time of the meeting, in the number of minutes passed since midnight, January 1, 1904.

location Specifies the location to be set. This parameter must be one of the following objects:

- a name reference (see the previous description under“SetMeetingInvitees” on page 18-70)

- a Names soup entry
- an alias to a Names soup entry
- a string (write-in location)
- `nil`, meaning clear the location for the meeting

This method always returns `nil`.

If the specified meeting is a repeating meeting and is not an exception meeting, this method sets the location for all meetings in the repeating series. If the specified meeting is a repeating meeting exception, the location applies to that exception meeting only.

SetMeetingNotes

calendar`:SetMeetingNotes(`*mtgText*`,` *mtgStartDate*`,` *notes*`)`

Sets the notes for a meeting or event specified by *mtgText* and *mtgStartDate*.

mtgText	A string that is the meeting text of the meeting or event for which you want to set the notes.
mtgStartDate	An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method finds an event scheduled any time during the day specified by *mtgStartDate*.
notes	Specifies the notes to be set. This parameter must be an array of paragraph and polygon frames, or it can be a string. The new notes replace all existing notes. You can specify `nil` to clear the notes.

This method always returns `nil`.

If the specified meeting is a repeating meeting, this method sets the notes for only the particular instance of the meeting identified by *mtgStartDate*.

SetRepeatingEntryStopDate

calendar:SetRepeatingEntryStopDate(*mtgText*, *mtgStartDate*, *mtgStopDate*)

Sets the stop date for the repeating meeting or event with the given text at the given date and time.

mtgText	A string that is the meeting text of the repeating meeting or event for which you want to set the stop time.
mtgStartDate	An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method will find an event scheduled any time during the day of *mtgStartDate*.
mtgStopDate	An integer specifying the stop date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. The stop date is the date after which the meeting or event will no longer repeat. If you specify `nil`, the meeting or event will repeat forever.

This method always returns `nil`.

UnRegInfoItem

calendar:UnRegInfoItem(*symbol*)

Removes an item previously added by `RegInfoItem` to the Info button picker in the base view of the calendar.

symbol	The symbol used to identify the item in the `RegInfoItem` method that added the item.

This method always returns `nil`.

UnRegMeetingType

calendar:UnRegMeetingType(*symbol*)

Removes a meeting type previously added by RegMeetingType to the New button picker in the base view of the calendar.

symbol The symbol used to identify the item in the RegMeetingType method that registered the item.

This method always returns nil.

Dates Soup Formats

This section describes the format of entries in the Dates soups. Dates soup entries consist of either meeting frames or notes frames.

Meeting Frames

Each meeting frame contains the following required slots.

Slot descriptions

viewStationery In the Calendar soup this slot always contains the value 'Meeting for a meeting. In the Repeat Meetings soup this slot always contains the value 'repeatingMeeting. In the Calendar Notes and Repeat Notes soups it always contains the value 'CribNote.

mtgStartDate Contains an immediate value; the start date of the meeting (or date the event was entered) in the number of minutes passed since midnight, January 1, 1904. For events the time is midnight at the beginning of the day.

mtgDuration Contains an immediate value; the duration of the meeting in minutes. For events this value is meaningless and should be set to zero.

`mtgText` A rich string containing the meeting or event text. A rich string is used because the user can enter an ink meeting or event.

A meeting frame may also contain the following optional slots.

Slot descriptions

`mtgStopDate` Used for repeating meetings and events. An immediate value; the date that the meeting should stop repeating, in the number of minutes passed since midnight, January 1, 1904.

`repeatType` Used for repeating meetings and events. Contains one of the following constants that describe how often the meeting repeats: `kDayofWeek`, `kWeekInMonth (1)`, `kDateInMonth (2)`, `kDateInYear(3)`, `kPeriod(4)`, `kNever(5)`, `kWeekInYear(7)`.

`mtgInfo` Used for repeating meetings and events. An immediate value containing packed repeating meeting information. This slot is interpreted differently, depending on the value of the `repeatType` slot, as follows:

`repeatType` = `kDayOfWeek`

`mtgInfo` is set to any combination of constants from the following two groups added together:

Constants for day of week

`kSunday`	0x00000800
`kMonday`	0x00000400
`kTuesday`	0x00000200
`kWednesday`	0x00000100
`kThursday`	0x00000080
`kFriday`	0x00000040
`kSaturday`	0x00000020
`kEveryday`	0x00000FE0

Constants for week in month

`kFirstWeek`	0x00000010

kSecondWeek	0x00000008
kThirdWeek	0x00000004
kFourthWeek	0x00000002
kLastWeek	0x00000001
kEveryWeek	0x0000001F

`repeatType` = `kWeekInMonth`
`mtgInfo` is set to a single constant from the first group above, added to any combination from the second group.

`repeatType` = `kDateInMonth`
`mtgInfo` is set to the date in the month on which the meeting or event is to repeat.

`repeatType` = `kDateInYear`
`mtgInfo` is set to (*month*<<8) + *date*, where *month* is the number of the month in the year (January = 1) and *date* is the date in the month on which the meeting is to repeat.

`repeatType` = `kPeriod`
`mtgInfo` is set to (*mtgDay*<<8) + *period*, where *mtgDay* is the date, measured in days, of the meeting. This is the same as `mtgStartDate`, but in days, instead of minutes—that is, more simply `(mtgStartDate DIV 1440)`. *period* is the number of days between meetings. Technically, *period* can range between 1 and 255; however, the current Newton user interface allows the user to choose only every other week (14 days) for this kind of meeting. Opening a `kPeriod` meeting will always display it as an "Every other week" meeting type and reset its *period* to 14.

`repeatType = kWeekInYear`
`mtgInfo` is set to (*month*<<<12) plus a

single constant from the day-of-week constants (for example, `kThursday`) plus a single constant from the week-in-month constants (for example `kThirdWeek`).

Note

`mtgInfo` uses only the least significant 24 bits of the integer. The remaining bits are reserved for future expansion. Always be sure to mask out the upper bits so a future change in format will not overflow your values. ◆

`mtgAlarm` Contains an immediate value. For single (non-repeating) meetings or events, this value is the time when the alarm should occur. This value is represented as the number of minutes passed since midnight, January 1, 1904. For repeating meetings or events, this slot contains the number of minutes before the meeting or event at which the alarm should occur.

`mtgIconType` This slot determines what kind of icon to display for this meeting or event and what kind of slip to display when the user taps the meeting marker. The valid values are `'Meeting`, `'WeeklyMeeting`, `'Event`, `'MultiDayEvent`, and `'AnnualEvent`. If the slot doesn't exist, or is `nil`, then the icon defaults to the Meeting icon or the Event icon.

`mtgInvitees` If the meeting frame is for a meeting, not for an event, then this slot contains an array listing the invitees to the meeting. The elements in the array are name references (see page 18-56). If the meeting has no invitees specified, `nil` is stored in the slot. Use the methods `GetMeetingInvitees` and `SetMeetingInvitees` to access this slot.

`mtgLocation` If the meeting frame is for a meeting, not for an event, then this slot stores the meeting location as a name reference (see page 18-56). If the meeting has no location specified, `nil` is stored in the slot. Use the methods `GetMeetingLocation` and `SetMeetingLocation` to access this slot.

`notesData` Contains an array of meeting (or event) note objects for non-repeating meetings and events. Meeting notes are the notes visible when the user taps on the Add notes or Editnotes button in a meeting slip to open its notes view. Meeting notes can consist of text objects, polygons, or ink objects. These objects have the same format as data objects in the Notes soup. For information on the format of these objects, see the description of the `data` slot in the section "Notes Soup Format" beginning on page 18-90. Text objects have, in addition, a `viewFont` slot specifying the font of the text.

Notes for repeating meetings and events are stored in the `instanceNotesData` slot.

`instanceNotesData`

Contains an array of aliases to notes for instances of repeating meetings and events that have notes. Each instance's notes are stored as a separate soup entry in the Repeat Meetings soup containing the repeating meeting, or in the Repeat Notes soup containing the repeating event.

`instanceNotesData` is an array of pairs. Each pair is an array of two elements: [*time*, *notesAlias*]. The first element, *time*, is the date and time of the meeting or event instance. The second element, *notesAlias*, is an alias to another entry in the same soup; that entry contains the actual notes for that instance. For a description of the format of a note entry (See "Notes Frames" on page 18-79.)

Use the methods `GetMeetingNotes` and `SetMeetingNotes` to access this slot.

Notes for non-repeating meetings and events are stored in the `notesData` slot.

`version` Contains the integer 2 if the meeting or event was created by version 2.0 of the Dates application. If this slot is missing or its value is `nil`, then the Dates application assumes the meeting or event was created

by the 1.x version of the application. When the Dates application converts a 1.x meeting or event to 2.0 format, it sets this slot to the value 2.

`viewBounds` A bounds frame used only for meetings that are not at the left edge of the Day view, such as double-booked meetings.

`exceptions` Used for repeating meetings or events. This is an array of arrays representing meetings and events that are exceptions to the normal repeating time, for example, when a user has erased one of the instances of a meeting or event or has changed the starting time or duration. It would then be listed in this array as an exception.

Warning

The internal format of exception meetings is subject to change, so you should treat this array as read-only, and not attempt to add to it. ▲

Each sub-array represents one exception. There are two elements in each array. The first is an integer specifying the normal time of the meeting or event, in the number of minutes since midnight, January 1, 1904. The second element is either `nil`, meaning the meeting or event has been erased, or an exception meeting frame that contains the changed information, such as a different `mtgStartDate` or `mtgDuration`.

Note that the `viewStationery` slot of an exception meeting frame contains the symbol `'exceptionMeeting`. The `viewStationery` slot of an exception event frame contains the symbol `'cribNote`.

Notes Frames

Notes frames occur in the Repeat Meetings and Repeat Notes soups. Notes frames contain the notes for a specific instance of a repeating meeting or event.

Each notes frame contains the following required slots.

Slot descriptions

`notes` An array of note objects. Notes can consist of text objects, polygons, or ink objects. These objects have the same format as data objects in the Notes soup. For information on the format of these objects, see the description of the `data` slot in the section "Notes Soup Format" beginning on page 18-90. Text objects have, in addition, a `viewFont` slot specifying the font of the text.

`repeatingMeetingAlias` An alias to the repeating meeting or event soup entry with which these notes are associated.

To Do List Reference

To Do List Soup Format

The To Do List uses a soup called To Do List where each day has a single entry with a topics slot containing an array element for each task. Topic frames do not use the `level` slot, since it would always be 1. See"Structured List Views" beginning on page 8-18 for a list of the topic frame slots. The To Do List uses one additional slot, `repeatInfo`.

Slot descriptions

`repeatInfo` A frame containing the meeting frame slots `mtgStartDate`, `mtgStopDate`, `repeatType`, and `mtgInfo` (see page 18-74.) These slots are used as in repeating meetings. The meeting frame slots appear in detail beginning on page 18-74.

To Do List Methods

These methods are sent to the To Do frame, which is obtained by `GetRoot().calendar:GetToDo()`.

CreateToDoItem

toDoframe`:toCreateToDoItem` (*aDate*, *aRichString*, *aReminder*, a*Frequency*)

This method adds a task with the specified text for the specified date.

aDate The date of the task in the number of minutes passed since midnight, January 1, 1904.

aRichString Text associated with the meeting. A rich string is necessary because the user can enter ink.

aReminder An integer. The number of days notice before the meeting, or `nil`.

aFrequency A frame specifying `mtgStartDate`, `mtgStopDate`, `repeatType`, and `mtgInfo`, or `nil`.

It returns an integer, which is the index of the item made (i.e. if the new task is the second one on the specified date, `CreateToDoItem` returns 1).

Slot descriptions

`mtgStartDate` Contains an immediate value; the start date of the meeting (or date the event was entered) in the number of minutes passed since midnight, January 1, 1904.

`mtgStopDate` Used for repeating meetings. An immediate value; the date that the meeting should stop repeating, in the number of minutes passed since midnight, January 1, 1904.

`repeatType` Contains one of the following constants that describe how often the meeting

repeats: `kDayofWeek` (0), `kWeekInMonth` (1), `kDateInMonth` (2), `kDateInYear` (3), `kPeriod` (4), `kNever` (5).

`mtgInfo` An immediate value containing packed repeating meeting information. This slot is interpreted differently, depending on the value of the `repeatType` slot, as describe on page 18-75.

CreateToDoItemAll

toDoframe`:CreateToDoItemAll (`*aDate*`,` *aRichString*`,` *aReminder*`,` *aFrequency*`,` *aPriority*`,` *aCompleted*`)`

This method adds a task with the specified text for the specified date and sets the priority and completion status.

aDate The date of the task in the number of minutes passed since midnight, January 1, 1904.

aRichString Text associated with the meeting. A rich string is used because the user can enter ink.

aReminder An integer. The number of days notice before the meeting, or `nil`.

aFrequency A frame specifying `mtgStartDate`, `mtgStopDate`, `repeatType`, and `mtgInfo`, or `nil`. Returns an integer, which is the index of the item made (i.e. if the new task is the second one on the specified date, `CreateToDoItem` returns 1).

slot descriptions

`mtgStartDate` Contains an immediate value; the start date of the meeting (or date the event was entered) in the number of minutes passed since midnight, January 1, 1904.

`mtgStopDate` Used for repeating meetings. An immediate value; the date that the

		meeting should stop repeating, in the number of minutes passed since midnight, January 1, 1904.
	`repeatType`	Contains one of the following constants that describe how often the meeting repeats: `kDayofWeek` (0), `kWeekInMonth (1)`, `kDateInMonth (2)`, `kDateInYear(3)`, `kPeriod(4)`, `kNever(5)`.
	`mtgInfo`	An immediate value containing packed repeating meeting information. This slot is interpreted differently, depending on the value of the `repeatType` slot.
aPriority		This parameter takes an integer: 0 = high, 1 = medium, 2 = low, 3 = none.
aCompleted		This parameter takes `true` or `nil`, depending on whether the task is completed or not.

This method returns the topics index in the entry of the specified date. If the ToDo is open, it returns the index of the topic in the local topics array.

GetToDoItemsForRange

toDoframe`:GetToDoItemsForRange` (*aDate1*, *aDate2*)

This method returns an array of frames of the format `{date: d, topics: []}`. `aDate1 >= aDate2.`

If *aDate2* is greater than *aDate1*, the method returns `nil`.

GetToDoItemsForThisDate

toDoframe`:GetToDoItemsForThisDate` (*aDate*)

This method returns an array of `ListView` topics for the date specified. It merges any multiple stores, such as those resulting from duplication on storage cards, repeating tasks, and tasks imported from earlier version of the software). This method has the side effect of sorting the soup entry's topics; it sorts them as the user sees them, by undone and done and then by priority.

aDate The date of the task.

GetTaskShapes

toDoframe:GetTaskShapes (*aShapes*, *aTask*, *aYOffset*, *aWidth*)

This method returns an array of shapes for the task. It returns a frame

```
{shapes: s, height: h}
```

where `s` is the *aShapes* array and `h` is the height of the added shapes. This frame is used in printing.

aShapes If *aShapes* is `nil`, it's an array to which the shapes are added, and which is returned. No style is added to the array. If *aShapes* is `nil`, then just the new shapes are returned, and a style is included.

aTask The task specified.

aYOffset All shapes receive this offset. The units are points.

aWidth The width to wrap text to, in points.

GetToDoShapes

toDoframe:GetToDoShapes (*aDate*, *aYOffset*, *aWidth*)

This method returns an array of shapes, which is used in printing.

aDate The date to get shapes for in the number of minutes passed since midnight, January 1, 1904.

aYOffset All shapes receive this offset. The units are points

aWidth The width to wrap text to, in points.

LastVisibleTopic

toDoframe:LastVisibleTopic()

This method returns the index of the last topic drawn in the view.

NextToDoDate

toDoframe:NextToDoDate(*aDate*)

This method returns the date of the next task on or after `aDate`, or `nil` if there are none.

aDate The date of the task in the number of minutes passed since midnight, January 1, 1904.

RemoveOldTodoItems

toDoframe:`RemoveOldTodoItems`(*aBeforeDate*, *aRemoveWhich*, *aProgressContext*)

This method removes any To Do item dated prior to *aBeforeDate*.

aBeforeDate The date of the oldest allowable To Do item, which you can specify in the number of minutes passed since midnight, January 1, 1904.

aRemoveWhich Set to `'done` or `nil`. If *aRemoveWhich* is `'done`, then only completed items are removed. If *aRemoveWhich* is `nil`, then every item before *aBeforeDate* is removed.

aProgressContext The location to which the To Do list sends *UpdateProgressBar*. If it is `nil`, no message is sent.

Time Zone Reference

The developer's interface to Time Zone consist of functions that support the ROM/RAM soup scheme, of wrapper functions that retrieve information about a particular location, and of general utility functions.

GetLocationSoupNames

`GetLocationSoupNames()`

This method returns the names of all of the location soups as an array of strings.

GetLocationSoup

`GetLocationSoup(`*name*`)`
This method returns the union soup for the specified location.

LocationEntryChange

`LocationEntryChange()`
This method updates the entry. It automatically clones a ROM entry into the RAM soup, adding a '`replace` slot.

LocationEntryRemoveFromSoup

`LocationEntryRemoveFromSoup(`*entry*`)`
If the entry is from the RAM soup and is not a replacement for a ROM soup entry, this method removes it from the soup. If it is a ROM entry, the method adds a new entry to the RAM soup with a '`delete` slot, and then removes from the frame all slots that are not indexed. If the entry is from the RAM soups and has a kind slot of `'replace`, the kind slot is changed to `'delete` and all non-indexed slots are removed.

ROM/RAM Location Soup Wrapper Functions

GetCityEntry

`GetCityEntry(`*name*`)`
This method returns an array of the cities in the Cities union soup whose name slot matches the name parameter, i.e. if *name* is “Portland” the routine returns an array containing both Portland, OR and Portland, ME. If no entries match an empty array returns. If the location store part is not found, this method returns `nil`.

GetCountryEntry

`GetCountryEntry(`*name*`)`
This method returns an array of the countries in the Country union soup whose name or symbol slot matches the name parameter. If the name

parameter is of class `'string` the query uses the `'name` index, otherwise it uses the `'symbol` index. If no entries match, an empty array returns. If the location store part is not found this routine returns `nil`.

GetDSTEntry

`GetDSTEntry(`*name*`)`
This method returns an array of the daylight savings time entries in the DST union soup whose name or symbol slot matches the name parameter. If the name parameter is of class `'string` the query uses the `'name` index, otherwise it uses the `'symbol` index. If no entries match, an empty array returns. If the location store part is not found this routine returns `nil`.

Utility Functions

ClosestLocations

`ClosestLocations (`*soup*`, `*count*`, `*radius*`, `*x*`, `*y*`, `*extra*`)`
This method returns an array of entries closest to the given location. The entries returned in the array are returned sorted by name.

soup	The soup of locations to search.
count	The maximum number of entries to return in the array.
radius	The radius of the search in miles.
x, *y*	The locations to use as the center of the search.
extra	A frame that contains the following slots:

Slot descriptions

`width`	Required.The width of the coordinate space relative to the *x* parameter.
`height`	Required. The height of the coordinate space relative to the *y* parameter in.
`slot`	Optional.The slot in each of the entries against which the filter parameter is tested for equality.
`filter`	Optional. The value to use in the above test.

If there are no entries within the specified radius that match the specified requirements the method returns an empty array.

SetExtrasInfo

Extrasdrawer`:SetExtrasInfo(`*appSymbol*`,` *newInfo*`)`

This method changes the Extras drawer information for the application specified by *appSymbol* to the information frame specified in *newInfo*. The return value of the function is the information frame that was in effect before this call.

appSymbol	An application symbol.
newInfo	A new information frame for the application represented by *appSymbol*.

The new information frame can have the following slots:

Slot description

`icon`	A BitMap or PICT image that appears in the extras drawer.
`text`	A text string displayed below the icon.

If either slot is missing from the information frame or is `nil`, the value of that slot is not changed.
If the application isn't found, this function returns `nil`.

NewCity

`NewCity(`*`newCity`*`, `*`workSite`*`, `*`makeHome`*`)`

This method adds the city specified by `newCity` to the cities soup. Always set `workSite` to `nil`. If `makeHome` is `true`, the current location is set to the location. For more information see “Adding a City to the City Soup” on page 18-26

Notes Reference

Notes Methods

The following methods let you add a note and warn a user about the size of a note

MakeTextNote

paperroll`:MakeTextNote (`*`string`*`, `*`add`*`)`
Use this method to add a note to the `paperroll`. The height is calculated automatically. `MakeTextNote` relies on the notepad being open.

string	The string is the text you want appear as a note.
add	A Boolean; if `true` the note is added to the paperroll; if `false` the note is returned.

AppMaxSizeExceeded

paperroll`:AppMaxSizeExceeded(`*`entry`*`)`
The parameter, *`entry`* is the soup entry for the offending note. Currently, this function opens the Notify slip, but developers can use it as they wish (see page 18-30).

MaxEntrySizeBytes

paperroll`:maxEntrySizeBytes (`*bytes*`)`
This function sets the limit in bytes that will trigger a warning to the user that a note is too large (See page 18-30.) If *bytes* is `nil` the warning does not take place.

bytes	The number of bytes that will trigger a warning to the user that a note is too big. There is no action beyond the warning.

Notes Soup Format

This section describes the format of entries in the Notes soup. Each entry consists of a frame with the following slots.

Slot descriptions

`viewStationery`	This slot always contains the symbol `'paperroll`.
`class`	The class symbol varies according to the type of stationary the user creates on the New picker, which may be a Note (class `paperroll`), an outline (class `list`), or a checklist (class `checkList`).
`paperroll`	A typical Note entry looks like this:

```
{
          class: paperroll,
          data: [#440EE11],
          viewStationery: paperroll,
          height: 200,
           timeStamp: 47863380,
}
```

The data slot contains an array of frames created and manipulated by `clEditView`, for example:

```
[
{#440EE49},
{#440F701},
{#440EF51}
]
```

Here is `data[0]`:

```
{
styles: [#440EA39],
viewBounds: {#440F221},
viewStationery: para,
text: "hello",
_proto: {#307}
}
```

`list`

A typical outline entry looks like:

```
{
topics: [#440ED59],
class: list,
data: nil,
viewStationery: paperroll,
height: 200,
timestamp: 47863531,
}
```

Note that data is `nil`, class is `'list`. The `topics` slot contains outline data. It is an array of frames, each with the following format:

```
{
styles: [#440EDB9],
hideCount: 0,
viewBounds: {#440EE81},
```

```
level: 1,
text: "hello"
}
```

hideCount specifies how many items are hidden at level *level*.

`checkList` A typical `checkList` entry looks like this:

```
{
topics: [#44170C9],
class: checkList,
data: nil,
viewStationery: paperroll,
height: 200,
timestamp: 47863531,
}
```

Note that data is `nil` and class is `'checkList`. Data is an array of frames, each with the following format:

```
{
mtgDone: NIL,
styles: [#4417161],
hideCount: 0,
viewBounds: {#44171B9},
level: 1,
text: "check"
}
```

mtgDone is `true` if the check box is checked or `nil` if it is unchecked. *hideCount* specifies how many items are hidden at level *level*.

`height` This slot contains an immediate value that is the height, in pixels, of the note.

`data` This slot holds an array of frames, which contains either text objects, polygons, or ink objects. There is one frame

for each text paragraph, polygon, or ink object in the note. The text object frames have these slots:

Slot description

`viewStationery`
: Required. Always contains the symbol `'para`.

`viewBounds`
: Required. The bounds of the text object.

`text`
: Required. A string that is the text contained in the paragraph.

`tabs`
: Optional. An array of tab stops.

`styles`
: Optional. An array holding font style information for the text.

For more information on the slots particular to paragraph views (See the section “Paragraph Views” beginning on page 8-12.)

The polygon object frames have these slots:

`viewStationery`
: This slot always contains the symbol `'poly`.

`viewBounds`
: The bounds of the polygon.

`points`
: Contains a binary data structure, which holds polygon data.

The ink object frames have these slots:

`ink`
: This slot contains a binary data structure of the class `'ink` that holds the ink data.

`viewBounds`
: The bounds of the ink object.

`timeStamp`
: Contains an immediate value; the date and time that this note was created, in the

number of minutes passed since midnight, January 1, 1904.

A note entry frame may also contain the following optional slot:

`labels` A symbol specified by the user as a label (file folder) for the note.

Auxiliary Button Reference

These methods let you to add buttons to the status bars of the Notes and Names applications and let your applications allow themselves to be extended.

RegAuxButton

`RegAuxButton (`*buttonSymbol*`,` *template*`)`

This method adds a button to the auxiliary button registry. The return value of `RegAuxButton` is currently undefined.

buttonSymbol The unique symbol for the button, with your signature. (See "Naming Soups" on page 11-64 for a discussion of signatures.)

template A view template for the button to be added, with one extra slot, `destApp`, which you should set to the symbol of the application that you want to add the button to. (For example, `'paperroll` or `'cardfile` to add the button to the Notes or the Names application, respectively.) If `destApp` is `nil`, this button is added to the background application, if it can support it.

UnRegAuxButton

UnRegAuxButton (*buttonSymbol*)

This function removes the button with the given symbol from the auxiliary button registry. The return value of UnRegAuxButton is currently undefined.

buttonSymbol The unique symbol for the button, with your signature. (See "Naming Soups" on page 11-64 for a discussion of signatures.)

GetAuxButtons

GetAuxButtons (*appSymbol*)
Returns an array that contains the buttons specific to your application and, if yours is the backdrop application, any other buttons designated for the backdrop application. Each array element is a frame with a butt slot, which holds the template for the button.
Other slots may exist in this frame, but are undefined and are subject to change.

appSymbol Your unique application symbol with your signature.

AddAuxButton

app:AddAuxButton *(buttonFrame*)

The AddAuxButton message is sent to your application when someone calls RegAuxButton specifying your application in the destApp slot, or when your application is the backdrop application and RegAuxButton is called with the destApp slot set to nil.

buttonFrame A frame that contains the butt slot. This slot holds the template for the button that was just added.

This method is optional; you are not required to implement it.

RemoveAuxButton

app:RemoveAuxButton (*buttonSymbol*)

The RemoveAuxButton message is sent to your application when someone calls UnRegAuxButton for a button that is specific to your application, or when your application is the backdrop application and UnRegAuxButton is called for a button whose destApp slot is set to nil.

buttonSymbol The unique symbol for the button, with your signature. (See "Naming Soups" on page 11-64 for a discussion of signatures.)

This method is optional; you are not required to implement it.

System Data Reference

System Data Structures

User Configuration Frame

This section describes most slots in the userConfiguration entry in the System soup that are available to your applications. Certain slots in this frame that are closely associated with text and shape recognition are described in the section "Using RecConfig Frames" beginning on page 10-29. Note that you should always use the functions GetUserConfig (page 18-101) and SetUserConfig (page 18-102) to access and change slots in this frame.

Slot descriptions

`address`	The string that the user entered as the first line of their Address in the Personal Preferences form.
`cityZip`	The string that the user entered as the second line of their Address in the Personal Preferences form.
`company`	The string that the user entered as their Company name in the Personal Preferences form.
`country`	The string that the user entered as their Country in the Personal Preferences form.
`countrySlot`	A symbol representing the country specified by the user as their Country in the Personal Preferences form.
`currentAreaCode`	The area code of the current or last number dialed. This is entered by the user in the Area Code field of the Call Options slip.
`currentCountry`	A symbol representing the country specified by the user as their Country in the Personal Preferences form, or as their current location in the Time Zones map, whichever was set last. This symbol is used to determine if inter-national dialing codes are needed when dialing phone numbers.
`currentPrinter`	A frame describing the last printer selected for use in a Print slip.
`dialingPrefix`	A string that is the dialing prefix of the current or last number dialed. This is entered by the user in the Prefix field of the Call Options slip.
`emailPassword`	A string that is the user's e-mailpassword, entered in the E-mail Account field of the Mail Preferences form.
`faxPhone`	The string that the user entered as their Fax phone in the Personal Preferences form.
`homePhone`	The string that the user entered as their Home phone in the Personal Preferences form.
`leftHanded`	This slot provides a single place for developers to look for a user's handedness preference. A non-`nil` value indicates the user is left handed. You may consider placing your views differently for left-handed users, for

	example, buttons that would appear on the right edge of the screen might instead be placed on the left edge.
`mailAccount`	A string that is the user's e-mail account name, entered in the E-mail Account field of the Mail Preferences form.
`mailNetwork`	A symbol representing the e-mail network selected in the Network field of the Mail Preferences form.
`mailPhone`	A string that is the number entered in the Phone Number field of the Mail Preferences form.
`name`	The string that the user entered as their Name in the Personal Preferences form.
`paperSizes`	The paper sizes currently installed in the system. They appear as choices for the Paper Size item in the Locale Preferences form. The dimensions here are independent of the current driver and tend to be somewhat less than the full printable area a driver can support. This leeway allows for printing to different drivers with the same word wrap and other issues. Note that page size is set as a system global and cannot be overridden for individual print jobs, so you cannot, for example, print a set of envelopes and a set of letters without changing the paper size globally.
`phone`	The string that the user entered as their Office phone in the Personal Preferences form.
`signature`	The signature the User enters north signature sip. It is text and ink.

Utility Functions

This section includes a description of some global functions applicable to the built-in applications and system data.

GetSysEntryData

`GetSysEntryData(`*entry*`, `*path*`)`

Returns the value of a slot from a built-in soup entry.

entry	The soup entry from which you want to read a slot.
path	A path expression specifying the data to read in the entry.

Use this function whenever you want to read the value of a slot in an entry from one of the built-in soups.

SetSysEntryData

`SetSysEntryData(`*entry*`, `*path*`, `*value*`)`

Sets the value of a slot in a built-in soup entry.

entry	The soup entry in which you want to set a slot.
path	A path expression specifying the location to set in the entry.
value	The value you want to set in *path*.

Use this function whenever you want to set the value of a slot in an entry from one of the built-in soups.

RegPrefs

`RegPrefs(`*appSymbol*`,`*prefsTemplate*`)`

Registers with the system a template used to add an item to the Preferences roll in the Extras drawer. The template must be based on the `protoPrefsRollItem` system prototype. Note that items added to the Preferences roll must specify system-wide preferences rather than application-specific ones.

▲ WARNING

This function's return value is unspecified and may change in the future; do not rely on values returned by this function. ▲

appSymbol A unique symbol identifying the application adding this item to the Preferences roll; normally, the value of this parameter is the application symbol, which includes your registered signature.

prefsTemplate A view template based on the `protoPrefsRollItem` system prototype; it describes the view to be added to the Preferences roll. Items in the Preferences roll must be used for settings that are global in nature (like to Control Panels in the Mac OS), not for application-specific settings.

UnRegPrefs

`UnRegPrefs(`*appSymbol*`)`

Unregisters the specified application's Preference roll items.

▲ WARNING

This function's return value is unspecified and may change in the future; do not rely on values returned by this function. ▲

appSymbol A unique symbol identifying the application adding this item to the Preferences roll; normally, the value of this parameter is the application symbol, which includes your registered signature, or some variation on it.

RegFormulas

`RegFormulas(`*appSymbol*`,` *formulasTemplate*`)`

Registers with the system a template used to add a view to the Formulas application in the Extras Drawer. The template must be based on the `protoFormulasPanel` system prototype.

▲ WARNING

This function's return value is unspecified and may change in the future; do not rely on values returned by this function. ▲

appSymbol A unique symbol identifying the application adding this item to the Preferences roll; normally, the value of this parameter is the application symbol, which includes your registered signature.

formulasTemplate A view template based on the `protoFormulasPanel` system prototype; it describes the view to be added to the Formulas roll.

UnRegFormulas

`UnRegFormulas(`*appSymbol*`)`

Unregisters the specified Formulas application item.

▲ WARNING

This function's return value is unspecified and may change in the future; do not rely on values returned by this function. ▲

appSymbol A unique symbol identifying the application adding this item to the Preferences roll; normally, the value of this parameter is the application symbol, which includes your registered signature.

GetUserConfig

`GetUserConfig (`*configSym*`)`

Retrieves the value of a slot in the system `userConfiguration` frame.

configSym A symbol naming a slot in the `userConfiguration` frame.

This function returns the value of the requested `userConfiguration` slot.

For future compatibility, use this function instead of directly reading the global `userConfiguration` frame.

Here is an example of how to use this function:

```
savedPrinter := GetUserConfig('currentPrinter);
```

SetUserConfig

`SetUserConfig (`*`configSym`*`, `*`theValue`*`)`

Sets the value of a slot in the system `userConfiguration` frame and writes them to the system soup.

configSym	A symbol naming a slot in the `userConfiguration` frame.
theValue	The new value of the `userConfiguration` slot identified by *configSym*.

This function returns *theValue*.

For future compatibility, use this function instead of directly modifying the global `userConfiguration` frame.

Here is an example of how to use this function:

```
SetUserConfig('currentPrinter, savedPrinter);
```

RegUserConfigChange

`RegUserConfigChange(`*`appSymbol`*`,`*`callBackFn`*`)`

Registers a function object to be called each time the `userConfiguration` frame changes.

callbackID	A unique symbol identifying the function object to be registered; normally, the value of this parameter is the application symbol, which includes your registered signature, or some variation on it.
callBackFn	The function object that is called when the `userConfiguration` frame changes. The function

must be of the form
`func(`*what*`);`
Its parameters are

what The slot that changed in the `userConfiguration` frame

The value returned by the *callBackFn* function is ignored.

▲ **WARNING**

This function's return value is unspecified and may change in the future; do not rely on values returned by this function. ▲

UnRegUserConfigChange

`UnRegUserConfigChange(`*appSymbol*`)`

Unregisters a function object registered by the `RegUserConfigChange` function.

appSymbol A unique symbol identifying the function object to be unregistered; normally, the value of this parameter is the application symbol, which includes your registered signature, or some variation on it.

This function's return value is unspecified and may change in the future; do not rely on values returned by this function.

Summary

Constants and Variables

Table 18-2 Names card layouts

Constant	Value	Description
kSquiggle	0	Layout that uses squiggly line
kPlain	1	Plain layout
kSeparate	2	Layout with dashed lines
kCross	3	Layout with crossed lines

Table 18-3 Dates variables

Variable	Description
`firstDayOfWeek`	Specifies what the first day of the week should be, for display purposes. It holds an integer value from 0 to 6, where 0 means Sunday, 1 means Monday, and so on. The default value is 0; that is, display all months with Sunday as the first day of the week. This variable is a slot in the global `userConfiguration` frame, or in the locale bundle frame.
`useWeekNumber`	If non-`nil`, the Dates application displays the week number in the upper left-hand corner of its view. The first week of the year is number 1 and the last week is number 52. This variable is a slot in the locale bundle frame.

Table 18-4 Dates constants for the day of the week

Constant	Value	Description
`kSunday`	0x00000800	Sunday
`kMonday`	0x00000400	Monday
`kTuesday`	0x00000200	Tuesday
`kWednesday`	0x00000100	Wednesday
`kThursday`	0x00000080	Thursday
`kFriday`	0x00000040	Friday
`kSaturday`	0x00000020	Saturday
`kEveryday`	0x00000FE0	Every day in the week

Table 18-5 Dates constants for `repeatType`

Constant	Value	Description
`kDayOfWeek`	0	Meeting recurs on a specific week day of any week in the month.
`kWeekInMonth`	1	Meeting recurs in a specified week of the month.
`kDateInMonth`	2	Meeting recurs on a certain day of each month.
`kDateInYear`	3	Meeting recurs on a certain day of each year.
`kPeriod`	4	Meeting recurs on a specific day every two weeks.
`kNever`	5	Meeting does not recur.
	6	Reserved for internal use.
`kWeekInYear`	7	Meeting recurs in a specified week of the year.

Table 18-6 Other date constants

Constant	Value	Description
`kForever`	0x1fffffff	A special value.
`kMaxyear`	2919	The largest year value handled.
`kYearMissing`	2920	The nearest leap year before `kForever`. The string parser uses it to indicate that the year is missing in the date string.

Table 18-7 Dates constants for the weeks in a month

Constant	Value	Description
`kFistWeek`	0x00000010	The first week in the month
`kSecondWeek`	0x00000008	The second week in the month
`kThirdWeek`	0x00000004	The second week in the month
`kFourthWeek`	0x00000002	The fourth week in the month
`kLastWeek`	0x00000001	The last week in the month
`kEveryWeek`	0x0000001F	Any week in the month

Dates Error Codes

A list of the error codes in the Dates API follows. It lists the method, the name of the error, the error code number, the value, and a text string. The

text string is a detailed explanation of the error, but that string is not shown to the user.
Several errors may be mapped into a smaller set of errors, so, for example, a throw of `kFramesErrUnexpectedImmediate` can result form several different problems.
In some cases error codes don't match the actual error very closely. The most inconsistent or misleading examples are marked with a dagger (†) and you should be thoughtful in interpreting them.

Method: `calendar.ValidatePeriod`
Error: `kFramesErrNotNil`
Error Code: 48422
Value: `'period2` argument
Message: "`Expected nil for 'period2 argument.`"

Method: `calendar.ValidatePeridod`
Error: `kFramesErrUnexpectedImmediate`
Error Code: 48418
Value: `'period2` argument
Message: `"Expected nil, 'daily, 'weekly, 'biweekly, 'monthlyByWeek, 'monthly, 'yearly, or 'yearlyByWeek for 'period argument."`

Method: `calendar.ValidatePeriod`
Error: `kFramesErrNotAnArrayOrNil`
Error Code: 48413
Value: `'period2` argument
Message: `"Expected nil or array of integers for 'period2 argument."`

Method: `calendar.ValidatePeriod`
Error: `kFramesErrNotAnArrayOrNil`
Error Code: 48413
Value: `'period2` argument
Message: "`Since 'period is 'yearlyByWeek, expected`

nil or array of one integer for 'period2 argument."

Method: calendar.ValidatePeriod
Error: kFramesErrNotAnArray†
Error Code: 48401
Value: 'period2 argument
Message: "Expected array of integers for 'period2 argument."

Method: calendar.ValidatePeriod
Error: kFramesErrUnexpectedImmediate
Error Code: 48418
Value: 'period2 argument
Message: "Since 'period is 'weekly, expected integers between 0 and 6 for 'period2 argument."

Method: calendar.ValidatePeriod
Error: kFramesErrUnexpectedImmediate
Error Code: 48418
Value: 'period2 argument
Message: "Since 'period is 'monthlyByWeek or 'yearlyByWeek, expected integers between 1 and 5 for 'period2 argument."

Method: calendar.ValidatePeriod
Error: kFramesErrUnexpectedImmediate
Error Code: 48418
Value: 'period2 argument
Message: "Meeting 'startdate does not fit in any value in 'period2 argument"

Method: calendar.ValidateTitleAndDatetimeArgs
Error: kFramesErrNotAString
Error Code: 48402
Value: 'title argument

Message: "Expected string for meeting 'title argument."

Method: calendar.ValidateTitleAndDatetimeArg
Error: kFramesErrUnexpectedImmediate
Error Code: 48418
Value: 'datetime argument
Message: "Expected positive integer for 'datetime argument."

Method: calendar.IdentifyAppointments
Error: kFramesErrUnexpectedFrame
Error Code: 48416
Value: 'titleOrFrame argument
Message: "The frame argument is not a meeting frame."

Method: calendar.IdentifyAppointment
Error: kFramesErrUnexpectedImmediate
Error Code: 48418
Value: nil
Message: "No meeting/event found with that title on that date"

Method: calendar.IdentifyAppointment
Error: kFramesErrUnexpectedImmediate
Error Code: 48418
Value: array of appointments found
Message: "More than one meeting/event found with that title on that date."

Method: calendar.AddAppointment
Error: kFramesErrUnexpectedImmediat
Error Code: 48418
Value: 'duration argument

Message: "Expected positive integer for 'duration argument."

Method: calendar.DeleteAppointment
calendar.DeleteRepeatingEntry
calendar.DeleteEvent
Error: kFramesErrNotTrueOrNil
Error Code: 48424
Value: 'deleteOneOnly argument
Message: "Expected true or nil for 'deleteOneOnly argument."

Method: calendar.SetRepeatingEntryStopDate
Error: kFramesErrNotAnInteger
Error Code: 48406
Value: 'stopDateTime argument
Message: "Expected non-negative integer for 'stopDatetime argument."

Method: calendar.SetEntryAlarm
Error: kFramesErrUnexpectedImmediate
Error Code: 48418
Value: 'minutesOrDaysBefore argument
Message: "Expected nil or non-negative integer for 'minutesOrDaysBefore argument."

Method: calendar.SetMeetingIconType
Error: kFramesErrNotASymbol†
Error Code: 48410
Value: 'newIconType argument
Message: "Expected 'Meeting, 'WeeklyMeeting, 'Event, 'MultiDayEvent, or 'AnnualEvent for 'newIconType argument.""

Method: calendar.SetMeetingIconTyp
Error: kFramesErrUnexpectedImmediatle†
Error Code: 48418
Value: nil

Message: "New icon type is incompatible with old type."

Method: calendar.MoveAppointment
calendar.MoveOnlyOneAppointment
Error: kFramesErrUnexpectedImmediate
Error Code: 48418
Value: 'newDatetime argument
Message: "Expected positive integer or nil for 'newDatetime argument."

Method: calendar.MoveAppointmen
calendar.MoveOnlyOneAppointment
Error: kFramesErrUnexpectedImmediate
Error Code: 48418
Value: 'newDuration argument
Message: "Expected positive integer or nil for 'newDuration argument."

Method: calendar.FindAppointment
Error: kFramesErrNotAStringOrNi
Error Code: 48414
Value: 'title argument
Message: "Expected string or nil for 'title argument."

Method: calendar.FindAppointmen
Error: kFramesErrNotAnArrayOrNil
Error Code: 48413
Value: 'findWords argument
Message: "Expected array of strings or nil for 'findWords argument."

Method: calendar.FindAppointmne
Error: kFramesErrNotAnArrayOrNil
Error Code: 48413
Value: 'dateRange argument

Message: "Expected nil or integer or array of two integers for 'dateRange argument."

Method: alendar.FindAppointmne
Error: kFramesErrNotAnArrayOrNil
Error Code: 48413
Value: 'type argument
Message: "Expected nil, a meeting type symbol, or an array of meeting type symbol(s) for 'type argument. The possible meeting type symbols are Meeting, Event, RepeatingMeeting, and RepeatingEvent."

Method: calendar.FindAppointmen
Error: kFramesErrUnexpectedImmediate
Error Code: 48418
Value: maxNumberTofind argument
Message: "Expected nil or number for maxNumberTofind argument"

Method: calendar.FindAppointmen
Error: kFramesErrUnexpectedImmediate
Error Code: 48418
Value: array of appointments found
Message: "More than one matching meeting/event found."

Method: calendar.FindAppointme
Error: kFramesErrUnexpectedImmediate
Error Code: 48148
Value: nil
Message: "Not matching meeting/event found."

Method: calendar.RegMeetingType
Error: kFramesErrUnexpectedFrame
Error Code: 48416
Value: 'frame argument
Message: "The 'frame argument to RegMeetingType is missing the item, icon or newMeeting slots."

Functions and Methods

Names Application Methods

cardfile:AddCard(*dataDefType*, *entryFrame*) // Creates a new card in the Names application

FormatFunc(*pathArray*) // Returns a string for the All Info view

cardfile:AddLayout (*layout*) // Specify a view definition in Show picker

cardfile:ReplaceInkData(*entry*, *layoutSym*, *oldString*, *checkPath*, *newString*) // replaces a specified ink string with a recognized string.

cardfile:RemoveLayout(*layoutSym*) // remove layout by symbol

cardfile:bcPhoneNumber(*entry*, *which*) // returns an array of legitimate phone numbers

cardfile:bcCreditCards(*entry*, *which*) // returns the credit card information

cardfile:bcEmailAddress(*entry*, *which*) // returns e-mail information

cardfile:bcEmailNetwork(*entry*, *type*) // returns e-mail network information

cardfile:bcCustomFields(*inEntry*, *inWhich*) // returns custom field information

Dates Application Methods

calendar:AddAppointment(*mtgText*, *mtgStartDate*, *mtgDuration*, *repeatPeriod*, *repeatInfo*) // Adds a meeting to Dates soup

calendar:AddEvent(*mtgText*, *mtgStartDate*, *repeatPeriod*, *repeatInfo*) //Adds an event

calendar:DeleteAppointment(*mtgTextOrFrame*, *mtgStartDate*, *deleteOneOnly*) // Deletes specified meeting(s)

calendar:DeleteRepeatingEntry(*mtgTextorFrame*, *mtgStartDate*,

```
   deleteOneOnly) // Delete specified repeating meeting or
   event series
calendar:DeleteEvent(mtgTextOrFrame, mtgStartDate, deleteOneOnly)
   // Deletes specifed events
calendar:DisplayDate(date, format) // Displays meetings,
   events, or To Do tasks for a date
calendar:FindAppointment(mtgText, findWords, dateRange, type,
   maxNumberToFind) // Returns specified meetings or
   events
calendar:FindExactlyOneAppointment(mtgText, findWords,
   dateRange, type)// Returns a specified meeting or
   event
calendar:FindNextMeeting(date) // Returns next meeting
   after a date
calendar:GetMeetingIconType(mtgTextOFrame, mtgStartDate) //
Returns the type of icon of a meeting or event
calendar:GetMeetingInvitees(mtgText, mtgStartDate) // Returns
   list of invitees
calendar:GetMeetingLocation(mtgText, mtgStartDate) // Returns
   meeting location
calendar:GetMeetingNotes(mtgText, mtgStartDate) // Returns
    notes for a meeting
calendar:GetSelectedDates() // Return currently selected
   date(s)
protoRepeatview:GetRepeatSpec()//Returns repeat information
calendar:MoveAppointment(mtgText, mtgStartDate, newStartDate,
   newDuration) // Changes date or duration of specified
   meetings or events or repeating series
calendar:MoveOnlyOneAppointment(mtgText, mtgStartDate,
newStartDate, newDuration) // Changes date or duration of
   a specified meeting or event or repeating meeting or
   event instance
```

```
calendar:OpenMeetingSlip (meetingEntry, date, openDefaultSlip)
   //Opens slip for specified meeting
calendar:RegInfoItem(symbol, frame) // Adds item to info
   picker
calendar:RegMeetingType(symbol, frame)// Adds meeting type to
   New picker
meetingtype:NewMeeting (date, parentBox) // Creates a new
   meeting
frame:OpenMeeting(meeting, date, parentbBox) // a slot of the
   frame registered with RegMeetingType
calendar:RememberedClose(view)// Closes view and reference
   to view
calendar:RememberedOpen(view) // Opens view and sets up
   closing of view with calendar
calendar:SetEntryAlarm(mtgText, mtgStartDate, minutesOrDaysBefore)
   // Sets alarm for specified meeting or event
calendar:SetMeetingIconType(mtgText, mtgStartDate, newIconType)
   // Sets icon type for specified meeting or event
calendar:SetMeetingInvitees(mtgText, mtgStartDate, invitees)
   // Sets list of invitees for specified meeting
calendar:SetMeetingLocation(mtgText, mtgStartDate, location)
   // Sets location for specified meeting
calendar:SetMeetingNotes(mtgText, mtgStartDate, notes)
   // Sets notes for specified meeting or event
calendar:SetRepeatingEntryStopDate(mtgText, mtgStartDate,
   mtgStopDate) // Sets last date for specified repeating
   meeting or event
calendar:UnRegInfoItem(symbol) // Removes item from info
   picker
calendar:UnRegMeetingType(symbol) // Removes meeting type
   from New picker
```

To Do List Methods

```
CreateToDoItem (aDate, aRichString, aReminder, aFrequency)
   // Adds a task on a date
CreateToDoItemAll (aDate, aRichString, aReminder, aFrequency,
   aPriority, aCompleted) // Adds a task with prority and
   completion on a date
GetToDoItemsForRange (aDate1, aDate2) //Returns topics for
   a range of dates
GetToDoItemsForThisDate (aDate) // Returns Topics for date
GetTaskShapes (aShapes, aTask, aYOffset, aWidth) // Returns an
   array of shapes for the task
GetToDoShapes (aDate, aYOffset, aWidth) // Returns an
   array of shapes for the task
LastVisibleTopic() // Returns the index of the last
   topic drawn in the view
NextToDoDate(aDate) // Returns the date of the next task
RemoveOldTodoItems(aNumDays) // Removes dates with past
   completion dates
```

Time Zone Methods

```
GetLocationSoupNames() // Returns names of location soups
GetLocationSoup(name) // Returns the union soup for the
   location
LocationEntryChange() // Updates the entry
LocationEntryRemoveFromSoup(entry) // Does RAM and ROM
   housekeeping
GetCityEntry(name) // Returns specified cities from the
   union soup
GetCountryEntry(name) // Returns specified countries
   from the union soup
GetDSTEntry(name) // Returns specified DST entry from
   the union soup
```

NewCity(*newCity*, *workSite*, *makeHome*) // Adds a city to the cities soup

Notes Methods

AppMaxSizeExceeded(*entry*) // Notifies user note too large
MakeTextNote (*string*, *add*) // Adds a note to paperoll
maxEntrySizeBytes (*bytes*) // Sets trigger for note size warning

Auxiliary Button Functions

RegAuxButton (*buttonSymbol*, *template*) // Add button TO registry

UnRegAuxButton (*buttonSymbol*) // Remove button from registry

GetAuxButtons (*appSymbol*) // Returns your application's buttons

app:AddAuxButton (*buttonFrame*)// message when someone Calls RegAuxButton

app:RemoveAuxButton (*buttonSymbol*) // message when someone calls UnRegAuxButton

Utility Functions

GetSysEntryData(*entry*, *path*) // Returns the value of a specifed slot
SetSysEntryData(*entry*, *path*, *value*) // Sets the value of a specifed slot
RegPrefs(*appSymbol*,*prefsTemplate*) // Adds an item to the Preferences roll in the Extras Drawer
UnRegPrefs(*appSymbol*) // Deregister an application's Preference roll items
RegFormulas(*appSymbol*,*formulasTemplate*) // Add a view to the Formulas roll in the Extras Drawer
UnRegFormulas(*appSymbol*) // Deregisters a specified Formulas roll item

```
GetUserConfig(configSym) // Returns the value of a slot in
   the system userConfiguration frame
SetUserConfig(configSym, theValue) // Sets the value of a
   slot in the system userConfiguration frame
RegUserConfigChange(callbackID,callBackFn) // Registers
   a function with the userConfiguration frame
UnRegUserConfigChange(appSymbol) // Deregisters a
   function registered by RegUserConfigChange
Extrasdrawer:SetExtrasInfo(appSymbol, newInfo) // changes the
   Extras drawer information for the application
```

Localizing Newton Applications

This chapter discusses how to support multiple languages and how to make use of locale-specific user preferences to customize your application's handling of numbers, dates and times. At the time of this book's printing, available Newton devices support ROMs based on the English, French, German and Japanese languages.

This chapter also discusses how locale settings affect the set of dictionaries used by the system for handwriting recognition. The recognition section in this chapter presumes that you already have a basic understanding of handwriting recognition issues; if you have not done so already, you should read Chapter 10, "Recognition."

About Localization

The goal of the localization functions is to let you set up your applications so you can build versions in different languages without changing the source files.

There are two basic approaches and types of localization:

- You can fully localize your application at compile time, replacing strings in English with strings in other languages. This is discussed in “Defining Language at Compile-Time” beginning on page 19-4.
- You can check preferences that the user sets to indicate preferred formats for output of dates and currency amounts. The next section discusses these user settings. “Determining Language at Runtime” on page 19-8 discusses this and related issues.

The Locale Panel and the International Frame

The user can use the Locale panel to tell the Newton device to specify the set of conventions the Newton system uses to interpret user input and display information to the conventions of a particular location. The user can specify values for the country, keyboard type, and paper size in this panel, which is shown in Figure 19-1.The system stores the settings from the Locale panel in the **International Frame**.

Figure 19-1 The Locale settings in Preferences

The most important of these settings is the Country popup menu. Every Newton device contains a number of frames that can be used to tailor the system’s responses to match the conventions of a specified location. These frames are called **locale bundles**. At any time, one and only one of these locale bundles is active, and that is called the **active locale bundle**. The user can change the active locale bundle by using the Country popup menu from the Locale panel.

The values in a locale bundle specify a variety of items, such as the set of dictionaries used for handwriting recognition, locally-popular formats for displaying currency values, and labels for commonplace entities. For example, the set of dictionaries used for handwriting recognition in the U.S. locale has an entry for the word “color,” while the corresponding entry in the set of dictionaries used for the United Kingdom locale is spelled “colour.”

Each Newton device may contain different locale bundles:

- Every Newton device contains locale bundles in its ROM, and these vary depending on what ROM the device has. For example, the German version of the Newton MessagePad does not have the same locale bundles as the English version.
- Applications can add additional locale bundles to provide locale settings for additional countries, or to override built-in locale bundles. For information on how to do this, see “Adding a New Bundle to the System” on page 19-10.

The Paper Size menu sets the default page format used for printing and faxing; this value is stored in the active locale bundle.

How Locale Affects Recognition

As the user changes settings in the Locale panel, the set of dictionaries used for word recognition also changes. In addition to changing the set of enumerated dictionaries used for recognition, changing locales changes the set of lexical dictionaries used to specify formats for dates, times and numbers.

Because the dictionaries vary in size, the amount of RAM used for system dictionaries varies as the set of dictionaries in use changes; however, this variation does not change the amount of RAM available to your application.

For more information about dictionary-based recognition, see Chapter 10, “Recognition.”

Using the Localization Features of the Newton

This section tells you how to localize your applications using the Newton's built-in features.

Defining Language at Compile-Time

You can write your application so that the language used in strings and other objects is determined at compile-time.

To do this, you use the `LocObj` function to tie objects in your code to alternative objects. The alternative objects are called **localization frames**.

The `LocObj` function depends on a project setting that tells it whether to use English—which appears in the source code—or whether to retrieve a specified localization frame. See the *Newton Toolkit User's Guide* for information on project settings.

The following two sections tell how to:

- define a localization frame
- use `LocObj`

Following those sections are two sections that discuss how to build strings from components and how to measure the lengths of strings at compile-time so that you can arrange your displays.

Defining a Localization Frame

You define the alternative language frames with the `SetLocalizationFrame` function in a text file that's included in the project, and you establish a language for a build through the Project Settings item in the Project menu.

If, for example, you display a message while searching for an object, you can set up the message for any language by wrapping the string in the `LocObj` function:

```
msg := LocObj("Searching for ^0…", 'find.searchfor)
```

To set up versions of this message in different languages, you would use the `SetLocalizationFrame` function, which establishes the alternative-language frame:

```
SetLocalizationFrame({
   Swedish: {
      find: {
         searchFor:
            "Söker efter ^0…",   // "Searching for ^0…"
      . . .}},
   French: {
      find: {
         searchFor:
            "Recherche dans ^0…",// "Searching for ^0…"
      . . .}}
});
```

When the Language setting in the Project Settings dialog box is English, NTK uses the string included in the code itself (“Searching for *name*”). When the Language setting is Swedish, NTK looks for the string contained in the slot `Swedish.find.searchFor` in the language frame.

To avoid name collisions, it’s a good idea to use at least one extra naming level (`find` in the example above). You can set up your data objects in as complex a hierarchy as you need within the language frame.

Using LocObj to Reference Localized Objects

The `LocObj` function takes two parameters:

- a string or other object; this is used in the English-language version of the application

- a frames path that the compiler uses to find the alternative object when the Language setting in the Project Settings dialog box is set to anything other than English; you should avoid having reserved words in the path—refer to Appendix A of *The NewtonScript Programming Language* for a complete list of reserved words in NewtonScript

When an English-language version of the application is compiled, the `LocObj` function simply returns its first argument; this implementation helps keep code readable by allowing you to use English strings inline. For non-English versions of the application, the `LocObj` function uses the value of the `language` slot in NTK Package Settings and the path expression passed as its second argument to return the appropriate object from the localization frame.

For example, an application that is not localized might provide user feedback by passing the string `"Not found."` to the `SetStatus` function, as in the following code fragment.

```
:SetStatus("Not found.");
```

The localized version of the same code uses the `LocObj` function to return a path expression based on the `'find.nope` argument and the language for which the application is compiled.The `SetStatus` function then uses this path expression to reference the localized string. Because the `LocObj` function returns this path expression as its result, the argument to `SetStatus` can be “wrapped” in a call to the `LocObj` function, as in the following line of code.

```
:SetStatus( LocObj("Not found", 'find.nope) );
```

The object passed to the `LocObj` method doesn’t have to be a string; it can be an array or an immediate value. However, the path expression used to reference the object must be unique across the entire system; thus, to avoid name collisions, it’s recommended that you use additional naming levels based on your application symbol; for example, `'myApp.find.nope` introduces a third naming level to the path expression.

When the application is compiled, the expression'`myApp.find.nope` evaluates to the appropriate object according to the language specified by value of the `language` slot in NTK Package Settings.

Use ParamStr Rather Than “&” and “&&” Concatenation

While it is often very convenient to use the inline ampersand string concatenators & and &&, the `ParamStr` function provides a much more flexible and powerful way to parameterize the construction of strings, which helps you customize your strings for different languages by, for example, varying word order.

For information about `ParamStr`, see its description on page 20-25.

Measuring String Widths at Compile Time

When the size of a screen element depends on the size of associated text, you can use the `MeasureString` function to determine how big to make the screen element.

You could establish the width of the search message, for example, by using `MeasureString` and `LocObj` together. You must place the calls in code that is executed at compile time. You could, for example, place in a view's template an evaluate slot with the name `holdWidth` that contains this statement:

```
MeasureString(LocObj("Searching for ^0",
                     'find.searchFor), simpleFont12)+6;
```

You could access the width at run time from the view’s `viewDrawScript` with this function:

```
func()
begin
local newBounds := deepClone(viewBounds);
newBounds.right := newBounds.left + holdWidth;
SetValue(self, 'viewBounds, newBounds);
end
```

During the build, the value of `holdWidth` is compiled into the function as a constant.

See page 19-23 for reference information on `MeasureString`.

Determining Language at Runtime

You can determine the language at runtime, and your program can use that information to modify its behavior.

There are two basic ways to determine the current language:

- You can examine the active locale bundle. You can also add new locale bundles to give the user new locale options and you can set the locale from within your program. The next sections discuss how to examine the active locale bundle
- You can use the `GetLanguageEnvironment` function to find out the native language for which the ROMs on the Newton device are implemented. See "GetLanguageEnvironment" on page 19-21 for more information.

Note that you need to determine which of these different methods your program should use to determine its behavior.

Examining the Active Locale Bundle

The global function `GetLocale` returns the active locale bundle, which is the locale bundle that the Country popup is currently set to. You should use this function rather than accessing the frame directly.

For example:

```
activeLocale:=GetLocale();
```

Once you've obtained a bundle, you can examine it to see how your program should interpret the user's input or how it should display output.

See "Contents of a Locale Bundle" on page 19-31 for details of what a locale bundle contains.

Changing Locale Settings

You cannot change settings in the active locale bundle. To change locale settings, you need to create a new locale bundle that has the values you want and make it the active locale. The best way to do this is to define in your custom locale bundle a `_proto` slot that references one of the existing locale bundles. This approach ensures that your custom locale bundle has the full complement of required slots, including those storing the lexical dictionaries used for handwriting recognition, which you cannot create.

Creating a Custom Locale Bundle

To create your application's custom locale bundle, use the `FindLocale` function to get the frame to be referenced by your custom locale bundle's `_proto` slot.

▲ **WARNING**

Your custom bundle's `_proto` slot must ultimately reference a system-supplied locale bundle. That does not have to be direct—you can proto from a custom bundle that you know protos from a system-supplied bundle, for example. ▲

Your custom locale bundle is simply a frame that includes this `_proto` reference and any additional slots that you wish to define to override the values of those inherited from the prototype. Your custom locale bundle should look like the code in the following example.

```
usLocaleBundle := :FindLocale('usa);
myLocaleBundle :=
         {
         _proto: usLocaleBundle,
         // add slots to be modified
         title: "myLocaleBundle:PIEDTS",
         localeSym: '|myLocaleBundle:PIEDTS|,
         };
```

The `FindLocale` function accepts as its argument a symbol specifying the locale bundle it is to return. This function returns the locale bundle that has this value in its `localeSym` slot.

In the preceding code example the `myLocaleBundle` frame is based on the U.S. locale bundle. It gives a new `title` and `localeSym` value, as it must, but implements no functional changes to the locale. To make changes to the locale bundle, you can add your own slots in place of the comment that says `// add slots to be modified`.

Your custom locale bundle needs to have a `localeSym` slot of its own, as well as a `title` slot that gives the string that you want to appear in the Country popup menu. There can't be another locale bundle in the system with the same symbol as your new bundle, and there should not be one with the same title, so when creating your bundle's `title` and `localeSym` values, you need to take appropriate precautions to avoid name clashes. The problems associated with creating a unique name string for a locale bundle are similar to those you might encounter when creating a name for a soup; for suggestions regarding the creation of unique name strings, see the "Naming Soups" section in Chapter 11, "Data Storage and Retrieval." Basically, you should incorporate your developer or application symbol in the title and symbol.

Adding a New Bundle to the System

Once you have created your locale bundle, you need to make it available to the system by using the `AddLocale` function. The following code sample shows how to pass the previously created locale bundle `myLocaleBundle` to this function.

```
:AddLocale(myLocaleBundle);
```

If the `localeSym` slot in `myLocaleBundle` is not unique, the new locale overrides the existing locale with the same symbol. You can override a build-in locale bundle.

Removing a Locale Bundle

You may need to remove the custom locale bundle you have installed. You can use the system-supplied `RemoveLocale` function to do so.

As with any shared data, when is an appropriate time to remove your locale bundle is left up to you, the developer, because it can be difficult to determine whether other applications are using your bundle. Even if your own application is the only one using the custom locale bundle, it can be difficult to determine whether to remove it. You wouldn't necessarily want to install it every time the application opened and remove it every time the application closed.

If you remove the active locale bundle, `RemoveLocale` makes one of the built-in locales the active locale; which locale it chooses depends on the ROM version. If your application makes a new bundle active, you may want to save the symbol of the previously active bundle, so that you can reset the value before you remove your locale bundle.

For the purposes of discussion, assume that you have determined very clear-cut circumstances under which it is appropriate to remove your custom locale bundle.

The `RemoveLocale` function accepts as its argument a symbol specifying the locale bundle it is to remove. The following code shows how to pass the locale bundle's symbol to this function.

```
:RemoveLocale('|myLocaleBundle:PIEDTS|);
```

Changing the Active Locale

The `SetLocale` function searches for a specified bundle and makes that bundle the active locale bundle. This is equivalent to the user's setting the Country value from the Country popup menu, and overrides the user's action; you should, therefore, save the previous setting and reset it when you are done using your locale.

This function accepts as its argument a symbol identifying the bundle to be installed. The following code example shows how to use the `SetLocale`

function to install the custom locale frame created in“Defining a Localization Frame” on page 19-4.

```
SetLocale('|myLocaleBundle:PIEDTS|);
```

Summary: Customizing Locale

The following code sample summarizes the information in the preceding sections.

```
// get a bundle to proto from
usLocaleBundle := :FindLocale('usa);

// define your custom locale bundle
myLocaleBundle :=
         {
         _proto: usLocaleBundle,
         // add slots to be modified here
         title: "myLocaleBundle:PIEDTS",
         localeSym: '|myLocaleBundle:PIEDTS|,
         }

// add myLocaleBundle to the system
:AddLocale(myLocaleBundle);

//save the current locale setting
previousLocale:=GetLocale().localeSym;

//install myLocaleBundle as the active locale bundle
SetLocale('|myLocaleBundle:PIEDTS|);

//reset the previous locale setting
SetLocale(previousLocale);
```

```
//remove your locale
:RemoveLocale('|myLocaleBundle:PIEDTS|);
```

Localized Output

Your application can make use of locale-specific user preferences to customize your application's handling of numbers, dates and times. In addition to using information available from the active locale bundle, your application can use the utility functions described here to display localized strings representing dates, times and monetary values.

Date and Time Values

The system provides several utility functions that return date and time values as strings formatted according to the format specification passed as one of their arguments. The `GetDateStrSpec` function is used to create the format specification passed to these functions.

A date or time format specification is an array, the elements of which are themselves two-element arrays. The first element in each two-element array is a constant specifying the item to be displayed and the second element is a constant specifying the format in which that item is displayed.

For example, the two-element array `[kElementDayOfWeek, kFormatAbbr]` specifies that the day in a date string is to be displayed in abbreviated format, such as "Wed". On the other hand, the two-element array `[kElementDayOfWeek, kFormatLong]` specifies that the day of the week is to be displayed in long format, such as "Wednesday".

The order in which date elements appear is specified by values stored in the active locale bundle. The `longDateOrder` slot stores the format specification for the order of all dates except short dates. The format specification for short dates is stored in the active locale bundle's `shortDateOrder` slot.

Times always appear in hour / minute / seconds order, although you can use format specifications to vary the display of individual elements and delimiters in the time string.

The delimiters used to separate the various elements of the date or time string are retrieved from the active locale bundle.

Formatting Date and Time Values

You can use format specifications to specify the manner in which date and time strings are displayed by the functions that accept them. The following code example uses system-supplied constants to build an array of [*element*, *format*] pairs specifying the output of a date string. The complete set of constants is listed in the section “Date and Time String Specification Constants” on page 19-29.

This array is supplied as the argument to the `GetDateStrSpec` function, which returns the format specification passed to the `LongDateStr` function. The `LongDateStr` function formats the current time (returned by the `Time` function) as specified by the format specification.

```
// at compile time, define my array of
// element and format pairs in Project Data
myArray:=
   [
      [kElementYear, kFormatNumeric],// year
      [kElementDay, kFormatNumeric],// day of month
      [kElementMonth, kFormatLong],// name of month
      [kElementDayOfWeek, kFormatLong]// day of week
   ];
// create the formatSpec
// this spec returns a string such as “February 1, 1994”
theSpec:= GetDateStringSpec(myArray);

// later on, at run time...
// initialize an evaluate slot with theSpec
mySpec := theSpec;
// get the current time
theTime:= Time();
```

```
// pass the time and the format to LongDateStr
LongDateStr(theTime,mySpec);
```

This example is deliberately verbose for purposes of illustrating how to build a format specification array. For commonly-used format specifications, the system defines formats that can be passed directly to the functions that accept format specifications. These formats are stored in the global variable `ROM_dateTimeStrSpecs`; to use one of these values, access the appropriate slot by dereferencing `ROM_dateTimeStrSpecs` with a dot operator, as in the following example.

```
// a more succinct approach
LongDateStr(Time(),ROM_datetimestrspecs.longDateStrSpec);
```

Using these predefined format specifications also saves you the trouble of having to define them at compile time and initialize slots with the compile time variables at run time. The format specifications available from the `ROM_dateTimeStrSpecs` global variable are listed in Table 19-1.

Table 19-1 Format specifications in `ROM_dateTimeStrSpecs` global

Slot	Note	Example of format
`longDateStrSpec`	1	"Wednesday, July 22, 1992"
`abbrDateStrSpec`	1	"Wed, Jul 22,1992"
`yearMonthDayStrSpec`	1	"July 22, 1992"
`yearMonthStrSpec`	1	"July 1992"
`dayStrSpec`	1	"Wed, Jul 22"
`monthDayStrSpec`	1	"July 22"
`numericDateStrSpec`	2	"7/22/92"
`numericMDStrSpec`	2	"7/22"
`numericYearStrSpec`	1,2	"1992"
`longMonthStrSpec`	1	"July"

Table 19-1 Format specifications in `ROM_dateTimeStrSpecs` global (continued)

Slot	Note	Example of format
`abbrMonthStrSpec`	1	"Jul"
`numericDayStrSpec`	1,2	"22"
`longDayOfWeekStrSpec`	1	"Wednesday"
`abbrDayOfWeekStrSpec`	1	"Wed"
`longTimeStrSpec`	3	"10:40:59 AM"
`shortTimeStrSpec`	3	"10:40 AM"
`shortestTimeStrSpec`	3	"10:40"
`hourStrSpec`	3	"10"
`minuteStrSpec`	3	"40"
`secondStrSpec`	3	"59"

[1] Argument to `LongDateStr` function
[2] Argument to `ShortDateStr` function
[3] Argument to `TimeStr` function

The kIncludeAllElements Constant

If you want to use the default format for time or date strings as specified by the active locale bundle, you can pass the `kincludeAllElements` constant

to the functions `LongDateStr`, `ShortDateStr` and `TimeStr`. You'll get the results summarized inTable 19-2.

Table 19-2 Using the `kIncludeAllElements` constant

Function	Format of output
`LongDateStr`	day of week, month, day, year in locale's default format
`ShortDateStr`	year, month, day in locale's default short date format
`TimeStr`	hour, minute, second, AM/PM, and suffix

Miscellaneous Date and Time Functions

The system supplies a number of additional functions that return date and time values from the system clock. For a complete description of these functions, see the "Function Reference" beginning on page 19-18.

Currency Values

Currency strings reflect localized formatting characteristics that distinguish them from other number strings. They typically display a prefix or suffix indicating their denomination and may additionally require one indicating whether the amount is negative. Currency strings must also adhere to regional conventions for grouping numbers, for the delimiter used to indicate these groupings, and for the character used to represent the decimal point. These values are stored in a frame in the active locale bundle's `numberFormat` slot. For descriptions of all of the values stored in this slot, see the section "Numeric Strings" on page 19-38.

For example, the `currencyPrefix` slot stores the value `$` for the U.S. locale and `£` for the United Kingdom locale. The `currencySuffix` slot stores a string that follows a monetary value in some locales; for example, in the locale `Canada (French locale)` this slot stores the value `$`.

Localization Reference

This section has reference information about localization.

Function Reference

The following functions and methods useful for implementing regional customization of application data. Some of the functions described here are used to install or remove a custom locale bundle. Others return information from the ROM and the active locale bundle. This section also describes functions that return date and time values from the system clock—some of these functions return a formatted string; others, a system value representing the date or time information. The remaining functions described here are utilities used to regionalize the format of numeric strings.

AddLocale

`:AddLocale(`*theLocaleBundle*`)`

Adds the specified frame to the available locales.

theLocaleBundle The locale bundle to be installed into the system.

Note

This function may not be defined in ROM—
it may be supplied by NTK. ◆

Date

`Date(`*time*`)`

Returns the specified time as a date frame having the slots described in Table 19-3. The formats of individual strings in the date frame are determined by values in the active locale bundle.

time The time as returned by the `Time` function

Table 19-3 Date frame slots and values

Slot name	Example value
`year`	1993
`month`	1
`date`	24
`dayofweek`	0 (Sunday=0, Saturday=6, and so on)
`hour`	15
`minute`	38
`second`	52

DateNTime

`DateNTime(`*time*`)`

Returns the specified time as a string having the format *MM*/*DD*/*YYYY HH*:*MM*; for example, `10/23/1993 12:45`. The formats used for individual elements and delimiters in the returned string are determined by values in the active locale bundle.

time The time as returned by the `Time` function

FindLocale

`:FindLocale(`*locSymbol*`)`

Returns from the available locales the frame having the specified string in its `title` slot. Use this function to get a frame to be referenced by your custom locale bundle's `_proto` slot.

locSymbol The symbol the locale bundle to be retrieved, as specified by the symbol in the bundle's `localeSym` slot

Note
This function may not be defined in ROM—it may be supplied by NTK. ◆

GetDateStringSpec

`GetDateStringSpec(`*formatArray*`)`

Returns a date or time format specification which is passed to one of the following built-in functions: `LongDateStr`, `ShortDateStr`, or `TimeStr`. Because the `GetDateStrSpec` function is available at compile time only, its return value must be stored in a compile time variable used to initialize an evaluate slot at run time. The slot value is then passed to date and time functions requiring the format spec at run time.

The order in which elements of a date or time string appear is not specified by the format specification, but by values stored in the active locale bundle. The delimiters used to separate the various elements of the date or time string are also not specified in the format spec, but retrieved from the active locale bundle.

formatArray an array of two-element arrays. Each two-element array lists a single date or time element and a corresponding format to be used to display that element. For example:

```
[[kElementMonth,kFormatAbbr],
[kElementDay, kFormatNumeric]]
```

The two-element subarrays can appear in any order; the order in which elements of the date or time string appear is defined in the active locale bundle, not by the format spec.

See the section “Date and Time String Specification Constants,” later in this chapter, for a complete listing of the values you can use for the date or time element in each subarray, and an example of each as returned by one of the built-in date or time functions.

Note

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

GetLanguageEnvironment

```
GetLanguageEnvironment()
```

Returns a value indicating the language for which the ROM in the current Newton device is implemented. These values are summarized in Table 19-4.

Table 19-4 ROM language codes

Language	Value
English	`0`
French	`1`
German	`2`
Japanese	`14`

Note
This function may not be defined in ROM—it may be supplied by NTK. ◆

GetLocale

```
GetLocale()
```

Returns the frame installed in the `currentLocaleBundle` slot of the `international` frame.

For more information, see the sections “Examining the Active Locale Bundle” on page 19-8 and “Contents of a Locale Bundle” on page 19-31.

HourMinute

`HourMinute(`*time*`)`

Returns the value of the *time* argument as a string in the format *HH*:*MM*; for example, `12:45`. The formats used for individual elements and delimiters in the returned string are determined by values in the active locale bundle

time	The time as returned by the `Time` function

LocObj

`LocObj(`*obj*`, `*pathexpr*`)`

Returns the object specified by the *obj* parameter or, if the Language setting in the Project Settings dialog box is set to something other than English, the object specified by the path name in the *pathexpr* parameter.

obj	The object. This needs to be a constant.
pathexpr	The frames path to an alternative version of the object. This also needs to be a constant.

You can reference `LocObj` from within a function that's executed at run time, because `LocObj` is evaluated at compile time and replaced with the string or other object appropriate to the language setting.

This is a compile-time function. Because `LocObj` is evaluated at compile time, its parameters must be constants or in-line values, not references to local variables that are created at run time.

LongDateStr

`LongDateStr(`*time*`,`*dateStrSpec*`)`

Returns the date as a string in the `kLongDateFormat` defined for the active locale bundle; for example `"Saturday, May 8, 1993"`.

time	The time as returned by the `Time` function
dateStrSpec	A format specification as returned by the `GetDateStringSpec` function. See the description

of the `GetDateStringSpec` function for more information on available formatting options.

MeasureString

`MeasureString(`*str*`, `*fontSpec*`)`

Measures the length of a text string in a specified font.

str	The text to be measured.
fontSpec	The font in which the text appears. You can specify the font using any of the constants or combinations of constants described in "Specifying a Font" in the chapter "Working With View Classes" in the *Newton Programmer's Guide*. If you're using your own front, you can pass a font frame.

The `MeasureString` function returns the length, in pixels, of the *str* parameter in the font specified by the *fontSpec* parameter.

This is a compile-time function. Because `MeasureString` is evaluated at compile time, its parameters must be constants or in-line values, not references to local variables that are created at run time.

See "Measuring String Widths at Compile Time" on page 19-7 for an example of using this function.

RemoveLocale

`:RemoveLocale(`*locSymbol*`)`

Removes the specified locale bundle from the available locales.

locSymbol	The symbol of the locale bundle to be retrieved, as specified by the string in the bundle's `localeSym` slot.

SetLocale

`SetLocale(`*locSymbol*`)`

Searches the system for the specified locale bundle and, if it is found, makes it the active locale bundle.

locSymbol The symbol of the locale bundle that you want to be the active locale, as specified by the string in the bundle's `localeSym` slot.

Note

This function may not be defined in ROM—it may be supplied by NTK. ◆

SetLocalizationFrame

`SetLocalizationFrame(`*frame*`)`

Establishes the language frame to be used by `LocObj` when the Language setting for a build is anything other than English.

frame The language frame, that is, the hierarchy of objects that maps to the path names used by the `LocObj` function. At its first level, the language frame contains one or more slots, whose names are the Language codes that can be specified through the Project Settings dialog box. Each language slot contains all objects established for that language.

If you call `SetLocalizationFrame` more than once, the most recent language frame replaces the previous language frame.

This is a compile-time function. Because `SetLocalizationFrame` is evaluated at compile time, its parameters must be constants or in-line values, not references to local variables that are created at run time.

SetLocation

worldClock:SetLocation(*newLocation*)

Sets the “I'm Here” location in Time Zones. This does not change the active locale setting.

newLocation A frame specifying a name, latitude, longitude, country, and area code; country and area code are optional.

SetTime

SetTime(*time*)

Sets the time of the system clock. This function always returns NIL.

time The time to which the system clock is to be set, specified as the number of minutes elapsed since midnight, January 1, 1904

ShortDate

ShortDate(*time*)

Returns the date as a string in the format *Day MM*/*DD*; for example, "Fri 12/25". The formats used for individual elements and delimiters in the returned string are determined by values in the active locale bundle.

time The time as returned by the Time function

ShortDateStr

ShortDateStr(*time*, *dateStrSpec*)

Returns the date as a string in the format specified in dateStrSpec; for example, "5/8/93".

time The time as returned by the Time function

dateStrSpec A format specification as returned by the GetDateStringSpec function. See the description

of the `GetDateStringSpec` function for more information on available formatting options.

StringToDate

`StringToDate(`*dateString*`)`

Parses a string for date or time information and returns the result as the number of minutes passed since midnight, January 1, 1904. The formats used for individual elements and delimiters in the input string are determined by values in the active locale bundle.

dateString The string to be parsed. If the year is omitted from the string, the current year is assumed. The following types of date/time strings can be parsed (the case of letters is not significant):

```
"12:05 a.m. sun, jan 2, 1992"
"jan 2, 1992"
"12:05 1/2/92"
"1/2/92"
"12:05mon,1/2"
"1/2"
```

To get a date frame from a string use the following line of code. (For more information about date frames, see“Date” on page 19-18.)

```
Date(StringToDate(dateString))
```

StringToDateFrame

`StringToDateFrame(`*str*`)`

Returns the input string as a date frame.

This function is similar to `StringToDate`, but has two differences:

- The `StringToDateFrame` function returns a date frame instead of the number of minutes since 1/1/1904. For example, the

`StringToDateFrame` function returns the following frame when passed the string "June 2" as its argument.

```
{
year: nil,
month: 6,
day: 2,
hour: nil,
minute: nil,
second: nil,
status: 0       // for internal use
}
```

- The `StringToDateFrame` function does not supply date or time elements missing from the input string. In the previous example, the `year`, `hour`, `minute`, and `second` slots are set to nil because the input string does not include these values. This behavior can be useful for determining what's really in the input string. On the other hand, the `StringToDate` function fills in the missing elements with the current date and time.

  Therefore one must be aware that

  `StringToDateFrame(`*str*`)`

  and

  `Date(StringToDate(`*str*`))`

  do not return the same result.

StringToTime

`StringToTime(`*timeString*`)`

This function is just like `StringToDate`, except that it parses a string for time information only, ignoring any date information in the *timeString* string, and returns the result as the number of minutes passed since midnight,

January 1, 1904. The formats used for individual elements and delimiters in the input string are determined by values in the active locale bundle.

timeString The string to be parsed for time information only; any date information in this string is ignored.

Ticks

`Ticks()`

Returns a number of ticks. A tick is one sixtieth of a second. There is no defined starting time for ticks; they are used to measure durations of time. So you would typically call `Ticks`, do something, or wait, and then call Ticks again and compare the values to see how much time has passed.

Time

`Time()`

Returns the time in minutes as an integer. This is the number of minutes passed since midnight, January 1, 1904.

TimeInSeconds

`TimeInSeconds()`

Returns the time in seconds as an integer. This is the number of seconds passed since midnight, January 1, 1993.

TimeStr

`TimeStr(`*time*`,` *timeStrSpec*`)`

Returns the specified time as a string in the format *HH*`:`*MM*`:`*SS*`[AM|PM]`; for example, `"10:40:59 AM"`.

time The time as returned by the `Time` function

timeStrSpec A format specification as returned by the `GetDateStringSpec` function. See the description of

the `GetDateStringSpec` function for more information on available formatting options.

TotalMinutes

`TotalMinutes(`*dateFrame*`)`

Returns the time in minutes since midnight, January 1, 1904, when passed a date frame have the same slots as described in the Date() function. (The dayofweek slot is actually not used.) You must pass in a date frame, or this function returns an error.

Date and Time String Specification Constants

This section contains several tables listing the system-supplied constants available for specifying the elements and formats of date and time strings.

The system-supplied constants for specifying the elements of date strings are described in Table 19-5.

Table 19-5 Elements of date strings

Constant	Element of string	Example values
`kElementNothing`	no element	*displays no value*
`kElementDayOfWeek`	day of the week	“Wednesday, Weds, W”
`kElementDay`	date	“22, 02, 2”
`kElementMonth`	month	“July, Jul, J”
`kElementYear`	year	“1992”

The system-supplied constants for specifying the elements of time strings are described inTable 19-6.

Table 19-6 Elements of time strings

Constant	Element of string	Example values
`kElementHour`	hour	12
`kElementMinute`	minute	33
`kElementSecond`	second	59
`kElementAMPM`	AM / PM	“AM”, “PM”
`kElementSuffix`	24-hour clock indicator	“GMT”

The system-supplied constants for specifying the format of date string elements are described inTable 19-7.

Table 19-7 Formats for date or time string elements

Constant	Element of string	Example of display
`kFormatLong`	full-length	“Wednesday”
`kFormatAbbr`	abbreviated	“Wed”
`kFormatTerse`	shortened abbreviation	“We”
`kFormatShort`	single-letter	“W”
`kFormatNumeric`	numeral	“1994”

The kIncludeEverything Constant

If you want to use the default format for time or date strings as specified by the active locale bundle, you can pass the `kIncludeEverything` constant

to the functions `LongDateStr`, `ShortDateStr` and `TimeStr`. You'll get the results summarized inTable 19-2.

Table 19-8 Using the `kIncludeEverything` constant

Function	Format of output
`LongDateStr`	year, month, day, day of week in locale's default format
`ShortDateStr`	year, month, day in locale's default short date format
`TimeStr`	hour, minute, second, AM/PM, and suffix

Contents of a Locale Bundle

This section shows the slots of a locale bundle that you can access. Slots that are not described here are for internal use, and you should not change their values or write your program to depend on the contents or existence of those slots.

String Slots

This section describes slots in the locale bundle that are used to display locale-specific strings. The strings stored in these slots vary according to the locale.

Your application can reference these slots directly to use the strings stored here as labels for text or fields in your application. You should never set the values of these slots directly except in your own custom locale bundle.

`postalCodeLabel`

A string used to label postal codes. For example, the built-in Names application uses this slot to display the

string "Zip Code" in the U.S. locale and the string "Post Code" in the Australia locale.

`longOrdinals` An array of strings intended to label items ordinally; for example, "First", "Second" and so on. This is not currently used by the system.

`shortOrdinals` An array of short strings intended to label items ordinally; for example, "1st", "2nd" and so on. This is not currently used by the system.

`distanceLabel` The string used to label distances. For example, this value is "miles" in the U.S. locale and "kilometers" in the Canada locale.

`distanceMeasure`
The unit of measure used for distances. For example, this value is `miles` in the U.S. locale and `kilometers` in the Canada locale. This value is used by the built-in World Clock application to display the distance between two cities in a format appropriate to the user's locale; your application can use it also.

`distanceLabelShort`
The short version of the string used to label distances. For example, this value is "km" as opposed to the corresponding `distanceLabelLong` value of "kilometers".

`title` A string that identifies this bundle in the Country popup menu in the Locale preference panel. You can also pass this string to locale functions such as `SetLocale`, though that may change in the future, so you should use the locale symbol to identify locale bundles.

`regionLabel` The string used to label regions. For example, this value is "State" in the U.S. locale and "Province" in the Canada locale.

`cityLabel` The string used to label cities. For example, this value is "City" in the U.S. locale and "Town" in the United Kingdom locale.

Date Strings

These slots contain strings used by the system in the graphical representation of various date values. The strings stored in these slots vary according to the locale.

Rather than using these values directly, you generally would use the appropriate `dateTimeStrSpec` to format the output of functions that return date information. However, your application can reference these slots directly to use the strings stored here as labels for text or fields in your application. You should never set the values of these slots directly except in your own custom locale bundle.

`longDateFormat` The frame containing strings for the graphical representation of the elements of the long (verbose) date format.This frame contains the slots listed immediately following.

`versionNumber`
For internal use only.

`longDofWeek`
An array of strings representing full-text names of the days of the week; for example, `"Sunday"`, `"Monday"`, `"Tuesday"` and so on.

`abbrDofWeek`
An array of strings representing abbreviated names of the days of the week; for example, `"Sun"`, `"Mon"`, `"Tue"` and so on.

`terseDofWeek`
An array of strings representing shorter abbreviations of names of days than are specified in the `abbrDofWeek` slot; for example, `"Su"`, `"Mo"`, `"Tu"` and so on.

`shortDofWeek`
An array of strings representing single-letter abbreviations of names of the days of the week; for example, `"S"`, `"M"`, `"T"` and so on.

`longMonth` An array of strings representing full-text names of the months of the year; for example, `"January"`, `"February"`, `"March"` and so on.

`abbrMonth` An array of strings representing abbreviated names of the months of the year; for example, `"Jan"`, `"Feb"`, `"Mar"` and so on.

`longDateOrder`
A `dateTimeStrSpec` value describing the order in which the elements of a long date are to appear; for example, month / day / year (`"January 31, 1994"`) as opposed to day / month / year (`"31 January 1994"`).

`dayLeadingZ`
For internal use only.

`longDateDelim`
An array of strings used to represent the character separating the elements in the graphical representation of a long date string; for example, the string `", "` (comma-space) used in the long date string `"January 31, 1994"`.

The system automatically selects a delimiter from this array according to the relative positions of the string elements to be separated. The 0th array element precedes the date string; the first element specifies the delimiter to be placed between the first and second elements in the date string; the second element in the array specifies the delimiter used to separate the second and third elements in the date string; and so on.

`shortDateFormat`

The frame containing strings for the graphical representation of the elements of all date strings other than those in the `longDateFormat` format.This frame contains the slots listed immediately following.

`versionNumber`

For internal use only.

`shortDateOrder`

A `dateTimeStrSpec` value describing the order in which the elements of a short date are to appear; for example, day / month / year (`"31/1/94"`) as opposed to month / day / year (`"1/31/94"`).

`shortDateDelim`

An array of strings used to represent the character separating the elements in the graphical representation of a short date string; for example, the string `"/"` (forward slash) used in the short date string `"1/31/94"`.

The system automatically selects a delimiter from this array according to the relative positions of the string elements to be separated. The 0th array element precedes the date string; the first element specifies the delimiter to be placed between the first and second elements in the date string; the second element in the array specifies the delimiter used to separate the second and third elements in the date string; and so on.

`dayLeadingZ`

A value of `kLeadZero` specifies that a leading zero is to be prefixed to representations of single-digit day values in short dates; for example, the `0` in `11/01/94`. A value of `kNoLeadZero` specifies suppresses the use of the leading zero prefix.

`monthLeadingZ`
A value of `kLeadZero` specifies that a leading zero is to be prefixed to representations of single-digit month values in short dates; for example, the `0` in `01/11/94`. A value of `kNoLeadZero` suppresses the use of the leading zero prefix.

`yearLeadingZ`
A value of `kLeadZero` specifies that a leading zero is to be prefixed to representations of single-digit year values in short dates; for example, the `0` in `12/12/04`. A value of `kNoLeadZero` suppresses the use of the leading zero prefix.

Time Strings

These slots contain strings used by the system in the graphical representation of various time values. The strings stored in these slots vary according to the locale.

Rather than using these values directly, you generally would use the appropriate `dateTimeStrSpec` to format the output of functions that return time information.

You should never set the values of these slots directly except in your own custom locale bundle.

`timeFormat` The frame containing strings for the graphical representation of the elements of time formats. This frame contains the slots listed immediately following.

`versionNumber`
For internal use only

`timeSepStr1`
The string used to represent the character separating the first and second elements in the graphical representation of a time

string; for example, the string "." (period) used in the time string "23.59:59".

timeSepStr2
: The string used to represent the character separating the second and third elements in the graphical representation of a time string; for example, the string ":" (colon) used in the time string "23.59:59".

morningStr
: The string used to annotate times from midnight to just before noon for a 12-hour clock cycle; for example, the string " am" (space-am) used in the time string 8:00 am.

eveningStr
: The string used to annotate times from noon to just before midnight for a 12-hour clock cycle; for example, the string " pm" (space-pm) used in the time string 8:00 pm.

suffixStr
: The string used as a suffix for graphical representations of times for a 24-hour clock cycle. This slot is not currently used by the MessagePad; however, it might contain, for example, the string " GMT" to indicate the use of Greenwich Mean Time.

hourLeadingZ
: A value of kLeadZero specifies that a leading zero is to be prefixed to representations of single-digit hour values. A value of kNoLeadZero suppresses the use of the leading zero prefix.

minuteLeadingZ
: A value of kLeadZero specifies that a zero is to be prefixed to representations of single-digit minute values. A value of

kNoLeadZero suppresses the use of the leading zero prefix.

secondLeadingZ
: A value of kLeadZero specifies that a zero is to be prefixed to representations of single-digit second values. A value of kNoLeadZero suppresses the use of the leading zero prefix.

timeCycle
: A value of kCycle12 specifies the use of a twelve-hour clock cycle; the value kCycle24 specifies the use of a 24-hour clock cycle.

midNightForm
: The value kUseHour0 specifies the representation of midnight as the numeric string "00:00".The value kUseHour12 specifies the representation of midnight as the numeric string "12:00". The value kUseHour24 specifies the representation of midnight as the numeric string "24:00".

noonForm
: The only valid value is kUseHour12, which specifies the representation of noon as the numeric string "12:00"

Numeric Strings

These slots contain strings used by the system in the graphical representation of various numeric values. The strings stored in these slots vary according to the locale.

Rather than using these values directly, use the appropriate dateTimeStrSpec to format the output of functions that return numeric data.

You should never set the values of these slots directly except in your own custom locale bundle.

`numberformat` The frame containing strings for the graphical representation of the elements of numeric formats. This frame contains the slots listed immediately following.

`versionNumber`
For internal use only

`decimalpoint`
The string used to represent the decimal character in the graphical representation of a numeric string; for example, the string `"."` (period) used in the currency string `"$123.45"`.

`groupSepStr`
The string used to represent the character separating the groupings of numbers in the graphical representation of a numeric string; for example, the string `","` (comma) used in the numeric string `"1,234"`.

`groupWidth`
The number of characters in each grouping of numeric characters separated by the `groupSepStr` character; for example, the string `"123,456,789"` is separated in groups of three.

`minusPrefix`
The string used as a prefix for graphical representations of negative numbers; for example, the minus sign (`-`) in the string `"-123"`.

`minusSuffix`
The string used as a suffix for graphical representations of negative numbers; for example, the minus sign (`-`) in the string `"123-"`.

`currencyPrefix`
: The string used as a prefix for graphical representations of currency values; for example, the dollar sign ($) in the string `"$123"`.

`currencySuffix`
: The string used as a suffix for graphical representations of currency values; for example, the dollar sign ($) in the string `"123$"`, as used for the locale Canada (French locale).

`decimalLeadingZ`
: A value of `kLeadZero` specifies that a zero is to be prefixed to representations of single-digit decimal values; for example, the `0` in the string `"$0.12"`. A value of `kNoLeadZero` suppresses the use of the leading zero prefix.

Other Slots in Locale Bundles

These slots are also contained in the locale bundles.

`_proto`	The locale bundle from which this frame inherits default attributes. Not all built-in locale frames have this slot, but all locale bundles that you define must have this slot.
`localeSym`	A symbol that uniquely identifies the locale. You must have this slot in any locale bundle that you define. You use this symbol to identify this locale bundle in calls to locale functions such as `SetLocale`.
`firstDayofWeek`	This integer value, ranging from 0 to 6, specifies the starting day of the week. 0 represents Sunday, 1

represents Monday, and so on. This information is used by, for example, clMonthView.

`postalCodeNumeric`
Has the value `TRUE` if postal code field accepts numeric input only; for example, in the U.S. locale, postal codes do not include alphabetic characters.

`versionNumber` For internal use

`wordBreakTable` For internal use; the word-selection table used to find word boundaries when selecting words.

`lineBreakTable` For internal use; the word-selection table used to find word boundaries when breaking lines of text.

`defaultPaperSize`
Specifies the default paper size to be used when formatting pages for printing and faxing; valid values are `'eightByEleven` and `'a4`.

`keycodeMapping` Specifies the Macintosh-style `'kchr'` resource used to map keys on the floating keyboard to appropriate keycodes when the application is compiled by NTK.

Summary of Localization Functions

This section categorizes the date, time, locale and utility functions in this chapter according to task.

Compile-Time Functions

These functions allows you to build your application for various language environments. For more details, see the *Newton ToolKit User's Guide* for version 1.5 of NTK.

`LocObj(`*obj*`, `*pathexpr*`)`
`MeasureString(`*str*`, `*fontSpec*`)`

Locale Functions

These functions manipulate locale bundles.

`:AddLocale(`*theLocaleBundle*`)`
`:FindLocale(`*locSymbol*`)`
`GetLocale()`
`:RemoveLocale(`*locSymbol*`)`
`SetLocale(`*locSymbol*`)`

Date and Time Functions

These functions return date or time information from the system clock. They are grouped into two categories: those that return formatted strings and those that do not.

Formatted Date/Time Functions

These functions return formatted date or time strings. Some of the functions here format the string according to a format specification supplied as one of their arguments; others format the string according to values stored in the active locale bundle. See the descriptions of individual functions for more information.

`Date(`*time*`)`
`DateNTime(`*time*`)`
`HourMinute(`*time*`)`
`LongDateString(`*time*`,`*dateStrSpec*`)`
`ShortDate(`*time*`)`
`ShortDateStr(`*time*`,`*dateStrSpec*`)`
`StringToDate(`*dateString*`)`
`StringToTime(`*timeString*`)`
`TimeStr(`*time*`,`*timeStrSpec*`)`

Miscellaneous Date/Time Functions

These functions get and set system clock values. The date and time values are returned as system clock values, not as strings.

`SetTime(`*time*`)`
`Ticks()`
`Time()`
`TimeInSeconds()`
`TotalMinutes(`*dateFrame*`)`

Utility Functions

These functions perform tasks related to the presentation of data in regionalized formats.

`GetDateStringSpec(`*formatArray*`)`
`GetLanguageEnvironment()`
worldClock`:SetLocation(`*newLocation*`)`

CHAPTER 20

Utility Functions

This chapter describes a number of utility functions. The following groups of functions are included here:

- Object system
- String
- Bitwise
- Array and sorted array
- Math
- Floating point math
- Control of floating point math
- Financial
- Exception handling
- Message sending and deferred message sending
- Data extraction
- Data stuffing
- Getting and Setting Global Variables
- Miscellaneous

Note
The inspector examples used throughout this document often include a number after a pound sign; for example, #4945. This information can be ignored as it is an internal pointer to data in the system. ◆

Compatibility

This section describes the changes made to the utility functions for Newton System Software 2.0.

New Functions

The following new functions have been added for this release.

New Object System Functions

The following new object system functions have been added.

`GetFunctionArgCount`
`IsCharacter`
`IsFunction`
`IsInteger`
`IsNumber`
`IsReadOnly` (existed in 1.0 but now documented)
`IsReal`
`IsString`
`IsSubclass` (existed in 1.0 but now documented)
`IsSymbol`
`MakeBinary`
`SetVariable`
`SymbolCompareLex`

New String Functions

The following new string functions have been added.

`CharPos`
`DurationStr`
`StrExactCompare`
`StrFilled` (existed in 1.0 but now documented)
`StrTokenize`
`StyledStrTruncate`
`SubstituteChars`

New Array Functions

The following new array functions have been added.

`ArrayInsert`
`InsertionSort`
`LFetch`
`LSearch`
`NewWeakArray`
`StableSort`

New Sorted Array Functions

The following new functions have been added that operate on sorted arrays. These functions are based on binary search algorithms, hence the “B” prefix to the function names.

`BDelete`
`BDifference`
`BFetch`
`BFetchRight`
`BFind`
`BFindRight`
`BInsert`
`BInsertRight`

```
BIntersect
BMerge
BSearchLeft
BSearchRight
```

New Integer Math Functions

The following new functions related to integer math have been added.

```
GetRandomState
SetRandomState
```

New Financial Functions

The following new functions that perform operations related to the currency exchange rate have been added.

```
GetExchangeRate
SetExchangeRate
GetUpdatedExchangeRates
```

New Exception Handling Functions

The following new exception handling function has been added.

```
RethrowWithUserMessage
```

New Message Sending Functions

The following new utility functions for sending immediate messages have been added.

```
IsHalting
PerformIfDefined
ProtoPerform
ProtoPerformIfDefined
```

New Deferred Message Sending Functions

The following new utility functions for delayed and deferred actions have been added.

```
AddDeferredCall
AddDelayedCall
AddProcrastinatedCall
AddDeferredSend
AddDelayedSend
AddProcrastinatedSend
```

These new functions replace `AddDelayedAction` and `AddDeferredAction` (although they both remain in the ROM for compatibility with existing applications). These two older functions have several problems, and you should not use them—they will likely be removed in future versions of system software.

New Data Stuffing Functions

The following new data stuffing functions have been added.

```
StuffCString
StuffPString
```

New Functions to Get and Set Globals

The following new functions that get, set, and check for the existence of global variables and functions have been added.

```
GetGlobalFn
GetGlobalVar
GlobalFnExists
GlobalVarExists
DefGlobalFn
DefGlobalVar
UnDefGlobalFn
UnDefGlobalVar
```

New Miscellaneous Functions

This following miscellaneous functions have been added.

```
AddMemoryItem
AddMemoryItemUnique
BinEqual
Gestalt
GetAppName
GetAppPrefs
GetMemoryItems
GetMemorySlot
MakePhone
MakeDisplayPhone
ParsePhone
Translate
```

Enhanced Functions

The following string function has been enhanced in Newton 2.0.

`ParamStr` has been enhanced to support conditional substitution.

Obsolete Functions

Some utility functions previously documented in the *Newton Programmer's Guide* are obsolete, but are still supported for compatibility with older applications. Do not use the following utility functions, as they may not be supported in future system software versions:

`AddDeferredAction` (use `AddDeferredCall` instead)
`AddDelayedAction` (use `AddDelayedCall` instead)
`AddPowerOffHandler` (use `RegPowerOff` instead)
`ArrayPos` (use `LSearch` instead)
`FormattedNumberStr`
`GetGlobals` use (`GetGlobalVar` or `GetGlobalFn` instead)

`RemovePowerOffHandler` (use `UnRegPowerOff` instead)
`SmartStart` (use other string manipulation functions)
`SmartConcat` (use other string manipulation functions)
`SmartStop` (use other string manipulation functions)
`StrTruncate` (use `StyledStrTruncate` instead)
`StrWidth` (use `StrFontWidth` instead)

Object System Functions

The functions described in this section operate on NewtonScript objects. They perform operations such as getting and checking for slots, removing slots, cloning frames, and so forth.

Four of the functions described in this section are designed to clone, or copy, objects. These functions each act slightly differently. Table 20-1 summarizes their actions. The “Recurs” column indicates if references within the object are copied. The “Follows Magic Pointers” column indicates if objects referenced through magic pointers are copied. The “Ensures Object is Internal” column indicates if the function ensures that all parts of the object exist in internal RAM or ROM. The “Copies Object” column indicates if the object is copied.

Table 20-1 Summary of Copying Functions

Function Name	Recurs	Follows Magic Pointers	Ensures Object is Internal	Copies Object
`Clone`	—	—	—	yes
`DeepClone`	yes	yes	—	yes
`EnsureInternal`	yes	—	yes	as needed
`TotalClone`	yes	—	yes	yes

ClassOf

`ClassOf(`*object*`)`

Returns the class of an object.

object The object whose class to return.

The return value is a symbol. Some of the common object classes are: `'int`, `'char`, `'boolean`, `'string`, `'array`, `'frame`, `'function`, and `'symbol`. Note that this is not necessarily the same as the primitive class of an object. For binary, array, and frame objects, the class can be set differently from the primitive class.

Frames or arrays without an explicitly assigned class are of the primitive class `'frame` or `'array`, respectively. If a frame has a class slot, the value of the class slot will be returned. Here are some examples:

```
f:={multiply:func(x,y) x*y};
classof(f);
#1294    Frame

f:={multiply:func(x,y) x*y, class:'Arithmetic};
classof(f);
#1294     Arithmetic

s:="India Joze";
classof(s);
#1237    String
```

See also “PrimClassOf” on page 20-16.

Clone

`Clone(`*object*`)`

Makes and returns a “shallow” copy of an object; that is, references within the object are copied, but the data pointed to by the references is not.

object The object to copy.

Here is an example:

```
SeaFrame :=  {Ocean: "Pacific", Size: "large" , Color: "blue"};
seaFrameCopy := clone(seaFrame);
seaFrameCopy.Deep := true;
seaFrame
#441896D  {Ocean: "Pacific", size: "large", Color: "blue"}
seaFrameCopy
#4418B0D  {Ocean: "Pacific", size: "large", Color: "blue",
   Deep: TRUE}
```

See Table 20-1 on page 20-7 for a comparison with other object copying functions.

DeepClone

`DeepClone(`*object*`)`

Makes and returns a “deep” copy of an object; that is, all of the data referenced within the object is copied, including that referenced by magic pointers (pointers to ROM objects).

object The object to copy.

It is not guaranteed that every part of the data structure is in RAM. (Certain information, such as the symbols naming frame slots, may be shared with the original object.)

Contrast this function with `Clone` that only makes a “shallow” copy, and the functions `TotalClone` and `EnsureInternal` that ensure that the object exists entirely in internal RAM. See Table 20-1 on page 20-7 for a comparison with other object copying functions.

EnsureInternal

`EnsureInternal(`*obj*`)`

Ensures that the object exists entirely in internal RAM or ROM. This function may copy all, some, or none of the object to ensure that it exists in RAM.

Note that magic pointers are not followed; that is, objects referenced through magic pointers are not copied.

obj The object to ensure exists in internal RAM.

This function returns an object, which may or may not be a copy of the original object.

See Table 20-1 on page 20-7 for a comparison with other object copying functions.

GetFunctionArgCount

`GetFunctionArgCount(`*function*`)`

Returns the number of arguments expected by a function.

function The function whose number of arguments you want to get.

GetSlot

`GetSlot(`*frame*`, `*slotSymbol*`)`

Returns the value of a slot in a frame. Only the frame specified is searched.

frame A reference to the frame in which to look for the slot.

slotSymbol A symbol naming the slot whose value you want to get.

If the slot doesn't exist, this function returns `nil`.

Unlike `GetVariable`, `GetSlot` searches for a slot only in the indicated frame. Inheritance is not used to find the slot.

The use of the NewtonScript dot operator is similar to the `GetSlot` function in that it also returns the value of a frame slot. For example, the expression `frame.slot` returns the value of the specified slot. However, when using the dot operator, if the slot is not found in the specified frame, proto frames are also searched for the slot (but not parent frames).

GetVariable

`GetVariable(`*frame*`, `*slotSymbol*`)`

Returns the value of a slot in a frame. If the slot is not found, `nil` is returned.

frame	A reference to the frame in which to begin the search for the slot.
slotSymbol	A symbol naming the slot whose value you want to get.

This function begins its search for the slot in the specified frame and makes use of the full proto and parent inheritance.

HasSlot

`HasSlot(`*frame*`, `*slotSymbol*`)`

Returns non-`nil` if the slot exists in the frame, otherwise, returns `nil`. Inheritance is not used to find the slot.

frame	The name of the frame in which to look for the slot.
slotSymbol	A symbol naming the slot whose existence you want to check.

HasVariable

`HasVariable(`*frame*`, `*slotSymbol*`)`

Returns non-`nil` if the slot exists in the frame, otherwise, returns `nil`. This function searches proto and parent frames of the specified frame if the slot is not found there.

frame	The name of the frame in which to begin the search for the slot.
slotSymbol	A symbol naming the slot whose existence you want to check. You must use a single quote before the slot name because it is a symbol.

Intern

`Intern(` *string* `)`

Creates and returns a symbol whose name is given as the string parameter *string*. If a symbol with that name already exists, the preexisting symbol is returned.

string The name of the symbol.

IsArray

`IsArray(`*obj*`)`

Returns non-`nil` if *obj* is an array.

obj The object to test.

IsBinary

`IsBinary(`*obj*`)`

Returns non-`nil` if *obj* is a binary object.

obj The object to test.

IsCharacter

`IsCharacter(`*obj*`)`

Returns non-`nil` if *obj* is a character, and returns `nil` otherwise.

obj The object to test.

IsFrame

`IsFrame(`*obj*`)`

Returns non-`nil` if *obj* is a frame.

obj The object to test.

IsFunction

`IsFunction(`*obj*`)`

Returns non-`nil` if *obj* is a function, and returns `nil` otherwise.

obj The object to test.

IsImmediate

`IsImmediate(`*obj*`)`

Returns non-`nil` if *obj* is an immediate.

obj The object to test.

IsInstance

`IsInstance(`*obj*`, `*class*`)`

Returns non-`nil` if *obj*'s class symbol the same as *class* or a subclass of *class*.

obj The object to test.

class A symbol specifying the class.

Note that this is equivalent to:

`IsSubclass(ClassOf(`*obj*`), `*class*`)`

IsInteger

`IsInteger(`*obj*`)`

Returns non-`nil` if *obj* is an integer, and returns `nil` otherwise.

obj The object to test.

IsNumber

`IsNumber(`*obj*`)`

Returns non-`nil` if *obj* is a number (integer or real), and returns `nil` otherwise.

obj The object to test.

IsReadOnly

`IsReadOnly(`*obj*`)`

Returns non-`nil` if *obj* is read-only, and returns `nil` otherwise. You can use `IsReadOnly` to determine if an array, frame, or binary object is writable.

obj An array, frame, or binary object to test. (Immediate objects such as integers are never read-only.)

Here is an example:

```
if IsReadOnly(viewBounds) then
   viewBounds := Clone(viewBounds);
```

This function should not be used to determine the location of an object, that is, whether it is in the heap, in ROM, or in protected memory. The NewtonScript language could permit read-only objects in the NewtonScript heap, or writable objects that exist in other locations.

IsReal

`IsReal(`*obj*`)`

Returns non-`nil` if *obj* is a real number, and returns `nil` otherwise.

obj The object to test.

IsString

`IsString(`*obj*`)`

Returns non-`nil` if *obj* is a string, and returns `nil` otherwise.

obj The object to test.

IsSubclass

`IsSubclass(`*sub*, *super*`)`

Checks if a class is a subclass of another class.

sub A class symbol you want to test.

super A class symbol.

This function returns non-`nil` if *sub* is a subclass of *super*, or is the same as *super*. Returns `nil` if *sub* is not a subclass of *super*. See also the related function `IsInstance` on page 20-13.

IsSymbol

`IsSymbol(`*obj*`)`

Returns non-`nil` if *obj* is a symbol, and returns `nil` otherwise.

obj	The object to test.

MakeBinary

`MakeBinary(`*length*`, `*class*`)`

Allocates a new binary object of the specified *length* and *class*.

length	The size of the binary object in bytes.
class	A symbol specifying the class

Map

`Map(`*obj*`, `*function*`)`

Applies a function to the slot name and value of each element of an array or frame.

obj	An array or frame.
function	Returns `nil`. A function to apply to the elements or slots in *obj*. The function is passed two parameters: *slot* and *value*. The *slot* parameter contains an integer array index if *obj* is an array, or it contains a symbol naming a slot, if *obj* is a frame. The *value* parameter contains the value of the array or frame slot referenced by the *slot* parameter.

This is equivalent to:

```
foreach slot,value in obj do call function with (slot,value)
```

PrimClassOf

`PrimClassOf(`*obj*`)`

Returns the primitive class of an object.

obj The object whose primitive class to return.

Returns a symbol identifying the primitive data structure type of the object, one of: `'immediate`, `'binary`, `'array`, or `'frame`.

See also “ClassOf” on page 20-8.

RemoveSlot

`RemoveSlot(`*obj*`, `*slot*`)`

Removes a slot from a frame or array.

obj The name of the frame or array from which to remove the slot.

slot A symbol naming the frame slot you want to remove, or the index of the array slot to remove. Note that no inheritance lookup is used to find this slot in *obj*.

This function returns the modified frame or array. If *slot* is not found, nothing is done and the unmodified frame or array is returned. Note that the system throws an exception if *obj* is read-only.

ReplaceObject

`ReplaceObject(`*originalObject*`, `*targetObject*`)`

Causes all references to an object to be redirected to another object.

originalObject The original object.

targetObject The object to which you want to redirect references to *originalObject*.

This function always returns `nil`.

Note that you cannot specify immediate objects as parameters to this function.

Here is an example:

```
x:={name:"Star"};
y:={name:"Moon"};
replaceobject(x,y);
x;
#469E69   {name: "Moon"}

y;
#46A1E9   {name: "Moon"}
```

SetClass

`SetClass(`*obj*`, `*classSymbol*`)`

Sets the class of an object.

obj	The object whose class to set.
classSymbol	A symbol naming the class to give to the object.

This function returns the object whose class was set.

You can set the class of the following kinds of objects: frames, arrays, and binary objects. Note that you cannot set the class of an immediate object.

When setting the class of a frame, if a `class` slot doesn't exist, one is created in the frame. For example:

```
x:={name: "Star"};
setclass(x, 'someClass);
#46ACC9   {name: "Star",
           class: someClass}
```

SetVariable

`SetVariable(`*frame*`, `*slotSymbol*`, `*value*`)`

Sets the value of a slot in a frame. The value is returned.

frame	A reference to the frame in which to begin the search for the slot.
slotSymbol	A symbol naming the slot whose value you want to set. If the slot is not found, it is created in *frame*.
value	The new value of the slot.

This function begins its search for the slot in the specified frame and makes use of the full proto and parent inheritance.

Note that if the slot is found in the proto chain, it is not set there, but is created and set in *frame*, or in its parent chain, following the usual inheritance rules as they apply to setting a value.

SymbolCompareLex

`SymbolCompareLex(`*symbol1*`, `*symbol2*`)`

Compares symbols lexically. This function returns a negative number if symbol *symbol1* is less than symbol *symbol2*. Returns zero if the two symbols are equal. Returns a positive number if *symbol1* is greater than *symbol2*. Case is not significant (that is, `'Hello` and `'hello` are equal).

symbol1	A symbol.
symbol2	A symbol.

TotalClone

`TotalClone(`*obj*`)`

Makes and returns a “deep” copy of an object; that is, all of the data referenced within the object is copied.

obj	The object to copy.

This function is similar to `DeepClone`, except that this function guarantees that the object returned exists entirely in internal RAM. Also, unlike

`DeepClone`, `TotalClone` does not follow magic pointers, so that objects referenced through magic pointers are not copied. See Table 20-1 on page 20-7 for a comparison with other object copying functions.

String Functions

These functions operate on and manipulate strings.

BeginsWith

`BeginsWith(` *string*`,` *substr* `)`

Returns non-`nil` if *string* begins with *substr*, or returns `nil` otherwise. This function is case and diacritical-mark insensitive. An empty *substr* matches any *string*.

string The string to test.

substr A string.

Capitalize

`Capitalize(` *string* `)`

Capitalizes the first character in *string* and returns the result. *string* is modified.

string The string to modify.

CapitalizeWords

`CapitalizeWords(` *string* `)`

Capitalizes the first character of each word in *string* and returns the result. *string* is modified.

string The string to modify.

CharPos

`CharPos(`*str*`, `*char*`, `*startpos*`)`

Returns the position of the next occurrence of character in the specified string, starting from the *startPos* (or `nil` if it's not found).

str	The specified string.
char	The specified character in the string.
startpos	The starting position of the character to return.

Downcase

`Downcase(` *string* `)`

Changes each character in *string* to lowercase and returns the result. *string* is modified.

string	The string to modify.

DurationStr

`DurationStr(`*minutes*`)`

Returns an array of strings giving the number of months, weeks, days, hours, and minutes represented by the specified number of minutes. The resulting strings are properly localized.

minutes	An integer specifying a number of minutes.

Here are some examples:

```
DurationStr(1000)
#4418D21  ["16 hours", "40 minutes"]
DurationStr(10000)
#4419179  ["", "6 days"]
DurationStr(100000)
#4419679  ["9 weeks", "6 days"]
```

EndsWith

`EndsWith(` *string*, *substr* `)`

Returns non-`nil` if *string* ends with *substr*, or returns `nil` otherwise. This function is case and diacritical-mark insensitive. An empty *substr* matches any *string*.

string	The string to test.
substr	A string.

EvalStringer

`EvalStringer(` *frame*, *array* `)`

Returns a string containing all of the elements in *array* concatenated. Any symbols in *array* are evaluated in the context of the specified *frame*.

frame	A frame used as the context for evaluating symbols in *array*.
array	An array.

Numbers, strings, characters, and symbols are converted to their natural string representation. For elements that are frames, arrays, and Booleans, this function converts them to an empty string.

FindStringInArray

`FindStringInArray(` *array*`,` *string* `)`

Finds a string in an array. This function compares the string to each element of the array. If the string matches the value of an array element, the index of that array element is returned. If the string is not found in the array, `nil` is returned.

array	An array to test.
string	A string.

The string comparison used to find a match is case sensitive. The string in the array must exactly match the string you specify in order for it to be found; a partial word will not be found.

FindStringInFrame

`FindStringInFrame(` *frame*`,` *stringArray*`,` *path* `)`

Finds one or more strings, specified by *stringArray*, in a frame.

frame	A frame to test.
stringArray	An array containing strings.
path	A Boolean indicating whether or not a description of the locations of successful searches should be returned.

This function compares the strings to each slot of the frame that contains a string. If all of the strings you specify in *stringArray* are found somewhere in the frame, this function returns non-`nil`. This function recursively searches arrays and frames referenced within the target frame for the strings. If all of the specified strings are not found within the target frame, including other frames and arrays referenced in it, `nil` is returned.

The string comparison used to find a match is not case sensitive (unlike `FindStringInArray`). And, the search looks for word beginnings, so it will not find a string unless it begins a word. For example in the string “blackboard”, this function would find the strings “blackboard” or “black”, but not “board”.

If *path* is non-`nil`, and the strings are found in the frame, this function returns an array of entries describing where each occurrence of the strings was found in the frame. A group of three entries is added to the array for each occurrence of a found string.

The first entry in each group is the complete value of the slot where the string was found.

The second entry is the path to the slot where the string was found (array elements are indicated by their index). This second entry can be either a slot access expression (e.g. `aSlot.anotherSlot.lastSlot`), or a path expression array (e.g. `[pathExpr: aSlot, 3, lastSlot]`) if the path includes an array.

And the third entry is the offset (in characters) of the string within the slot where it was found.

Here is an example:

```
myframe:={type: 'person,
             data: {name: "Christine Morrison",
                  employer: {company: "Apple",
                              years: 4,
                              boss: "John Morris"}}}
findstringinframe(myframe, ["Morris"], true)
#52185B1  ["Christine Morrison",
           data.name,
           10,
           "John Morris",
           data.employer.boss,
           5]
```

IsAlphaNumeric

`IsAlphaNumeric(`*char*`)`

Returns non-`nil` if *char* is a number or a letter; otherwise, this function returns `nil`.

char A character to test.

IsWhiteSpace

`IsWhiteSpace(`*char*`)`

Returns non-`nil` if *char* is a space ($\20), tab ($\09), linefeed ($\0A), or carriage return ($\0D) character; otherwise, this function returns `nil`.

char A character.

LatitudeToString

`LatitudeToString(`*latitude*`)`

Returns a string representation of the encoded latitude value.

latitude The latitude value.

LongitudeToString

`LongitudeToString(`*longitude*`)`

Returns a string representation of the encoded longitude value

longitude The longitude value.

NumberStr

`NumberStr(` *number* `)`

Returns a string representation of the number passed in.

number An integer or real number to be converted.

For example, if you pass in the value 1234.56, you'll get back this string: `"1234.56"`. If you pass in an integer, the string will contain an integer, and if you pass in a real number, the string will contain a real number.

ParamStr

`ParamStr(` *baseString*`,` *paramStrArray* `)`

Returns a new string that is the result after the substitution has been performed. The original *baseString* is not modified.

baseString	The base string containing substitution placeholders.
paramStrArray	An array of strings to be substituted for the placeholders in the base string. You can also specify numbers, characters, or symbol data types and these will be converted to their natural string representation.

This function returns the base string after the substitutions have been made.

The substitution placeholders in the base string are the following character pairs: `"^0"`, `"^1"`, and so on up to `"^9"`. There can be a maximum of 10 placeholders specified in any order in the base string. But no numbers can be skipped; that is, if the string contains ^2, it must also contain ^1 and ^0. The substitution is done by replacing placeholder ^0 with the first element from the string array. Then placeholder ^1 is replaced by the second element, and so on.

Placeholders can be nested up to three levels deep. This means that the substitution strings can themselves contain placeholders, which will be replaced on subsequent passes up to two additional times after the initial replacement.

If you need to specify a caret (^) as part of a string, use two carets together (^^).

`ParamStr` also supports conditional substitution using this syntax:

^?*X true*|*false*|

The value *X* is an integer from 0 through 9, representing a standard placeholder, as above. If the element in *paramStrArray* corresponding to this placeholder is non-`nil` and not the empty string, then the *true* characters are interpreted. Otherwise, the *true* characters are skipped, and the *false* characters are interpreted. The vertical bars act to delimit the *true* and *false*

portions of the string. Note that the *true* or *false* portions of the string may contain no characters.

Conditional operators can be nested, and any character can appear between the delimiters. If you need to use the vertical bar character as part of a *true* or *false* string, specify `^|`.

The conditional operator is useful for avoiding the insertion of unnecessary punctuation or spaces when building a string from elements that may include optional or potentially empty items.

Here are some examples. If your *baseString* is this:

```
"^2 ^0 of each ^1."
```

And your *paramStrArray* is this:

```
["Monday", "week", "Every"]
```

Then `ParamStr` would return this string:

```
"Every Monday of each week."
```

If your *baseString* is this:

```
"^?0^0, ||^?1^1, ||^2"  // false branches are empty
```

And your *paramStrArray* is this:

```
["Sarah", "", "Smith"]
```

Then `ParamStr` would return this string:

```
"Sarah, Smith"
```

SPrintObject

`SPrintObject(` *obj* `)`

Returns a string of the object passed in. Numbers, strings, characters, and symbols are converted to their natural string representation. For frames, arrays, and Booleans, this function returns an empty string.

To convert the contents of a frame or array into strings, use the `Foreach` statement along with the `Stringer` function to iterate over each slot.

To convert a Boolean into a string, you must check for non-`nil` or `nil` and return the appropriate string.

Note
This function changes the number format depending on the current locale setting. Real numbers may be formatted unexpectedly. ◆

StrCompare

`StrCompare(` *a*, *b* `)`

Returns a negative number if string *a* is less than string *b*. Returns zero if string *a* and *b* are equal. Returns a positive number if string *a* is greater than string *b*. Case is not significant (that is, “Hello” and “hello” are equal).

a A string.

b A string.

Note that this is a content comparison of the two strings, not a pointer comparison.

Use `StrExactCompare` to do a case-sensitive comparison of strings.

StrConcat

`StrConcat(` *a*, *b* `)`

Concatenates string *b* onto string *a* and returns the result as a new string.

a A string.

b A string.

StrEqual

`StrEqual( a, b )`

Returns non-`nil` if the two strings, *a* and *b*, are equal.

a A string.

b A string.

Case is not significant. Note that this is a content comparison of the two strings, not a pointer comparison.

Use `StrExactCompare` to do a case-sensitive comparison of strings.

StrExactCompare

`StrExactCompare( a, b )`

Returns a negative number if string *a* is less than string *b*. Returns zero if string *a* and *b* are equal. Returns a positive number if string *a* is greater than string *b*. Case and diacritical marks are significant (that is, “Hello” and “hello” are not equal).

a A string.

b A string.

Note that this is a content comparison of the two strings, not a pointer comparison.

Use `StrCompare` or `StrEqual` to do a case-insensitive comparison of strings.

StrFilled

`StrFilled(`*string*`)`

Returns non-`nil` if the expression *string* evaluates to a string with a length greater than zero. This function returns `nil` if the expression *string* is either `nil` or evaluates to an empty string.

string An expression that evaluates to a string.

StrFontWidth

`StrFontWidth(` *string*`,` *fontSpec*`)`

Returns the width of the string in pixels, if drawn in the specified font.

string	A string.
fontSpec	A frame having the following format:

`{family:`*familyName*`,face:`*faceName*`,size:`*pointSize*`}`

For more information about specifying fonts, see the section "Specifying a Font" in Chapter 8, "Text and Ink Input and Display."

Stringer

`Stringer(` *array* `)`

Returns a string containing all of the elements in the array concatenated.

array	An array.

Numbers, strings, characters, and symbols are converted to their natural string representation. For elements that are frames, arrays, and Booleans, this function converts them to an empty string.

StringFilter

`Stringfilter(`*str*`,` *filter*`,` *instruction*`)`

Returns a string filtered according to the instruction.

str	The string to be filtered.
filter	A string containing characters to be filtered from the string.
instruction	One of the symbols shown in Table 20-2.

Table 20-2 Instruction symbols for StringFilter

Instruction Symbol	Meaning
`'passAll`	Return any letter in *str* also in *filter*.
`'passBeginning`	Look for any character in *filter*, and return everything in *str* after and including that character.
`'passOne`	Pass only the first letter of a group in the filter and pass everything else. This is useful to collapse an arbitrary number of spaces to one.
`'rejectAll`	Returns any letter in *str* not in *filter*
`'rejectBeginning`	Rejects any letter that is in *filter* until it reaches a letter that isn't in *filter*. It returns everything past that point.

StringToNumber

`StringToNumber(` *string* `)`

Parses a string representing a number and returns the real number value (never an integer).

string A string.

The format of the real number returned by this function is determined by values in the current locale bundle. The number of digits allowed on both sides of the decimal is 63. Instead of simply changing the constants, a more space-efficient way is to calculate the value. If the number of digits on either side of the decimal point exceeds 63, `StringToNumber` returns `nil`. For more information, see Chapter 19, "Localizing Newton Applications."

Strings with the following kinds of numbers can be parsed:

```
1
1.2
```

```
-12,345
(12,345.78)
```

StrLen

`StrLen(` *string* `)`

Returns the number of characters in a string, excluding the null terminator (if one exists).

string	A string.

StrMunger

`StrMunger(` *dstString, dstStart, dstCount, srcString, srcStart, srcCount* `)`

Replaces characters in *dstString* using characters from *srcString* and returns the destination string after munging is complete. This function is destructive to *dstString*.

dstString	The destination string. The string must be writable, you can't specify a string literal, or an exception will be thrown. You'll have to use `Clone` (page 20-8) or a similar function to make a writable copy from a string literal.
dstStart	The starting position within *dstString*.
dstCount	The number of characters to be replaced in *dstString*. You can specify `nil` for *dstCount* to go to the end of the string.
srcString	A string. This can be `nil` to simply delete the characters.
srcStart	The starting position in *srcString* from which to begin taking characters to place into *dstString*.
srcCount	The number of characters to use from *srcString*. You can specify `nil` to go to the end of *srcString*.

Here is an example:

```
StrMunger("abcdef", 2, 3, "ZYXWV", 0, nil)
"abZYXWVf"
```

`StrMunger` can also be used to concatenate large strings; for example:

```
StrMunger(str1, StrLen(str1)+1, nil, str2, 0, nil);
```

StrPos

`StrPos(` *string*`,` *substr*`,` *start* `)`

Returns the position of *substr* in *string*, or `nil` if *substr* is not found. The search begins at character position *start*. (The first character position in a string is zero.) This function is not case sensitive.

string	A string.
substr	A string.
start	An integer.

Here is an example:

```
StrPos("abcdef", "Bcd", 0)
1
```

StrReplace

`StrReplace(` *string*`,` *substr*`,` *replacement*`,` *count* `)`

Replaces each occurrence of *substr* in *string* with *replacement*. *count* is the number of replacements to perform, or `nil` to replace all occurrences. This function returns the number of replacements performed. This function is destructive to *string*.

string	A string.
substr	A string.
replacement	A string.
count	An integer.

`StrReplace` positions the replacement pointer after the current replacement for each iteration, so a three time replacement of “a” in “aaa” with “ab” will yield “ababab”, not “abbbaa” as in some editors.

StrTokenize

`StrTokenize(`*`str`*`,` *`delimiters`*`)`

Breaks up a string into chunks for you as defined by the *delimiters* argument. Each time you call the closure (passing it no arguments) you will get back the next token, until there are no more tokens and it returns `nil`.

str	A string to be broken up into tokens
delimiters	Either a character or string (list of characters) that are the delimiters separating the pieces of the string.

For example, to break up a sentence into space separated words you do something like the following:

```
fn := StrTokenize("the quick green fox", $ );
#441BE8D  <function, 0 arg(s) #441BE8D>
      while x := call fn with () do Print(x);
"the"
"quick"
"green"
"fox"
#2        NIL
```

StyledStrTruncate

`StyledStrTruncate(`*string*`, `*length*`, `*font*`)`

Truncates a string to the indicated length, in pixels. (Of course, the length does not include the null terminator.) Returns the truncated string.

string	A string.
length	An integer specifying the length, in pixels, at which to truncate the string.
font	A font specification, which is used to determine how many characters of the string will fit in the specified length. For details on specifying a font, refer to the section "Specifying a Font" in Chapter 8, "Text and Ink Input and Display."

This function adds an ellipsis (...) to the end of the truncated string.

SubstituteChars

`SubstituteChars(`*targetStr*`, `*searchStr*`, `*replaceStr*`)`

Substitutes characters in *targetStr* by searching for each character in *searchStr* and replacing it by the value of string length in *replaceStr*. That is, for each offset character "`x`" in *targetStr*, if it exists in *searchStr*, it will, in a copy of *targetStr*, replace

```
copy[x]
```

with

```
replaceStr[y mod StrLength(replaceStr)]
```

.

If no substitutions are made, the original string is returned unmodified; otherwise, a modified copy is returned.

For example:

```
SubstituteChars("Text with spaces\tand\ttabs", " \t",
"-")
```

Creates

```
Text-with-spaces-and-tabs
```

or

```
SubstituteChars( "(800) 41PHONE",
"ADGJMPTWBEHKNRUXCFILOSVY", "23456789"
```

Creates

```
(800) 4174663
```

SubStr

`SubStr(` *string*`,` *start*`,` *count* `)`

Returns a new string containing *count* characters from *string*, starting at position *start*. Character positions begin with zero for the first character.

string	A string.
start	An integer.
count	An integer.

TrimString

`TrimString(` *string* `)`

Removes any white space (spaces, tabs, and new line characters) from the beginning and end of *string* and returns the result. *string* is modified.

string	A string.

Upcase

`Upcase(` *string* `)`

Capitalizes each character in *string* and returns the result. *string* is modified.

string	A string.

Bitwise Functions

These functions perform logical operations on bits.

Band, Bor, Bxor, and Bnot

`Band(`*a*`, `*b*`)`
`Bor(`*a*`, `*b*`)`
`Bxor(`*a*`, `*b*`)`
`Bnot(`*a*`)`

These bitwise functions each return an integer result of their operation on one or two integer parameters. They perform bitwise AND, OR, XOR, and NOT, respectively.

a	An integer.
b	An integer.

Array Functions

These functions operate on and manipulate arrays.

AddArraySlot

`AddArraySlot (`*array*`, `*value*`)`

Appends a new element onto an array.

array	An array.
value	A value to be added as new element in the array.

For example:

```
myArray := [123, 456]
#1634 myArray
addArraySlot (myArray, "I want chopstix")
#12 "I want chopstix"
myArray
#1634 [123, 456, "I want chopstix"]
```

Array

`Array(` *size*`,` *initialValue* `)`

Returns a new array with *size* number of elements that each contain *initialValue*.

size	An integer.
initialValue	A value.

ArrayInsert

`ArrayInsert(`*array*`,` *element*`,` *position*`)`

Inserts an element into an array and returns the modified array.

array	The array to be modified.
element	The element to be inserted into the array.
position	The index where the new element is to be inserted. Specify zero to insert the element at the beginning of the array. Specify the result of `Length(`*array*`)` to insert the element at the end of the array.

The length of the array is increased by one.

ArrayMunger

`ArrayMunger(` *dstArray*`,` *dstStart*`,` *dstCount*`,` *srcArray*`,` *srcStart*`,` *srcCount* `)`

Replaces elements in *dstArray* using elements from *srcArray* and returns the destination array after munging is complete. This function is destructive to *dstArray*.

dstArray	The destination array.
dstStart	The starting element in the destination array.
dstCount	The number of elements to be replaced in *dstArray*. You can specify `nil` for *dstCount* to go to the end of the array.
srcArray	An array. You can specify `nil` for *srcArray* to delete the elements.
srcStart	The starting position in the source array from which to begin taking elements to place into the destination array.
srcCount	The number of elements to use from the source array. You can specify `nil` to go to the end of the source array.

Here is an example:

```
ArrayMunger([10,20,30,40,50], 2, 3, [55,66,77,88,99], 0, nil)
[10, 20, 55, 66, 77, 88, 99]
```

Using `ArrayMunger` is the most efficient way to join two arrays.

To put B at the front of A:

```
ArrayMunger(A, 0, 0, B, 0, nil)
```

To put B at the end of A:

```
ArrayMunger(A, Length(A), 0, B, 0, nil)
```

You can also do this with `SetUnion` (page 20-47), which has the additional property that it eliminates duplicates, but `ArrayMunger` is much faster if you don't need that property.

ArrayRemoveCount

`ArrayRemoveCount(` *array*`,` *startIndex*`,` *count* `)`

Removes one or more elements from an array.

array	The array from which to remove elements.This parameter is modified by this function.
startIndex	An integer that is the index of the first element to remove.
count	An integer specifying the number of elements to remove.

Any elements following those removed are shifted left so that no empty elements remain.

InsertionSort

`InsertionSort(`*array*`,` *test*`,` *key*`)`

Sorts an array, preserving the original relative ordering of equivalent elements.

array	The array to modify by sorting.
test	Indicates how the array is to be sorted. See the description of the *test* parameter on page 20-49.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 20-50.

This sort performs very well on arrays that are nearly sorted already and on very small arrays. This sort is an $O(n^2)$ sort. To sort larger arrays, use `Sort` or `StableSort`.

Length

`Length (`*array*`)`

Returns the number of elements in an array, the number of slots in a frame, or the size, in bytes, of a binary object.

array An array or frame or binary object.

For example:

```
myArray := [123, 456, "I want chopstix"]
length (myArray)
#12    3
```

Note that arrays are indexed from 0, but length returns a count of the number of characters. Therefore, the last element of this example is element 2.

Note

If you pass a string to this function, you will get the number of bytes that a string occupies. To get the length of strings, use `StrLen` instead. ◆

LFetch

`LFetch(`*array*`, `*item*`, `*start*`, `*test*`, `*key*`)`

Linearly searches an array for the specified element and returns the element, or `nil` if it is not found or if *start* is equal to or greater than the length of the array.

array The array in which to search.

item The key value for which to search.

start The array index at which to begin searching.

test Indicates how to compare key values to test for a match. Specify one of the following symbols for *test*:

`'|=|` If the objects being compared are immediates and reals, their values are

compared for equivalency. For reference objects, their identity is compared.

`'|str=|` For string objects, the contents of the strings are compared for equivalency.

Alternatively, for nonstandard sorting situations, you can specify a function object that compares two key values and returns a Boolean or integer value indicating whether or not they are equivalent. This function will be called to test for matches. The function is passed two parameters, *A* and *B*, where *A* is the *item* parameter passed to `LFetch` and *B* is the array element being tested.

The function must return a non-`nil` value (or zero) if the items are equivalent, or `nil` (or a non-zero integer) if the items are not equivalent.

Note that specifying a function object for *test* results in much slower performance than using one of the predefined symbols.

key Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 20-50.

This function works just like `LSearch`, except that `LSearch` returns the index of the found item.

If you know that the array you are working with is sorted, you can use the function `BFetch` to search for an element. This function, based on binary search algorithms, is much faster on large arrays than `LFetch` or `LSearch`, though it can be used only on sorted arrays.

LSearch

`LSearch(`*array*`, `*item*`, `*start*`, `*test*`, `*key*`)`

Linearly searches an array for the specified element and returns the index of the element, or `nil` if it is not found or if *start* is equal to or greater than the length of the array.

array — The array in which to search.

item — The key value for which to search.

start — The array index at which to begin searching.

test — Indicates how to compare key values to test for a match. Specify one of the following symbols for *test*:

- `'|=|` — If the objects being compared are immediates and reals, their values are compared for equivalency. For reference objects, their identity is compared.
- `'|str=|` — For string objects, the contents of the strings are compared for equivalency.

Alternatively, for non-standard sorting situations, you can specify a function object that compares two key values and returns a Boolean or integer value indicating whether or not they are equivalent. This function will be called to test for matches. The function is passed two parameters, *A* and *B*, where *A* is the *item* parameter passed to `LSearch` and *B* is the array element being tested. The function must return a non-`nil` value (or zero) if the items are equivalent, or `nil` (or a non-zero integer) if the items are not equivalent. Note that specifying a function object for *test* results in much slower performance than using one of the predefined symbols.

key — Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 20-50.

This function works just like `LFetch`, except that `LFetch` returns the found item instead of its index.

If you know that the array you are working with is sorted, you can use the function `BFind` to search for an element. This function, based on binary search algorithms, is much faster than `LSearch`, though it can be used only on sorted arrays.

NewWeakArray

`NewWeakArray(`*length*`)`

Returns a new weak array with *length* number of elements, which are initialized to `nil`.

length An integer specifying the size of the array to create.

A **weak array** is an array that does not prevent the objects it refers to from being garbage-collected. That is, if the only references to an object are from weak arrays, the object is destroyed during the next garbage collection cycle. When that happens, the references in the weak arrays are replaced with `nil`.

The purpose of weak arrays is to cache objects without preventing them from being garbage collected. For example, if you wanted to keep an array of all objects in existence of a certain type, you could add each object to an array as it's created. If you use a regular array, those objects will never be garbage-collected, because there will always be references to them in your array, and the system will eventually run out of memory. However, if you use a weak array, its references don't affect garbage collection, so the objects will be garbage-collected normally, freeing memory when it is needed.

SetAdd

`SetAdd (`*array*`,` *value*`,` *uniqueOnly*`)`

Appends an element to the specified array and returns the modified array, or `nil` if the element was not added.

array	The array to which `SetAdd` appends the element in *value*.
value	The element to append to the array specified by *array*.
uniqueOnly	Whether only unique elements are to be added to the array; if the value of this parameter is non-`nil`, `SetAdd` appends *value* to the array only if it is not already present in the array. If the element specified by the *value* parameter is already present in the array, `SetAdd` returns `nil` and does not append the element. If *uniqueOnly* is `nil`, the item is appended to the array without checking whether it is unique.

Note
The type of comparison used in this function is pointer comparison, not content comparison. ◆

SetContains

`SetContains(` *array*`,` *item* `)`

array	An array.
item	An item that may be in the array.

Searches each element of an array to determine if *item* is equal to one of the array elements. If a match is found, this function returns the array index of the matching array element. If *item* is not found in the array, `nil` is returned.

Note
The type of comparison used in this function is pointer comparison, not content comparison. ◆

SetDifference

`SetDifference(` *array1*`,` *array2* `)`

Returns an array that contains all of the elements in *array1* that do not exist in *array2*.

array1 An array.

*array*2 An array.

If *array1* is nil, `nil` is returned.

Note

The type of comparison used in this function is pointer comparison, not content comparison. ◆

SetLength

`SetLength (`*array*`,` *length*`)`

Sets the length of an array.

array An array.

length An integer.

This function is useful for increasing or decreasing the size of an array. If you increase the size of the array, new elements are filled with a `nil` value. For example:

```
myArray := [123, 456, "I want chopstix"]
#1634 myArray
setLength (myArray, 4)
#1634 [123, 456, "I want chopstix", NIL]
myArray [3] := 789
#3156 789
myArray
#1634 [123, 456, "I want chopstix", 789]
```

SetOverlaps

`SetOverlaps(` *array1*`,` *array2* `)`

Compares each element in *array1* to each element in *array2*, and returns the index of the first element in *array1* that is equal to an element in *array2*. If no equivalent elements are found, `nil` is returned.

array1	An array.
*array*2	An array.

Note
The type of comparison used in this function is pointer comparison, not content comparison. ◆

SetRemove

`SetRemove (`*array*`,` *value*`)`

`SetRemove` removes the specified element from the specified array and returns the modified array. The length of the array is shifted left by one and all of the elements after the deleted element are shifted by one to the next lowest numbered array position. If the item is not found in the array, this function returns `nil`.

array	The array from which `SetRemove` removes the specified element.
value	The element to remove from the array specified by *array*.

Note
The type of comparison used in this function is identity comparison, not pointer comparison. ◆

SetUnion

`SetUnion( array1, array2, uniqueFlag )`

Returns an array that contains all of the elements in *array1* and all of the elements in *array2*.

array1	An array.
*array*2	An array.
uniqueFlag	If any non-`nil` value, `SetUnion` will not include any duplicate items in the array it returns. If *uniqueFlag* is `nil`, all elements from both arrays are included, even if there are duplicates.

If both of the arrays are `nil`, an empty array is returned.

`SetUnion` can eliminate duplicates. If you do not need that property, you can combine two arrays more efficiently using `ArrayMunger` (page 20-38).

Note

The type of comparison used in this function is identity comparison, not pointer comparison. ◆

Sort

`Sort( array, test, key )`

Sorts an array and returns it after it is sorted. The sort is destructive; that is, the array you give it is modified. The sort also is not stable; that is, elements with equal keys won't necessarily have the same relative order after the sort.

array	An array.
test	Defines the sort order. It can be a function object that takes two parameters *A* and *B* and returns a positive integer if *A* sorts after *B*, returns zero if *A* sorts equivalently to *B*, and returns a negative integer if *A* sorts before *B*.

For much greater speed, specify one of the following symbols for *test*:

`'|<|`	Sort in ascending numerical order
`'|>|`	Sort in descending numerical order
`'|str<|`	Sort in ascending string order
`'|str>|`	Sort in descending string order

key Defines the sort key within each array element. Specify `nil` to use the array elements directly as they are. You can specify a path expression, in which case the array elements are assumed to be frames or arrays and the path is applied to each element to find the sort key. Or, you can specify a function that takes one parameter and returns the key.

This example sorts `myArray` in ascending numerical order according to the `timestamp` slot of the entries:

```
Sort(myArray, '|<|, 'timestamp)
```

This example sorts `myArray` in descending string order according to the first and last names concatenated together:

```
Sort(myArray, '|str>|, func (e) e.first && e.last)
```

StableSort

`StableSort(`*array*`, `*test*`, `*key*`)`

Sorts an array, preserving the original relative ordering of equivalent elements.

array The array to modify by sorting.

test Indicates how the array is to be sorted. See the description of the *test* parameter on page 20-49.

key Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one

parameter. See the description of the *key* parameter on page 20-50.

This sort requires working memory, so may not be suitable for extremely large arrays or in low memory conditions.

Sorted Array Functions

This section describes new functions that operate on sorted arrays. These functions are based on binary search algorithms, hence the "B" prefix to the function names.

IMPORTANT

The arrays you pass to these functions must be ordered, otherwise the results are undefined. To sort an array, you can use the functions `Sort`, `InsertionSort`, or `StableSort`.◆

These sorted array functions each use *test* and *key* parameters to allow them to be adapted to different data structures. Typically, these functions search, or iterate over several items in an array. As each element in an array is examined, the *key* argument is used to extract a value, called the key, from the element. Then that key is treated as specified by the *test* argument.

Here's an explanation of these parameters:

test Indicates the sort order of the array. Specify one of the following symbols for *test*, to indicate how the array is sorted:

`'|<|`	Sorted in ascending numerical order
`'|>|`	Sorted in descending numerical order
`'|str<|`	Sorted in ascending string order
`'|str>|`	Sorted in descending string order
`'|sym<|`	Sorted in ascending symbol order, based on lexical comparison of symbol name

`'|sym>|`	Sorted in descending symbol order, based on lexical comparison of symbol name

Alternatively, for non-standard sorting situations, you can specify a function object that compares two key values and returns an integer that indicates how they are sorted relative to each other. This function will be called by any of the sorted array functions to determine sorting relationships between elements. The function is passed two parameters, *A* and *B*, and must return a positive integer if *A* sorts after *B*, must return zero if *A* sorts equivalently to *B*, and a must return a negative integer if *A* sorts before *B*. Note that specifying a function object for *test* results in much slower performance than using one of the predefined symbols.

key — Defines the key within each array element. Specify `nil` to use the array elements directly as they are. You can specify a path expression, in which case the array elements are assumed to be frames or arrays and the path is applied to each element to find the key. You can also specify a function that takes one parameter (the element) and returns the key.

BDelete

`BDelete(`*array*`, `*item*`, `*test*`, `*key*`, `*count*`)`

Deletes elements from an ordered array.

This function returns the number of elements deleted.

array	The array to be modified.
item	The key value for which to search. Elements with this key are deleted.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 20-49.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 20-50.
count	The maximum number of elements to delete. Specify `nil` to indicate that all matching elements are to be deleted.

BDifference

`BDifference(`*array1*`, `*array2*`, `*test*`, `*key*`)`

Returns a new sorted array containing those elements from *array1* that do not have equivalent elements in *array2*.

array1	The first array. This array is not modified.
array2	The second array. This array is not modified.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 20-49.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 20-50.

BFetch

`BFetch(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses a binary search to find an element in a sorted array. The leftmost found element is returned, or `nil` is returned if none are found.

array	The array to be searched.
item	The key value for which to search.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 20-49.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 20-50.

This function works just like `BFind`, except that `BFind` returns the index of the found item.

BFetchRight

`BFetchRight(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses a binary search to find an element in a sorted array. The rightmost found element is returned, or `nil` is returned if none are found.

array	The array to be searched.
item	The key value for which to search.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 20-49.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 20-50.

This function works just like `BFindRight`, except that `BFindRight` returns the index of the found item.

BFind

`BFind(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses a binary search to find an element in a sorted array. The index of the leftmost found element is returned, or `nil` is returned if none are found.

array	The array to be searched.
item	The key value for which to search.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 20-49.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 20-50.

This function works just like `BFetch`, except that `BFetch` returns the found item instead of its index.

BFindRight

`BFindRight(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses a binary search to find an element in a sorted array. The index of the rightmost found element is returned, or `nil` is returned if none are found.

array	The array to be searched.
item	The key value for which to search.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 20-49.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 20-50.

This function works just like `BFetchRight`, except that `BFetchRight` returns the found item instead of its index.

BInsert

`BInsert(`*array*`, `*element*`, `*test*`, `*key*`, `*uniqueOnly*`)`

Inserts an element into the proper position in a sorted array. In the case of equivalent elements, the element is inserted to the left of its equivalent.

array	The array to be modified.
element	The new element to be inserted. Note that the *key* parameter is used to extract its key value.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 20-49.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 20-50.
uniqueOnly	Specify non-`nil` to indicate that the element is not to be inserted if the array already contains an element with an equivalent key value. Specify `'returnElt` to indicate the same thing, and also that this function should return an array element. It returns either the element that was inserted, or if a matching element is found in the array, that element is returned. This is useful when you want to reuse existing objects in order to conserve space or ensure pointer equality. Specify `nil` to indicate that the element is to be inserted even if the array already contains an element with an equivalent key. In this case, the new element is inserted to the left of the existing equivalent elements.

This function has three possible return values, as follows:

- It can return `nil`, signaling that the element was not inserted.
- It can return an integer, which is the index at which the element was inserted.

- It can return an array element—either the element that was inserted (if it was unique), or an element that already exists in the array, whose key value matches the key value of the element you wanted to insert. This type of return value can occur only if you specify `'returnElt` for *uniqueOnly*.

Here is an example of how you might use this function with *uniqueOnly* set to `'returnElt` to ensure pointer equality:

```
// :GetStr() returns a string input by the user
bodyColor := BInsert(colorList,:GetStr(),'|str<|,nil,'returnElt);
interiorColor:= BInsert(colorList,:GetStr(),'|str<|,nil,'returnElt);
if bodyColor = interiorColor then Print("bad idea");
```

If `GetString` returns a string already in `colorList`, this code makes sure that the original string is reused. This is why using the `=` operator to test for equality works. It also allows the duplicate string to be garbage collected, provided there are no remaining references to it.

BInsertRight

`BInsertRight(`*array*`,` *element*`,` *test*`,` *key*`,` *uniqueOnly*`)`

Inserts an element into the proper position in a sorted array. In the case of equivalent elements, the element is inserted to the right of its equivalent. The index at which it was inserted is returned, or `nil` is returned if it was not inserted.

array	The array to be modified.
element	The new element to be inserted. Note that the *key* parameter is used to extract its key value.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 20-49.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one

parameter. See the description of the *key* parameter on page 20-50.

uniqueOnly — A Boolean value. Specify a non-`nil` value to indicate that the element is not to be inserted if the array already contains an element with an equivalent key value. Specify `nil` to indicate that the element is to be inserted even if the array already contains an element with an equivalent key. In the later case, the new element is inserted to the right of the existing equivalent elements.

BIntersect

`BIntersect(`*array1*`, `*array2*`, `*test*`, `*key*`, `*uniqueOnly*`)`

Returns a new sorted array consisting of the equivalent elements from the two specified arrays.

array1 — The first array. This array is not modified.

array2 — The second array. This array is not modified.

test — Indicates the sort order of the array. See the description of the *test* parameter on page 20-49.

key — Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 20-50.

uniqueOnly — A Boolean value. Specify a non-`nil` value to indicate that elements with duplicate key values are not allowed in the resulting array. Note that this works only if *array1* and *array2* are both free of equivalent elements.

Specify `nil` to indicate that elements with duplicate key values are allowed in the resulting array. Note that this guarantees that the resulting array has at least two

equivalent elements for every intersecting value, since intersection finds equivalent elements.

If equivalent elements are found in the resulting array, they are ordered as follows: equivalent elements from the same source array retain their original ordering, and equivalent elements from *array1* come before those in *array2*.

BMerge

`BMerge(`*array1*`, `*array2*`, `*test*`, `*key*`, `*uniqueOnly*`)`

Merges two ordered arrays into one new ordered array, which is returned.

array1	The first array. This array is not modified.
array2	The second array. This array is not modified.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 20-49.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 20-50.
uniqueOnly	A Boolean value. Specify a non-`nil` value to indicate that elements with duplicate key values are not allowed in the resulting array. Note that this works only if *array1* and *array2* are both free of equivalent elements. Specify `nil` to indicate that elements with duplicate key values are allowed in the resulting array. If equivalent elements are found in the resulting array, they are ordered as follows: equivalent elements from the same source array retain their original ordering, and equivalent elements from *array1* come before those in *array2*.

BSearchLeft

`BSearchLeft(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses binary search to find an element in a sorted array. The index of the smallest and leftmost element that is greater than or equal to *item* is returned. The value `Length(`*array*`)` is returned if *item* is larger than all elements.

array	The array to be searched.
item	The key value for which to search.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 20-49.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 20-50.

Here is an example of how this function might be used:

```
// Extract all elements between "F" and "Na"
array := ["Ag","C","F","Fe","Hg","K","N","Na","Ni","Pu","Zn"];
pos1   := Min(Length(array)-1,BSearchLeft(array,"F",'|str<|,nil));
pos2   := Max(0,BSearchRight(array,"Na",'|str<|,nil));
ArrayMunger([],0,nil,array, pos1, pos2-pos1+1);
```

BSearchRight

`BSearchRight(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses binary search to find an element in a sorted array. The index of the largest and rightmost element that is less than or equal to *item* is returned. The value –1 is returned if all elements are larger than *item*.

array	The array to be searched.
item	The key value for which to search.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 20-49.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 20-50.

For an example of how this function might be used, see `BSearchLeft`.

Integer Math Functions

These math functions operate on or return integers. (Some of the floating point functions can also operate on integers.)

Abs

`Abs(`*x*`)`

Returns the absolute value of an integer or real number.

x	An integer or real number.

Ceiling

`Ceiling(`*x*`)`

Returns the smallest integer not less than the specified real number. (Rounds up the real number to an integer.)

x A real number.

Floor

`Floor(`*x*`)`

Returns the largest integer not greater than the specified real number. (Rounds down the real number to an integer.)

x A real number.

GetRandomState

`GetRandomState()`

Returns the current state of the random number generator as a binary object of unspecified format. The random state object is only useful for passing to `SetRandomState`.

Max

`Max(` *a*`,` *b* `)`

Returns the maximum value of the two integers *a* and *b*.

a An integer.

b An integer.

Min

`Min(` *a*`,` *b* `)`

Returns the minimum value of the two integers *a* and *b*.

a An integer.

b An integer.

Real

`Real(`*x*`)`

Converts the specified integer to a real number.

x An integer.

Random

`Random (`*low*`, `*high*`)`

Returns a random integer in the range between the two integers `low` and `high`. The range is inclusive of the numbers `low` and `high`.

low An integer.

high An integer.

For example:

```
random (0, 100)
#120      72
```

SetRandomSeed

`SetRandomSeed (`*seedNumber*`)`

Seeds the random number generator with the number you specify.

seedNumber An integer.

When seeded with the same number, the random number generator (`Random` function) will return the same sequence of random numbers each time you reseed it. Do not use 0 to seed the generator as it will return 0 instead of a random number. To generate virtually random numbers, you can seed it with the value returned from the time function `Ticks`, like this:

```
SetRandomSeed(Ticks());
```

Note
There is only one random number generator on the Newton, so calls by other functions may interfere with your function getting a consistent sequence of values. ◆

SetRandomState

`SetRandomState(`*randomState*`)`

Used to reset the random number generator to a previously saved state.

randomState A random state object returned by `GetRandomState`.

The return value of this function is unspecified.

Note that this function provides different functionality than `SetRandomSeed`, which lets you conveniently initialize the random state by providing an integer seed value.

Floating Point Math Functions

NewtonScript provides the floating point math functions documented in this section.

The NewtonScript floating point number system is based on standards 754 and 854 adopted by the Institute of Electrical and Electronics Engineers (IEEE). For more details on IEEE-standard arithmetic than are given here, refer to the *PowerPC Numerics* volume of *Inside Macintosh* or to the *Apple Numerics Manual, Second Edition*. These books describe SANE, the standard Apple numeric environment. The NewtonScript environment supports many features of SANE.

NewtonScript floating point numbers (also called *real* numbers) correspond to the double format of the IEEE standards. The number system supports representations for the following values:

- Normal numbers—numbers with approximately 16 decimal digits of precision, ranging from 1.8×10^{308} down to 2.2×10^{-308}.

- Subnormal numbers—numbers ranging from 2.2×10^{-308} down to 4.9×10^{-324}, whose precision diminishes from approximately 16 decimal digits down to less than one digit.
- Signed zeros—the values +0 and –0, which compare equal, but whose behavior differs when, for example, divided into nonzero values.
- Signed infinities—the values `+INF` and `-INF`, which represent results too large to represent or the result of dividing a nonzero numerator by a zero denominator.
- Not-a-Number symbols, or NaNs—values used to represent missing or uninitialized data, or the results of operations, such as $\sqrt{-3}$, which have no meaning in the real number system.

In some application areas, you may find it useful to think of signed zeros and infinities in terms of mathematical *limits*. For example, although +0 and –0 compare equal, it may be the case for a function *f* that $\lim_{x \to 0-} f(x) \neq \lim_{x \to 0+} f(x)$, and you may find it useful to exploit that fact. Similarly, you may find it useful to interpret *g*(`+INF`) as $\lim_{y \to \infty} g(y)$.

The functions in this section follow the model of the arithmetic operations set forth in the IEEE standards, namely, they produce results that are exact when the results are exactly representable in the number system, and otherwise they deliver the nearest (or nearly so) representable number to the mathematically correct result. The IEEE standards specify that one or more exceptions be raised when the result of an operation is different from the mathematical result, or when the result is not defined in the real number system. The possible exceptions are

- Inexact—the result is *rounded* or otherwise altered from the mathematical result.
- Underflow—the nonzero result is too tiny to represent except as zero or a subnormal number, and is rounded to less precision than a normal number.
- Overflow—the result is too huge to represent as a normal number.

- Divide by Zero—the quotient of a nonzero value divided by zero produces `+INF` or `-INF`, according to the arguments' signs.
- Invalid—the result is not mathematically defined, as is the case with 0/0.

See "Managing the Floating Point Environment" on page 20-79 for further discussion of the handling of floating point exceptions.

One feature of the IEEE standards and SANE is the choice of rounding direction for results not exactly representable. In NewtonScript systems, rounding is *always* to the nearest representable number (with ties going to the value whose least significant bit is zero). The IEEE standards also specify rounding to the nearest value toward 0, toward `+INF`, or toward `-INF`. But the standards are written as though rounding direction is determined by a state variable in the floating point environment (see "Managing the Floating Point Environment"), while on the ARM family of processors used by NewtonScript systems, rounding direction is determined on an instruction-by-instruction basis.

Acos

`Acos(`*x*`)`

Returns the inverse cosine in radians of *x*. `Acos` raises invalid for $x < -1$ or $x > 1$. It raises inexact for all values except 1. `Acos` returns values between zero and π.

x An integer or real number.

Acosh

`Acosh(`*x*`)`

Returns the inverse hyperbolic cosine of *x*. `Acosh` raises invalid for $x < 1$. It raises inexact for all values except 1. `Acosh(+INF)` returns `+INF`, but `Acosh` never overflows. Its value at the largest finite real number is approximately 710.

x An integer or real number.

Asin

`Asin(`*x*`)`

Returns the inverse sine in radians of *x*. `Asin` raises invalid for $x < -1$ or $x > 1$. It raises inexact for all values except zero and raises underflow for all finite *x* near zero. `Asin` returns values between $-\pi/2$ and $\pi/2$.

x An integer or real number.

Asinh

`Asinh(`*x*`)`

Returns the inverse hyperbolic sine of *x*. `Asinh` raises inexact for all values except zero. `Asinh(-INF)` returns `-INF` and `Asinh(+INF)` returns `+INF`. `Asinh` raises underflow for *x* near zero.

x An integer or real number.

Atan

`Atan(`*x*`)`

Returns the inverse tangent in radians of *x*. It raises inexact for all values except zero. `Atan(-INF)` returns $-\pi/2$ and `Atan(+INF)` returns $\pi/2$. `Atan` returns values between $-\pi/2$ and $\pi/2$. It raises inexact for all nonzero *x*.

x An integer or real number.

Atan2

`Atan2(`*x*`,`*y*`)`

Returns the inverse tangent in radians of *x*/*y*. `Atan2` uses the algebraic signs of *x* and *y* to determine the quadrant of the result. It returns values between $-\pi$ and π. Its special cases are those of `Atan`.

x An integer or real number.

y An integer or real number.

Atanh

`Atanh(`*x*`)`

Returns the inverse hyperbolic of *x*. `Atanh` raises invalid for $x < -1$ or $x > 1$. It raises inexact for all valid arguments except zero and raises underflow near zero.and raises underflow for all finite *x* near zero. `Atanh(-1.0)` returns `-INF` and `Atan(+1.0)`returns `+INF`.

x An integer or real number.

CopySign

`CopySign(`*x*`,`*y*`)`

Returns the value with the magnitude of *x* and sign of *y*.

x An integer or real number.

y An integer or real number.

Note

The order of the parameters for `CopySign` matches the recommendation of the IEEE 754 floating point standard, which is opposite from the SANE `copysign` function. ◆

Cos

`Cos(`*x*`)`

Returns the cosine of the radian value *x*. `Cos` raises inexact for all finite arguments except zero. It is periodic with period 2π. `Cos` raises invalid when *x* is infinite.

x An integer or real number.

Cosh

`Cosh(`*x*`)`

Returns the hyperbolic cosine of *x*. `Cosh` raises inexact for all finite arguments except zero. `Cosh(-INF)` and `Cosh(+INF)` return `+INF`. `Cosh` raises overflow for finite values of large magnitude.

x An integer or real number.

Erf

`Erf(`*x*`)`

Returns $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$, the *error function* of *x*. `Erf` raises inexact for all arguments except zero. It raises underflow for arguments near zero. `Erf(-INF)` returns –1 and `Erf(+INF)` returns 1.

x An integer or real number.

Mathematically, the sum of `Erf(`*x*`)` and `Erfc(`*x*`)` should be 1, though the relationship may not hold when roundoff or underflow affect the results significantly.

Erfc

`Erfc(`*x*`)`

Returns $\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}dt$, the *complementary error function* of *x*. `Erfc` raises inexact for all arguments except zero. `Erfc(-INF)` returns 2 and `Erfc(+INF)` returns +0.

x An integer or real number.

Exp

`Exp(`*x*`)`

Returns e^x, the exponential of the *x*. `Exp` is inexact for all nonzero finite arguments. `Exp(-INF)` returns +0 and `Exp(+INF)` returns `+INF`. `Exp` raises

overflow for large, positive, finite *x*, and raises underflow for negative, finite *x* of large magnitude.

x An integer or real number.

Expm1

`Expm1(`*x*`)`

Returns $e^x - 1$, one less than the exponential of *x*. `Expm1` avoids loss of accuracy when *x* is nearly zero, and the difference is nearly zero. `Expm1` is inexact for all nonzero finite arguments. `Expm1(-INF)` returns –1 and `Expm1(+INF)` returns `+INF`. `Expm1` raises overflow for large, positive, finite *x*, and raises underflow for *x* near zero.

x An integer or real number.

Fabs

`Fabs(`*x*`)`

Returns the absolute value of *x*. It never raises an exception.

x An integer or real number.

FDim

`FDim(`*x*`,`*y*`)`

Returns the *positive difference* between its parameters:

- If *x* > *y*, `FDim` returns *x* – *y*
- Otherwise, if *x* <= *y*, `FDim` returns +0
- Otherwise, if *x* is a NaN, `FDim` returns *x*.
- Otherwise (*y* is a NaN), `FDim` returns *y*.

x An integer or real number.

y An integer or real number.

FMax

`FMax(`*x*`,`*y*`)`

Returns the maximum of its two parameters. NaN parameters are treated as missing data:

- If one parameter is a NaN and the other is a number, then the number is returned.
- Otherwise, if both are NaNs, then the first parameter is returned.

(This corresponds to the `max` function in FORTRAN.)

x An integer or real number.

y An integer or real number.

FMin

`FMin(`*x*`,`*y*`)`

Returns the minimum of its two parameters. NaN parameters are treated as missing data:

- If one parameter is a NaN and the other is a number, then the number is returned.
- Otherwise, if both are NaNs, then the first parameter is returned.

(This corresponds to the `min` function in FORTRAN.)

x An integer or real number.

y An integer or real number.

Fmod

`Fmod(`*x*`,`*y*`)`

Returns the remainder when *x* is divided by *y* to produce a truncated integral quotient. That is, `Fmod` returns the value *x* `-` *y*`*Trunc(`*x*`/`*y*`)`.

x An integer or real number.

y An integer or real number.

Gamma

`Gamma(`*x*`)`

Returns Γ(*x*), the gamma function applied to *x*. `Gamma` raises inexact for all non-integral *x*. It raises invalid for non-positive integral arguments *z*. `Gamma(`*p*`)` returns (*p*-1)! for positive, integral *p*, with 0! defined to be 1. `Gamma(+INF)` returns `+INF`. `Gamma` can raise overflow.

x An integer or real number.

Hypot

`Hypot(`*x*`,`*y*`)`

Returns the square root of the sum of the squares of *x* and *y*, avoiding the hazards of overflow and underflow when the arguments are large or tiny in magnitude but the result is within range.

x An integer or real number.

y An integer or real number.

IsFinite

`IsFinite(`*x*`)`

Returns `true` if *x* is finite; returns `nil` if *x* is infinite.

x An integer or real number.

IsNaN

`IsNaN(`*x*`)`

Returns `true` if *x* is a NaN; returns `nil` if *x* is a number.

x An integer or real number.

Note

Saying that *x* "is a NaN" and "is not a number" are not the same thing. A NaN is a non-numerical value in a numerical format; on the other hand, a string such as `"foo"` is not a number because it is not a numerical object. ◆

IsNormal

`IsNormal(`*x*`)`

Returns `true` if *x* is a normal number; returns `nil` if *x* is zero, subnormal, infinite, or a NaN.

x An integer or real number.

LessEqualOrGreater

`LessEqualOrGreater(`*x*`, `*y*`)`

Returns `true` if neither *x* nor *y* is a NaN, and therefore the two arguments are ordered; otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

LessOrGreater

`LessOrGreater(`*x*`, `*y*`)`

Returns `true` if either $x < y$ or $x > y$; otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

LGamma

`LGamma(`*x*`)`

Returns the natural logarithm of $\Gamma(x)$, the gamma function applied to *x*. `LGamma` raises inexact for all positive *x*. It raises invalid for negative or zero *x*. `LGamma(+INF)` returns `+INF`.

x An integer or real number.

Log

`Log(`*x*`)`

Returns the natural logarithm of *x*. `Log` raises inexact for positive, finite arguments except 1. `Log(0.0)` returns `-INF` and raises divide by zero. `Log(+INF)` returns `+INF`. `Log` raises invalid for $x < 0$.

x An integer or real number.

Logb

`Logb(`*x*`)`

Returns the integral value *k* such that $1 \leq |x| * 2^{-k} < 2$, when *x* is finite and nonzero. `Logb(0.0)` returns `-INF` and raises divide by zero. `Logb(-INF)` and `Logb(+INF)` return `+INF`.

Log1p

`Log1p(`*x*`)`

Returns the natural logarithm of `1+`*x*. While accurate for all arguments no less than –1, `Log1p` preserves accuracy when *x* is nearly zero—when computing `Log(1.0 +` *x) would* suffer from the mere addition of *x* to 1. `Log1p` raises inexact for all finite arguments greater than –1 except 0. It raises invalid for all *x* less than –1 and raises underflow for *x* near zero. `Log1p(-1.0)` returns `-INF` and raises divide by zero. `Log1p(+INF)` returns `+INF`.

x An integer or real number.

Log10

`Log10(`*x*`)`

Returns the logarithm base 10 of *x*. Because of the mathematical relationship $\log_{10}(x) = \log(x)/\log(10)$, `Log10` shares the computational properties of `Log`.

x An integer or real number.

NearbyInt

`NearbyInt(x)`

Returns *x* rounded to the nearest integral value. `NearbyInt` differs from `Rint` only in that it does not raise the inexact exception.

x An integer or real number.

Note
`NearbyInt` always rounds to nearest. ◆

NextAfterD

`NextAfterD(x,y)`

Returns the next representable number after *x* in the direction of *y*.

If *x* and *y* are equal, then the result is *x*. If either argument is a NaN, `NextAfterD` returns one of the NaN arguments. When *x* is finite but the result is infinite, `NextAfterD` raises overflow. When the result is zero or subnormal, `NextAfterD` raises underflow.

x An integer or real number.

y An integer or real number.

Pow

`Pow(x,y)`

Returns x^y. When $x < 0$, `Pow` raises invalid unless *y* is an integral value. It can raise inexact, overflow, underflow, and invalid.

x An integer or real number.

y An integer or real number.

RandomX

`RandomX(x)`

Returns a two-element array, based on the random seed *x*. The first element of the result is a pseudo-random number that is the result of the SANE

`randomx` function. The second element is the new seed returned by the `randomx` function. The result is an integral value between 0 and $2^{31} - 1$.

x An integer or real number.

Remainder

`Remainder(`*x*`,`*y*`)`

Returns the *exact* difference *x* – *n***y*, where *n* is a mathematical integer (as opposed to a NewtonScript integer—*n* may be thousands of bits wide) to *x* / *y* in the sense of rounding to nearest. The magnitude of the result is no greater than half the magnitude of *y*. When the result is zero, it has the sign of *x*. `Remainder` raises invalid when *y* is zero or *x* is infinite. It never raises overflow, underflow, or inexact.

x An integer or real number.

y An integer or real number.

RemQuo

`RemQuo(`*x*`,`*y*`)`

Returns a two-element array. The first element is `Remainder(`*x*`, `*y*`)`. The second element is the seven low-order bits of the quotient *x* / *y* rounded to the nearest integer and given the sign of the quotient.

x An integer or real number.

y An integer or real number.

Rint

`Rint(`*x*`)`

Is identical to `Nearbyint` except that it raises inexact when its result differs from *x*.

x An integer or real number.

RintToL

`RintToL(`*x*`)`

Returns an integer obtained by rounding *x* to an integral (real) value and then converting that value to an integer. `RintToL` raises inexact when its result differs in value from *x*. It raises invalid and returns an unspecified value when the rounded value of *x* cannot be represented exactly as an integer object.

x An integer or real number.

Note

`RintToL` always rounds to nearest. ◆

Round

`Round(`*x*`)`

Returns the integral real number obtained from *x* by adding 1/2 to *x* and truncating the result to the nearest integer toward 0. It raises inexact when the result differs from *x*.

x An integer or real number.

Scalb

`Scalb(`*x*`, `*k*`)`

Returns $x * 2^k$. `Scalb` avoids explicit computation of 2^k and so avoids the complications of overflow or underflow when 2^k is out of range but the result isn't. `Scalb` can raise overflow, underflow, and inexact. `Scalb` and `Logb` are related by the formula $1 \le$ `Scalb(`*x*`, RintToL(-Logb(`*x*`)))` < 2 for finite, nonzero *x*.

x An integer or real number.

y An integer.

SignBit

`SignBit(`*x*`)`

Returns a nonzero integer if the sign of *x* is negative; otherwise (the sign of *x* is positive), returns the integer 0.

x An integer or real number.

Signum

`Signum(`*x*`)`

Returns the integer value –1 if *x* < 0, 0 if *x* = 0, or 1 if *x* > 0. If *x* is an integer, `Signum` returns an integer; otherwise, if *x* is a real, `Signum` returns a real. If *x* is neither an integer nor a real, `Signum` throws the exception `kFramesErrNotANumber`.

x An integer or real number.

Sin

`Sin(`*x*`)`

Returns the sine of the radian value *x*. `Sin` raises inexact for all finite values except zero. It is periodic with period 2π. `Sin` raises invalid for infinite *x* and raises underflow for *x* near zero.

x An integer or real number.

Sinh

`Sinh(`*x*`)`

Returns the hyperbolic sine of *x*. `Sinh` raises inexact for all finite arguments except zero. `Sinh(-INF)` returns `-INF` and `Sinh(+INF)` returns `+INF`. `Sinh` raises overflow for large finite values and raises underflow near zero.

x An integer or real number.

Sqrt

`Sqrt(`*x*`)`

Returns the square root of *x*. It raises invalid for $x < 0$, and can raise inexact for positive *x*.

x An integer or real number.

Tan

`Tan(`*x*`)`

Returns the tangent of the radian value *x*. `Tan` raises inexact for all finite values except zero. It is periodic with period π. `Tan` raises invalid for infinite *x* and raises underflow for *x* near zero.

x An integer or real number.

Tanh

`Tanh(`*x*`)`

Returns the hyperbolic tangent of *x*. `Tanh` raises inexact for all finite arguments except zero. `Tanh(-INF)` returns –1 and `Tanh(+INF)` returns +1. `Tanh` raises overflow for large finite values and raises underflow near zero.

x An integer or real number.

Trunc

`Trunc(`*x*`)`

Returns the integral real number nearest to but no larger in magnitude than *x*.

x An integer or real number.

Unordered

`Unordered(`*x*`, `*y*`)`

Returns `true` if *x* and *y* satisfy none of $x < y$, $x = y$, or $x > y$ (because one or both of *x* and *y* are NaNs); if neither *x* nor *y* is a NaN, they satisfy one of the three order relations and `Unordered` returns `nil`.

x An integer or real number.

y An integer or real number.

UnorderedGreaterOrEqual

`UnorderedGreaterOrEqual(`*x*`, `*y*`)`

Returns `true` if *x* and *y* satisfy $x \geq y$ or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

UnorderedLessOrEqual

`UnorderedLessOrEqual(`*x*`, `*y*`)`

Returns `true` if *x* and *y* satisfy $x \leq y$ or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

UnorderedOrEqual

`UnorderedOrEqual(`*x*`, `*y*`)`

Returns `true` if *x* and *y* satisfy $x = y$ or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

UnorderedOrGreater

`UnorderedOrGreater(`*x*`, `*y*`)`

Returns `true` if *x* and *y* satisfy *x* > *y* or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

UnorderedOrLess

`UnorderedOrLess(`*x*`, `*y*`)`

Returns `true` if *x* and *y* satisfy *x* < *y* or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

Managing the Floating Point Environment

The floating point environment is a set of state variables maintained by the Newton system and the underlying processor. The environment contains information about which floating point exceptions have occurred. Floating point exceptions are distinct from NewtonScript exceptions. When floating point exceptions arise (for example, overflow arises when the sum of two huge numbers is too large to represent in the number system), the system raises an exception flag in the environment. Exception flags can be tested, cleared, or raised by functions in this section. Once raised, an exception flag remains raised until you clear it using calls from this section. The predefined

constants used to select the floating point exception flags are shown in Table 20-3.

Table 20-3 Floating point exceptions

Constant	Value	Meaning
`fe_Inexact`	`0x010`	inexact
`fe_DivByZero`	`0x002`	divide-by-zero
`fe_Underflow`	`0x008`	underflow
`fe_Overflow`	`0x004`	overflow
`fe_Invalid`	`0x001`	invalid
`fe_All_Except`	`0x01F`	all exceptions

You can refer to multiple exceptions in a single function invocation by forming the bitwise-OR of the predefined constants, using expressions like `Bor(Bor(fe_Invalid, fe_DivByZero), fe_Overflow)`.

NOTE

The representation of the floating point environment is implementation-dependent. Functions that manipulate the environment and its components do so without exposing their implementation. In particular, the floating point exception flags may or may not be implemented as single bits. ◆

The functions that manage the floating point environment are based on recommended numerical extensions to the ANSI C language. The recommendations for C include functions to test and alter the direction of rounding. Although the direction of rounding is determined by the environment on most systems, Newton systems based on the ARM family of processors determine the rounding direction on an instruction-by-instruction basis, so rounding is not determined by the environment.

You can pass the predefined constant `fe_Dfl_Env` to the functions `FeSetEnv` and `FeUpdateEnv`, which take an environment object as a

parameter. `Fe_Dfl_Env` indicates the default environment, in which all exception flags are clear.

FeClearExcept

`FeClearExcept(`*excepts*`)`

Clears the floating point exception flags indicated by *excepts*.

excepts The integer bitwise-OR of one or more floating point exceptions.

FeGetEnv

`FeGetEnv()`

Returns a data object representing the current floating point environment.

FeGetExcept

`FeGetExcept(`*excepts*`)`

Returns a data object representing the current state of the exception flags indicated by *excepts*.

excepts The integer bitwise-OR of one or more floating point exceptions.

Note

The representation of the exception flags is unspecified. ◆

FeHoldExcept

`FeHoldExcept()`

Returns a data object representing the current floating point environment, and clears the exception flags.

FeRaiseExcept

`FeRaiseExcept(`*excepts*`)`

Raises the floating point exception flags indicated by *excepts*.

excepts The integer bitwise-OR of one or more floating point exceptions.

Note

Because floating point exceptions are not tied to the general NewtonScript exception-handling mechanism, raising a flag merely sets an internal variable; raising a flag will not alter the flow of control. ◆

FeSetEnv

`FeSetEnv(`*envObj*`)`

Installs the floating point environment represented by the object *envObj*.

envObj Either the predefined constant `fe_Dfl_Env` or an object returned by a call to `FeGetEnv` or `FeHoldExcept`.

FeSetExcept

`FeSetExcept(`*flagObj*`, `*excepts*`)`

The parameter *flagObj* is an object containing an implementation-dependent representation of one or more floating point exception flags; *flagObj* must have been set by a previous call to `FeGetExcept`. `FeSetExcept` alters the current environment so that those floating point exception flags indicated by *excepts* match the corresponding values in *flagObj*.

flagObj An object (returned by a previous call to `FeGetExcept`) containing a representation of one or more floating point exception flags.

excepts The integer bitwise-OR of one or more floating point exceptions.

This function does not raise exceptions; it just alters the state of the flags.

FeTestExcept

`FeTestExcept(`*excepts*`)`

Returns the bitwise-OR of the floating point exceptions indicated by *excepts* whose flags are raised in the current environment.

excepts	The integer bitwise-OR of one or more floating point exceptions.

FeUpdateEnv

`FeUpdateEnv(`*envObj*`)`

Saves the state of the current exception flags, installs the environment represented by *envObj*, and then re-raises the saved exceptions.

envObj	Either the predefined constant `fe_Dfl_Env` or an object returned by a call to `FeGetEnv` or `FeHoldExcept`.

You can use `FeUpdateEnv` in conjunction with `FeHoldExcept` to write functions which hide spurious exceptions from their callers:

```
func() begin
   savedEnv := FeHoldExcept(); // clears flags
   result := ...; // ecomputation in which underflow and
                  // divide by zero are benign
   FeClearExcept(BOR(fe_Underflow, fe_DivByZero));
   FeUpdateEnv(savedEnv); // merge old flags with new
   return result
end
```

Financial Functions

These functions perform financial calculations.

Annuity

`Annuity(r, n)`

Returns the value of the financial formula $\frac{1-(1+r)^{-n}}{r}$. When *r* is the periodic interest rate and *n* the number of periods, *p*`*Annuity(r, n)` is the *present value* of a series of *n* periodic payments of size *p*. `Annuity` is robust over the entire range of *r* and *n*, whether financially meaningful or not.

`Annuity` raises invalid for $r < -1$. When $r = -1$:

- `Annuity(-1, n)` returns –1 for $n < 0$.
- `Annuity(-1, 0)` returns 0.
- `Annuity(-1, n)` returns `+INF` and raises divide by zero for $n > 0$.

Otherwise, $r > -1$. When *r* is nonzero, `Annuity(r, 0)` returns *r*; otherwise, `Annuity(0, n)` returns *n*. `Annuity` raises inexact in all other cases, and can raise overflow or underflow.

r	An integer or real number.
n	An integer or real number.

Compound

`Compound(r, n)`

Returns the value of the financial formula $(1+r)^n$. When *r* is the periodic interest rate and *n* the number of periods, *P*`*Compound(r, n)` is the *future value* of a principal amount *P*. `Compound` is robust over the entire range of *r* and *n*, whether financially meaningful or not.

`Compound` raises invalid for $r < -1$. When $r = -1$:

- `Compound(-1, n)` returns `+INF` and raises divide by zero for $n < 0$.
- `Compound(-1, 0)` returns 1.
- `Compound(-1, n)` returns +0 for $n > 0$.

Otherwise, $r > 0$. `Compound(`*r*`, 0)` returns 1; `Compound(0,` *n*`)` raises invalid when *n* is infinite. `Compound` can raise inexact, overflow or underflow.

r	An integer or real number.
n	An integer or real number.

GetExchangeRate

`GetExchangeRate(`*country1*`,` *country2*`)`

Returns the currency exchange rate between two countries as a floating point number. This function first checks for an updated rate stored in the System soup and then checks for the rate stored in ROM. This function returns `nil` if it can't find the rate in either place.

country1	A symbol identifying a country.
country2	A symbol identifying a country.

Here is an example:

```
rate := GetExchangeRate('USA, 'Japan);
```

SetExchangeRate

`SetExchangeRate(`*country1*`,` *country2*`,` *rate*`)`

Saves the currency exchange rate between any two countries as a floating point number in the System soup. Subsequent calls to `GetExchangeRate` will return this value instead of the original value stored in ROM.

country1	A symbol identifying a country.
country2	A symbol identifying a country.
rate	The currency exchange rate between *country1* and *country2*.

Here is an example:

```
SetExchangeRate('USA, 'Japan, 87.5);
```

GetUpdatedExchangeRates

```
GetUpdatedExchangeRates()
```

Returns a frame containing all of the updated currency exchange rates that have been stored in the System soup by use of the `SetExchangeRate` function. The `GetUpdatedExchangeRates` function is called by both `GetExchangeRate` and `SetExchangeRate`. Normally, you do not need to call this function unless you want to retrieve all of the updated exchange rates together.

Exception Functions

These functions are used to raise and handle NewtonScript exceptions in an application. For more information about exception handling and how to use these functions, refer to the chapter “Flow of Control” in *The NewtonScript Programming Language*. For a list of system exceptions, see Appendix A, “Errors.”

The section “Managing the Floating Point Environment” beginning on page 20-79 describes some functions that deal with floating-point exceptions, which are not related to NewtonScript exceptions.

Throw

`Throw(`*name*`, `*data*`)`

Raises an exception and creates an exception frame with the specified name and data.

name	An exception symbol that names the exception being raised.
data	The data for the exception. The possible values for this parameter depend on the composition of *name* and are shown inTable 20-4.

Table 20-4 Exception frame data slot name and contents

Exception symbol	Slot name	Slot contents
contains part with prefix `type.ref`	`data`	a data object, which can be any NewtonScript object
contains part with prefix `evt.ex.ms|`	`message`	a message string
any other	`error`	an integer error code

See the chapter “Flow of Control” in *The NewtonScript Programming Language* for more information on `Throw`.

Rethrow

```
Rethrow()
```

Reraises the current exception to allow the next enclosing `Try` statement an opportunity to handle it. `Rethrow` throws the current exception again, passing along the same parameters are were passed with the original call to the `Throw` function. This functionality lets you pass control from within an exception handler to the next enclosing `Try` statement.

IMPORTANT

You can call the `Rethrow` function only from within the dynamic extent of an `onexception` clause. ◆

CurrentException

```
CurrentException()
```

During exception processing (that is, inside the dynamic extent of an `onexception` block), returns the frame that is associated with the current exception. You can examine the frame returned by `CurrentException` to determine what kind of exception you are handling. For example, you can call the `HasSlot` function to determine if the frame contains a slot named

`error`, and take appropriate action thereafter. (The format of the frame depends on the exception, but it always contains a *name* slot with the exception symbol.)

`CurrentException` gives a meaningful response only from within the dynamic extent of an `onexception` clause. Outside the extent of `onexception`, it returns `nil`.

RethrowWithUserMessage

`RethrowWithUserMessage(`*userTitle*`,`*userMessage*`,`*override*`)`

Unhandled exceptions currently end up displaying a `Notify` dialog whose contents are sometimes not very informative to the user. This function allows you to catch an exception, specify a more descriptive message, and then rethrow the exception. If it is then unhandled, the system will use the *userTitle* and *userMessage* in the `Notify` dialog.

userTitle	A string used as the title of the `Notify` dialog.
userMessage	A string used as the body text of the `Notify` dialog
override	If the exception has already been annotated with a title and a message, this flag controls whether or not you want to override the existing annotations. Set this slot to non-`nil` to override any existing annotations, or `nil` to preserve them.

This function does not return.

If the exception has a `type.ref` part, the *userTitle* and *userMessage* are added to the existing data. Otherwise, the exception that is rethrown is changed to have a `type.ref` part. For example, an exception named `|evt.ex.bozo|` becomes `|evt.ex.bozo;type.ref|`, and the error is put into the `error` slot of the data frame. Because this change adds an exception part—leaving the existing ones intact—it shouldn't interfere with other `try` blocks looking for the exception (unless they make dangerous assumptions about the format of the exception frame).

Note

Exceptions of the type `|evt.ex.msg|` are changed to `|evt.ex;type.ref|`. You shouldn't use exceptions of the type `|evt.ex.msg|` in final code anyway—they're only for debugging. ◆

Message Sending Functions

These functions send messages or execute functions.

Apply

`Apply(`*function*`, `*parameterArray*`)`

Calls a function, passing the supplied parameters. The `Apply` function returns the return value of the function it called.

function	The function to call.
parameterArray	An array of parameters to be passed to the function. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).

`Apply` respects the environment of the function object it is passed. Using `Apply` is similar to using the NewtonScript `call` statement.

`Apply` is useful when you want to call a function, but don't know until run time the number of parameters it takes. If you do know ahead of time the number of parameters the function takes, then you can use the NewtonScript `call` statement to call the function.

Here's an example of using this function in the Inspector:

```
f:=func(x,y) x*y;
Apply(f, [10,2]);
#50      20
```

The `Apply` call is equivalent to:

```
f(10,2);
```

IsHalting

`IsHalting(`*functionObject*`, `*args*`)`

Returns non-`nil` if function object will return. `IsHalting` can be used to test a function object before calling it with the specified arguments. It does not actually call the function object; instead; it determines if it will ever return a value (as opposed to getting into an infinite loop). If the function will throw an exception, it returns the symbol `'throws`.

functionObject	The function object you want to test.
args	Array of arguments for the function object.

Perform

`Perform(`*frame*`, `*message*`, `*parameterArray*`)`

Sends a message to a frame; that is, a method with the name of the message is executed in the frame. Both parent and proto inheritance are used to search for the method if it does not exist in the frame. If the method is not found, an exception is thrown.

frame	The frame to which to send the message.
message	A symbol naming the message to send.
parameterArray	An array of parameters to be passed along with the message. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).

The `Perform` function returns the return value of the message it sent.

Note that the method named by *message* is executed in the context of *frame*, not in the context of the frame from within which `Perform` is called.

The `Perform` function is useful when you want to send a message, but you don't know until run time the name of the message or the number of parameters it takes. If you do know these things ahead of time, then you can just use the standard NewtonScript message sending syntax.

For variations of the `Perform` function, see `PerformIfDefined`, `ProtoPerform`, and `ProtoPerformIfDefined`.

Here's an example of using this function in the Inspector:

```
f:={multiply: func(x,y) x*y};
perform(f, 'multiply, [10,2]);
#50      20
```

Note that

```
f:multiply(10,2)
```

is equivalent to

```
Perform(f, 'multiply,[10,2])
```

PerformIfDefined

`PerformIfDefined(`*receiver*`,` *message*`,` *paramArray*`)`

Sends a message to a frame; that is, a method with the name of the message is executed in the frame. Both parent and proto inheritance are used to search for the method if it does not exist in the frame. If the method is not found, an exception is not thrown.

receiver	The frame to which you want the message sent.
message	A symbol that is the name of the message to send to *receiver.*
paramArray	An array of parameters to be passed with the *message.* You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).

This function returns the return value of the message it sent. If the method is not found, this function returns `nil`.

Contrast this function with `Perform` (page 20-91), which is exactly the same, except that `Perform` throws an exception if the method is not found.

Also, contrast this function with `ProtoPerform` and `ProtoPerformIfDefined` (page 20-93), which search only the proto chain for the method.

ProtoPerform

`ProtoPerform(`*receiver*`,` *message*`,` *paramArray*`)`

Sends a message to a frame; that is, a method with the name of the message is executed in the frame. Only proto inheritance is used to search for the method if it does not exist in the frame. If the method is not found, an exception is thrown.

receiver	The frame to which you want the message sent.
message	A symbol that is the name of the message to send to *receiver*.
paramArray	An array of parameters to be passed with the *message*. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).

This function returns the return value of the message it sent.

Contrast this function with `Perform` (page 20-91), which is exactly the same, except that `Perform` searches both the parent and proto chains for the method.

Also, contrast this function with `PerformIfDefined` (page 20-92) and `ProtoPerformIfDefined` , which do not throw exceptions if the method is not found.

ProtoPerformIfDefined

`ProtoPerformIfDefined(`*receiver*`,` *message*`,` *paramArray*`)`

Sends a message to a frame; that is, a method with the name of the message is executed in the frame. Only proto inheritance is used to search for the

method if it does not exist in the frame. If the method is not found, an exception is not thrown.

receiver	The frame to which you want the message sent.
message	A symbol that is the name of the message to send to *receiver*.
paramArray	An array of parameters to be passed with the *message*. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).

This function returns the return value of the message it sent. If the method is not found, this function returns `nil`.

Contrast this function with `PerformIfDefined` (page 20-92), which is exactly the same, except that `PerformIfDefined` searches both the parent and proto chains for the method.

Also, contrast this function with `Perform` (page 20-91) and `ProtoPerform` (page 20-93), which search both the parent and proto chains for the method.

Deferred Message Sending Functions

This section describes utility functions for delayed and deferred actions.

AddDeferredCall

`AddDeferredCall(`*functionObject*`,`*paramArray*`)`

Queues a function object to be executed the next time the system main event loop is executed.

functionObject	The function object to be executed.
paramArray	An array of parameters to be passed to the *functionObject*. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).

This function always returns non-`nil`.

You use this function so that the currently executing method (within which this function is called) has a chance to finish its execution and return up the call chain before the deferred function object is called. The function object is called before the next event is handled.

The `AddDeferredCall` function puts a type of event in a first-in-first-out queue which also contains user actions. Normally, this means that if you call `AddDeferredCall` and then the user clicks, the deferred function call will occur first. However, just because the user takes an action does not mean that it is processed immediately. Different components of the Newton operating system are processed in separate threads and thus you cannot rely on events being processed in a predictable order.

Note also that `ViewIdleScript` methods can be called several times before deferred function calls are executed. Suppose that you have, for example, some networking code that initializes a view. Since this is networking code, you have a `ViewIdleScript` method that's being called every 200 milliseconds, to look for new names on the network. You then have the view initialized by a deferred function call. The `ViewIdleScript` method may be called two or three times before the deferred function call is made.

AddDelayedCall

`AddDelayedCall(`*functionObject*`,`*paramArray*`,`*delay*`)`

Schedules a function object to be executed after a specific delay.

functionObject	The function object to be executed.
paramArray	An array of parameters to be passed to the *functionObject*. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).
delay	The time in milliseconds after which the *functionObject* is executed

This function always returns non-`nil`.

AddDeferredSend

`AddDeferredSend(`*receiver*`,` *message*`,` *paramArray*`)`

Queues a message to be sent the next time the system main event loop is executed.

receiver	The frame to which you want the message sent.
message	A symbol that is the name of the message to send to *receiver*.
paramArray	An array of parameters to be passed with the *message*. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).

This function always returns non-`nil`.

You use this function so that the currently executing method (within which this function is called) has a chance to finish its execution and return up the call chain before the deferred message is sent. The message is sent before the next event is handled.

The `AddDeferredSend` function puts a type of event in a first-in-first-out queue which also contains user actions. Normally, this means that if you call `AddDeferredSend` and then the user clicks, the deferred message send will occur first. However, just because the user takes an action does not mean that it is processed immediately. Different components of the Newton operating system are processed in separate threads and thus you cannot rely on events being processed in a predictable order.

Note also that `ViewIdleScript` methods can be called several times before deferred message sends are executed. Suppose that you have, for example, some networking code that initializes a view. Since this is networking code, you have a `ViewIdleScript` method that's being called every 200 milliseconds, to look for new names on the network. You then have the view initialized by a deferred message send. The `ViewIdleScript` method may be called two or three times before the deferred message send is made.

AddDelayedSend

`AddDelayedSend(`*receiver*`,` *message*`,` *paramArray*`,` *delay*`)`

Schedules a message to be sent after a specific delay.

receiver	The frame to which you want the message sent.
message	A symbol that is the name of the message to send to *receiver*.
paramArray	An array of parameters to be passed with the *message*. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).
delay	The time in milliseconds after which the *message* is sent.

This function always returns non-`nil`.

AddProcrastinatedCall

`AddProcrastinatedCall(`*funcSymbol*`,` *functionObject*`,` *paramArray*`,` *delay*`)`

Queues a function object to be executed at a later time.

funcSymbol	A unique symbol identifying the function object to be executed. You should append your developer signature to form this symbol, to ensure that it is unique in the system.
functionObject	The function object to be executed at a later time.
paramArray	An array of parameters to be passed to the *functionObject*. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).
delay	The approximate time in milliseconds after which the *functionObject* is executed. Specify zero to cause the function to be executed the next time the system main event loop is executed. Zero does not cause immediate execution of the function.

The return value of this function is undefined.

If, prior to executing *functionObject*, another function object with the same identifying *funcSymbol* is queued, the originally queued function is cancelled. Similarly, the execution of this second function can be preempted by yet another queued function with the same *funcSymbol*, and so on.

This function is useful for preventing lengthy operations from happening multiple times in a row when a single operation would suffice. For example, you might call `EntryChange` in several places in your code, to flush an entry to a soup. However, you really need to call `EntryChange` only once, after the last slot is changed. You could use a function call like this to help prevent multiple calls to `EntryChange` from happening one after another:

```
AddProcrastinatedCall('|flush:mySignature|,
                      functions.EntryChange, [entry], 0);
```

AddProcrastinatedSend

`AddProcrastinatedSend(`*msgSymbol*`,`*receiver*`,`*message*`,`*paramArray*`,` *delay*`)`

Queues a message to be sent at a later time.

msgSymbol	A unique symbol identifying the message to be sent. You should append your developer signature to form this symbol, to ensure that it is unique in the system.
receiver	The frame to which you want the message sent.
message	A symbol that is the name of the message to send to *receiver*.
paramArray	An array of parameters to be passed with the *message*. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).
delay	The approximate time in milliseconds after which the *message* is sent. Specify zero to cause the function to be executed the next time the system main event loop is executed. Zero does not cause immediate execution of the function.

The return value of this function is undefined.

If, prior to sending *message*, another message with the same identifying *msgSymbol* is queued, the originally queued message is cancelled. Similarly, the sending of this second message can be preempted by yet another queued message with the same *msgSymbol*, and so on.

This function is useful for preventing lengthy operations from happening multiple times in a row when a single operation would suffice. Here is an example of calling this function:

```
AddProcrastinatedSend('|update:mySignature|, base,
'updateViews, nil, 0);
```

Data Extraction Functions

These functions are used to extract chunks of data out of other objects of various types.

All integers are stuffed and extracted in two's-complement **big-endian** form. In this form, byte 0 is the most significant byte, as found on the Newton and Macintosh. The opposite of this is **little-endian,** where byte 0 is least significant byte, as found on Intel-based computers. For example, the number 0x12345678 is stored as:

big-endian 12 34 56 78

little-endian 78 56 34 12

All Unicode conversions use the Macintosh extended character set for codes greater than or equal to 128.

ExtractByte

`ExtractByte(`*data*`, `*offset*`)`

Returns one signed byte from the given offset.

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

For example:

```
ExtractByte("\u12345678",0);
#3FC      255
```

ExtractBytes

`ExtractBytes(`*data*`, `*offset*`, `*length*`, `*class*`)`

Returns a binary object of class *class* containing *length* bytes of data starting at *offset* within *data.*

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.
length	An integer giving the number of bytes to extract.
class	A symbol specifying the class of the return value.

ExtractChar

`ExtractChar(`*data*`, `*offset*`)`

Returns a character object of the character at the given *offset* in the *data.*

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

Gets one byte at the specified offset, converts it to Unicode and returns the character it makes from it.

For example:

```
ExtractChar("\uFFFFFFFF",0);
//$\u02C results from a ASCII to UNICODE conversion.
#2C76     $\u02C7
//Note $a is at offset 1 in a Unicode string
ExtractChar("abc",0);
#6        $\00
ExtractChar("abc",1);
#616      $a
```

ExtractLong

`ExtractLong(`*data*`, `*offset*`)`

Returns an integer object of the low 29 bits of an unsigned long at the given offset, right-justified (that is, the low 29 bits of a four-byte value).

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

Reads four bytes at the specified offset, but ignores the high-order bits (first two). Returns a 30 bit signed value

```
ExtractLong("\uFFFFFFFF",0);
#FFFFFFFC -1
ExtractLong("\uC0000007",0);
#1C       7
```

ExtractXLong

`ExtractXLong(`*data*`, `*offset*`)`

Returns an integer object of the high 29 bits of an unsigned long at the given offset, right-justified (that is, the high 29 bits of a four-byte value).

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

For example:

```
ExtractXLong("\u0000000F",0);
#4        1
```

ExtractWord

`ExtractWord(`*data*`, `*offset*`)`

Returns an two-byte signed integer object from the given offset.

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

For example:

```
ExtractWord("\uFFFFFFFF",0);
#FFFFFFFC -1
//if you want unsigned use:
band(ExtractWord(-),0xFFFF);
#40004    65535
```

ExtractCString

`ExtractCString(`*data*`, `*offset*`)`

Returns a Unicode string object derived from the null-terminated C-style string at the given offset.

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

ExtractPString

`ExtractPString(`*data*`, `*offset*`)`

Returns a Unicode string object derived from the Pascal-style string (a length byte followed by text) at the given offset.

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

ExtractUniChar

`ExtractUniChar(`*data*`, `*offset*`)`

Gets two bytes at the specified offset and returns the Unicode character represented by those bytes.

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

For example:

```
ExtractUniChar("abc",0);
#616      $a
```

Data Stuffing Functions

These functions are used to stuff chunks of data into objects of various types.

All integers are stuffed in two's-complement **big-endian** form. For a discussion of this, see "Data Extraction Functions" on page 20-99.

WARNING

It is important that the destination for the data stuffing functions is large enough to hold the data being stuffed. If the destination is not large enough, the NewtonScript heap may become corrupted. Be sure to take into account the offset. Here is a formula you can use:

Length(*destObj*) – *offset* >= *size of stuffed data*

In this formula, *destObj* is the destination object and *offset* is the position within the destination object where the data is to be stuffed. ◆

StuffByte

`StuffByte(`*obj*`, `*offset*`, `*toInsert*`)`

Writes the low order byte of *toInsert*, at the specified *offset* in *obj*.

obj	A binary object into which data is to be stuffed.
offset	The position in *obj* at which stuffing is to begin.
toInsert	The data to be stuffed in *obj*.

For example:

```
x := "\u00000000";
StuffByte(x,0,-1);
x[0]
#FF006    $\uFF00
```

```
x := "\u00000000";
StuffByte(x,0,0xFF);
x[0]
#FF006    $\uFF00
```

StuffChar

`StuffChar(`*obj*`, `*offset*`, `*toInsert*`)`

Stuffs one byte into *obj* at the specified offset.

obj	A binary object into which data is to be stuffed.
offset	The position in *obj* at which stuffing is to begin.
toInsert	A character or integer to be stuffed in *obj*. You pass it a two byte Unicode value as *toInsert*. The function makes a one-byte character from that value and stuffs the one-byte character.

This accepts a character or integer as its third parameter, *toInsert*:

- If *toInsert*: is an integer: writes the low byte of *toInsert*.
- *If toInsert*: is a character: converts from Unicode and writes a byte.

For example:

```
x := "\u00000000";
StuffChar(x,1,Ord($Z));
x[0]
#5A6      $Z

x := "\u00000000";
StuffChar(x,1,-1);
x[0]
#1A6      $\1A

ExtractByte(x,1)
#68       26
```

```
ExtractByte(x,0)
#0        0
```

StuffCString

`StuffCString(`*obj*`, `*offset*`, `*aString*`)`

Converts a Newton Unicode string into a null-terminated C-style string and stuffs it at the given offset into a binary object.

obj	A binary object into which data is to be stuffed.
offset	The position in *obj* at which stuffing is to begin.
aString	A Unicode string to be stuffed into *obj*.

The string *aString* is converted into ASCII format using Macintosh roman string encoding. It is then stuffed into *obj*, beginning at the byte offset *offset*. It is followed by a null byte terminator.

This function throws an exception if *aString* will not fit into *obj* beginning at the given offset, or if the offset is negative. The length of *obj* will not be altered.

StuffLong

`StuffLong(`*obj*`, `*offset*`, `*toInsert*`)`

Writes four bytes at the specified offset using the 30 bit signed value you pass it as the third parameter, and sign extends it to 32 bytes.

obj	A binary object into which data is to be stuffed.
offset	The position in *obj* at which stuffing is to begin.
toInsert	The data to be stuffed in *obj*.

For example:

```
x := "\u00000000";
StuffLong(x,0,-1);
x[0]
#FFFF6    $\uFFFF
```

```
x[1]
#FFFF6    $\uFFFF
x := "\u00000000";
StuffLong(x,0,0x3FFFFFFA);
x[0]
#FFFF6    $\uFFFF
x[1]
#FFFA6    $\uFFFA
```

StuffPString

`StuffPString(`*obj*`, `*offset*`, `*aString*`)`

Converts a Newton Unicode string into a Pascal-style string (a length byte followed by text) and stuffs it at the given offset into a binary object.

object	A binary object into which data is to be stuffed.
offset	The position in *obj* at which stuffing is to begin.
aString	A Unicode string to be stuffed into *obj*. This string must be no longer than 255 characters.

The string *aString* is converted into ASCII format using Macintosh roman string encoding. Then a length byte followed by the string is stuffed into *obj*, beginning at the byte offset *offset*. The length byte indicates the number of characters in the string.

This function throws an exception if *aString* will not fit into *obj* beginning at the given offset, or if the offset is negative. The length of *obj* will not be altered.

StuffUniChar

`StuffUniChar(`*obj*`, `*offset*`, `*toInsert*`)`

Stuffs the two-byte Unicode encoding for the character indicated by *toInsert* into *obj* at the specified offset.

obj	A binary object into which data is to be stuffed.
offset	The position in *obj* at which stuffing is to begin.
toInsert	A character or integer to be stuffed in *obj*.

For example:

```
x := "\u00000000";
StuffUniChar(x,0,"\uF00F"[0]);
x[0]
#F00F6    $\uF00F

x := "\u00000000";
StuffUniChar(x,0,0x0AA0);
x[0]
#AA06     $\u0AA0
```

StuffWord

`StuffWord(`*obj*`, `*offset*`, `*toInsert*`)`

Writes the low order two bytes of *toInsert* at the specified offset.

obj	A binary object into which data is to be stuffed.
offset	The position in *obj* at which stuffing is to begin.
toInsert	The data to be stuffed in *obj*.

For example:

```
x := "\u00000000";
StuffWord(x,0,0x3FFF1234);
x[0]
#12346    $\u1234
```

```
x := "\u00000000";
StuffWord(x,0,-1);
x[0]
#FFFF6    $\uFFFF
```

Getting and Setting Global Variables and Functions

These functions get, set and test for the existence of global variables and functions.

GetGlobalFn

`GetGlobalFn(`*symbol*`)`

Returns a global function. If the function is not found, `nil` is returned.

symbol A symbol naming the global function you want to get.

GetGlobalVar

`GetGlobalVar(`*symbol*`)`

Returns the value of a slot in the system globals frame. If the slot is not found, `nil` is returned.

symbol A symbol naming the global variable whose value you want to get.

GlobalFnExists

`GlobalFnExists(`*symbol*`)`

Returns non-`nil` if the global function identified by *symbol* exists, otherwise returns `nil`.

symbol A symbol naming the global function whose existence you want to check.

GlobalVarExists

`GlobalVarExists(`*symbol*`)`

Returns non-`nil` if the global variable identified by *symbol* exists, otherwise returns `nil`.

symbol A symbol naming the global variable whose existence you want to check.

DefGlobalFn

`DefGlobalFn(`*symbol*`,` *function*`)`

Defines a global function. The symbol identifying the function is returned.

symbol A symbol naming the global function you want to define. To avoid naming conflicts with other global functions, you should choose a name that includes your `appSymbol`, which includes the developer signature you have registered with Newton DTS.

function A function object.

Note that the global function is destroyed if the system is reset.

It is very important to remove any global functions created by your application when your application is removed. You can do this with `UnDefGlobalFn` in the application `RemoveScript` function.

IMPORTANT

Do not create global functions unless it is absolutely necessary. Global functions occupy NewtonScript heap space. They can conflict with system global functions and other applications' global functions. In most cases, you can use methods in your application base view instead of global functions. ◆

DefGlobalVar

`DefGlobalVar(`*symbol*`, `*value*`)`

Defines a global variable—that is, a slot in the system globals frame. The value of the variable is returned.

symbol A symbol naming the global variable you want to define. To avoid naming conflicts with other globals, you should choose a name that includes your `appSymbol`, which includes the developer signature you have registered with Newton DTS.

value The value you want to assign to the global variable.

The system ensures that the object created exists entirely in internal RAM (it calls `EnsureInternal` on the object identified by *symbol*. Note that the global variable is destroyed if the system is reset.

It is very important to remove any globals created by your application when your application is removed. You can do this with `UnDefGlobalVar` in the application `RemoveScript` function.

IMPORTANT

Do not create global variables unless it is absolutely necessary. Global variables occupy NewtonScript heap space. They can conflict with system globals and other applications' globals. In most cases, you can put any global data that you need in your application base view or in a soup. ◆

UnDefGlobalFn

`UnDefGlobalFn(`*symbol*`)`

Removes a global function you previously defined. This function returns `nil`.

symbol A symbol naming the global function you want to remove.

UnDefGlobalVar

`UnDefGlobalVar(`*symbol*`)`

Removes a global variable you previously defined. This function returns `nil`.

symbol A symbol naming the global variable you want to remove.

Miscellaneous Functions

These functions send messages or execute functions.

AddMemoryItem

`AddMemoryItem(`*memSymbol*`, `*value*`)`

Adds a memorized value that can be any string that you want to pass as the second parameter to this function. Unlike `AddMemoryItemUnique` (page 20-113), this function does not test for uniqueness.

memSymbol An identifier symbol that names the memorized value that can be retrieved later with `GetMemoryItems` (page 20-121). You must use a symbol that has your developer signature appended to ensure that the symbol is unique to the system.

value The string to add to the memorized items.

For example, if you call

```
AddMemoryItem('|widget:MYSIG|, "Frazzle Wrench") ;
```

you can later call

```
GetMemoryItems('|widget:MYSIG|) ;
```

to get

```
[{item:"Frazzle Wrench"}]
```

This function returns an array of memory items that have been added under that *memSymbol*.

AddMemoryItemUnique

`AddMemoryItemUnique(`*memorySlot*`, `*value*`, `*testFunc*`)`

Adds a memorized value which can be any object that you want to pass as the second parameter to this function. For example, when used with a picker, the second parameter would usually be an object from the picker.

memSymbol	An identifier symbol that names the memorized value that can be retrieved later with `GetMemoryItems`. You must use a symbol that has your developer signature appended to ensure that the symbol is unique to the system. See `AddMemoryItem` (page 20-112) for an example.
value	The object to add to the memorized items.
testFunc	A function object that must accept two parameters, which are two memorized values. The system will call this function object and compare the memorized values returning non-`nil` if the values are equivalent and `nil` otherwise. If you pass `nil` for the *testFunc* parameter, this function will behave like `AddMemoryItem`; that is, the item is added even if it's not unique.

BinEqual

`BinEqual(`*a*`, `*b*`)`

a	A binary object
b	A binary object

Compares two binary objects' data as raw bytes. Returns non-`nil` if they are identical.

BinaryMunger

`BinaryMunger(` *dst*`,` *dstStart*`,` *dstCount*`,` *src*`,` *srcStart*`,` *srcCount* `)`

Replaces bytes in *dst* using bytes from *src* and returns *dst* after munging is complete. This function is destructive to *dst*.

dst	A value to be changed.
dstStart	The starting position in *dst*.
dstCount	The number of bytes to be replaced *in dst*. You can specify `nil` for *dstCount* to go to the end of *dst*.
src	A value. Can be `nil` to simply delete the contents of *dst*.
srcStart	The starting position in the source binary from which to begin taking elements to place into the destination binary.
srcCount	The number of bytes to use from the source binary. You can specify `nil` to go to the end of the source binary.

Bytes are numbered counting from zero.

Chr

`Chr(`*integer*`)`

Converts a decimal integer to its Unicode character equivalent.

integer	An integer.

Here is an example:

```
chr(65)
$A
```

Compile

`Compile(`*string*`)`

Compiles an expression sequence and returns a function that evaluates it.

string	The expression to compile.

Here are two examples. Note that, in the first example, x is a local variable.

```
compile("x:= {a:self.b, b:1234}")
#440F711  <CodeBlock, 0 args #440F711>
f:=compile("2+2")
f();
#440F712 4
```

Note

All characters used in NewtonScript code must be 7-bit ASCII. This usually is no problem, but can create problems with `Compile` in certain situations. Suppose you tried this call:

```
Compile ("blah, blah, blah,  \u0F0F\u")
```

The Unicode character is not a 7-bit character, it is 16 bits. Therefore, you get an error. (The `\u` switch turns on Unicode character mode.) You should do this instead:

```
Compile ("blah, blah, blah,  \\u0F0F\\u")
```

The backslash escape character preceding the `\u` prevents Unicode mode from being turned on for the compile. (The `\u` is read simply as the string `"\u"` instead of the Unicode switch.)

Note, also, that:

```
compile("func()...")
```

returns a function that constructs the function. The environment is captured when the function constructor is executed:

```
f := compile("func()b");
x := {a:f, b:0};
```

```
g:=x:a();
#440F713 <CodeBlock, 0 args #440F711>
```

Executing the function construction captures the message environment with x as receiver.

```
g();
#440F714 0
```

So now it can find b. ◆

Gestalt

`Gestalt(`*selector*`)`

Returns information about the Newton System depending on the value of the selector parameter.

selector A constant that specifies the type of information that will be returned on the system. The `kGestalt_SystemInfo` is the only constant currently supported.

`kGestalt_System`, which has a value of 0x1000003, returns a frame with the following slots:

`manufacturer`
A decimal integer indicating the manufacturer of the Newton Device.

`machineType`
A decimal integer indicating the hardware type this ROM was built for.

`ROMStage`
A decimal integer indicating the language (English, German, French) and the stage of the ROM (alpha, beta, final)

`ROMVersion`
A decimal integer indicating the major and minor ROM version numbers. The

major number is in front of the decimal place ; the minor number is after it.

Note

The `Machinetype`, `ROMStage` and `ROMVersion` slots provide internal configuration information and should not be relied on. ◆

`screenWidth`

An integer representing the width of the screen in pixels. The width takes into account the current screen orientation.

For example, on the MessagePad 120, because the screen width is 240 and the screen height is 320, when in portrait orientation, `Gestalt` will return a width of 240. If the screen is then rotated, `Gestalt` will return a width of 320.

`screenHeight`

An integer representing the height of the screen in pixels.

`screenResolutionx`

An integer representing the number of horizontal pixels per inch. For screens that have square pixels, `screenResolutionx` equals `screenResolutiony`. On the MessagePad 120, for example, both `screenResolutionx` and `screenResolutiony` equals 85.

`screenResolutiony`

An integer representing the number of vertical pixels per inch.

`screenDepth`

The bit depth of the LCD screen. For the MessagePad 120, the LCD supports a monochrome screen depth of 1.

patchVersion

Returns 0 on an unpatched newton and non-zero on a patched newton.

`ROMVersionString`

The user visible string that identifies the version of the installed ROM and the installed patch if any.

The first part of the string is a "functionality level" indicating whether the ROM has 1.x or 2.x functionality. All Pre-2.x units, except the original MessagePads, have "1.3" as their functionality level. 2.x and later units have "2.0".

The second part of the string is a six digit number in parentheses that is an encoded representation of ROM and Update information.

Here is an example of some code that you can use to decode the value of the `ROMVersion` slot in the returned frame.

```
global VersionDecode(ROMVersion)
begin
        local minor := BAND(ROMVersion, 0xFFFF);
        local major := BAND(ROMVersion>>16, 0xFFFF);
        [Floor(StringToNumber(BAND(major>>12, 0xF)
                              & BAND(major>>8, 0xF)
                              & BAND(major>>4, 0xF)
                              & BAND(major, 0xF))),
        Floor(StringToNumber(BAND(minor>>12, 0xF)
                              & BAND(minor>>8, 0xF)
                              & BAND(minor>>4, 0xF)

                              & BAND(minor, 0xF)))];
```

```
end;
VersionDecode(Gestalt(0x1000003).ROMVersion)
```

Here is another example of some code you can use to test if your Newton is running 2.0. It returns true if the major version is 2.

```
global VersionTwo() BAND((Gestalt(0x1000003).ROMVersion)
   >>16, 0xFFFF) = 0x0002;
```

IMPORTANT

Do not assume that if the Newton is running version 2.0 or later that a particular feature exists. You still need to test the Newton to make sure the feature exits. ◆

GetAppName

`GetAppName(`*appSymbol*`)`

Retrieves a user-visible application name for another application. `GetAppName` returns a string that is the name of the application.

appSymbol A symbol identifying the application whose name you want.

This function looks several places to find a string that is the application name. Here is how it searches:

1. First this function checks in the application base view for a slot named `appName`, and if found, returns the string found therein.
2. Next, this function looks in the application base view for a `title` slot, and if found, returns the string found therein.
3. Next, this function returns the string used for the application name below its icon in the Extras Drawer.
4. If none of the above attempts succeed in finding a name, the *appSymbol* is converted to a string and returned.

GetAppParams

`GetAppParams( )`

Returns a frame containing information about the screen size and other system configuration items. The frame returned contains these slots:

`appAreaTop` The Y coordinate of the top left corner of the screen.

`appAreaLeft` The X coordinate of the top left corner of the screen.

`appAreaWidth` The width of the screen in pixels.

`appAreaHeight` The height of the screen in pixels.

`buttonBarPosition`
A symbol ('`top`, '`left`, '`bottom`, or '`right`) indicating where the button bar is. This is useful if you want to locate your application flush against the button bar. (The button bar contains the Newton application / scroller icons.)

In order to make your applications compatible on future Newton systems, you must compute the size and location of your application based on the values in this frame.

This frame may be expanded to include additional slots as other system parameters become relevant on future Newton systems.

GetAppPrefs

`GetAppPrefs(`*appSymbol*`,` *defaultFrame*`)`

Retrieves the preferences for an application from the System soup.

appSymbol A symbol identifying the application whose preferences you want.

defaultFrame The default frame to be used for the application preferences.

This function returns a system soup entry that is the application preferences entry.

The *appSymbol* is stored in the `tag` slot of the entry. If no entry exists for the specified *appSymbol*, a new entry is created, filled in with the contents of the

defaultFrame, entered into the System soup and the entry is returned. If *defaultFrame* does not have a `tag` slot, the *appSymbol* is turned into a string and entered into the `tag` slot.

GetMemoryItems

`GetMemoryItems(`*memSymbol*`)`

The function returns an array of the memory items, suitable for use in a picker, that have been added under *memSymbol*.

memSymbol A symbol used in your memory slot.

For example:

```
self.currentMem := GetMemoryItems('|wiggys:Wiggy:PIEDTS|) ;
   if currentMem AND Length(currentMem) > 0 then
   :PopupMenu(currentMem, nil);
```

GetMemorySlot

`GetMemorySlot(`*memorySlot*`, `*op*`)`

Removes the storage used for the memorized items.You should call this from the `DeletionScript` function for your application. For more information on `DeletionScript`, see page 2-14

memorySlot A symbol that identifies a group of memorized items.

op The symbol, `'remove`.

For example:

```
GetMemorySlot('|wiggys:Wiggy:PIEDTS|, 'remove);
```

Note:
Other values of *op* and the return value for this function are undefined and subject to change. ◆

MakePhone

`MakePhone (`*phoneFrame*`)`

Takes a phone frame and creates a phone string with the different parts encoded. These phone strings are used in Names and in the rest of the system. To display one of these real phone strings, call `MakeDisplayPhone`.

phoneFrame	A frame with optional slots: `areacode`, `phone`, `extension`

For example, if you call:

```
ParsePhone(MakePhone({areacode: "617", phone:
   "965-4322"}))
```

you'll get

```
back {areacode: "617", phone: "965-4322"}
```

MakeDisplayPhone

`MakeDisplayPhone(`*phoneStr*`)`

`MakeDisplayPhone` takes a phone string or phone frame (a frame with slots `areacode`, `phone`, and `extension`), and formats it with `Userconfiguration.phoneformat` to return another string, which is how a phone number really should be displayed. `Userconfiguration.phoneformat` contains a formatting string; for example, `^0/^1x^2` if you want your phone number displayed as `408/ 555-1212x111`. The default is to display numbers as `408 555-1212x111`. Note that there is currently no way for a user to change the default `userconfiguration.phoneformat.`

phoneStr	A phone string or phone frame (a frame with slots `areacode`, `phone`, and `extension`)

MungePhone

rootView:MungePhone(*inNum*, *country*)

This is the main method that you call to find the correct dialing sequence to use in your current location. It is a method of the root view. Builds from *inNum* a dialing string based on current user configuration settings. The returned string prefixes *inNum* with an international access code, country code, and area code as appropriate.

inNum — The phone number to convert. It should generally be of the form <*area-code*> <*phone-number*> For example:

415 555 1212 —A phone number in San Francisco, USA

81 555 1212— phone number in London, UK

country — A string that is the name of the country in which *inNum* is to be dialed. For example:

"USA" for "415 555 1212"

"UK" for "81 555 1212"

Note that the country must be one from ROM_countries, which is discussed below.

If you give nil instead of a string, MungePhone assumes that the call is within the country specified in the Country setting in the user's personal preferences.

If the user has stored a calling card number with *inNum* in the calling options in the Names application, the calling card number is appended to the phone dialing string.

ROM_countries is a frame that has one slot for each country for which Newton can convert phone numbers. Each slot is a frame, though the only slot you care about is the name slot.

You can get an array of the names of all known countries like this:

```
local countryNames := foreach item in ROM_countries
                                  collect item.name ;
```

`CallOptionsSlip` is the system slip for getting and setting the user call options. This includes the current area code, dialing prefix, long distance access code and any calling card numbers the user may have.

`MungePhone` automatically checks these user configuration items, so they will be figured into the return string.

ParsePhone

`ParsePhone(`*phoneStr*`)`

Takes a phone string and parses it into a frame with the slots `'areacode`, `'phone`, and `'extension`. The slots may be `nil` if there's no corresponding string. You should call `ParsePhone` if you want to get the component parts of the phone number. (If you just want to display the phone number, call `MakeDisplayPhone` instead.)

phoneStr Phone string to parse into a frame.

Ord

`Ord (`*char*`)`

Converts a character to its Unicode decimal integer equivalent.

char A character.

Here is an example:

```
ord($A)
65
```

ShowManual

`ShowManual()`

Opens the system-supplied help browser; this has the same effect as tapping How Do I in the assist drawer.

Sleep

`Sleep(`*ticks*`)`

Puts the Newton to sleep (suspends processing) for the number of ticks given.

ticks An integer giving the number of ticks to sleep. A tick is one sixtieth of a second.

For example:

```
for i:= 5 to 1 by -1 do
begin
   SetValue(infoView, 'text,
                  "Will restart in " & i & " seconds!");
   RefreshViews();
   Sleep(6);
end;
```

If you leave out the `RefreshViews` call, the view is not updated until after the last iteration, because calling sleep postpones the view event handling.

Notice that the screen is not updated during `sleep`, even if there is a pending update.

IMPORTANT

Do not use the `sleep` function to put the Newton to sleep for very long. The `sleep` function suspends the Application task, in which all NewtonScript code runs, so the user can do nothing else during a sleep. Occasionally, particularly during communications, you may need to sleep for several seconds, even half a minute, but, in general, sleeping for more than a fraction of a second is too much. If you need a longer delay, consider `AddDeferredCall` (or a related function) or a `ViewIdleScript` method as alternatives. Those methods return control to the main event loop. ◆

SysBeep

rootView:SysBeep()

Plays the system beep sound. This message must be sent to the root view. For example:

```
:SysBeep();
```

Translate

Translate(*data*, *translator*, *store*, *callback*)

This function translates data, returning an object that contains the translated data.

data	The data to be translated. The type of this object depends on the translator used.
translator	A symbol indicating the type of translator to use. Table 20-5 lists the translators available, and the corresponding type of the *data* object to be used with each.
store	Specifies the store on which you want the translated object to be created. If you specify a valid store, the translated object is created as a virtual binary object on that store. If you specify `nil`, a normal object is created on the NewtonScript heap.
callback	A function that may be called periodically during the translation process to inform your application of progress. This function is passed two parameters: the first is the number of bytes that have been translated so far, and the second is the total number of bytes that are to be translated. If either of these parameters is `nil`, that signifies that the number is unknown. This callback function must return a Boolean value. A return value of non-`nil` means continue translation, and `nil` means abort the translation.

Note that this callback function is not implemented by either of the two existing translators, so this parameter is currently ignored.

Table 20-5 Data Translators

Translator	Data Type	Description
`'flattener`	frame	Translates a frame into a binary object
`'unflattener`	binary object	Translates a binary object containing a flattened frame into a frame

Summary of Functions and Methods

This section contains a summary of the functions and methods described in this chapter.

Object System Functions

`ClassOf(`*object*`)`
`Clone(`*object*`)`
`DeepClone(`*object*`)`
`EnsureInternal(`*obj*`)`
`GetFunctionArgCount(`*function*`)`
`GetSlot(`*frame*`, `*slotSymbol*`)`
`GetVariable(`*frame*`, `*slotSymbol*`)`
`HasSlot(`*frame*`, `*slotSymbol*`)`
`HasVariable(`*frame*`, `*slotSymbol*`)`
`Intern( `*string*` )`
`IsArray(`*obj*`)`
`IsBinary(`*obj*`)`
`IsCharacter(`*obj*`)`

IsFrame(*obj*)
IsFunction(*obj*)
IsImmediate(*obj*)
IsInstance(*obj*, *class*)
IsInteger(*obj*)
IsNumber(*obj*)
IsReadOnly(*obj*)
IsReal(*obj*)
IsString(*obj*)
IsSubclass(*sub*, *super*)
IsSymbol(*obj*)
MakeBinary(*length*, *class*)
Map(*obj*, *function*)
PrimClassOf(*obj*)
RemoveSlot(*obj*, *slot*)
ReplaceObject(*originalObject*, *targetObject*)
SetClass(*obj*, *classSymbol*)
SetVariable(*frame*, *slotSymbol*, *value*)
SymbolCompareLex(*symbol1*, *symbol2*)
TotalClone(*obj*)

String Functions

BeginsWith(*string*, *substr*)
Capitalize(*string*)
CapitalizeWords(*string*)
CharPos(*str*, *char*, *startpos*)
Downcase(*string*)
DurationStr(*minutes*)
EndsWith(*string*, *substr*)
EvalStringer(*frame*, *array*)
FindStringInArray(*array*, *string*)
FindStringInFrame(*frame*, *stringArray*, *path*)
IsAlphaNumeric(*char*)
IsWhiteSpace(*char*)
NumberStr(*number*)
ParamStr(*baseString*, *paramStrArray*)

SPrintObject(*obj*)
StrCompare(*a*, *b*)
StrConcat(*a*, *b*)
StrEqual(*a*, *b*)
StrExactCompare(*a*, *b*)
StrFilled(*string*)
StrFontWidth(*string*, *fontSpec*)
Stringer(*array*)
Stringfilter(*str*, *filter*, *instruction*)
StringToNumber(*string*)
StrLen(*string*)
StrMunger(*dstString, dstStart, dstCount, srcString, srcStart, srcCount*)
StrPos(*string*, *substr*, *start*)
StrReplace(*string*, *substr*, *replacement*, *count*)
StrTokenize(*str*, *delimiters*)
StyledStrTruncate(*string*, *length*, *font*)
SubstituteChars(*targetStr*, *searchStr*, *replaceStr*)
SubStr(*string*, *start*, *count*)
TrimString(*string*)
Upcase(*string*)

Bitwise Functions

Band(*a*, *b*)
Bor(*a*, *b*)
Bxor(*a*, *b*)
Bnot(*a*)

Array Functions

AddArraySlot (*array*, *value*)
Array(*size*, *initialValue*)
ArrayInsert(*array*, *element*, *position*)
ArrayMunger(*dstArray*, *dstStart*, *dstCount*, *srcArray*, *srcStart*, *srcCount*)
ArrayRemoveCount(*array*, *startIndex*, *count*)
InsertionSort(*array*, *test*, *key*)
Length (*array*)
LFetch(*array*, *item*, *start*, *test*, *key*)

LSearch(*array*, *item*, *start*, *test*, *key*)
NewWeakArray(*length*)
SetAdd (*array*,*value*,*uniqueOnly*)
SetContains(*array*, *item*)
SetDifference(*array1*, *array2*)
SetLength (*array*, *length*)
SetOverlaps(*array1*, *array2*)
SetRemove (*array*, *value*)
SetUnion(*array1*, *array2*, *uniqueFlag*)
Sort(*array*, *test*, *key*)
StableSort(*array*, *test*, *key*)

Sorted Array Functions

BDelete(*array*, *item*, *test*, *key*, *count*)
BDifference(*array1*, *array2*, *test*, *key*)
BFetch(*array*, *item*, *test*, *key*)
BFetchRight(*array*, *item*, *test*, *key*)
BFind(*array*, *item*, *test*, *key*)
BFindRight(*array*, *item*, *test*, *key*)
BInsert(*array*, *element*, *test*, *key*, *uniqueOnly*)
BInsertRight(*array*, *element*, *test*, *key*, *uniqueOnly*)
BIntersect(*array1*, *array2*, *test*, *key*, *uniqueOnly*)
BMerge(*array1*, *array2*, *test*, *key*, *uniqueOnly*)
BSearchLeft(*array*, *item*, *test*, *key*)
BSearchRight(*array*, *item*, *test*, *key*)

Integer Math Functions

Abs(*x*)
Ceiling(*x*)
Floor(*x*)
GetRandomState()
Max(*a*, *b*)
Min(*a*, *b*)
Real(*x*)
Random (*low*, *high*)
SetRandomSeed (*seedNumber*)
SetRandomState(*randomState*)

Floating Point Math Functions

Acos(*x*)
Acosh(*x*)
Asin(*x*)
Asinh(*x*)
Atan(*x*)
Atan2(*x*,*y*)
Atanh(*x*)
CopySign(*x*,*y*)
Cos(*x*)
Cosh(*x*)
Erf(*x*)
Erfc(*x*)
Exp(*x*)
Expm1(*x*)
Fabs(*x*)
FDim(*x*,*y*)
FMax(*x*,*y*)
FMin(*x*,*y*)
Fmod(*x*,*y*)
Gamma(*x*)
Hypot(*x*,*y*)
IsFinite(*x*)
IsNaN(*x*)
IsNormal(*x*)
LessEqualOrGreater(*x*, *y*)
LessOrGreater(*x*, *y*)
LGamma(*x*)
Log(*x*)
Logb(*x*)
Log1p(*x*)
Log10(*x*)
NearbyInt(*x*)
NextAfterD(*x*,*y*)
Pow(*x*,*y*)
RandomX(*x*)

Remainder(*x*,*y*)
RemQuo(*x*,*y*)
Rint(*x*)
RintToL(*x*)
Round(*x*)
Scalb(*x*, *k*)
SignBit(*x*)
Signum(*x*)
Sin(*x*)
Sinh(*x*)
Sqrt(*x*)
Tan(*x*)
Tanh(*x*)
Trunc(*x*)
Unordered(*x*, *y*)
UnorderedGreaterOrEqual(*x*, *y*)
UnorderedLessOrEqual(*x*, *y*)
UnorderedOrEqual(*x*, *y*)
UnorderedOrGreater(*x*, *y*)
UnorderedOrLess(*x*, *y*)
FeClearExcept(*excepts*)
FeGetEnv()
FeGetExcept(*excepts*)
FeHoldExcept()
FeRaiseExcept(*excepts*)
FeSetEnv(*envObj*)
FeSetExcept(*flagObj*, *excepts*)
FeTestExcept(*excepts*)
FeUpdateEnv(*envObj*)

Financial Functions

Annuity(*r*, *n*)
Compound(*r*, *n*)
GetExchangeRate(*country1*, *country2*)
SetExchangeRate(*country1*, *country2*, *rate*)
GetUpdatedExchangeRates()

Exception Functions

`Throw(`*name*`, `*data*`)`
`Rethrow()`
`CurrentException()`
`RethrowWithUserMessage(`*userTitle*`,`*userMessage*`,`*override*`)`

Message Sending Functions

`Apply(`*function*`, `*parameterArray*`)`
`IsHalting(`*functionObject*`, `*args*`)`
`Perform(`*frame*`, `*message*`, `*parameterArray*`)`
`PerformIfDefined(`*receiver*`,`*message*`,`*paramArray*`)`
`ProtoPerform(`*receiver*`,`*message*`,`*paramArray*`)`
`ProtoPerformIfDefined(`*receiver*`,`*message*`,`*paramArray*`)`

Deferred Message Sending Functions

`AddDeferredCall(`*functionObject*`,`*paramArray*`)`
`AddDelayedCall(`*functionObject*`,`*paramArray*`,`*delay*`)`
`AddDeferredSend(`*receiver*`,`*message*`,`*paramArray*`)`
`AddDelayedSend(`*receiver*`,`*message*`,`*paramArray*`,`*delay*`)`
`AddProcrastinatedCall(`*funcSymbol*`,`*functionObject*`,`*paramArray*`,`*delay*`)`
`AddProcrastinatedSend(`*msgSymbol*`,`*receiver*`,`*message*`,`*paramArray*`,`*delay*`)`

Data Extraction Functions

`ExtractByte(`*data*`, `*offset*`)`
`ExtractBytes(`*data*`, `*offset*`, `*length*`, `*class*`)`
`ExtractChar(`*data*`, `*offset*`)`
`ExtractLong(`*data*`, `*offset*`)`
`ExtractXLong(`*data*`, `*offset*`)`
`ExtractWord(`*data*`, `*offset*`)`
`ExtractCString(`*data*`, `*offset*`)`
`ExtractPString(`*data*`, `*offset*`)`
`ExtractUniChar(`*data*`, `*offset*`)`

Data Stuffing Functions

`StuffByte(`*obj*`, `*offset*`, `*toInsert*`)`
`StuffChar(`*obj*`, `*offset*`, `*toInsert*`)`
`StuffCString(`*obj*`, `*offset*`, `*aString*`)`

StuffLong(*obj*, *offset*, *toInsert*)
StuffPString(*obj*, *offset*, *aString*)
StuffUniChar(*obj*, *offset*, *toInsert*)
StuffWord(*obj*, *offset*, *toInsert*)

Getting and Setting Global Variables and Functions

GetGlobalFn(*symbol*)
GetGlobalVar(*symbol*)
GlobalFnExists(*symbol*)
GlobalVarExists(*symbol*)
DefGlobalFn(*symbol*, *function*)
DefGlobalVar(*symbol*, *value*)
UnDefGlobalFn(*symbol*)
UnDefGlobalVar(*symbol*)

Miscellaneous Functions

AddMemoryItem(*memSymbol*, *value*)
AddMemoryItemUnique(*memorySlot*, *value*, *testFunc*)
BinEqual(*a*, *b*)
BinaryMunger(*dst*, *dstStart*, *dstCount*, *src*, *srcStart*, *srcCount*)
Chr(*integer*)
Compile(*string*)
Gestalt(selector)
GetAppName(*appSymbol*)
GetAppParams( )
GetAppPrefs(*appSymbol*, *defaultFrame*)
GetMemoryItems(*memSymbol*)
GetMemorySlot(*memorySlot*, *op*)
MakePhone (*phoneFrame*)
MakeDisplayPhone(*phoneStr*)
rootView:MungePhone(*inNum*, *country*)
ParsePhone(*phoneStr*)
Ord (*char*)
ShowManual()
Sleep(*ticks*)
rootView:SysBeep()
Translate(*data*, *translator*, *store*, *callback*)

Utility Functions

Utility Functions

Errors

This appendix lists the exceptions and error codes that the Newton system software generates. These are grouped into the following categories:

- system exceptions
- system errors
- hardware errors
- communications errors
- system services errors
- NewtonScript environment errors
- device driver errors
- other errors

Each of the categories is subdivided into several tables of related error codes to make it easier to find an error. All errors in this appendix are listed in ascending numeric order.

System Exceptions

These are the two main types of exceptions that can be raised by the Newton system software.

Exception symbol	Description
`|evt.ex.fr|`	NewtonScript environment exception
`|evt.ex.comm|`	Communications toolbox exception

System Errors

This section lists the different kinds of Newton system software errors.

Common Errors

These are errors that can occur at almost any time.

Error code	Description
0	No error
–7000	Not enough memory available

Application Errors

These are the application errors.

Error code	Description
–8001	PCMCIA card battery must be replaced
–8002	PCMCIA card battery is running low
–8003	Nothing to undo
–8004	The routing slip is already open
–8005	Close box must be tapped to hang up the modem
–8006	Nothing to print
–8007	Exception not handled
–8008	The length of a styles slot had to be extended
–8009	A length in the read-only styles slot is too short to display the text
–8010	Communications card has been inserted
–8011	Note has too many items
–8012	Note is too large
–8013	Note is too long
–8100	Blank note could not be created
–8101	Item could not be moved

Error code	Description
–8102	Changes could not be saved
–8103	A problem has occurred
–8104	Problem with the PCMCIA card
–8105	Note could not be changed

I/O Box Errors

These are the I/O Box errors.

Error code	Description
–8301	Missing transport
–8302	Missing slip
–8303	Cannot convert

View System Errors

These are the view system errors.

Error code	Description
–8501	Could not create view
–8502	Missing class slot
–8503	Unknown view stationery
–8504	Missing view flags
–8505	Missing view bounds

State Machine Errors

These are the state machine errors.

Error code	Description
–8601	Invalid state
–8602	No state
–8603	No wait state
–8604	No polling routine
–8605	Polling timed out
–8606	Aborted
–8607	No reentrance
–8608	Invalid mode

Operating System Errors

These are the operating system errors.

Error code	Description
–10000	Bad domain object ID
–10001	Bad physical page object ID
–10002	Unexpected object type
–10003	No page table
–10004	Allocation on an uninitialized heap
–10005	Call not implemented
–10006	Bad parameters
–10007	Not enough memory
–10008	Item not found
–10009	Could not create object
–10010	Must use a remote procedure call
–10011	Bad object
–10012	Not a user call

Error code	Description
–10013	Task does not exist
–10014	Unexpected end of message
–10015	Bad object ID
–10016	Bad message object ID
–10017	Message already posted
–10018	Cannot cash token
–10019	Port no longer exists
–10020	No message waiting
–10021	Communications problem (message timed out)
–10022	Bad semaphore group ID
–10023	Bad semaphore operation list ID
–10024	Semaphore group no longer exists
–10025	Semaphore would cause blocking
–10026	Task no longer exists
–10027	Task aborted
–10028	Cannot suspend blocked task
–10029	Bad register number
–10030	Bad monitor function
–10031	No such monitor
–10032	Not a monitor
–10033	Size too large in shared memory call
–10034	Shared memory mode violation
–10035	Object not owned by task
–10036	Object not assigned to task
–10037	Total confusion
–10038	Another task already blocking
–10039	Cancelled

Error code	Description
–10040	Object already initialized
–10041	Nested collection
–10042	Shared memory message no longer exists
–10043	Receiver did not perform remote procedure call
–10044	Copy aborted
–10045	Bad signature
–10046	Call not in progress
–10047	Token expected
–10048	Receiver object no longer exists
–10049	Monitor is not suspended
–10050	Not a fault monitor
–10051	No available page
–10052	Interrupt not enabled
–10053	Interrupt not implemented
–10054	Tric interrupt not enabled
–10055	Tric interrupt not implemented
–10056	Unresolved fault
–10057	Call already in progress
–10058	Offset beyond data
–10059	Bus access
–10060	Access permission
–10061	Permission violation
–10062	Duplicate object
–10063	Ill formed domain
–10064	Out of domains
–10065	Write protected
–10066	Timer expired

Error code	Description
–10067	Not registered
–10068	Already registered
–10069	System restarted due to a power fault
–10070	System restarted because the battery was dead
–10072	System restarted because a PCMCIA card was removed while in use.
–10073	RAM table is full
–10074	Unable to satisfy request
–10075	System error
–10076	System failure
–10077	New system software
–10078	Resource is claimed
–10079	Resource is unclaimed

Stack Errors

These are the stack errors.

Error code	Description
–10200	Stack too small
–10201	No room for heap
–10202	Stack is corrupted
–10203	Stack overflow
–10204	Stack underflow
–10205	Address out of range
–10206	Bad domain

Package Errors

These are the package errors.

Error code	Description
–10401	Bad package
–10402	Package already exists
–10403	Bad package version
–10404	Unexpected end of package
–10405	Unexpected end of package part
–10406	Part type is already registered
–10407	Part type is not registered
–10408	No such package exists
–10409	Newer package already exists
–10410	Newer version of application already installed

Newton Hardware Errors

This section lists the different kinds of Newton hardware errors.

PCMCIA Card Errors

These are the PCMCIA card errors.

Error code	Description
–10501	Unrecognized card
–10502	Card not ready
–10503	Bad power on card
–10504	Unexpected card error
–10505	Card reset
–10506	Card is not initialized
–10507	Card service is not installed

Error code	Description
–10508	Card service is not suspended
–10509	Card service has not been resumed
–10510	No usable configurations on card
–10511	Card could not be formatted
–10512	Card could not be formatted because it is write-protected
–10520	Bad CIS parser procedure pointer
–10521	Unknown tuple in CIS
–10522	Unknown subtuple in CIS
–10523	CIS tuple order is bad
–10524	CIS tuple size is bad
–10525	CIS tuple specified as no link has a link
–10526	CIS tuple specified with a link has no link
–10527	CIS tuple link target is bad
–10528	Bad CIS tuple version 1
–10529	Bad CIS tuple version 2
–10530	Bad CIS JEDEC tuple
–10531	Bad CIS checksum
–10532	Missing CIS
–10533	Blank CIS
–10534	Bad CIS
–10535	Bad link target

Flash Card Errors

These are the flash card errors.

Error code	Description
–10551	Flash card is busy
–10552	Flash card is not erasing
–10553	Flash card erase is not suspended
–10554	Flash card suspend erase error
–10555	Flash card erase failed
–10556	Flash card write failed
–10557	Flash card Vpp is low
–10558	Flash card error in sleep
–10559	Flash card does not have enough power

Card Store Errors

These are the card store errors.

Error code	Description
–10600	Attempt to read or write outside of object bounds
–10601	Bad buffer pointer
–10602	Bad card access
–10603	Bad storage type
–10604	Store not found
–10605	The store has been write-protected by the user
–10606	Object not found
–10607	Flash card block is full
–10608	Flash card is not virgin
–10609	Write error (one or more bits failed to assert)
–10610	No more objects
–10611	Flash card erase in progress

Error code	Description
–10612	Card is full
–10613	No more blocks left in search on flash card
–10614	Flash card log is full
–10615	Card needs to be formatted
–10616	Bad or unknown PSSID
–10617	Card memory is full
–10618	Missing or low battery on SRAM card
–10619	Attempt to modify store without a transaction in effect
–10620	Transaction aborted
–10621	Card needs recovery, but it is write-protected
–10622	Object too large for store

DMA Errors

These are the DMA errors.

Error code	Description
–10800	DMA mode
–10801	DMA bus access
–10802	DMA buffer doesn’t exist
–10803	DMA address word alignment
–10804	DMA count word alignment
–10805	DMA count size
–10806	DMA offset size
–10820	DMA PCMCIA ready
–10821	DMA PCMCIA input acknowledgment
–10822	DMA PCMCIA write protect
–10823	DMA PCMCIA time out

Heap Errors

These are the heap errors.

Error code	Description
–10900	Heap odd block size
–10901	Heap block out of range
–10902	Heap preferred free not found
–10903	Heap free accounting error
–10904	Heap accounting error
–10905	Heap block too big
–10906	Heap bad prior pointer
–10907	Heap bad last pointer in prior
–10908	Heap bad last pointer in last

Communications Errors

This section lists the different kinds of Newton communications errors.

Generic AppleTalk Errors

These are the generic AppleTalk errors.

Error code	Description
–12001	Buffer too small or corrupted
–12002	Event is pending
–12003	Cancelled
–12004	Attempt to cancel failed
–12005	No handler for cancel
–12006	Unknown message receiver
–12007	Cannot create AppleTalk port
–12008	Cannot create AppleTalk task

Error code	Description
–12009	Not implemented
–12010	Data length error
–12011	No such subject available to open
–12012	Not opened
–12014	AppleTalk is already open
–12015	Duration is too small
–12016	Duration is too large

LAP Protocol Errors

These are the LAP protocol errors.

Error code	Description
–12100	LAP read link failed
–12101	LAP all protocols in use
–12102	No protocol handler
–12103	No such command
–12104	Bad link

DDP Protocol Errors

These are the DDP protocol errors.

Error code	Description
–12200	No such DDP command
–12201	Invalid socket
–12202	Not in static socket range
–12203	Not in dynamic socket range
–12204	Socket is already open
–12205	Socket not open

Error code	Description
–12206	Socket internal socket
–12207	Socket is in use
–12208	Unknown LAP type
–12209	DDP back check sum
–12210	Bad packet size
–12211	No listener for socket
–12212	No such protocol type known
–12213	External client timed out

NBP Protocol Errors

These are the NBP protocol errors.

Error code	Description
–12300	Bad form
–12301	Name is already registered
–12302	Too many names
–12303	Name is not registered
–12304	Too many names requested
–12305	Too many lookups are pending
–12306	Not a NBP packet DDP type
–12307	Unknown NBP function
–12308	Unknown NBP lookup reply
–12309	Too many tuples in lookup request
–12311	NBP index out of range
–12312	NBP lookup aborted
–12313	No such command
–12314	No names found

AEP Protocol Errors

These are the AEP protocol errors.

Error code	Description
–12400	No such command
–12401	Not an echo packet DDP type
–12402	AEP packet size is zero
–12403	AEP function not requested

RTMP Protocol Errors

These are the RTMP protocol errors.

Error code	Description
–12500	No such command
–12502	Packet size is zero
–12503	RTMP routed
–12504	RTMP address unresolved
–12505	RTMP no router available

ATP Protocol Errors

These are the ATP protocol errors.

Error code	Description
–12600	No such command
–12601	No ATP packet DDP type
–12602	Unknown ATP function
–12603	ATP request data length is zero
–12604	Expected responses are out of range
–12605	Response buffer is too small
–12606	ATP retry duration too small

Error code	Description
–12607	ATP transaction timed out
–12608	Responding socket already open
–12609	Responding socket not open
–12610	Response packet length bad
–12611	Bad number of response packets
–12612	Socket already has a request on autorequest

PAP Protocol Errors

These are the PAP protocol errors.

Error code	Description
–12700	No such command
–12701	Unexpected connection ID
–12702	Invalid connection ID
–12703	Invalid responder socket
–12704	Unexpected function
–12705	Printer is busy
–12706	Unexpected connection open result
–12707	Bad flow quantum requested
–12708	Connection timed out
–12709	EOF sent
–12710	PAP flushed
–12711	Printer terminated connection
–12712	Printer not found
–12713	No status available
–12714	No data available
–12715	The buffer that was passed is too small
–12716	Put data operation timed out

ZIP Protocol Errors

These are the ZIP protocol errors.

Error code	Description
–12800	No zones

ADSP Protocol Errors

These are the ADSP protocol errors.

Error code	Description
–12900	Too many ADSP connections
–12901	ADSP mode invalid
–12902	ADSP packet size bad
–12903	ADSP control type bad
–12904	Remote end disconnected

Utility Class Errors

These are the utility class errors.

Error code	Description
–14001	Not implemented
–14002	Out of memory
–14003	Bad position
–14004	Already initialized
–14005	Invalid size
–14006	Overflow
–14007	Underflow
–14008	Range check failed

Error code	Description
–14009	Element sizes do not match
–14010	Not initialized
–14011	Pointer is `nil`

Communications Tool Errors

These are the communications tool errors.

Error code	Description
–16001	Command in progress
–16002	Bad command
–16003	Tool already has maximum requests pending
–16004	Buffer overflow
–16005	Request canceled by kill request
–16006	Bad parameter in request
–16007	Connection end has not been created yet
–16008	Invalid call when connected
–16009	Phone connection was cut off, or invalid call when not connected
–16010	Connection negotiation failed because remote end is not compatible with local end configuration
–16011	Connection terminated or failed due to retransmission limit of data or connect packet
–16012	No data available for `TCommToolGetRequest` when `fNonBlocking` is true.
–16013	Connection aborted by disconnect
–16014	Call not supported
–16015	Request not pending
–16016	Event not pending
–16017	Time-out waiting for connection

Error code	Description
–16018	Connection end is already bound
–16019	Connection end was not bound before use
–16020	Connection end is being released
–16021	No phone number was provided
–16022	Operation failed because a resource was not available
–16023	Call failed because the option passed is not supported
–16024	The method is not implemented

Serial Tool Errors

These are the serial tool errors.

Error code	Description
–18000	Serial channel is in use
–18001	Memory error
–18002	Not current owner of the serial port
–18003	Framing or parity overrun, or bad connection
–18004	CRC error on input framing
–18005	An internal error has occurred
–18006	Packet size too large or too small in an output request
–18007	Unexpected packet length
–18008	EOF not found
–18009	Overrun bit was set
–18010	Too many collisions when sending packet
–18011	Too many deferrals when sending packet
–18012	Timed out waiting for an event
–18013	Serial tool is not active or ready

MNP Tool Errors

These are the MNP tool errors.

Error code	Description
–20001	Connection parameter negotiation failed
–20002	Acceptor of connect request timed out
–20003	Not connected
–20004	Request aborted by disconnect request
–20005	Link attention service is not enabled
–20006	Request retry limit of connect initiator reached
–20007	Command already in progress
–20008	Connection already established
–20009	Connection failed due to incompatible protocol levels
–20010	Connection handshake failed
–20011	Memory for MNP not allocated

FAX Tool Errors

These are the FAX tool errors.

Error code	Description
–22001	Lost connection while sending or receiving FAX
–22002	FAX machine is not compatible
–22003	Transmission error
–22005	FAX machine had a problem sending some pages
–22006	Transmission error
–22007	Transmission error

Modem Tool Errors

These are the modem tool errors.

Error code	Description
–24000	No modem is connected
–24001	There is no dial tone
–24002	There is no answer
–24003	The phone number is busy
–24004	There is no answer
–24005	The modem is not responding properly
–24006	FAX carrier error
–24007	The modem is not responding properly
–24008	The modem connected to the serial port does not support cellular connection
–24009	The AT+FRH command timed out when receiving flags

Communications Manager Errors

These are the Communications Manager errors.

Error code	Description
–26000	Service already initialized
–26001	Unknown command
–26002	Unknown service
–26003	Service already exists
–26004	No service specified in the options array
–26005	There is no registered service matching the type specified in the options array
–26006	No endpoint exists; this is usually because `CMStartService` has not been called
–26007	No public port exists; this is usually because `CMGetEndPoint` has not been called

Error code	Description
–26008	No known last connected device
–26009	A tuple has been received, but no the device ID tuple
–26010	A service information response tuple was expected
–26011	Unsupported service; can only load packages
–26012	An SCP load is in progress and another cannot be issued
–26013	The SCP load call is not supported on this machine
–26014	Cannot process this speed
–26015	The SCP loader did not previously load a package

Docker Errors

These are the docker errors.

Error code	Description
–28001	Invalid store signature
–28002	Invalid entry
–28003	Aborted
–28004	Invalid query
–28005	Read entry error
–28006	Invalid current soup
–28007	Invalid command length
–28008	Entry not found
–28009	Bad connection
–28010	File not found
–28011	Incompatible protocol
–28012	Protocol error
–28013	Docking canceled
–28014	Store not found
–28015	Soup not found

Error code	Description
–28016	Invalid header
–28017	Out of memory
–28018	Newton version too new
–28019	Package cannot load
–28020	Protocol already registered
–28021	Remote import error
–28022	Bad password error
–28023	Password retry
–28024	Idle too long
–28025	Out of power
–28026	Invalid cursor
–28027	Already busy
–28028	Desktop error
–28029	Cannot connect to modem
–28030	Disconnected
–28031	Access denied
–28100	Disconnect during read
–28101	Read failed
–28102	Communications tool not found
–28103	Invalid modem tool version
–28104	Card not installed
–28105	Browser File Not Found
–28106	Browser Volume Not Found
–28107	Browser Path Not Found

Docker Import and Export Errors

These are the docker import and export errors.

Error code	Description
–28200	Syntax error
–28201	Invalid version
–28202	Could not open temporary store
–28203	Could not convert
–28204	Invalid criteria
–28205	Error applying script
–28206	Missing meta data
–28207	Unknown error
–28208	Scanner overflow error
–28209	Data Viz translator error
–28210	Invalid type

Docker Disk Errors

These are the docker disk errors.

Error code	Description
–28300	Disk full
–28301	File not found
–28302	File is write protected
–28303	Duplicate file name
–28304	Too many files open

Docker Desktop DIL Errors

These are the docker desktop DIL errors.

Error code	Description
–28700	No Error
–28701	Out of memory
–28702	Invalid pipe state
–28703	Exception error
–28704	Queue full
–28705	Pipe not initialized
–28706	Invalid parameter
–28707	Pipe not ready
–28800	No Error
–28801	Out of object heap memory
–28802	Out of temporary memory
–28803	Unknown slot
–28804	Slot size exceeded
–28805	Slot size required

System Services Errors

This section lists the different kinds of Newton system services errors.

Sound Errors

These are the sound errors.

Error code	Description
–30000	Generic sound error
–30001	Not enough memory available
–30002	Invalid message

Error code	Description
–30003	Sound was not played
–30004	No channel decompressor
–30005	Destination buffer too small
–30006	Sound player busy
–30007	Sound recorder busy
–30008	No samples provided
–30009	Unsupported sound configuration
–30010	Sound channel closed
–30011	Sound cancelled
–30012	The sound volume is set to zero

Compression Errors

These are the compression errors.

Error code	Description
–32001	Cannot compress in place
–32002	Parsing error
–32003	Invalid type
–32004	Compression not achieved
–32005	Key not found
–32006	Compression index error
–32007	Cannot decompress in place
–32008	Decompression not achieved
–32009	Unexpected end of source
–32100	Buffer overflow
–32101	Buffer underflow

Memory Errors

These are the memory errors.

Error code	Description
-34000	Not free, direct or indirect
-34001	Pointer not aligned to 4-byte boundary
-34002	Pointer to outside of heap
-34003	Unknown infrastructure type
-34004	Free block where there shouldn't be one
-34005	Free list pointer points outside of heap
-34006	Free-list pointer doesn’t point at a free block
-34007	Invalid block size
-34008	Forbidden bits set in block size
-34009	Less than minimum size for heap block
-34010	Heap block tool large
-34011	Total free space is more than space for entire heap
-34012	Nil pointer where not allowed
-34013	Actual free space does not match tracked free space
-34014	Linked free space does not match tracked free space
-34015	Master pointer doesn’t point back to a handle block
-34016	Invalid block size adjustment
-34017	Internal block may be mangled
-34018	The heap is invalid
-34019	Caught an exception while checking the heap
-34020	Invalid heap header

Communications Transport Errors

These are the communications transport errors.

Error code	Description
–36001	Incorrect address format
–36002	Incorrect option format
–36003	Incorrect permissions
–36005	Could not allocated address
–36006	Operation not supported in the current tool state
–36008	System error
–36012	Flow control problem
–36018	Unsupported primitive
–36019	State change is in process
–36030	There's already a synchronous call pending

Sharp IR Errors

These are the Sharp infrared errors.

Error code	Description
–38001	No response - protocol time out
–38002	Cancelled - remote side cancelled operation
–38003	Protocol error
–38004	Data checksum failed
–38005	Remote side receive failed
–38006	Bad connection - allowed number of retries exceeded
–38007	SCC data errors on receive
–38008	Unspecified beaming error

Online Service Errors

These are the online service errors.

Error code	Description
–40102	Lost connection to host
–40103	Lost connection to host
–40104	The host is not responding
–40105	There is a problem reading from the host
–40106	Failed to connect to local access number

Printing Errors

These are the printing errors.

Error code	Description
–44000	Printer problem
–44001	Newton is unable to print
–44002	No printer is connected
–44003	Printer busy
–44004	Printing stopped
–44005	Lost contact with the printer
–44006	Image too complex for printer
–44100	The next sheet of paper must be inserted
–44101	The phone number must be dialed now
–44102	There is no paper tray
–44103	The wrong paper tray is attached
–44104	The printer has no paper
–44105	The printer has no ink
–44106	The printer is jammed
–44107	The printer door is open
–44108	The printer is off-line

Newton Connection Errors

These are the Newton connection errors.

Error code	Description
–46001	Connection initialization failed
–46002	Timer error
–46003	Connection request was denied by the remote
–46004	Unable to connect because there are no endpoints available
–46005	A connect request was received but no service name was given

NewtonScript Environment Errors

This section lists the different kinds of NewtonScript error codes.

Store and Soup Errors

These errors are related to stores and soups.

Error code	Description
–48001	The PCMCIA card is not a data storage card
–48002	Store format is too old to understand
–48003	Store format is too new to understand
–48004	Store is corrupted, can't recover
–48005	Single object is corrupted, can't recover
–48006	Object stream has unknown format version
–48007	Fault block is invalid
–48008	Not a fault block
–48009	Not a soup entry
–48010	Tried to remove a store that was not registered
–48011	Soup index has an unknown type
–48012	Soup index has an unknown key structure

Error code	Description
–48013	Soup index does not exist
–48014	A soup with this name already exists
–48015	Tried to `CopyEntries` to a union soup
–48016	Soup is invalid (probably from a removed store)
–48017	Soup is invalid (probably from a removed store)
–48018	Entry is invalid (probably from a removed store)
–48019	Key does not have the type specified in the index
–48020	Store is in ROM
–48021	Soup already has an index with this path
–48022	Internal error–something unexpected happened
–48023	Tried to call `RemoveIndex` on the `_uniqueID` index
–48024	Query type missing or unknown
–48025	Discovered index inconsistency
–48026	Maximum number of soup tags reached
–48027	Soup does not have a tags index
–48028	Invalid tags specification in the query
–48029	Store cannot handle the feature (for example, large objects)
–48030	Unknown sorting table
–48031	Cannot do union soup because of different sorting tables
–48032	Invalid index description
–48033	Cannot use virtual objects for soup entry keys

Object System Errors

These errors are related to the object system.

Error code	Description
–48200	Expected a frame, array, or binary object
–48201	Invalid magic pointer
–48202	Empty path
–48203	Invalid segment in path expression
–48204	Path failed
–48205	Index out of bounds (string or array)
–48206	Source and destination must be different objects
–48207	Long out of range
–48210	Bad arguments
–48211	String too big
–48212	Expected a frame, array, or binary object
–48213	Expected a frame, array, or binary object
–48214	Object is read-only
–48216	Out of heap memory
–48217	Invalid attempted use of magic pointer
–48218	Cannot create or change an object to negative size
–48219	Value out of range
–48220	Could not resize locked object
–48221	Reference to deactivated package
–48222	Exception is not a subexception of `|evt.ex|`

Bad Type Errors

These errors are caused by data of the wrong type.

Error code	Description
–48400	Expected a frame
–48401	Expected an array
–48402	Expected a string
–48403	Expected a frame, array, or binary object
–48404	Expected a number
–48405	Expected a real
–48406	Expected an integer
–48407	Expected a character
–48408	Expected a binary object
–48409	Expected a path expression (or a symbol or integer)
–48410	Expected a symbol
–48411	Expected a function
–48412	Expected a frame or an array
–48413	Expected an array or `nil`
–48414	Expected a string or `nil`
–48415	Expected a binary object or `nil`
–48416	Unexpected frame
–48417	Unexpected binary object
–48418	Unexpected immediate
–48419	Expected an array or string
–48420	Expected a virtual binary object
–48421	Expected a package
–48422	Expected `nil`

Error code	Description
–48423	Expected `nil` or a symbol
–48424	Expected `nil` or `true`
–48425	Expected an integer or an array

Compiler Errors

These errors are generated by the compiler.

Error code	Description
–48600	Could not open a listener window
–48601	Syntax error
–48603	Cannot assign to a constant
–48604	Cannot test for subscript existence; use length
–48605	Global variables not allowed in applications
–48606	Cannot have a global variable and a global constant with the same name
–48607	Cannot redefine a constant
–48608	Cannot have a variable and a constant with the same name in the same scope
–48609	Non-literal expression for constant initializer
–48610	End of input inside a string
–48611	Odd number of digits between \\u's
–48612	No escapes but \\u are allowed after \\u
–48613	Invalid hex character in \\u string
–48617	Two-digit hex number required after $\\ escape
–48618	Four-digit hex number required after $\\u
–48619	Illegal character '%c'
–48620	Invalid hexadecimal integer: %s (out of range)
–48621	Invalid real number (out of range)
–48622	Invalid decimal integer: %s (out of range)

Error code	Description
–48626	#xxxx not allowed from NTK
–48627	Not a constant
–48628	Decimal digit required after @

Interpreter Errors

These are interpreter errors.

Error code	Description
–48800	Not in a break loop
–48803	Wrong number of arguments
–48804	FOR loop BY expression has value zero
–48806	No current exception
–48807	Undefined variable
–48808	Undefined global function
–48809	Undefined method
–48810	No `_proto` for inherited send
–48811	Tried to access slot of `nil`
–48814	Local variables and FOR / WITH loops not allowed at top level
–48815	The operation would make the rich string invalid

Communications Endpoint Errors

These are the communications endpoint errors.

Error code	Description
–54000	No active input script
–54001	Badly formed script
–54002	Zero length data
–54003	Expected a specification

Error code	Description
–54004	Invalid option
–54005	Invalid end sequence
–54006	Inappropriate partial
–54007	Inappropriate termination
–54008	Inappropriate target
–54009	Inappropriate filter
–54010	Expected target
–54011	Expected template
–54012	Input specification already active
–54013	Invalid proxy
–54014	No endpoint available
–54015	Inappropriate call
–54016	Character is not single byte
–54021	Option failure
–54022	Option partial success
–54023	Option read only
–54024	Option not supported
–54025	Invalid option op code
–54026	Option not found
–54027	Option truncated

Device Driver Errors

This section lists the device driver error codes.

Tablet Driver Errors

These are the tablet errors.

Error code	Description
–56001	Attempted to call the tablet driver before it was loaded
–56002	Attempted to create a tablet driver a second time
–56003	Creation of tablet driver failed
–56004	Unable to enter bypass mode
–56005	Not in bypass mode
–56006	Cannot add sample to buffer
–56007	No new data since last polling time
–56008	Unsupported function
–56101	Timeout when calibrating
–56102	Calibration aborted

Battery Driver Errors

These are the battery driver errors

Error code	Description
–56201	Could not find battery driver
–56202	Battery error
–56203	Invalid battery selector

Other Services Errors

This section lists the error codes for other services.

Alien Store Errors

These are the alien store errors

Error code	Description
–58001	Oversize page
–58002	No such page
–58003	Cannot repage ID
–58004	No more for that page
–58005	Store is damaged

The Inside Story on Declare

This appendix describes the technical details of the declare mechanism. Knowing these technical details is not necessary to understanding what declaring a view means; they are provided primarily for completeness and to help you when you are debugging. You shouldn't write code that depends on these details.

For a basic discussion of the declare mechanism, see the section "View Instantiation" beginning on page 3-36. You should be familiar with that material before reading this appendix.

To serve as an example here, imagine a calculator application whose base view is named "Calculator." It has (among others) a child view named "Display." The Display view is declared in the Calculator view. See Figure B-1 for an illustration of this example.

In the following sections, we'll explain what happens at compile time and at run time as a result of the declare operation. A number of slots are created, which you may see in the Newton ToolKit (NTK) Inspector if you are examining the view templates.

Compile-Time Results

As a result of the declare operation, at compile time, NTK creates a slot in the place where the view is declared—that is, in the Calculator template. The name of the slot is the name of the declared view, `Display`. This slot's value is initially set to `nil`.

Another slot, called `stepAllocateContext`, is also created in the Calculator template. This slot holds an array of values (two for each view declared there). The first value in each pair is a symbol used by the system at run time to identify the name of the slot in the Calculator view that holds a reference

to the declared view. This symbol is simply the name of the declared view, `Display`.

The second value in each pair is a reference to the template for the declared view. At run time, the system will preallocate a view memory object for the declared view from this template.

Note

Protos built into the system use an analogous slot called `allocateContext`, that holds the same thing as `stepAllocateContext`. The `allocateContext` slot is for declared children from the `viewChildren` array and the `stepAllocateContext` slot is for declared children from the `stepChildren` array. ◆

Also, as a result of the declare operation, NTK creates a slot in the Display template called `preallocatedContext`. This slot holds a symbol that is the name of the template, in this case `'Display`. This symbol will be used by the system when the view is instantiated time to find the preallocated view memory object for the Display view.

Run-Time Results

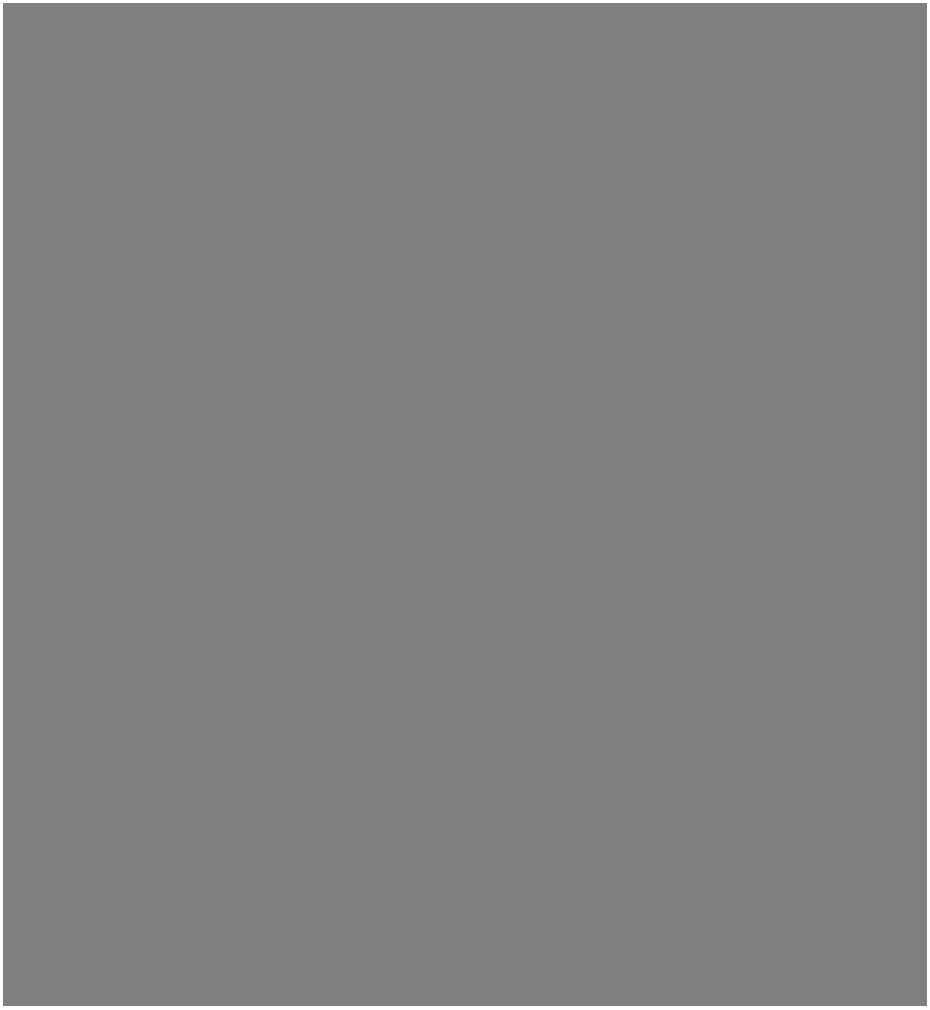
When the Calculator view is opened (even before its `viewSetupFormScript` is executed), a view memory object is preallocated for each view declared in Calculator. (The information required to do this is obtained from the `allocateContext` and `stepAllocateContext` slots.) In our example, a view memory object is created for the Display view.

The `Display` slot in the Calculator view is updated so that it points to the newly allocated Display view object.

Later in the instantiation process for the Calculator view, its child views are created and shown, including the Display view. At this time, the view system looks at the template for the Display view, sees the `preallocatedContext` slot, and knows that a view memory object has been preallocated for this view. Using this slot, the system can find the preallocated view.

The value of the `preallocatedContext` slot is the name of another slot in the Calculator view. The system locates this slot in the Calculator view, and finds there a reference to the preallocated view object. Instead of creating a new view object for the Display view, the system uses the preallocated view.

Figure B-1 Declare example



Glossary

application base view
: The topmost parent view in an application. The application base view typically encloses all other views that make up the application.

arc
: A portion of the circumference of an oval bounded by a pair of radii joining at the oval's center; a wedge includes part of the oval's interior. Arcs and wedges are defined by the bounding rectangle that encloses the oval, along with a pair of angles marking the positions of the bounding radii.

array
: A sequence of numerically indexed slots (also known as the array elements) that contain objects. The first element is indexed by zero. Like other non-immediate objects, an array can have a user-specified class, and can have its length changed dynamically.

away city
: The **emporium** that's displayed as a counterpoint to your **home city**. It defines information such as dialing area, time zone, and so on. Sometimes it is called the "I'm here" city.

binary object
: A sequence of bytes that can represent any kind of data, can be adjusted in size dynamically, and can have a

	user-specified class. Examples of binary objects include strings, real numbers, sounds, and bitmaps.
Boolean	A special kind of immediate value. In NewtonScript, there is only one Boolean, and it is called `true`. Functions and control structures use `nil` to represent false. When testing for a true / false value, `nil` represents false, and any other value is equivalent to `true`.
button host	An application that receives buttons from other applications (**button providers**).
button provider	An application that adds a button to another application (the **button host**).
card	1. Short for a **PCMCIA** card. 2. A view of information about an entry in the Names soup, formatted as a business card.
child	A frame that references another frame (its parent) from a `_parent` slot. With regard to views, a child view is enclosed by its parent view.
class	A symbol that describes the data referenced by an object. Arrays, frames, and binary objects can have user-defined classes.
constant	A value that does not change. In NewtonScript the value of the constant is substituted wherever the constant is used in code.
cursor	An object returned by the `Query` method. The cursor contains methods used to iterate over a set of soup entries meeting the criteria specified in the query. The addition or deletion of entries matching the query specification is automatically reflected in the set of entries referenced by the cursor, even if the changes occur after the original query was made.
data definition	A frame containing slots that define a particular type of data and the methods that operate on that data. The soup entries which it defines are to be used by an application and stored in its soup. A data definition is

registered with the system. The shortened term dataDef is sometimes used.

declaring a template
: Registering a template in another view (usually its parent) so that the template's view is pre-allocated when the other view is opened. This allows access to methods and slots in the declared view.

deferred recognition
: The process of recognizing an ink word that was drawn by the user at an earlier time. Deferred recognition is usually initiated by the user by double-tapping on an ink word. See also **ink** and **ink word**

desktop computer
: Either a Mac OS or Windows-based computer. Sometimes called simply "desktop."

emporium
: The permanent internal descriptions of places the user works with the Newton PDA. (Home and Office are obvious examples, but so might be "Tokyo Office" if the user travels a lot). Choosing an emporium sets up information such as local area code, dialing prefixes, time zone, and so on. This term is sometimes called "locale." The plural is "emporia."

endpoint
: A NewtonScript object that encapsulates a real-time communication session. The endpoint maintains the details of the connection and contains methods that perform communication operations.

entry
: A frame stored in a soup and accessed through a cursor. An entry frame contains special slots that identify it as belonging to a soup.

entry alias
: An object that provides a standard way to save a reference to a soup entry. Entry aliases themselves may be stored in soups.

enumerated dictionary
: A list of words that can be recognized when this dictionary is enabled. See also **lexical dictionary.**

evaluate slot
: A slot that's evaluated when NTK (Newton Toolkit) compiles the application.

event	An entry in the Dates application for a day, but not a particular time during that day.
field	An area in a view where a user can write information.
flag	A value that is set either on or off to enable a feature. Typically flag values are single bits, though they can be groups of bits or a whole byte.
font spec	A structure used to store information about a font, including the font family, the font style, and the point size.
frame	An unordered collection of slots, each of which consists of a name and value pair. The value of a slot can be any type of object, and slots can be added or removed from frames dynamically. A frame can have a user-specified class. Frames can be used like records in Pascal and structs in C, but can also be used as objects which respond to messages.
free-form entry field	A field of a `protoCharEdit` view that accepts any characters as user input.
function object	A frame containing executable code. Function objects are created by the function constructor: `func(`*args*`)` *funcBody* An executable function object includes values for its lexical and message environment, as well as code. This information is captured when the function constructor is evaluated at run time.
gesture	A handwritten mark that is recognized as having a special meaning in the Newton system, such as tap, scrub, caret, and so on.
global	A variable or function that is accessible from any NewtonScript code.
grammar	A set of rules defining the format of an entity to be recognized, such as a date, time, phone number or currency value. Lexical dictionaries are composed of sets of grammars.

home city	The **emporium** that the system uses to modify dialing information, time zone, and so on. It is usually the user's home, but when traveling, the user may set it to another city.
immediate	A value that is stored directly rather than through an indirect reference to a heap object. Immediates are characters, integers, or Booleans. See also **reference.**
implementor	The frame in which a method is defined. See also **receiver.**
inheritance	The mechanism by which attributes (slots or data) and behaviors (methods) are made available to objects. Parent inheritance allows views of dissimilar types to share slots containing data or methods. Prototype inheritance allows a template to base its definition on that of another template or prototype.
ink	The raw data for input drawn by the user with the stylus. Also known as raw ink or sketch ink.
ink word	The grouping of ink data created by the recognition system, based on the timing and spacing of the user's handwriting. Ink words are created when the user has selected "Ink Text" in the Recognition Preferences slip. Ink words can subsequently be recognized with deferred recognition.
instantiate	To make a run-time object in the NewtonScript heap from a template. Usually this term refers to the process of creating a view from a template.
lexical dictionary	A list of valid grammars, each specifying the format of an entity to be recognized, such as a date, time, phone number or currency value. See also **enumerated dictionary.**
line	A shape defined by two points: the current x and y location of the graphics pen and the x and y location of its destination.
local	A variable whose scope is the function within which it is defined. You use the `local` keyword to explicitly create a local variable within a function.

magic pointer	A constant that represents a special kind of reference to an object in the Newton ROM. Magic pointer references are resolved at run time by the operating system, which substitutes the actual address of the ROM object for the magic pointer reference.
meeting	An entry in the Dates application for a specific time during the day. People can be invited and the meeting can be scheduled for a particular location.
message	A symbol with a set of arguments. A message is sent using the message send syntax, *frame*`:`*messageName*`()`, where the message, *messageName*, is sent to the receiver, *frame*.
method	A function object in a frame slot that is invoked in response to a message.
name reference	A frame that contains a soup entry or an alias to a soup entry, often, though not necessarily, from the Names soup. The frame may also contain some of the individual slots from the soup entry.
NewtonScript heap	An area of RAM used by the system for dynamically allocated objects, including NewtonScript objects.
`nil`	A value that indicates nothing, none, no, or anything negative or empty. It is similar to `(void*)0` in C. The value `nil` represents “false” in boolean expressions; any other value represents “true.”
object	A typed piece of data that can be an immediate, array, frame, or binary object. In NewtonScript, only frame objects can hold methods and receive messages.
origin	The coordinates of the top-left corner of a view. This is usually (0, 0), however, it can be shifted, for example to scroll the contents of a view.
oval	A circular or elliptical shape defined by the bounding rectangle that encloses it.
package	The unit in which software can be installed on and removed from the Newton. A package consists of a header, which contains the package name and other

	information, and one or more **parts**, which contain the software.
package file	A file that contains downloadable Newton software.
package store	See **store part**.
parent	A frame that is referenced through the `_parent` slot of another frame. With regard to views, a parent view encloses its child views.
part	A unit of software—either code or data—held in a part frame. The format of the part is identified by a four-character identifier called its type or its part code.
part frame	The top-level frame that holds an application, book, or auto part.
PCMCIA	Personal Computer Memory Card International Association. This acronym is used to describe the memory cards used by the Newton PDA. Newton memory cards follow the PCMCIA standards.
persona	The permanent internal description of an individual person that works with a particular Newton PDA, or a particular public image of a single owner. The owner is the obvious example, but there can be many others. Choosing a persona sets up information such as name, title, birthday, phone numbers, e-mail addresses, and so on. The plural is “personae.”
picker	A type of view on the Newton that pops up and contains a list of items. The user can select an item by tapping it in the list. This type of view closes when the user taps an item in the list or taps outside of it without making a selection.
picture	A saved sequence of drawing operations that can be played back later.
polygon	A shape defined by a sequence of points representing the polygon’s vertices, connected by straight lines from one point to the next.
pop-up	See **picker**.

project	The collected files and specifications that NTK uses to build a package that can be downloaded and executed on the Newton.
proto	A frame that is referenced through another frame's `_proto` slot. With regard to views, a proto is not intended to be directly instantiated—you reference the proto from a template. The system supplies several view protos, which an application can use to implement user interface elements such as buttons, input fields, and so on.
protocol	An agreed-upon set of conventions for communications between two computers, such as the protocol used to communicate between a desktop computer and a Newton device.
raw ink	See **ink**.
receiver	The frame that was sent a message. The receiver for the invocation of a function object is accessible through the pseudo-variable `self`. See also **implementor.**
recognized text	Ink words that have been processed by the recognition system. Ink drawn by the user is converted into recognized text when the user has selected "Text" in the Recognition Preferences slip or after deferred recognition has taken place.
rectangle	A shape defined by two points—its top-left and its bottom-right corners—or by four boundaries—its upper, left, bottom, and right sides.
reference	A value that indirectly refers to an array, frame, or binary object. See also **immediate.**
region	An arbitrary area or set of areas on the coordinate plane. The outline of a region should be one or more closed loops.
resource	Raw data—usually bitmaps or sounds—stored on the development system and incorporated into a Newton application during the project build.
restore	To replace all of the information in a Newton with information from a file on the desktop.

restricted entry field
: A field of a `protoCharEdit` view that accepts as user input only the values specified in the view's template slot. For example, a field for entering phone numbers might restrict acceptable user input to numerals.

rich string
: A string object that contains imbedded ink words. Rich strings create a compact representation for strings that contain ink words and can be used with most of the string-processing functions provided in the system software. See also **rich string format**.

rich string format
: The internal representation used for rich strings. Each ink word is represented by a special placeholder character (`kInkChar`) in the string. The data for each ink word is stored after the string terminator character. The final 32 bits in a rich string encode information about the rich string.

root view
: The topmost parent view in the view hierarchy. All other views descend from the root view.

rounded rectangle
: A rectangle with rounded corners. The shape is defined by the rectangle itself, along with the diameter of the circles forming the corners (called the diameter of curvature).

self
: A pseudo-variable that is set to the current receiver.

shape
: A data structure used by the drawing system to draw an image on the screen.

siblings
: Child frames that have the same parent frame.

slot
: An element of a frame or array that can hold an immediate or reference.

soup
: A persistently stored object that contains a series of frames called entries. Like a database, a soup has indexes that can be used to access entries in a sorted order.

sketch ink
: See **ink**.

stationery
: Refers to the capability of having different kinds of data within a single application (such as plain notes and outlines in the Notepad) and/or to the capability of

	having different ways of viewing the same data (such as the Card and All Info views in the Names file). Implementing stationery involves writing data definitions and view definitions. See also **data definition** and **view definition**.
store	A physical repository that can contain soups and packages. A store is like a volume on a disk on a personal computer.
store part	A part that encapsulates a read-only store. This store may contain one or more soup objects. Store parts permit soup-like access to read-only data residing in a package. Store parts are sometimes referred to as package stores.
template	A frame that contains the data description of an object (usually a view). A template is intended to be instantiated at run time. See also **proto.**
text run	A sequence of characters that are all displayed with the same font specification. Text is represented in paragraph views as a series of text runs with corresponding style (font spec) information. See also **font spec**.
tick	A sixtieth of a second.
transport	A NewtonScript object that provides a communication service to the Newton In / Out Box. It interfaces between the In / Out Box and an endpoint. Examples include the print, fax, beam, and mail transports. See also **endpoint**.
user proto	A proto defined by an application developer, not supplied by the system.
view	The object that is instantiated at run time from a template. A view is a frame that represents a visual object on the screen. The `_proto` slot of a view references its template, which defines its characteristics.
view class	A primitive building block on which a view is based. All view protos are based directly or indirectly (through another proto) on a view class. The view class of a view is specified in the `viewClass` slot of its template or proto.

view definition	A view template that defines how data from a particular data definition is to be displayed. A view definition is registered with the system under the name of the data definition to which it applies. The shortened term viewDef is sometimes used.
wedge	A pie-shaped segment of an oval, bounded by a pair of radii joining at the oval's center.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro 630 printer. Final page negatives were output directly from the text and graphics files. Line art was created using Adobe™ Illustrator. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®.

LEAD WRITER
Christopher Bey

WRITERS
Bob Anders, Christopher Bey, Cheryl Chambers, Gary Hillerson, John Perry, Jonathan Simonoff, Yvonne Tornatta, Dirk van Nouhuys

PROJECT LEADER
Christopher Bey

ILLUSTRATOR
Peggy Kunz

EDITORS
Linda Ackerman, David Schneider, Anne Szabla

PRODUCTION EDITOR
Rex Wolf

PROJECT MANAGER
Gerry Kane

Special thanks to J. Christopher Bell, Gregory Christie, Bob Ebert, Mike Engber, Maurice Sharp, and Fred Tou.

Newton Programmer's Guide: System Software

Volume 2

Early Release

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