EXHIBIT 64 PART 3

The `wordInfo` frame provides methods that you can use to manipulate its contents; for more information, see "WordInfo Methods" (page 8-62) in *Newton Programmer's Reference.*

The alternate interpretations of a recognized word are provided as `wordInterp` frames based on the `protoWordInterp` system prototype. An array of `wordInterp` frames resides in the `wordInfo` frame's `words` slot.

Each `wordInterp` frame contains the following information:

- a string that is one interpretation of the original input strokes.
- a score indicating the recognizer's confidence in the accuracy of the interpretation.
- the dictionary identifier of the recognized word (for internal use only).
- the position occupied by this word in the original list of interpretations returned by the recognizer.

For more information, see the descriptions of the `protoCorrectInfo`, `protoWordInterp`, and `protoWordInfo` prototypes in *Newton Programmer's Reference.*

You can provide an optional `ViewCorrectionPopupScript` method that modifies or replaces the picker that displays correction information when a word is double-tapped. For a description of this method, see "Application-Defined Recognition Methods" (page 8-66) in *Newton Programmer's Reference.*

Using Custom Dictionaries

In addition to the system-supplied dictionaries, your application can use custom dictionaries to facilitate the recognition of specialized vocabulary such as medical or legal terms. It's relatively easy to create a RAM-based enumerated dictionary at run time; however, this approach is not recommended for dictionaries containing more than a few words.

Note that you cannot cause the built-in applications (Names, Dates and so on) to use custom dictionaries. The only way to enable these applications to recognize specialized terminology is to add words to the user dictionary. However, you are strongly discouraged from doing so, because each entry added to the user dictionary reduces the amount of system RAM available to the user. For more information, see "System Dictionaries" beginning on page 9-11.

Creating a Custom Enumerated Dictionary

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 NTK project data; they might 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. This function adds its argument to the specified custom dictionary.

The `AddWordToDictionary` function does not place any restrictions on the strings to be entered in the dictionary; however, your intended use of the dictionary entry may influence its content. For nonrecognition purposes, such as validating input to a field, any string is a valid dictionary entry. For use in stroke recognition, strings in enumerated dictionaries must not include spaces. The printed recognizer accepts the full set of ASCII characters; the cursive recognizer does not. Digits or non-alphabetic characters in dictionary entries used by the cursive recognizer must appear in the input string in order to be recognized. Do not use the `AddWordToDictionary` function to add items to the review dictionary; use the appropriate `reviewDict` methods instead.

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" (page 10-28), 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
   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 NTK Inspector, the type of value returned by this function may change on future Newton devices. Do not rely on the `NewDictionary` function returning an integer. ◆

Adding Words to RAM-Based Dictionaries

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, this function returns `nil`.

The `AddWordToDictionary` function may return `nil` when the word to be added is already present in the specified dictionary, or it may return `nil` because of resource limitations. It is possible to run out of system memory for dictionaries, with potentially serious consequences. Do not rely on a specific number of dictionary entries as the maximum amount that may be added safely. It is strongly recommended that you use custom dictionaries sparingly and keep them as small as possible, taking into account the possibility that other applications may require system memory for their own dictionaries or for other uses.

To populate the dictionary, you need to call the `AddWordToDictionary` function once for each item to be added. There are many ways to call a function iteratively; 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 soft resets.

IMPORTANT
Do not use the `AddWordToDictionary` global function to add words to the review dictionary; instead, use the appropriate review dictionary methods. ▲

Removing Words From RAM-Based Dictionaries

You can use the `DeleteWordFromDictionary` function to remove a specified word from a specified RAM-based dictionary. Note that this function does not make permanent changes to soups. After calling this function you must write your changes to the appropriate soup.

IMPORTANT
Do not use the `DeleteWordFromDictionary` function to remove words from the review dictionary; instead, use the appropriate review dictionary methods. ▲

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 inefficient to build it from scratch each time you need it, especially if it 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 `mySpecialDictionary` slot defined in the base view of the application. You can pass this identifier as the `GetDictionaryData` function's argument. This function returns a binary representation of the dictionary data (the words or items). You can then place this binary object in a slot in a frame and add the frame to a soup. The following code fragment assumes that the soup `kSoupName` is a valid soup created according to the Newton DTS soup-creation guidelines.

```
// get a soup in which to save the data
mySoup := GetUnionSoupAlways (kSoupName);
// create binary representation of dictionary data
local dict := GetRoot().appSym.mySpecialDictionary;
local theObj:= GetDictionaryData(dict);
```

```
// store the dictionary data
dictData := {data:theObj};
mySoup:AddXmit(dictData, nil);
```

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 part's `InstallScript` function or in the `ViewSetupFormScript` method of the view that uses the custom dictionary, as shown 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);
```

Note that RAM-based dictionaries are lost when the system resets. However, the system calls your application part's `InstallScript` function after a reset. This function can determine whether the dictionary exists and recreate it if necessary. Because this function is also called when a card with your application on it is inserted, as well as when the application is installed initially, it provides an ideal place from which to install your custom dictionary.

Using Your RAM-Based Custom Dictionary

Take the following steps to make your RAM-based dictionary available to each view that is to use it for recognition:

1. Set the view's `vCustomDictionaries` flag.
2. Create a `dictionaries` slot. You can create this slot in the view itself or in its `recConfig` frame.
3. Place your dictionary's identifier in the `dictionaries` slot.

To enable the use of custom dictionaries, you must set the `vCustomDictionaries` flag for the view that is to use the custom dictionary. This flag indicates that the view has access to a slot named `dictionaries` that specifies dictionaries to be used for recognition. The dictionaries specified in this slot are used in conjunction with any other dictionaries that may be specified for this view's use.

In addition to setting the view's `vCustomDictionaries` flag, you need to create a `dictionaries` slot in either the view or its `recConfig` frame. The `dictionaries` slot stores a single dictionary identifier or an array of dictionary identifiers. You need to install the custom dictionary in this slot using code similar to the following example.

```
// vCustomDictionaries flag already set
dictionaries := mySpecialDictionary;
```

To use system-supplied dictionaries in addition to your custom dictionary, you can enable additional view flags in the Entry Flags editor in NTK or set these additional flags procedurally. If you prefer to set view flags procedurally, you must use the `Bor` function to bitwise OR the `vCustomDictionaries` flag with any bits already set in the `viewFlags` slot. In either case, your custom dictionary must still be specified in the `dictionaries` slot.

Note that some view flags enable combinations of system dictionaries. If you want to specify explicitly which system dictionaries the view can use, set no dictionary-enabling flags other than the `vCustomDictionaries` flag and use system-supplied dictionary ID constants to add specific dictionaries to the `dictionaries` slot. For descriptions of the system-supplied dictionary ID constants, see "System-Supplied Dictionaries" (page 8-16) in *Newton Programmer's Reference*.

The following code fragment shows how you can specify dictionaries explicitly by including the appropriate constants as elements of the array in the `dictionaries` slot.

```
dictionaries :=[mySpecialDictionary, kUserDictionary,
                kCommonDictionary]
```

Regardless of the order of elements in the `dictionaries` array, the system always searches the user dictionary first. The system then searches all of the specified dictionaries in the order that they appear in the `dictionaries` array. In general, the order in which this array's items appear is not critical, except in the case of conflicting capitalization information for representations of the same word in multiple dictionaries. When multiple dictionary entries match the input, the system uses the first dictionary entry that was matched.

Note that the printed recognizer can always return words not present in dictionaries. Only the cursive recognizer may be restricted to returning only words present in dictionaries (and then only when letter-by-letter recognition is not enabled). To test your dictionary settings, use the cursive recognizer while its letter-by-letter option is disabled.

Removing Your RAM-Based Custom 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.

The `DisposeDictionary` function accepts one argument, the dictionary identifier returned by `NewDictionary`. If this identifier was stored in a slot named `mySpecialDictionary`, a line of code similar to the following example would be used to remove the custom dictionary.

```
DisposeDictionary(mySpecialDictionary);
```

Using System Dictionaries Individually

The system provides several constants that you can use to refer to system dictionaries conveniently; see "System-Supplied Dictionaries" (page 8-16) in *Newton Programmer's Reference*. You can set the `vCustomDictionaries` flag and place one or more of these constants in your view's `dictionaries` slot to specify explicitly the vocabulary it can recognize, such as first names only or names of days and months only. Note that a single constant may represent multiple dictionaries; for example, when the `kCommonDictionary` constant is specified, the system may actually add several dictionaries to the set that the view uses for recognition. The rest of this section describes the use of individual system dictionaries.

The `vNumbersAllowed` flag enables both the numeric lexical dictionary and the monetary lexical dictionary. To create a view that recognizes numeric values but not monetary values, set the `vCustomDictionaries` flag and place the `kNumbersOnlyDictionary` constant in the view's `dictionaries` slot.

Note that both the `vCustomDictionaries` and `vCharsAllowed` flags enable text recognition. The difference between these flags is in the set of dictionaries they enable. The `vCustomDictionaries` flag enables only those dictionaries specified by the `dictionaries` slot of the view performing recognition. The `vCharsAllowed` flag, on the other hand, enables several system-supplied dictionaries. To avoid unexpected results when working with custom dictionaries, be aware that setting other flags may enable additional dictionaries. Remember, also, that the printed recognizer can always return words not appearing in dictionaries.

Working With the Review Dictionary

The review dictionary object provides methods for manipulating the contents of the user dictionary (personal word list), and the expand dictionary. Although the auto-add dictionary is also part of the review dictionary, the auto-add dictionary has its own interface.

Do not use the global functions `AddWordToDictionary` and `RemoveWordFromDictionary` to make changes to the review dictionary; instead, use the appropriate review dictionary methods.

The dictionaries themselves are stored as entries in the system soup. This section describes how to manipulate these dictionaries programmatically. All of the functions and methods named in this section are described completely in "User Dictionary Functions and Methods" beginning on page 10-54.

Retrieving the Review Dictionary

To manipulate the contents of the user dictionary or expand dictionary, you send messages to the `reviewDict` object, which resides in the root view.

To obtain a reference to the `reviewDict` object, you can use code similar to the following example.

```
local reviewDict := GetRoot().reviewDict;
```

Note
Future versions of the system are not guaranteed to have the `ReviewDict` slot. You must verify that the returned value is non-`nil` before using it. ◆

You usually do not need to load the review dictionary into RAM yourself—the system does so automatically when it is reset and most flags that enable text recognition include the user dictionary automatically in the set of dictionaries they enable. You usually do not need to load the auto-add or expand dictionaries explicitly, either—the user dictionary consults these additional dictionaries automatically. However, the `LoadUserDictionary`, `LoadExpandDictionary`, and `LoadAutoAddDictionary` functions are provided for your convenience.

For general information about the user dictionary, expand dictionary and auto-add dictionary, see "System Dictionaries" beginning on page 9-11.

Displaying Review Dictionary Browsers

You can send the `Open` message to the `reviewDict` object to display the Personal Word List slip. If words have been added to the auto-add dictionary, this function displays the Recently Written Words slip automatically as well.

To display the Recently Written Words slip alone, send the `Open` message to the `autoAdd` object residing in the system's root view, as shown in the following example.

```
local auto := GetRoot().autoAdd:Open();
if auto then auto:Open();
```

Note
Future versions of the system are not guaranteed to have the `autoAdd` slot. You must verify that the returned value is non-`nil` before using it. ◆

Adding Words to the User Dictionary

The following code fragment uses the `AddWord` method of the `reviewDict` object to add words to the user dictionary. After adding one or more words, you must call the `SaveUserDictionary` function to make your changes to the user dictionary's system soup entry persistent.

```
local reviewDict := GetRoot().reviewDict;
if reviewDict then
   begin
      reviewDict:AddWord("myWord");
      reviewDict:AddWord("myOtherWord");
      SaveUserDictionary();
   end;
```

The `AddWord` method returns `true` if the word was added successfully and returns `nil` if the word was not added; however, this function may also return `nil` due to resource limitations.

It is possible to run out of system memory for dictionaries, with potentially serious consequences. Do not rely on a specific number as the maximum amount of dictionary entries that may be added safely.

If the Personal Word List slip is open when you add words to the user dictionary, its display is updated automatically. An undo action is posted for this update.

IMPORTANT
Do not use the `AddWordToDictionary` global function to add words to the review dictionary. ▲

Removing Words From the User Dictionary

The following code fragment uses the `RemoveWord` method of the `reviewDict` object to remove a word from the user dictionary. After deleting the word, you must call the `SaveUserDictionary` function to write the changes to the user dictionary's system soup entry.

```
local reviewDict := GetRoot().ReviewDict;
if reviewDict then
   begin
      reviewDict:RemoveWord("myWord");
      reviewDict:RemoveWord("myOtherWord");
      SaveUserDictionary();
   end;
```

The `RemoveWord` method returns `true` if the word was removed successfully and returns `nil` if the word was not removed. This method returns `nil` and does not remove the specified word if there are differences in case between the word in the dictionary and the word passed as the argument to the `RemoveWord` method. This method also returns `nil` when the word to be removed is not present in the review dictionary.

IMPORTANT
Do not use the `RemoveWordFromDictionary` global function to make changes to the review dictionary; instead, use the appropriate review dictionary methods. ▲

Adding Words to the Expand Dictionary

The expand dictionary (the dictionary that defines word expansions) is kept in RAM, and its size is limited to 256 words. To manipulate the expand dictionary, you send messages to the `reviewDict` object residing in the root view. The system provides methods for adding words and their associated expansions to this dictionary; retrieving the expansions associated with words; removing words and expansions from this dictionary; and saving expansion dictionary changes to the system soup.

To add a word and its expansion to the expand dictionary, you must send the `AddExpandWord` message to the `reviewDict` object. Words added to the expand dictionary must first be recognized and present in the user dictionary. If necessary, you can use the `AddWord` method of the `reviewDict` object to add the word to the user dictionary before adding it to the expand dictionary. After adding one or more words to the expand dictionary, you must call the `SaveExpandDictionary` function to write your changes to the system soup, as the following code fragment illustrates.

```
local reviewDict := GetRoot().ReviewDict;
// word must be present in user dict before adding to expand dict
if reviewDict then
   begin
      if not LookupWordInDictionary(reviewDict, "BTW") then
         begin
            reviewDict:AddWord("BTW");
            SaveUserDictionary();
         end;
      reviewDict:AddExpandWord("BTW", "by the way");
      // write changes to system soup
      SaveExpandDictionary();
   end;
```

Removing Words From the Expand Dictionary

Normally, words are added to both the expand dictionary and the user dictionary simultaneously. As a result, words removed from the expand dictionary generally must also be removed from the user dictionary. The following code fragment uses the `RemoveWord` method to remove a word from both the expand and the user dictionaries. After deleting the word, you must call the `SaveUserDictionary` function to write the changes to the system soup.

```
local reviewDict := GetRoot().ReviewDict;
if reviewDict then
   begin
      // remove word & expansion from dictionaries
      reviewDict:RemoveWord("BTW");
      SaveUserDictionary();
   end;
```

Retrieving Word Expansions

The following code fragment uses the `GetExpandedWord` method of the `reviewDict` object to retrieve the expansion associated with a specified word. This method returns `nil` if the specified word is not found in the expand dictionary.

```
local reviewDict := GetRoot().ReviewDict;
if reviewDict then
   local theExpansion := reviewDict:GetExpandedWord("BTW");
```

Retrieving the Auto-Add Dictionary

The auto-add dictionary (the list of new words to add to the user dictionary automatically) resides in RAM and its size is limited to 100 words. The system adds new words to this dictionary automatically when the cursive recognizer is enabled and the Add New Words to Personal Word List checkbox in the Text Editing Settings preferences slip is selected.

The Recently Added Words slip provides the NewtonScript interface to the auto-add dictionary. You can use code similar to the following example to obtain a reference to the RecentlyAdded Words slip.

```
local autoAddDict := GetRoot().AutoAdd;
```

Note
Future versions of the system are not guaranteed to have this slot. You must verify that the returned value is non-`nil` before using it. ◆

Usually, you do not need to load the auto-add dictionary into RAM yourself—the system does so automatically whenever the Personal Word List slip is opened or the system is reset. However, the system provides the `LoadAutoAddDictionary` function for your convenience.

Disabling the Auto-Add Mechanism

When the cursive recognizer is enabled, words not appearing in any of the currently enabled dictionaries are added to the auto-add and user dictionaries automatically as they are recognized or corrected. The value of the `doAutoAdd` slot in the system's user configuration data controls this default behavior.

However, not all input to a view is appropriate to add to dictionaries; for example, consider a spreadsheet that allows the user to select cells by entering row and column numbers—you wouldn't want to add these strings to the dictionaries as they are recognized. To disable the automatic addition of new words to the user and auto-add dictionaries, you can use either of the following techniques:

- Set the `_noautoadd` slot in the view or its `recConfig` frame to a non-`nil` value.
- Set the `_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.

Alternatively, you can set the value of the `doAutoAdd` slot in the system's user configuration data to `nil` and call the `ReadCursiveOptions` function; however, it is not recommended that you change user configuration settings without first obtaining confirmation from the user.

Adding Words to the Auto-Add Dictionary

The `AddAutoAdd` function adds a specified word to both the user and auto-add dictionaries. This function returns the value `true` after adding the word successfully. The word is not added if its unpunctuated form is present in the standard set of dictionaries enabled by the `vCharsAllowed` flag.

If the auto-add dictionary already contains its maximum of 100 words, this function does not add the new word but displays the notify icon instead. When the user taps the notify icon, it posts a notify action that displays the Recently Written Words slip. The user can then edit the Recently Written Words slip before attempting to add more words; if the user responds immediately, no new words are lost. For more information on the notify icon and notify actions, see Chapter 17, "Additional System Services."

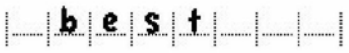
Removing Words From the Auto-Add Dictionary

The `RemoveAutoAdd` function deletes a specified word from both the user and auto-add dictionaries. This function returns `true` if the word was removed and returns `nil` if the word was not removed. This method does not remove the word if it is not present in the auto-add dictionary or if there are case inconsistencies between the argument to this function and the word actually found in the dictionary.

Using protoCharEdit Views

The `protoCharEdit` proto provides a comb-style view that facilitates the correction of individual characters in misrecognized words. The view provided by this proto uses an `rcGridInfo` frame internally to provide a horizontal row of single-character input areas. The system-supplied corrector available from the picker displayed when the user taps a recognized word makes use of this view. Figure 10-7 illustrates a typical `protoCharEdit` view.

Figure 10-7 Example of a `protoCharEdit` view



This section describes how to position a `protoCharEdit` view, how to manipulate the text string it displays, and how to restrict its input to a specified set of characters.

Positioning protoCharEdit Views

There are two ways to position a `protoCharEdit` view within its parent view. You can set the values of its `top` and `left` slots to values that position it at the top left corner of the view, or you can provide a similar value for its `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 the `viewBounds` slot based on the values of the `cellHeight`, `cellWidth`, and `maxChars` slots. On the other hand, if you provide the values of the `viewBounds` and `cellWidth` slots, then this view supplies appropriate values for the `maxChars` and `cellHeight` slots for you. This proto provides useful default values for the `cellWidth` and `cellHeight` slots; it is recommended that you do not change these values.

The technique you use depends on how you want to set the slots that this proto provides. For detailed information, see "protoCharEdit" (page 8-41) in *Newton Programmer's Reference*.

Manipulating Text in protoCharEdit Views

The default view provided by the `protoCharEdit` proto is an unformatted comb view (see page 10-4). You can provide an optional template that customizes this view's appearance and behavior. The template itself is a frame residing in the view's `template` slot. This frame may provide the following slots and methods:

- The template's `filter` slot defines a set of permissible input values. For example, a view for correcting phone numbers might restrict the set of permissible characters to numerals.
- The template's `format` slot can specify the length of the comb view and the editing characteristics of its entry fields. For example, the phone number correction view might use a format template to restrict input to a fixed number of characters and make certain entry fields non-editable. When the comb view erases invalid characters it displays the animated cloud and plays the `ROM_poof` sound that normally accompanies the scrub gesture.
- The template's `text` slot supplies a string to be displayed initially when the comb view opens. The comb view retrieves this value when its `ViewSetupFormScript` is executed.
- You can also supply optional `SetupString` and `CleanupString` functions that manipulate the string in the `text` slot.

For complete descriptions of these slots, see "Template Used by ProtoCharEdit Views"(page 8-45) and "Application-Defined protoCharEdit Template Methods" (page 8-52) in *Newton Programmer's Reference*.

The system also provides several global functions that are useful for manipulating `protoCharEdit` views and the strings they display.

To change the comb view's text string or template dynamically, call the `UseTextAndTemplate` function after setting appropriate values for the `text` or `template` slots. Alternatively, you can use the `SetNewWord` or `SetNewTemplate` and `UseNewWord` or `UseNewTemplate` functions to perform the same operations; in fact, calling these functions yourself is faster than calling the `UseTextAndTemplate` function.

To get the current value of the text in the comb view, you can send the `CurrentWord` message to the view. You must not use the value of the `text` slot directly, because unformatted comb views may add extra spaces to the string in this slot. To get a special version of the text that is formatted for display in a view other than the comb view, use the `GetWordForDisplay` function. If you are using a template, this function may return the string in a more standardized format, because it calls the template's optional `CleanupString` function before returning the string.

You may also need to know the boundaries of the word in the `text` slot when working with certain `protoCharEdit` methods and functions. The `protoCharEdit` view's `wordLeft` and `wordRight` slots provide indexes into the `text` string that you can use to determine the boundaries of a substring suitable for external display or for use as an argument to these routines. The `wordLeft` slot contains the index of the first externally-displayed character in the `text` slot. The `wordRight` slot contains the index of the position immediately following the last externally-displayed character in the `text` slot. For example, when the `text` slot holds the `"one  "`string, 1 is the value of the `wordLeft` slot and 4 is the value of the `wordRight` slot. The `dispLeft` slot contains the index of the first character in the `text` slot that is displayed—this character occupies the leftmost position in the comb view. The `dispLeft` slot normally has the value 0, but after scrolling it may have values greater than 0. The `dispIndent` slot is the offset from the leftmost edge of the view to the leftmost edge of the first character displayed.

For more information, see "protoCharEdit Functions and Methods" (page 8-47) in *Newton Programmer's Reference.*

Restricting Characters Returned by protoCharEdit Views

This section provides code examples illustrating the use of templates to restrict the set of characters that may appear in a comb view. Note that templates post-process the characters returned by the recognition system before the view displays them, rather than limiting the set of characters that the view can recognize.

The templates defined by the following code fragments are intended to serve as examples only. The system provides templates that handle formatting conventions for dates, times, phone numbers, and numeric values properly according to the user's locale. For complete descriptions of these templates, see "System-Supplied protoCharEdit Templates" (page 8-46) in *Newton Programmer's Reference.*

The following code example defines a template for a date field:

```
digits :=   "0123456789";// filters[0]
digits1 :=  "01";        // filters[1]
digits3 :=  "0123";      // filters[2]

dateTemplate := {
   string:"  /  /  ",// slashes locked by "_" in format
   format:"10_20_00",// indexes into filters array
   filters:[digits, digits1, digits3],
   };
```

This example template is used in a `protoCharEdit` view that specifies a value of `8` or more for its `maxChars` slot; hence, the eight-character strings in the `format` and `string` slots.

The cells in this example template use filters defined by the `format` and `filters` slots to restrict input to valid values.

The `format` slot specifies the valid input for each position in the comb view. Each character in the `format` string is an index into the `filters` array. In this example, the first position in the comb view is filtered by the element 1 of the `filters` array, which is the `digits1` template; the second position is filtered by element 0 of the filters array, which is the `digits` template.

You can write-protect any position in the comb view by placing an underscore (_) in the appropriate position in the `format` string. In this example, the string positions that display slashes between the digits of the date field do not allow input. These are indicated by the underscores at corresponding positions in the `format` string.

The `text` slot is not used by `protoCharEdit` views, but may be used as a default value by optional `SetupString` and `CleanupString` methods that your template supplies.

Note that the template itself does not restrict any values unnecessarily. For example, it is not wise to restrict date values according to the value of the month, because the user might enter the date before the month or the month value might not be recognized correctly. Instead, you can define a `CleanupString` function that examines the input string and indicates an error condition or modifies the string.

The following code fragment provides examples of typical `SetupString` and `CleanupString` functions.

```
myTemplate := {
   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 zeros
      str := NumberStr(StringToNumber(str));
      str;
   end,
};
```

Customized Processing of Input Strokes

Setting the `vStrokesAllowed` flag provides the view with a means of intercepting raw input data for application-specific processing. If this flag is set, strokes are passed one at a time as the argument to the view's `ViewStrokeScript` method. Your `ViewStrokeScript` method can then process the strokes in any manner that is appropriate. The view's `ViewStrokeScript` method is invoked when the user lifts the pen from the screen at the end of each input stroke.

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 in `clEditView` and `clParagraphView` views, while the `vStrokesAllowed` flag does not provide any behavior that you don't implement yourself, regardless of the kind of view performing recognition.

For example, `clEditView` and `clParagraphView` views handle system-defined gestures automatically. Thus, scrubbing in a `clParagraphView` view that sets the `vGesturesAllowed` flag does not invoke the `ViewGestureScript` method because the view handles this gesture automatically. On the other hand, a `clView` view would need to supply a `ViewGestureScript` method to process the scrub gesture because this kind of view does not provide any gesture-handling behavior of its own. Finally, remember that any view setting the `vStrokesAllowed` flag must also supply a `ViewStrokeScript` method.

Setting these flags causes the recognition system to send messages such as `ViewClickScript` or `ViewStrokeScript`, passing a unit (an object that describes the interaction of the pen with the tablet) as the argument to the corresponding methods. Units are only valid when accessed from within the methods invoked during the recognition process—you cannot save them for later use. However, you can distribute the processing of unit data as appropriate; for example, you might call the `GetPointsArray` function from within your `ViewClickScript` method and use the result later in your `ViewIdleScript` method.

IMPORTANT
Do not save units for later use—they are valid only during the recognition process. After the user interaction is complete and the various scripts utilizing a particular unit have returned, the memory allocated for that unit is freed explicitly. Subsequent use of the unit may produce bus errors or loss of significant data. ▲

Customized Processing of Double Taps

To process double taps reliably, your view's `ViewGestureScript` method can test for the presence of the `aeDoubleTap` gesture. The gesture recognizer measures time between pen events reliably even when the main NewtonScript thread is busy.

The recognition system considers a second tap to be part of a double tap when it occurs within a specified amount of time and distance relative to the first tap.

The second tap must be within 6 pixels of the first to be considered part of a double tap. Any stroke longer than 6 pixels is not recognized as a tap (or as the second tap). Measurement of the distance between taps is based on the midpoint of the start and end points of the stroke.

The amount of time that determines whether a second tap is considered part of a double tap is specified by the value of the `timeoutCursiveOption` slot in the system's user configuration data. This value ranges between 15 and 60 ticks, with a default value of 45 ticks. The user sets the value of this slot by moving the "Transform my handwriting" slider in the Fine Tuning slip. The Fine Tuning slip is available from the picker displayed by the Options button in the Handwriting Recognition preferences slip.

Your `ViewGestureScript` method is called only if the view does not handle the gesture automatically. Your `ViewGestureScript` method should return the value `true` to avoid passing the gesture unit to other `ViewGestureScript` methods, such as those supplied by views in the `_parent` chain. If you do want to pass the gesture unit to other views, your method should return the value `nil`.

Changing User Preferences for Recognition

When you must make system-wide changes in recognition behavior, you can set the values of slots in the system's user configuration data to do so. However, in most cases it is more appropriate to change the behavior of individual views, rather than system-wide settings. For information on using `recConfig` frames to specify the recognition behavior of individual views, see "Using recConfig Frames" beginning on page 10-8.

Take the following steps to change recognition settings used by all views:

1. Call the `SetUserConfig` function to set the values of one or more slots in the system's user configuration data. For a complete listing of the recognition-related slots, see "System-Wide Settings" (page 8-2) in *Newton Programmer's Reference*.
2. Call the `ReadCursiveOptions` function to cause the system to use the new settings.

Note
Normally, slot values in the system's user configuration data are set by the user from various preference slips. It is strongly recommended that you do not change any user preferences without first obtaining confirmation from the user. ◆

Modifying or Replacing the Correction Picker

Views that recognize text can provide an optional `ViewCorrectionPopupScript` method that modifies or replaces the picker displayed when a recognized word is double-tapped. For more information, see "ViewCorrectionPopupScript" (page 8-75) in *Newton Programmer's Reference.*

Using Stroke Bundles

The system provides functions that allow you to retrieve or manipulate stroke data, such as the tablet 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.

To access the ink in a view, use one of the functions documented in "Text and Ink Input and Display Reference" (page 7-1) in *Newton Programmer's Reference.* Functions that allow you to manipulate ink include the `ParaContainsInk`, `PolyContainsInk`, and `GetInkAt` functions.

To perform deferred recognition on the strokes in a stroke bundle, pass the stroke bundle to one of the `Recognize`, `RecognizePara` or `RecognizePoly` functions. For more information, see "Deferred Recognition" on page 10-5.

The system software provides a number of functions for working with stroke bundles. These functions allow you to extract information from a stroke bundle and convert the information in stroke bundles into other forms. The stroke bundle functions are documented in "Stroke Bundle Functions and Methods" (page 8-83) in *Newton Programmer's Reference.*

Stroke Bundles Example

This section shows an example of working with stroke bundles before they are passed to the view performing recognition. One way to do this, as shown in the following code fragment, is to implement the `ViewInkWordScript` method of an input view. The `ViewInkWordScript` method is described in "ViewInkWordScript" (page 7-56) in *Newton Programmer's Reference.*

```
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.

Summary of Advanced Topics in Recognition

See also “Summary” beginning on page 9-31 in Chapter 9, “Recognition.”

Constants

See also Chapter 9, “Recognition,” which includes the following summaries: “Text Recognition View Flags” on page 9-31; “Non-Text Recognition View Flags” on page 9-32; and “View Flags Enabling Lexical Dictionaries” on page 9-33.

```
kStandardCharSetInfo // cursive recognizer
kUCBlockCharSetInfo   // printed recognizer
ROM_canonicalBaseInfo // System-supplied rcBaseInfo frame
ROM_canonicalCharGrid // System-supplied rcGridInfo frame
```

Enumerated Dictionaries

Dictionary ID Constant	Value	Contents
`kUserDictionary`	31	Words added by the user.
`kCommonDictionary`	0	Commonly-used words.
`kCountriesDictionary`	8	Names of countries.
`kDaysMonthsDictionary`	34	Names of days and months.
`kFirstNamesDictionary`	48	First names.
`kLocalCitiesDictionary`	41	Names of cities.
`kLocalPropersDictionary`[1]	2	Proper names.
`kLocalStatesDictionary`	43	Names of states, provinces, etc.
`kSharedPropersDictionary`	1	Proper names, company names, state or province names and abbreviations.

[1] Locale-specific dictionary

Lexical Dictionaries

Dictionary ID Constant	Value	Contents
`kLocalDateDictionary`	`110`	Date formats.
`kLocalNumberDictionary`[1]	`113`	Currency and numeric formats.
`kLocalPhoneDictionary`	`112`	Phone number formats.
`kLocalTimeDictionary`	`111`	Time formats.
`kMoneyOnlyDictionary`[1]	`118`	Currency values and formats.
`kNumbersOnlyDictionary`[1]	`117`	Numeric values and formats.
`kPostalCodeDictionary`	`116`	Postal code formats.

[1] Locale-specific dictionary

System-Supplied RecConfig Frames

RecConfig Constant	Behavior of recConfig frame
`ROM_rcInkOrText`	Recognize ink text or text.
`ROM_rcPrefsConfig`	Recognize according to user settings.
`ROM_rcDefaultConfig`	None; you supply slot values.
`ROM_rcSingleCharacterConfig`	Recognize single characters.
`ROM_rcTryLettersConfig`	Recognize letter-by-letter.
`ROM_rcRerecognizeConfig`	Deferred recognition.
`rcBaseInfo`	Defines baseline.
`rcGridInfo`	Defines single-letter input view.

Data Structures

See also Chapter 9, “Recognition,” which includes the following summaries: “Recognition-Related User Configuration Slots” on page 9-33;

RecConfig Frame

See `protoRecConfig` in “Recognition System Prototypes” beginning on page 10-49.

System-Supplied RecConfig Frames

```
// recognize ink or text
ROM_rcInkOrText :=
   {
   // allow user to enable text recog from recToggle
   allowTextRecognition: true, // default
   // return ink text when text recognizer disabled
   doInkWordRecognition: true, // default
   …}

// recognize according to user prefs
ROM_rcPrefsConfig :=
   {
   // allow user to enable text recog from recToggle
   allowTextRecognition: true, // default
   // allow user to enable shape recog from recToggle
   allowShapeRecognition: true, // default
   …}

// generic recConfig frame - you supply useful values
ROM_rcDefaultConfig :=
   {
   // true enables recognition of punctuation marks
   punctuationCursiveOption: nil, // default
   // list of dictionaries used for recognition
   dictionaries: nil, // default
   // true enables letter-by-letter option
   rcSingleLetters: nil, // default
   // Holds an rcBaseInfo frame
   rcBaseInfo: nil, // default
   // bitfield specifying recognition configuration
   inputMask: 0x0000, // default
   …}

// use as-is to configure single-character input views
ROM_rcSingleCharacterConfig :=
   {
   // do not change value of this slot
   _proto: ROM_rcDefaultConfig, // default
   //interpret all input strokes as a single word
   letterSpaceCursiveOption: nil, // default
   // enable letter-by-letter option
   rcSingleLetters: true, // default
   // use custom dictionaries only
   inputMask: vCustomDictionaries, // default
   // dictionaries to use for recognition
   dictionaries: kSymbolsDictionary, // default
   // don't enable symbols dictionary twice
   inhibitSymbolsDictionary: true // default
   …}

// recognize letter-by-letter instead of w/ dictionaries
ROM_rcTryLettersConfig :=
   {
   // do not change value of this slot
   _proto: ROM_rcDefaultConfig, // default
   //interpret all input strokes as a single word
   letterSpaceCursiveOption: nil, // default
   // recognize non-dictionary words and numbers
   inputMask: vLettersAllowed+vNumbersAllowed, // default
   …}
// use as-is to implement your own form of deferred recog
ROM_rcRerecognizeConfig :=
   {
   // use value of doTextRecognition slot
   allowTextRecognition: true, // default
   // text recognition enabled
   doTextRecognition: true, // default
   // amount of time to spend analyzing input
   speedCursiveOption: 2, // default
   // do not segment strokes
   letterSpaceCursiveOption: nil, // default
   …}
```

Supporting Frames Used In RecConfig Frames

```
// specifies baseline info to recognizer
rcBaseInfo :=
   {
   // y-coordinate of the view's baseline
   // in screen coords (global coords).
   base: int,
   // Positive offset (in pixels) from base
   // to the top of a lowercase "x"
   smallHeight: int,
   // Positive offset (in pixels) from base
   // to the top of an uppercase "X"
   bigHeight: int,
   // Positive offset (in pixels) from base
   // to the bottom of a lowercase "g"
   descent: int,
   …}

// use w/ rcBaseInfo to define grids of input cells
rcGridInfo :=
   {// all coordinates are global (screen) coordinates
   // coord of left edge of upper-left box in grid
   boxLeft: int,
   // coord of right edge of upper-left box in grid
   boxRight: int,
   // distance in pixels from one boxLeft to next boxLeft
   xSpace: int,
   // coord of topmost edge of upper-left box in grid
   boxTop: int,
   // coord of bottom edge of upper-left box in grid
   boxBottom: int,
   // distance in pixels from one boxTop to next boxTop
   ySpace: int
   }
```

ProtoCharEdit Template

```
aCharEditTemplate :=
   {
   format: string, // string array indexes or underscores
   filters: [str1, str2, … strN], // valid input values
   string: string // initial string to display
   // optional method you supply
   // sets value of charEditView.text slot
   SetupString: func (str) begin … end,
   // optional method you supply
   // formats charEditView.text string for ext display
   CleanupString: func (str)begin … end
   }
```

System-Supplied ProtoCharEdit Templates

```
GetLocale().phoneFilter // phone number template
GetLocale().dateFilter // date template
GetLocale().timeFilter // time template
ROM_numberFilter // general-purpose integer template
```

Stroke Bundle Frame

```
aStrokeBundle :=
   {
   //bounding rectangle of ink strokes
   bounds: {top, left, right, bottom},
   // strokes in the bundle
   strokes: [ binaryObj1, binaryObj2, ... binaryObjN]
   }
```

Recognition System Prototypes

protoRecConfig

```
aRecConfigFrame := {
   // enabled recognizers and dicts
   inputMask: bitField,
   // true enables text recog if doTextRecognition
   // is also true
   allowTextRecognition: Boolean,
   // true enables shape recog if doShapeRecognition
   // is also true
   allowShapeRecognition:Boolean,
   // true enables text recognition unconditionally
   doTextRecognition:Boolean,
   // true enables shape recognition unconditionally
   doShapeRecognition:Boolean,
   // true enables ink text unconditionally
   doInkWordRecognition:Boolean,
   // amount of time to spend recognizing input
   speedCursiveOption:int,
   // relative amount of time between distinct strokes
   timeoutCursiveOption:int,
   // true enables letter-by-letter option
   letterSpaceCursiveOption:Boolean,
    // dictionaries to use when vCustomDictionaries is set
    // single values need not reside in an array
    dictionaries: [dictId1, dictID2, … dictIdN],
    // optional baseline info
    rcGridInfo: frame,
    // optional single-letter input view info
    rcSingleLetters: frame,
    // true disables symbols dictionary
    inhibitSymbolsDictionary: Boolean,
    …}
```

protoRecToggle

```
aRecToggleView :=
    {
    // current setting of recToggle view
    // this slot may be provided by _parent chain
    _recogSettings: systemSuppliedValue,
    // order of items in recToggle picker
    _recogPopup: [sym1, sym2 … symN],
    // optional index into _recogPopup array
    defaultItem: int,
    …}
```

protoCharEdit

```
aCharEditView :=
    {
    // screen coordinates of top edge of comb view
    top:int, // Required when viewBounds not provided
    // screen coordinates of left edge of comb view
    left: int, // Required when viewBounds not provided
    // dimensions & placement of comb view
    viewBounds: frame, // Required when top & left not provided
    // maximum number of cells in comb view; default value is 8
    maxChars: int, // Required; sys provides if you provide viewBounds
    // true causes comb view to display divider lines between cells
    frameCells: Boolean,// Optional; default value is nil
    // width of a cell in pixels; must be even number; default is 12
    cellWidth: int, // system calculates from your top & left values
    // pixels of blank space between cells in comb view
    cellGap: int, // system-provided default value is 6
    // pixels from top of viewBounds to dotted line at bottom of comb
    viewLineSpacing: int, // system-provided default is 30
// height of cells in pixels
cellHeight: int, // system-provided default is 50
// recConfig frame specifying this view's recog behavior
recConfig: frame, // system provides default
// specifies appearance & behavior of formatted comb view
template: frame, // optional protoCharEdit template
// string displayed when view opens; arg to SetupString method
text: string, // optional
// index of leftmost non-space character in comb view
wordLeft: int, // system-provided value
// index of cell to the right of rightmost non-space character
wordRight: int, // system-provided value
// index into text slot of character occupying leftmost cell
dispLeft: int, // system-provided value; changes after scrolling
// offset in pixels from leftmost edge of comb view
// to leftmost edge of first cell displayed
dispIndent: int,
// return word from comb view w/out leading/trailing spaces
CurrentWord: function,
// return cleaned-up version of word suitable for ext display
GetWordForDisplay: function, // calls CleanupString if provided
// deletes specified text from comb view
DeleteText: function,
// scrolls contents of comb view left or right as specified
Scroll: function,
// makes comb view use current values of text & template slots
UseTextAndTemplate: function,
// Sets the string displayed by the comb view
SetNewWord: function,
// performs internal initialization using current values of
// text and template slot; call after calling SetNewWord
UseNewWord: function,
// Returns true when template's format slot is non-nil
FixedWord: function,
// Returns number of chars residing in templates format slot
FixedWordLength: function,
// optional app-defined methods
// you supply optional fn to update external display
DisplayExternal: function, // message sent when comb view changes
// you supply optional fn to save your undo info
SaveUndoState: function, // message sent when comb view changes
// you supply optional fn to do app-specific tasks for undo
RestoreUndoState: function, // msg sent to undo comb view changes
...}
```

protoCharEdit Templates

```
ROM_numberFilter // general-purpose numeric template
GetLocale().timeFilter // time template
GetLocale().dateFilter// date template
GetLocale().phoneFilter // phone numnber template
```

ProtoCharEdit Functions

```
MapAmbiguousCharacters(str)
UnmapAmbiguousCharacters(str)
```

ProtoCorrectInfo

```
aCorrectInfoFrame :=
   {
   info: [frame1, frame2 … frameMax] // wordInfo frames
   // maximum number of frames in info array
   max: 10, // default value
   // system-supplied methods
   Offset: function, // move, merge or delete wordInfo
   // remove view's wordInfo from correctInfo
   RemoveView: function,
   // return wordInfo frame at specified offset
   Find: function,
   // return wordInfo frame at specified offset,
   // creating one if none found
   FindNew: function,
   // extract wordInfo from unit & add to correctInfo
   AddUnit: function,
   // add specified wordInfo to correctInfo
   AddWord: function,
   // delete range of wordInfo frames from correctInfo
   Clear: function,
   // copy range of wordInfo frames from view
   // into a new correctInfo frame
   Extract: function,
   // copy range of wordInfo frames from source
   // correctInfo frame into dest correctInfo frame
   Insert: function,
   …}
```

ProtoWordInfo

```
aWordInfoFrame :=
   {
   // ID of view that owns this data; returned by GetViewID
   id: int,
   // first char's offset into clParagraphView view
   Start: int,
   // last char's offset into clParagraphView view
   Stop: int,
   flags: forSystemUseOnly, // do not use this slot
   unitID: forSystemUseOnly, // do not use this slot
   // array of wordInterp frames; see page 10-53
   words: [wordInterp1, wordInterp2, ... wordInterpN]
   // stroke data from original input
   strokes: strokeBundleFrame, // see page 10-49
   unitData: forSystemUseOnly, // do not use this slot
   // sets list of words held by this wordInfo frame
   SetWords: function,
   // returns array of strings held by wordInterp frames
   GetWords: function,
   // Adds first word in this word list to auto-add and user dicts
   AutoAdd: function,
   // Removes first word in this list from auto-add and user dicts
   AutoRemove: function,
   }
```

protoWordInterp

```
aWordInterpFrame :=
   {
   // one interpretation of input strokes
   word: string,
   // recognizer's confidence in this interpretation
   score: int, // 0 is good score, 1000 is bad score
   // dictionary id of recognized word
   label: int, // internal use only
   // this word's rank in orig list of interpretations
   index:int, // internal use only
}
```

Additional Recognition Functions and Methods

Dictionary Functions

`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

`AddAutoAdd(`*word*`)`
`RemoveAutoAdd(`*word*`)`
reviewDict`:AddWord(`*word*`)`
reviewDict`:RemoveWord(`*word*`)`
`LoadUserDictionary()`
`SaveUserDictionary()`
reviewDict`:AddExpandWord(`*word*`,` *expandedWord*`)`
reviewDict`:GetExpandedWord(`*word*`)`
reviewDict`:RemoveExpandedWord(`*word*`)`
`LoadExpandDictionary()`
`SaveExpandDictionary()`

Recognition Functions

`BuildRecConfig(`*viewRef*`)`
`GetPoint(`*selector*`,` *unit*`)`
`GetPointsArray(`*unit*`)`
`GetScoreArray(`*unit*`)`
`GetViewID(`*viewRef*`)`
`GetWordArray(`*unit*`)`
`StrokeBounds(`*unit*`)`
`StrokeDone(`*unit*`)`
`PurgeAreaCache()`

Deferred Recognition Functions

`Recognize(`*strokes*`, `*config*`, `*doGroup*`)`
`RecognizePara(`*para*`, `*start*`, `*end*`, `*hilite*`, `*config*`)`
`RecognizePoly(`*poly*`, `*hilite*`, `*config*`)`

Application-Defined Methods

view`:ViewClickScript(`*stroke*`)`
view`:ViewStrokeScript(`*stroke*`)`
view`:ViewGestureScript(`*stroke*`, `*gesture*`)`
view`:ViewWordScript(`*stroke*`)`

CorrectInfo Functions

```
GetCorrectInfo() // return correctInfo frame
// return view identifier for use w/ correctInfo methods
GetViewID(view)
// extract wordInfo from word unit
GetCorrectionWordInfo(wordUnit) // call in ViewWordScript
```

Inker Functions

`InkOff(`*unit*`)`
`InkOffUnHobbled(`*unit*`)`
`SetInkerPenSize(`*size*`)`

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)*

C H A P T E R 1 1

Data Storage and Retrieval

The Newton operating system supplies a suite of objects that interact with each other to provide data storage and retrieval services. This chapter describes the use of these objects—stores, soups, cursors, and entries—to save and retrieve data. If you are developing an application that saves data, retrieves data, or provides preexisting data, you should familiarize yourself with the contents of this chapter.

Before reading this chapter, you should understand the following sections in Chapter 1, "Overview."

- "Memory" on page 1-3 describes the use of random access memory (RAM) by the operating system and applications.
- "Packages" on page 1-4 describes the object that encapsulates code, scripts, and resources as a Newton application.
- "Object Storage System" on page 1-5 provides a brief introduction to the most important data storage objects provided by the Newton operating system.

About Data Storage on Newton Devices

This section introduces Newton data storage objects and describes their interaction and use. Additional special-purpose data storage objects are described in Chapter 12, "Special-Purpose Objects for Data Storage and Retrieval."

Introduction to Data Storage Objects

Newton devices represent data as objects. The NewtonScript programming language provides four basic object types that applications can use to represent data:

Immediate	A small, immutable object such as a character, integer or Boolean value.
Binary	Raw binary data.
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 common to many programming languages, they are not discussed further here. For complete descriptions 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 NewtonScript object to which you can send messages. In addition, the following characteristics of frames make them a particularly flexible and efficient way to store data:

- Frames are sized dynamically—they grow and shrink as necessary.
- All frames support a common set of predefined NewtonScript data types that allows them 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 held in a **slot,** which may be thought of as a field in the database record. Unlike database records, however, frames need not contain the same complement of slots.

Any slot can hold any NewtonScript data type, including 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 hold other frames, as well as references to frames, slots, and NewtonScript objects. A frame's ability to reference other frames from its slots allows it 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 between frames. For detailed descriptions of NewtonScript syntax, system-supplied data types, dynamic slot lookup, message-passing, and inheritance in NewtonScript, see *The NewtonScript Programming Language.*

Other than the requirement that data must reside in a slot, frames don't impose any structure on their data. In practical use, though, the slots in a frame tend to be related in some way, usually holding related data and methods which operate on that data. In this way, the frame exemplifies 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 saved in a data structure called a **soup,** which is an opaque object that provides a persistent, dynamic repository for data. Unless removed intentionally, soups remain resident on the Newton device even when the application that owns them is removed.

The only NewtonScript object you can save in a soup is a frame; recall, however, that any slot in the frame can hold any NewtonScript data type and multiple data types can reside in a single frame. 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 an application, or the user may introduce them by inserting a storage card in the Newton device.

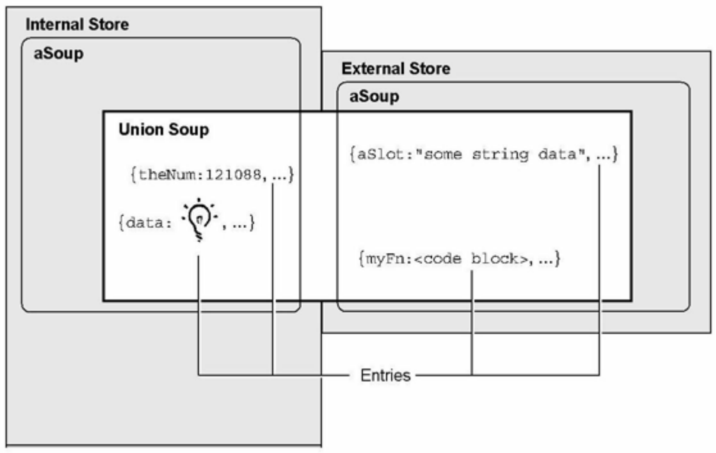
The soup resides on a **store,** which is a logical data repository on a physical storage device. A store may be likened to a disk partition or volume on a conventional computer system; just as a disk can be divided logically into multiple partitions, a physical storage device can house multiple stores. The Newton platform supports a single internal store and one or more external stores on PCMCIA devices. Applications can use as many soups as they need, subject to the availability of memory space on stores and in the NewtonScript heap.

Each store is identified by a name, which is not necessarily unique, though each store has a nearly unique random number identifier called a **signature.** The store's signature is assigned by the system when the store is created.

Soups can reside on internal or external stores; a special kind of object, the **union soup,** represents multiple soups as a single entity, regardless of their locations on various physical stores. For example, when a PCMCIA card is installed, application data may be distributed between the internal and card-based soups. The union soup object provides a way to address multiple soups having the same name as a single "virtual" soup. Figure 11-1 illustrates the concept of a union soup graphically.

It's important to understand that there is only one kind of soup object in the system; a union soup object simply represents the 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 (also called *member soups*) are no different from soups that are not part of a union. Unless specifically noted otherwise, anything said about soups in this text applies equally to union soups.

Figure 11-1 Stores, soups and union soups



In general, you'll want to use union soups for most purposes, especially for saving most data the user creates with your application. Applications must allow the user to choose whether to save new data on the internal or external store; using union soups makes this easy to do.

An application creates a union soup by registering a **soup definition** frame with the system; registering the soup definition causes the system to return a union soup object to which the application can send messages that save and retrieve data. This object may represent a new soup, one created previously, or no soups (if, for some reason, all of the union's member soups are unavailable). For a detailed discussion of soup creation, see "Soups" beginning on page 11-7.

All soups save frame data as objects called entries. An **entry** is a frame that has been added to a soup by means of any of several system-supplied methods provided for this purpose. Note that you cannot create a valid entry by simply adding certain slots and values to a frame—the system must create the entry for you from a frame presented as an argument to one of the entry-creation methods.

Returning to the database analogy, you can think of entries as 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 data by requesting it, rather than by examining its records directly.

Your request for soup data takes the form of a `Query` message sent to the soup or union soup object. The `Query` method accepts as its argument a frame known as the **query specification** or **query spec.** The query spec describes the kind of information the query returns. The order in which soups return data items is imposed by an **index** you define for a specified soup.

If you've ever used an array, you are already familiar with the concept of an index. Each element of the array is associated with a unique numeric value called a **key**. These key values can be sorted, thus imposing order on their associated data items (the elements of the array). In the case of a common array, a single numeric index sorts the array elements in ascending key order.

Key values can also be used to reference or retrieve an indexed item. For example, arrays allow you to reference or retrieve the data at a particular position in the array without regard to the actual content stored at that position. Soup indexes provide similar capabilities for soup data: they allow you to find and sort soup entries associated with specified key values without specific knowledge of the data associated with a particular key value.

You can index soup entries on any slot value you need to use as a key for extracting them from the soup. For example, you could retrieve entries having a certain creation date, or entries in which a particular string is present, and so on. Soups can be created with a set of default indexes you specify and you can also add new indexes to existing soups. Indexes are discussed in more detail in "Indexes" beginning on page 11-8.

A soup responds to a query by returning a **cursor** 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 reflected automatically in the set of entries that the cursor returns.

The cursor responds to messages that position it within the set of entries it references and extract individual entries from this set. Until an entry is extracted from the cursor, its data resides in the soup that was queried to generate the cursor.

The first time a slot in the entry is referenced—whether to read its value, set its value, or to print its value in the Inspector—the system creates a normal frame from it that is referenced by a special area of the NewtonScript heap known as the **entry cache.** Changes to the entry's soup data are actually made in the cached frame, not the permanent store; hence, changes to a soup entry are not persistent until the cached frame is written back to a soup. This scheme 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, unmodified soup entry.

Because the frame-based storage model facilitates the sharing of data, the system provides a soup change notification mechanism that you can use to advise other objects of changes to soups or soup data. All the methods that add, modify, or delete soups or soup entries provide the option to execute registered callback functions in response to changes in specified soups. Soup changes for which applications might require notification include creating soups; deleting soups; and adding, removing, or changing individual soup entries. The soup change notification mechanism is discussed in more detail in “Using Soup Change Notification” beginning on page 11-63.

In summary, most applications that work with dynamic data perform the following operations, which are described in this chapter:

- creating and using frames
- storing frames as soup entries
- querying soups to retrieve sets of entries
- using cursor objects to work with 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

Where to Go From Here

You should now have a general understanding of how stores, soups, queries, cursors, and entries interact. It is strongly recommended that you read the remainder of this section now—it provides important details you’ll need to know in order to work with the Newton data storage system. However, if you are anxious to begin experimenting with Newton data storage objects, you can skip ahead to “Programmer’s Overview” on page 11-25 and read the remainder of this section at another time.

Stores

Although soups and packages reside on stores, the occasions on which you’ll need to interact with stores directly are rare—the system manages hardware interaction for you, creates union soups automatically as needed, and provides a programming interface that allows you to perform most union soup operations without manipulating the stores on which individual member soups reside. Occasionally, you may need to message a store directly in order to create or retrieve a soup that is not part of a union, or you may need to pass a store object as an argument to certain methods; otherwise, most applications’ direct interaction with stores is limited.

In general, only specialized applications that back up and restore soup data need to manipulate stores directly. However, the system provides a complete developer interface to stores, as documented in “Data Storage and Retrieval Reference” (page 9-1) in *Newton Programmer’s Reference.*

For information on using store objects, see “Using Stores” beginning on page 11-29.

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 applications such as Newton Package Installer or Newton Backup Utility.

Packages can be read from a data stream or directly from memory. For example, Newton Connection Utility uses a data stream protocol to load a package into the Newton system from a MacOS or Windows computer. However, it is much more common to use packages directly from memory, as the user does after the package has been installed on the Newton device.

For a more detailed discussion of packages, see “Parts” on page 12-3 in Chapter 12, “Special-Purpose Objects for Data Storage and Retrieval.”

Soups

This section provides important background information about soup objects. Topics discussed here include

- soups vs. union soups
- related data structures such as soup definitions, indexes, index specification frames, and tags
- automatic creation of soups
- saving user preferences in the system soup

Applications using soup-based data storage must respect the user’s default store preferences for writing soup entries and create soups only as necessary. The use of union soups makes it easy to observe these requirements. Union soups provide methods that respect the user’s default store preferences automatically when adding new entries. These ROM-based methods are also much faster than equivalent NewtonScript code. Union soups also provide methods you can use on those occasions when you must specify the store on which to save soup entries.

Another good reason to use union soups is that applications almost never need to create them explicitly. Once a soup definition is registered with the system, individual members of the union soup it defines are created automatically as needed.

A **soup definition** is a frame that provides information about a particular union soup. The soup definition supplies descriptive information about the union soup and information required to create its member soups.

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 default set of indexes with which the soup is created. For a complete description of the slots in the soup definition frame, see the section "Soup Definition Frame" (page 9-2) in *Newton Programmer's Reference*.

Methods that add an entry to a union soup use the information in its soup definition to create a member soup to hold the new entry if the member soup is not present on the appropriate store at the time the entry is added. If a member of the union is present on the specified store, the new entry is added to the existing member soup and a new soup is not created. In most cases, the store in question is specified by the user's preferences for the default storage of new data items; if necessary, however, you can specify by store the member soup in which the new entry is to reside. Note also that you can create union soup members explicitly, if necessary.

If no frames have ever been added to a particular union soup, the union's member soups may not exist at all. You can add entries to a union soup in this state (member soups are created automatically), but you cannot query a union soup that has no members.

Member soups may be unavailable for other reasons, as well. For example, the user might have removed a member soup temporarily by ejecting the card on which it resides or might have removed the soup permanently by scrubbing it in the Extras Drawer.

The descriptive information in a soup definition frame can be used to supply information about a soup for use by the system, applications, or 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.

To make a soup definition available for use, you must first register it with the system. For information on registering and unregistering soup definitions, see the section "Registering and Unregistering Soup Definitions" beginning on page 11-33.

NewtApp applications also make use of soup definitions; for more information, see Chapter 4, "NewtApp Applications."

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. Because the system can retrieve and sort index key values without reading their associated soup entries into memory, indexes provide a fast and efficient means of finding soup entries.

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.

You can create your own specialized indexes for any soup. You need to create an index for each slot or set of slots on which the soup will be searched frequently. It is preferable to define your indexes in the appropriate soup definition, but you can add indexes to an existing soup if necessary.

An index generated against a single key value is called a **single-slot index.** A single-slot index selects and sorts soup entries according to the value of a single slot specified when the index is created. An index generated against multiple key values is called a **multiple-slot index.** A multiple-slot index can select and sort soup entries according to the values of multiple slots. A multiple-slot index can associate up to a total of six key values with each entry it indexes. You can create multiple indexes for any soup.

The characteristics of an index are specified by an **index specification frame** or **index spec.** 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 the indexed slots. The precise format of the index spec frame varies according to whether it defines a single-slot index or a multiple-slot index. For complete descriptions of index specs, see "Single-Slot Index Specification Frame" on page 9-5 and "Multiple-Slot Index Specification Frame" (page 9-6) in *Newton Programmer's Reference*.

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 into memory; for example, one could select the subset of entries tagged `'business` from the `ROM_CardFileSoupName` soup used by the built-in Names application. In fact, "filing" Newton data items in "folders" is a user-interface illusion—the data really resides in soup entries 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. The system treats missing tags as `nil` values. For more information, see "Tag-based Queries" on page 11-14.

A **tags index specification frame,** or **tags index spec,** defines the characteristics of a soup's tags index. Like an index spec, a tags index spec can be used to create a default tags index on a new soup or add a tags index to an existing soup. For a complete description of the slots in a tags index spec frame, see the section "Tags Index Specification Frame" (page 9-8) in *Newton Programmer's Reference*.

To better support the use of languages other than English, soup indexes and queries can be made sensitive to case and diacritical marks in string values. (Normally, string comparison in NewtonScript is insensitive to case and diacritics.) This behavior is intended to allow applications to support the introduction of non-English data easily; for example, the user might insert a PCMCIA card containing data from a different locale. To take advantage of this behavior, the application must create an internationalized index for the soup and the query must request the alternate sorting behavior explicitly in its query spec. For more information, see “Internationalized Sorting Order for Text Queries” on page 11-45.

Saving 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 your application’s user preferences data. There are several good reasons for always saving user preferences data on the internal store:

- If your application is on a card that is moved from one Newton device to another, it acts the way the users of the respective Newton devices think it should.
- It rarely makes sense to distribute preferences data among several storage cards.
- It’s difficult to guarantee that your application will always have access to any particular card.
- If your application is on the internal store and it simply adds preference data to the default store, the preference data could be saved on an external store that becomes unavailable to the application when a card is ejected.

Hence, the built-in `ROM_SystemSoupName` soup on the internal store is usually the ideal place to keep 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 26, “Utility Functions.” For more information about the `ROM_SystemSoupName` soup itself, see Chapter 19, “Built-in Applications and System Data.”

Queries

To retrieve entries from a soup or union soup, you perform a **query** by sending the `Query` message to the soup or union soup. The `Query` method accepts as its argument a frame known as a **query specification** or **query spec.** The query spec defines criteria for the inclusion of entries in the query result. You can think of the query spec as a filter that the `Query` method uses to select a subset of the soup’s data. Queries can test index key values or string values and perform customized tests that you define.

A single query spec can specify multiple criteria that entries must meet in order to be included in the result of the query. For example, you can specify that your query return all entries created after a certain date that are tagged `'business` but do not contain the `"paid"` string. For instructional purposes, this chapter discusses separately each test that a query spec may include.

Querying for Indexed Values

Queries can retrieve items according to the presence of one or more index keys and can test key values as well. A query that tests for the presence or value of an index key is called an **index query.**

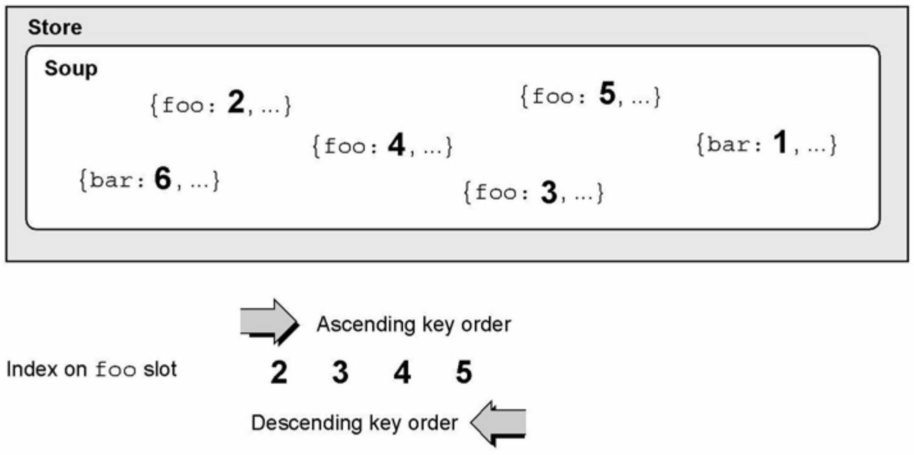
Soups that have single-slot indexes allow queries to use a single index key to select soup entries. Detailed information is provided in "Querying on Single-Slot Indexes" beginning on page 11-39.

Soups that have multiple-slot indexes allow queries to use multiple index keys to select soup entries. Detailed information is provided in "Querying on Multiple-Slot Indexes" beginning on page 11-47.

Index queries can be based only on slot names for which an index has been generated. For example, to select entries according to the presence of the `foo` slot, the soup that receives the `Query` message must be indexed on the `foo` slot. Entries not having a `foo` slot are not included in the set of entries referenced by the `foo` index.

Although the entries in the soup are not actually in any particular order themselves, the index keys associated with them can be sorted in a specific order that is defined for each NewtonScript data type. Thus, you can envision the contents of an index as a sequence of entries arranged in key order, as shown in Figure 11-2.

Figure 11-2 An index provides random access and imposes order



The `aSoup` soup shown in Figure 11-2 is indexed on the `foo` slot, which means that the value of each entry's `foo` slot is used as its index key. Only those entries containing a `foo` slot are included in this index. By sorting key values, the index imposes order on their corresponding soup entries, which are otherwise unordered.

Indexes sort key values in ascending order unless the index spec frame used to create a particular index specifies descending order.

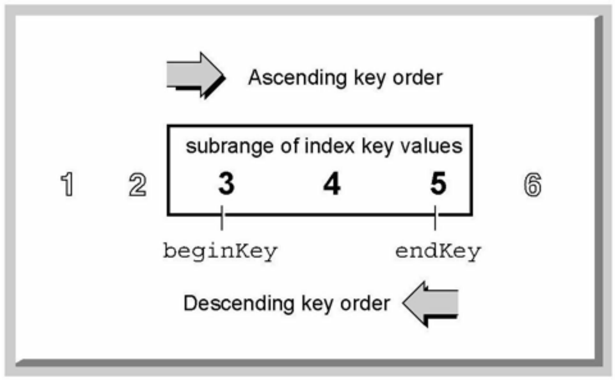
Begin Keys and End Keys

Because index keys are sorted by value, you can improve the speed of an index query significantly by limiting the range of index key values it tests. One way to do this is to eliminate from the search any index key values that fall outside 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 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 can use these optional `beginKey` and `endKey` values together to specify a subrange of index key values, as shown in Figure 11-3. Note that if an endrange value is not specified, it is unbounded; for example, if you don’t specify an `endKey` value the query result potentially includes all entries through the end of the index.

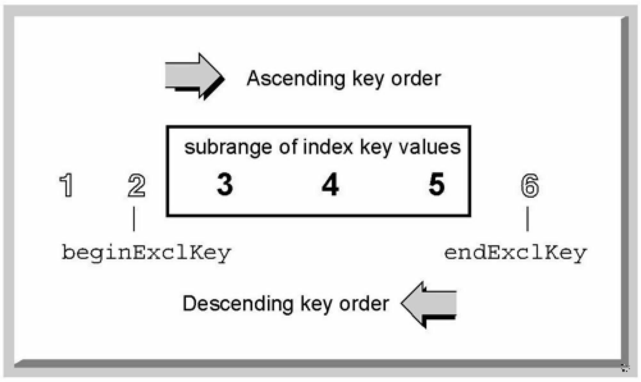
Figure 11-3 Using `beginKey` and `endKey` values to specify an index subrange



You can also define a special kind of key that is itself excluded from the valid subrange of index values. These keys are defined as `beginExclKey` and `endExclKey` values in the query spec. Figure 11-4 depicts the use of `beginExclKey` and `endExclKey` values to define the same index subrange shown in Figure 11-3. Note that you cannot specify both the inclusive and exclusive forms of the same endrange selector; for example, a single query spec cannot specify both a `beginKey` value and a `beginExclKey` value.

Another important point to understand is that there is only one `beginKey` or `beginExclKey` value, and only one `endKey` or `endExclKey` value associated with any query and the cursor it returns.

Figure 11-4 Using `beginExclKey` and `endExclKey` values to specify a subrange



Each `beginKey`, `beginExclKey`, `endKey`, or `endExclKey` specification evaluates to a single value that has a unique position in the sorted index key data. This position marks one end of the range over which the cursor iterates. The cursor never moves to a position outside the range specified by these keys.

If any endrange selectors are defined for a query, the relationship of the cursor's entries to the endrange selectors may be summarized as follows:

entry > `beginExclKey`
entry ≥ `beginKey`
entry ≤ `endKey`
entry < `endExclKey`

You can think of these values as being used by the system in an inequality expression to specify the range of the cursor; for example,

`beginKey` ≥ *entry* < `endExclKey`

Note that if a valid entry is not found at the key value specified for an endrange selector, the cursor is positioned on the nearest entry in index key order that falls within the range specified by the endrange selectors. For example, if a valid entry is not found at the key value specified for a `beginKey` or `beginExclKey` value, the cursor is positioned on the next valid entry in index key order. Similarly, if a valid entry is not found at the key value specified for an `endKey` or `endExclKey` value, the cursor is positioned on the previous valid entry in index key order. (The cursor is never positioned beyond the `endKey` value or before the `beginKey` value.)

For information on using index queries, see “Querying on Single-Slot Indexes” beginning on page 11-39 and “Querying on Multiple-Slot Indexes” beginning on page 11-47.

Tag-based Queries

Index queries can also include or exclude 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—it does not sort entries (as other indexes do), it only selects or eliminates entries according to their associated tag values. However, you need not specify an additional index in order to query on tag values; when a separate index is not specified, queries on tags test all entries in the soup.

The tags for which the query tests are specified by a **tags query specification frame** or **tags query spec** supplied as part of the query spec. The tags query spec can specify set operators such as `not`, `any`, `equal`, and `all` to create complex filters based on tag values. For example, you could use these operators to query for entries having the `'USA` or `'west` tags that do not have the `'California` tag.

The set operators used by tags query specs are described in greater detail in “Tag-based Queries” beginning on page 11-14 of this book and “Tags Query Specification Frame” (page 9-13) in *Newton Programmer’s Reference.*

Customized Tests

The use of indexes, begin keys, end keys, and tags provides sufficient control over query results for most uses; however, you can specify additional customized tests when necessary. These tests take the form of an `indexValidTest` or `validTest` function that you define in the query spec.

The `indexValidTest` function tests the index key values associated with each entry in the range of values over which the cursor iterates. This function returns `nil` for an entry that is to be rejected, and returns any non-`nil` value for an entry that is to be included in the results of the query. Like all tests that manipulate index key values, `indexValidTest` functions are fast and efficient because index key values are always kept in memory.

Another kind of customized test, the `validTest` function, works like the `indexValidTest` function but tests the soup entry itself rather than its associated index key value. To perform this test, the query must actually read the soup entry into the NewtonScript heap, which takes more time and uses more memory than tests which operate on index key values only. 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 range of entries passed to the `validTest` function.

For information on using `indexValidTest` and `validTest` functions, see "Querying on Single-Slot Indexes" beginning on page 11-39.

Text Queries

Queries can also select entries according to the presence of one or more specified strings. For instructional purposes, this section describes separately each of the text searches that queries can perform—remember, though, that a single query spec can specify multiple tests for the query to perform on each soup entry it examines.

A **words query** tests all strings in each soup entry for a word beginning or for an entire word. A **text query** is similar to a words query but its test is not limited to word boundaries.

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. Because words queries are not case sensitive, the word `"Smithsonian"` would also be found by this query.

If you specify that the words query match only entire words, it returns only entries containing the entire word `"smith"` or `"Smith"` and does not return any other variations. You can also specify explicitly that the query be sensitive to case and diacritics, causing it to return only the `"smith"` entry.

A words query is slower than a similar index query because it takes some time to test all the string values in a soup entry. For information about performing words queries, see "Querying for Text" beginning on page 11-43.

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 unless this behavior is requested explicitly, the words `"blackSmith"` and `"Smithsonian"` would also be found by this query.

A text query is slower than its words query counterpart. Text queries do not require significantly more heap space than other kinds of queries.

For more information about performing text queries, see “Querying for Text” beginning on page 11-43.

Cursors

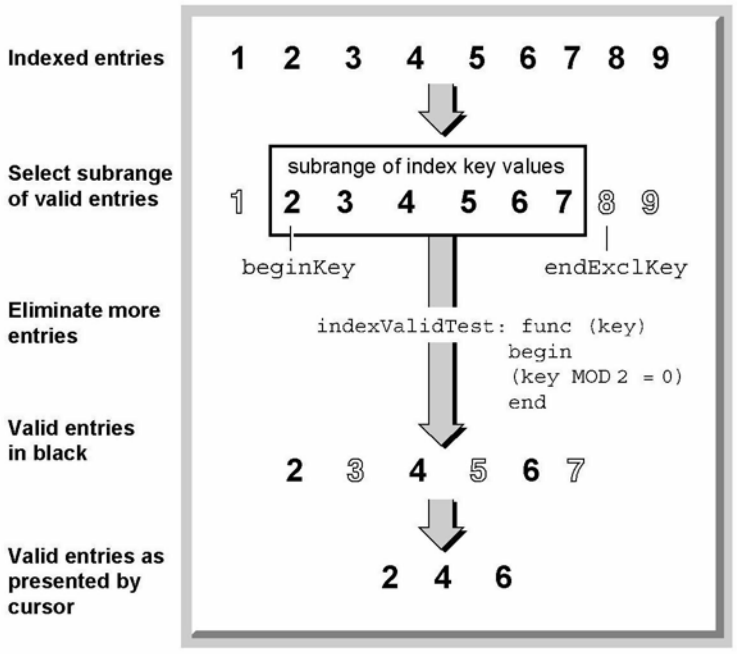
The `Query` method returns a **cursor,** which is an object that iterates over the set of entries satisfying the query spec and returns entries in response to the messages it receives. Cursors return entries in index key order. As entries in the soup are added, deleted, and changed, the set of entries the cursor references is updated dynamically, even after the original query has been performed.

Recall that after selecting a subrange of all entries in the soup, a query can use 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 is discontiguous—that is, it includes gaps occupied by entries that did not meet the criteria established by the query spec. However, the cursor presents this subset as a continuous range of entries, as depicted in Figure 11-5.

Initially, the cursor points to the first entry in index order 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 incrementally, moved to the position occupied by a specified entry or key, or reset to an initial position that is not necessarily the first entry in the valid set. Note that it is possible to move the cursor incrementally to a position outside the valid range of entries, in which case the cursor returns `nil` instead of an entry.

For information about using cursors, see “Using Cursors” beginning on page 11-53.

Figure 11-5 Cursor presents discontiguous index key values contiguously



Entries

An **entry** is a special kind of frame that resides in a soup. Valid entries can be created only by system-supplied methods provided for this purpose—you cannot create an entry by creating a frame having certain slots and values. The entry that these methods create consists of the frame presented to the entry-creation method, along with copies of any data structures the frame references, as well as copies of any data structures those structures reference, and so on. An exception to this rule is that `_proto` slots are not saved in soup entries. Circular references within an entry are allowed.

All frames are compressed automatically when they are stored as soup entries and all 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.

If you add a frame that references another entry, the referenced entry is copied as a frame into the new soup entry that is created. Similarly, if you move that entry to another store, any data it references is moved to the new store as well.

The only way to retrieve an entry is to send the `Query` message to the soup or union soup in which the entry resides. This method returns a cursor, which is an object that returns entries in response to messages it receives.

As first returned by the cursor, the entry is a frame that holds references to the entry's data. Soup data referenced by this frame is not decompressed until it is referenced—for example, to get or set the value of one of the entry's slots. When a slot in the entry is referenced, the system constructs the entire entry frame in the NewtonScript heap.

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 information about using entries, see the section "Using Entries" beginning on page 11-57.

Alternatives to Soup-Based Storage

Although soup-based data storage offers many advantages, you may improve your application's performance or reduce its RAM requirements by storing data in other formats.

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 uses 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.

When choosing schemes for storing your application's data, you need to consider factors such as:

- the kind of data to be saved
- the quantity of data to be saved
- 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 trade-offs 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 mechanisms for searching and sorting soup entry data.

Dynamic Data

Data 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 soups outweigh these costs; however, other approaches may be more suitable for certain data sets, especially large sets of read-only data.

For example, a large, read-only list of provinces and postal codes is saved most compactly as a single array, frame, or binary object residing in a slot in the application base view's template or 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 the conveniences provided by soup queries are not available.

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 compressed 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 data for individual countries in separate packages.

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's template and use a soup only for the user's additions.

A special-purpose object called a store part allows you to provide read-only soups as package data; however, a soup residing on a store part cannot participate in a union. For information about store parts, see Chapter 12, "Special-Purpose Objects for Data Storage and Retrieval."

If you decide not to store your data in a soup, consider the following points:

- Don't be too quick to discount frames as your data structure of choice—slot lookup is very fast.
- Storing data as a binary object can help you avoid some of the overhead associated with array and frame data structures. In general, binary objects may let you store your data more compactly, but make it more difficult to access: you'll need to use the various `Extract`*DataType* functions to retrieve items. Note that the `ExtractCString` and `ExtractPString` functions create a string object in the NewtonScript heap for each string extracted from a binary object.
- Consider storing symbols for repeated strings rather than storing the strings themselves. When you define a symbol for an object (such as a string or frame), only one instance of the object is stored in the application package, and all the symbols reference that instance. Remember that symbols are limited to 7-bit ASCII values. Symbols (slot names) can include nonalphanumeric ASCII characters if the name is enclosed by vertical bars; for example, the space in the symbol `'|Chicken Little|` would normally be illegal syntax, but the vertical bars suppress the usual evaluation of all characters they enclose.

Compatibility Information

This section provides version 2.0 compatibility information for applications that use earlier versions of the data storage and retrieval interface.

Obsolete Store Functions and Methods

The following store methods and functions are obsolete:

```
store:CreateSoup (soupName, indexArray) // use CreateSoupFromSoupDef
RemovePackage(pkgFrmOrID) // use SafeRemovePackage instead
store:RestorePackage(packageObject) // use SuckPackageFromBinary instead
```

Soup Compatibility Information

This section contains compatibility information regarding

- the new soup format introduced with version 2.0 of the Newton operating system
- obsolete soup functions and methods
- the new soup change notification mechanism introduced in version 2.0 of the Newton operating system
- soup information frame changes
- null union soups on Newton 1.*x* devices

New Soup Format

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 Newton 2.0 system, the system allows the user to choose read-only access or permanent conversion of the 1.*x* soup data to the Newton 2.0 format.
- Older systems display a slip that says "This card is too new. Do you want to erase it?" when a Newton 2.0 soup is introduced to the system.

Obsolete Soup Methods and Functions

The system's approach to creating soups automatically 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 creates the members of union soups automatically only when they are actually needed to store data. Thus, the `RegCardSoups`, `SetupCardSoups`, `RegisterCardSoup`, and `UnRegisterCardSoup` functions have been made obsolete by the `RegUnionSoup` and `UnRegUnionSoup` functions. Similarly, the `CreateSoup` store method has been made obsolete by the `RegUnionSoup` function. For more information, see "Soups" beginning on page 11-7.

The following soup methods and functions are obsolete:

```
SetupCardSoups() // use RegUnionSoup instead
RegisterCardSoup(soupName, indexArray,
                  appSymbol, appObject) // useRegUnionSoup instead
UnRegisterCardSoup(soupName)// use UnRegUnionSoup instead
BroadcastSoupChange(soupNameString) // use -xmit methods or
                                    // XmitSoupChange fn instead
UnionSoupIsNull(unionSoup)// no null uSoups from GetUnionSoupAlways
GetUnionSoup(soupNameString)// use GetUnionSoupAlways instead
soup:Add(frame) // use -xmit version instead
soup:AddIndex(indexSpec) // use -xmit version instead
soupOrUSoup:AddTags(tagsToAdd)// use -xmit version instead
unionSoup:AddToDefaultStore(frame)// use -xmit version instead
soup:AddWithUniqueId(frame)// use -xmit version instead
sourceSoup:CopyEntries(destSoup)// use -xmit version instead
soup:RemoveAllEntries() // use -xmit version instead
soup:RemoveFromStore() // use -xmit version instead
soup:RemoveIndex(indexPath) // use -xmit version instead
soupOrUSoup:RemoveTags(tagsToRemove)// use -xmit version instead
soup:SetInfo(slotSymbol) // use -xmit version instead
```

New Soup Change Notification Mechanism

Applications no longer modify system data structures 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. Consequently, the global function `BroadcastSoupChange` is obsolete.

For more details, see the section "Using Soup Change Notification" beginning on page 11-63.

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 obsolete global function `RegisterCardSoup` have a default soup information frame that contains only the slots `applications` and `itemNames`.

Soups created by the obsolete store method `CreateSoup` do not contain a default soup information frame.

Null Union Soups

Under unusual circumstances a 1.*x* application may encounter a union soup that doesn't contain any member 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 member 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 NTK 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.

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 member 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 member is null when the card is ejected. If you follow the guidelines outlined in "Registering and Unregistering Soup Definitions" on page 11-33 this problem does not occur.

Null union soups can also occur when another application deletes one or more soups that your application uses. Any application that deletes soups should at least transmit a soup change 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 a 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 constituent soups in the union.

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 new-style queries provide. The following list summarizes changes to queries. The remainder of this section explores query compatibility issues in more detail.

```
Query (soupOrUSoup, querySpec) // use soupOrUSoup:Query(querySpec) instead

querySpec := {type : symbol, // obsolete, do not use
              startKey: keyValue, // use beginKey or beginExclKey
              endTest: keyValue, // endKey or endExclKey instead
              … }
```

Query Global Function Is Obsolete

Queries are now performed by the `Query` method of soups or union soups; however, the `Query` global function still exists for compatibility with applications using version 1.*x* of the Newton application programming interface. The `Query` method accepts the same query specification frame argument that the `Query` global function did; however, version 2.0 query specs provide additional features that 1.*x* queries do not. For examples of the use of the `Query` method, see "Using Queries" beginning on page 11-38. For a complete description of the query spec frame and its slots, see "Query Specification Frame" (page 9-9) in *Newton Programmer's Reference*.

Query Types Obsolete

Query specs no longer require a `type` slot; if this slot is present, it is ignored.

StartKey and EndTest Obsolete

Because the order in which the cursor returns entries is determined entirely by index values, specifying key values is sufficient to determine a range. Hence, the use of an `endTest` function in a query spec is always unnecessary. Instead, your query spec should specify an `endKey` or `endExclKey` value.

The `endTest` function was sometimes used for other purposes, such as stopping the cursor after the visible portion of a list had 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 `endTest` function.

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 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 Unsupported

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.

In Newton system software version 2.0, the behavior of queries on `nil`-value slots is unspecified. For best performance, make sure your indexed slots store appropriate values.

Heap Space Requirements of Words and Text Queries

On systems prior to version 2.0, words and text queries generally require more memory than index queries, because each entry to be tested must first be read into the NewtonScript heap. System software version 2.0 uses virtual binary objects to reduce the memory requirements of words and text queries significantly; however, you need not be familiar with these objects yourself in order to query on string values. Virtual binary objects are described in Chapter 12, "Special-Purpose Objects for Data Storage and Retrieval."

Obsolete Entry Functions

The following entry functions are obsolete:

```
EntryChange(entry) // use -xmit version instead
EntryCopy(entry, newSoup) // use -xmit version instead
EntryMove(entry, newSoup)// use -xmit version instead
EntryRemoveFromSoup(entry)// use -xmit version instead
EntryReplace(oldEntry, newEntry)// use -xmit version instead
EntryUndoChanges(entry)// use -xmit version instead
```

Obsolete Data Backup and Restore Functions

The utility functions and methods in the following list are obsolete. Note that these functions and methods are intended for use only by utility programs that back up and restore Newton data.

```
soup:AddWithUniqueId (entry)// use -xmit version instead
soup:SetAllInfo (frame)// use -xmit version instead
EntryChangeWithModTime(entry)// use -xmit version instead
EntryReplaceWithModTime(original, replacement)// use -xmit version
```

Using Newton Data Storage Objects

This section describes how to use the most common Newton data storage objects and methods. It presumes knowledge of the material in preceding sections. This section begins with a programmer's overview, which is followed by sections providing detailed explanations of the use of stores, soups, queries, cursors, and entries.

Programmer's Overview

This section provides a code-level overview of common objects, methods, and functions that provide data storage and retrieval services to Newton applications.

This section presumes understanding of the material in “About Data Storage on Newton Devices” beginning on page 11-1.

Most applications store data as frames that reside in soup entries. You can create a frame by simply defining it and saving it in a variable, a constant, or a slot in another frame. For example, the following code fragment defines a frame containing the `aSlot` and `otherSlot` slots. The frame itself is stored in the `myFrame` 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 saved in the `myFrame` variable as the `myFrame` frame.

```
myFrame := {aSlot: "some string data", otherSlot: 9258};
```

The `myFrame` frame contains two slots: the `aSlot` slot stores the `"some string data"` string and the `otherSlot` slot stores the `9258` integer value. Because every NewtonScript object encapsulates its own class data, you need not declare types for NewtonScript data structures, including slots.

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 or union soup object by creating a new one or by retrieving a reference to one that is already present.

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 member soups as needed to store soup entries.

The following code fragment saves the union soup object `RegUnionSoup` returns in the `myUSoup` local variable. You might place code like this example in your application (`form`) part’s `InstallScript` function or your application base view’s `ViewSetupFormScript` method:

```
local aSlotIndexSpec := {structure: 'slot, path: 'aSlot,
                        type: 'string};
local otherSlotIndexSpec := {structure: 'slot, path: 'otherSlot,
                        type: 'int};
local mySoupDef := {name: "mySoup:mySig",
                    userName: "My Soup",
                    ownerApp: '|MyApp:MySig|,
                    ownerAppName : "My Application",
                    userDescr: "This is the My Application soup.",
                    indexes: [aSlotIndexSpec,otherSlotIndexSpec]
                    };
local myUsoup := RegUnionSoup('|MyApp:MySig|,mySoupDef);
```

Note the use of the `mySig` developer signature as a suffix to ensure the uniqueness of the values of the `name` and `ownerApp` slots. For more information regarding developer signatures, see Chapter 2, “Getting Started.”

When creating soups from within your application (`form`) part's `InstallScript` function, remember that this function calls the `EnsureInternal` function on all values it uses. Thus, instead of passing references such as `partFrame.theForm.myMainSoupDef` to the `RegUnionSoup` function, paste a local copy of your soup definition into your application part's `InstallScript` function for its use.

The `RegUnionSoup` function uses the value of your soup definition's `name` slot to determine whether a particular soup definition has already been registered. You need not be concerned with registering a soup definition twice as long as you don't register different soup definitions that have the same name. An application that registers a soup definition when it opens can always use the union soup object returned by the `RegUnionSoup` function—if the union soup named by the soup definition exists, this function returns it; otherwise, this function uses the specified soup definition to create and return a new union soup.

The next code fragment uses the `AddToDefaultStoreXmit` function to add the `myFrame` frame to the `myUSoup` union soup. This function creates a new member soup to hold the entry if necessary. The soup is created on the store indicated by the user preference specifying where new items are kept.

```
myUSoup:AddToDefaultStoreXmit(myFrame, '|MyApp:MySig|);
```

At this point, we have created a soup on the 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 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` union soup that the `'|MyApp:MySig|` application added an entry to this union soup. For more information, see "Callback Functions for Soup Change Notification" (page 9-14) in *Newton Programmer's Reference.*

Most of the time, your application needs to work with existing soups rather than create new ones. You can use the `GetUnionSoupAlways` function to retrieve an existing soup by name.

Once you have a valid soup object, you can send the `Query` message to it to retrieve soup entries. The `Query` method accepts a query specification frame as its argument. This frame defines the criteria soup entries must meet in order to be retrieved by this query. Although you can pass `nil` as the query spec in order to retrieve all the entries in a soup, usually you'll want to retrieve some useful subset of all entries. For example, the following code fragment retrieves from `myUsoup` all entries having an `aSlot` slot. For an overview of the use of query specifications, see "Using Queries" beginning on page 11-38.

```
// get from myUSoup all entries having an aSlot slot
local myCursor := myUSoup:Query({indexPath: 'aSlot});
```

The `Query` method returns a cursor object that iterates over the set of soup entries satisfying the query specification passed as its argument. You can send messages to the cursor to change its position and to retrieve specified entries, as shown in the following example. For an overview of cursor-manipulation functions, see "Moving the Cursor" beginning on page 11-55.

```
// 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, assume that the cursor returned the entry holding the `myFrame` frame. When accessing this frame, use the NewtonScript dot operator (`.`) to dereference any of its slots. In the current example, the expression `myEntry.aSlot` evaluates to the `"some string data"` value and the expression `myEntry.otherSlot` evaluates to the `9258` value.

As soon as any slot in the entry is referenced, the system reads entry data 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 the following reasons:

- 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.
- Changing the value of a slot in the entry really changes the cached entry frame, not the original soup entry; changes to the soup entry are not persistent until the cached entry frame is written back to the soup, where it takes the place of the original entry.

You can treat the cached entry frame as the `myFrame` frame and assign a new value to the `aSlot` slot directly, as shown in the following code fragment:

```
myEntry.aSlot := "new and improved string data";
```

To make the changes permanent, you must use `EntryChangeXmit` or a similar function to write the cached entry frame back to the soup, as in the following example:

```
EntryChangeXmit(myEntry, '|MyApp:MySig| );
```

Like the other functions and methods that make changes to soups, the `EntryChangeXmit` function transmits an appropriate soup change notification message after writing the entry back to its soup; in this case, the notification specifies that the `'|MyApp:MySig|` application made an `'entryChanged` change to the 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 if you have not yet written the cached entry back to the soup. Because this function throws away the contents of the entry cache, referencing a slot in the entry after calling the `EntryUndoChangesXmit` function causes entry data to be read into the cache again.

Most applications unregister their soup definitions when they are closed or removed. To facilitate the automatic creation of soups when the user files or moves soup entries in the Extras drawer, you may want your soup definition to remain registered while your application is closed—to unregister only when your application is removed, call the `UnRegUnionSoup` function from your application (`form`) part's `DeletionScript` function.

The following code example uses the `UnRegUnionSoup` function to unregister a soup definition. Because a single application can create multiple soups and soup definitions, soup definitions are unregistered by name and application symbol:

```
// usually in your app part's DeletionScript fn
UnRegUnionSoup("mySoup:mySig",'|MyApp:MySig|);
```

Using Stores

Because the system manages stores automatically, most NewtonScript applications' direct interaction with store objects is limited. This section describes the use of system-supplied functions and methods for

- getting store objects
- retrieving packages from stores
- testing stores for write protection
- getting and setting store information

Procedures for manipulating other objects that reside on stores (such as soups, store parts and virtual binary objects) are described in "Using" sections for each of these objects; for detailed information, see "Using Soups" on page 11-32; "Using Virtual Binary Objects" on page 12-8; and "Using Store Parts" on page 12-12.

Store Object Size Limits

The system imposes a hard upper limit of 64 KB on store object sizes for any kind of store. SRAM-based stores impose a further limitation of 32 KB on block size. Trying to create an entry larger than 32 KB causes the system to throw `|evt.ex.fr.store|` exceptions. These limits apply to the encoded form the data takes when written to a soup, which varies from the object's size in the NewtonScript heap.

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();
```

▲ **WARNING**
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 on different 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 Packages From Stores

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 the package reference information frame associated with the package. This frame is returned by the `GetPkgRefInfo` function.

To load a package procedurally, use either of the store methods `SuckPackageFromBinary` or `SuckPackageFromEndpoint`. For more information, see the descriptions of these methods in "Data Storage and Retrieval Reference" (page 9-1) in *Newton Programmer's Reference.*

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. Do not use the global function `IsReadOnly` to test store objects; use only the `IsReadOnly` store method for this purpose.

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. Normally, applications using union soups do not need to get or set the default store. The system-supplied functions that accept union-soup arguments handle the details of saving and retrieving soup data according to preferences specified by the user.

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. ◆

Getting and Setting the Store Name

Normal NewtonScript applications rarely need to get or set store names. A store's name is the string that identifies the store in slips displayed to 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 names of stores.

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();
```

Before attempting to set the store's name or write any other data to it, you can use the store methods `IsReadOnly` or `CheckWriteProtect` to determine whether the store can be written.

▲ **WARNING**
Renaming a store renders invalid all aliases to entries residing on that store. See "Using Entry Aliases" on page 12-7. ▲

Accessing the Store Information Frame

Each store may hold an optional information frame that applications can use to save information associated with the store itself. Note that unless an application stores data in this frame, it may not exist on every store.

The `GetInfo` and `SetInfo` store methods are intended for use by backup/restore applications only; most applications need not use them at all. 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.

Using Soups

This section discusses the functions and methods used to work with soup objects. Individual entries in soups and union soups are manipulated by means of queries, cursors, and entry functions, as described in subsequent sections of this chapter. This section describes procedures for

- creating soups and indexes
- retrieving existing soups
- indexing existing soups
- reading and writing soup data
- accessing information about the soup itself and the store on which it resides
- removing soups

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 as the name of its soup. Your package name is created by using a colon (`:`) to append your package's Extras Drawer name to 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 another colon, followed by 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 duplicate those used by other applications.

Normally, each soup appears under its own icon in the Extras Drawer. If your application creates multiple soups, it is recommended that you group them under a single Extras Drawer icon. For more information, see "About Icons and the Extras Drawer" on page 19-38 in Chapter 19, "Built-in Applications and System Data."

For additional information regarding naming conventions for your developer signature and other items, see "Developer Signature Guidelines" on page 2-9 in Chapter 2, "Getting Started."

Registering and Unregistering Soup Definitions

The `RegUnionSoup` global function registers a soup definition with the system and returns a union soup object to which you can send messages. Once the soup definition is registered, various union soup methods create the union's member soups as needed to save entries. A corollary function, `UnRegUnionSoup`, unregisters a specified soup definition.

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. Normally, code that registers soup definitions is provided by your application part's `InstallScript` function or your application base view's `ViewSetupFormScript` method. You need not be concerned with registering a soup definition twice as long as you don't register different soup definitions that have the same name.

Code to unregister soup definitions is usually provided either by your application base view's `ViewQuitScript` method (to unregister when the application closes) or your application part's `DeletionScript` function (to unregister only when the application is removed.) An application that allows the user to file or move data items from the Extras Drawer should allow its soup definitions to remain registered while the application is closed, unregistering them only when the application is removed. For more information on manipulating soup entries from the Extras Drawer, see "About Icons and the Extras Drawer" on page 19-38 in Chapter 19, "Built-in Applications and System Data."

Your application can also call the `RegUnionSoup` function to retrieve its own union soups that already exist. If you call `RegUnionSoup` on a soup definition that is already registered, this function replaces the currently registered soup definition with the new one and returns the union soup named by the soup definition passed as its argument; if that union soup does not exist, this method uses the soup definition passed as its argument to create a new union soup that it returns. Alternatively, you can call the `GetUnionSoupAlways` global function to retrieve any extant union soup, not just those your application registers. For more information, see "Retrieving Existing Soups" beginning on page 11-34.

To use the `RegUnionSoup` function, you might put code like the following example in the `ViewSetupFormScript` method of your application's base view:

```
local mySoupDef := {name: "mySoup:mySig",
                    userName: "My Soup",
                    ownerApp: '|MyApp:MySig|,
                    ownerAppName : "My Application",
                    userDescr: "This is the My Application soup.",
                    indexes: [{structure: 'slot, path: 'aSlot,
                                  type: 'string}]
                    };
// register soup or retrieve already-registered soup
   local myUsoup := RegUnionSoup('|myApp:mySig|, mySoupDef);
```

You can unregister a soup definition whenever 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 memory available to other applications.

The following code fragment illustrates the use of the `UnRegUnionSoup` function:

```
// unregister my soup def
   UnRegUnionSoup (mySoupDef.Name, '|myApp:mySig|);
// don't forget to set all unused references to nil
   myUsoup := nil;
```

Retrieving Existing Soups

To retrieve your own union soups, you can use the `RegUnionSoup` function as described in "Registering and Unregistering Soup Definitions" beginning on page 11-33. Alternatively, you can call the `GetUnionSoupAlways` global function to retrieve any union soup by name.

Use of the `GetUnionSoupAlways` global function is straightforward, as the following example shows. Note that you can pass system-supplied constants to this function to retrieve soups used by the system and the built-in applications. For more information, see Chapter 19, "Built-in Applications and System Data."

```
// retrieve "mySoup:mySig" union soup by name
local myUSoup := GetUnionSoupAlways("mySoup:mySig");
// retrieve soup used by built-in Names application
local names := GetUnionSoupAlways(ROM_CardFileSoupName);
```

Note that you can use the `IsInstance` utility function to determine whether a specified soup is a union soup. Pass either of the symbols `'PlainSoup` or `'UnionSoup` as the value of the *class* parameter to this function, as shown in the following code fragment.

```
IsInstance(mySoup, 'UnionSoup);
```

Adding Entries to Soups

This section describes how to add a frame to a union soup or a specified member soup in a union. For information on creating union soups, see "Registering and Unregistering Soup Definitions" on page 11-33. For information on retrieving union soups, see "Retrieving Existing Soups" on page 11-34.

You can use either of the `AddToDefaultStoreXmit` or `AddToStoreXmit` methods to save frames as soup entries. Both of these methods create a single soup in which to save the new entry when the appropriate member of the union is not already present on the store. The `AddToDefaultStoreXmit` method adds its entry to the member soup on the store specified by the user as the destination for new entries. The `AddToStoreXmit` method allows you to specify according to store the member soup to which it adds the new entry.

Methods that create soup entries—such as the `AddToDefaultStoreXmit`, `AddToStoreXmit`, and `AddXmit` methods—destructively modify the frame presented as their argument to transform it into a soup entry. Thus, any frame passed to these methods must allow write access. If the original frame must remain unmodified, pass a copy of it to these methods.

The following code fragment saves a frame in the default store member of the `myUsoup` union by sending the `AddToDefaultStoreXmit` message to the union soup object that the `RegUnionSoup` function returns:

```
// register soup def'n or get reference to already registered soup
   local myUsoup := RegUnionSoup('|myApp:mySig|, mySoupDef);
// add the entry and transmit notification
   local myEntry := myUSoup:AddToDefaultStoreXmit(
               {aSlot:"my data"}, // frame to add to soup
               '|myApp:mySig|); // app that changed soup
```

The following code fragment saves a frame in the internal store member of the `myUsoup` union by sending the `AddToStoreXmit` message to the union soup object that the `GetUnionSoupAlways` function returns:

```
// get pre-existing uSoup by name
   local myUSoup := GetUnionSoupAlways("mySoup:mySig");
// add entry to member on internal store and transmit notification
   local myEntry := myUSoup:AddToStoreXmit(
               {aSlot:"my data"}, // frame to add to soup
               (GetStores()[0]), // add to member on internal store
               '|myApp:mySig|); // app that changed soup
```

After creating the new soup entry, these methods transmit a soup change notification message. To suppress the soup change notification message that -`Xmit` functions and methods transmit, pass `nil` as the value of their *changeSym* argument. For more information, see "Using Soup Change Notification" beginning on page 11-63; also see the descriptions of the `AddToDefaultStoreXmit` and `AddToStoreXmit` methods in "Soup Functions and Methods" (page 9-35) in *Newton Programmer's Reference.*

Normally the member soups in a union are created automatically by the system as needed to save frames as soup entries. If you need to force the creation of a union soup member on a specified store without adding an entry to the new member soup, use the `GetMember` union soup method to do so. For more information, see the description of this method in "Soup Functions and Methods" (page 9-35) in *Newton Programmer's Reference.*

Adding an Index to an Existing Soup

Normally, applications create an index for each slot or set of slots on which a soup may 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 an index to an already existing soup and transmit a soup change notification message. Indexes must be added individually—you can't pass arrays of index specs to the `AddIndexXmit` method.

▲ **WARNING**
You cannot query a union soup on an index that is not present in all its member soups. Sending the `AddIndexXmit` message to a union soup adds the specified index to all soups currently available to the union; however, any soup introduced to the union subsequently has only its original complement of indexes, which may not include the index this method added. Similarly, any member soup created by the system has only the indexes specified by its soup definition, which may not include the index this method added. ▲

The following code fragment adds an index to the `"mySoup:myApp"` union soup, enabling queries to search for integer data in that soup's `mySlot` slot:

```
// get my union soup
   local myUSoup := GetUnionSoupAlways("mySoup:mySig");
// add a new single-slot index on the'mySlot slot
   local myISpec := {structure:'slot, path:'mySlot, type:'int};
   local myUSoup:AddIndexXmit(myISpec,'|myApp:mySig|);
```

▲ **WARNING**
Each soup has only one tags index; if you add a tags index to a soup that already has one, it replaces the original tags index. For more information, see the description of the `AddIndexXmit` method (page 9-42) in *Newton Programmer's Reference*. ▲

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` function is an optional function that you supply in your application's `form` part. This function accepts no arguments. You can remove your application's soups from within this function by invoking the `RemoveFromStoreXmit` soup method. The `RemoveFromStoreXmit` method is defined only for single soups; in other words, you must remove each member of a union soup separately.

For more information on the `DeletionScript` method, see the *Newton Toolkit User's Guide*. See also "RemoveFromStoreXmit" (page 9-47) in *Newton Programmer's Reference*.

Do not delete soups from within your application's `viewQuitScript` method—user data needs to be preserved until the next time the application is run. For similar reasons, do not remove soups from within your application's `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.)

Using Built-in Soups

The soup-based data storage model makes it easy for applications to reuse existing system-supplied soups for their own needs and to share their own soups with other applications. Refer to Chapter 19, "Built-in Applications and System Data," to see descriptions of the soups used by the applications built into the Newton 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 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 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 prevents soups from becoming cluttered with excessive numbers of entries.
- It facilitates 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 11-63.

Adding Tags to an Existing Soup

You can add tags only to a soup that has a 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 include the new tags.

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 `AddIndexXmit` 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 replace a soup's tags index.

Using Queries

To retrieve soup entries, you need to query a soup or union soup object by sending the `Query` message to it. The `Query` method accepts a query specification frame, or query spec, as its argument. The query spec specifies the characteristics that soup entries must have in order to be included in the query result.

Note
For instructional purposes, this section describes each item that may appear in a query specification separately. Normally, a single query spec defines multiple criteria that soup entries must meet to be included in the results of the query; for example, you can create a single query spec that specifies tests of index key values, string values, and tags. ◆

This section describes how to perform various queries to retrieve soup data. It includes examples of

- simple queries on index values, tags, or text
- the use of ascending and descending indexes

- the use of internationalized sorting order
- queries on multiple-slot indexes

Querying Multiple Soups

Soups having the same name can be associated logically by a union soup object. To retrieve entries from all the available soups in a union, just send the `Query` message to the union soup object.

You must query differently named soups separately, however. For example, before scheduling a meeting, you might send the `Query` message to the `ROM_CardfileSoup` soup for information regarding its participants, and send another `Query` message to the `ROM_CalendarSoupName` soup to determine whether you have conflicting appointments at the proposed meeting time.

Entry aliases provide a handy way to save references to soup entries. You can use entry aliases to reference entries from different soups more easily. For more information, see "Using Entry Aliases" on page 12-7.

Querying on Single-Slot Indexes

This section provides code examples illustrating a variety of queries on single-slot indexes. For more information on indexes, see "Introduction to Data Storage Objects" on page 11-2 and "Indexes" on page 11-8.

The following code fragment presents an example of the simplest kind of index query—it returns all entries in the soup:

```
local myUSoup := GetUnionSoupAlways("mySoup:mySig");
local allEntriesCursor := myUSoup:Query(nil);
```

When `nil` is passed as the query spec, as in the example above, the query result potentially includes all entries in the soup. The cursor generated by such a query returns entries in roughly the same order that they were added to the soup; however, this sorting order is not guaranteed because the system recycles the values it uses to identify entries internally. The only way to guarantee that entries are sorted in the order they were added to a soup is to index them on your own time stamp slot.

Most situations will require that you query for a subset of a soup's entries, rather than for all of its entries. 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 entries in which the value of a specified slot falls within a certain range. The next several examples illustrate the use of single-slot index queries for these kinds of operations.

To find all entries that have a particular slot, specify a path to that slot as the query spec's `indexPath` value. Note that in order to query on the presence of a particular 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 sorts 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 entry holding a specified name or value in an indexed slot. For examples of the use of this method, see “Moving the Cursor” beginning on page 11-55.

Using `beginKey` and `endKey` values to limit your search can improve query performance significantly. The following example is an index query that uses a `beginKey` value and an `endKey` value to return entries for which `(11 ≥ entry.number ≤ 27 )`.

```
// mySoup is indexed on the 'number slot
local numCursor := mySoup:Query({indexPath: 'number,
                                 beginKey: 11,
                                 endKey: 27});
```

The index on the `number` slot potentially includes all entries that have a `number` slot. The index sorts entries on their index key values; unless otherwise specified, the default index order is ascending. Thus, the query can use a `beginKey` value of `11` to skip over entries holding a value less than `11` in the `number` slot. The test can be concluded quickly by specifying a maximum value beyond which the cursor generated by this query does not proceed. In this case, the `endKey` value specifies that the query result does not include entries having values greater than `27` in the `number` slot. When multiple entries hold a specified endrange value, all of them are included in the result of a query that specifies that endrange value; for example, if multiple entries in the `mySoup` soup hold the value `27` in their `number` slot, the previous example includes all of these entries in its result.

The `beginKey` specification evaluates to a value that occupies a unique position in the sorted index data for the soup. If no entry is associated with this value, the cursor is positioned at the next valid entry in index order. For example, if the`mySoup` soup in the previous code fragment does not contain an entry having a `number` slot that holds the value `11`, the next valid entry in index order is the first entry in the range over which the cursor iterates.

Similarly, the `endKey` specification evaluates to a value that occupies a unique position in the sorted index data for the soup. If no entry is associated with this value, the cursor stops on the first valid entry in index order before the `endKey` value. For example, if the `mySoup` soup in the previous code fragment does not contain an entry having a `number` slot that holds the value `27`, the last valid entry at or before the position that would be occupied by `27` in the index is the last entry in the range over which the cursor iterates.

To conduct the same query while excluding the endrange values, specify a `beginExclKey` value instead of a `beginKey` value, and specify an `endExclKey` value instead of an `endKey` value, as shown in the following code fragment:

```
// mySoup is indexed on the 'number slot
// return entries for which (11 > entry.number < 27 )
local numCursor := mySoup:Query({indexPath: 'number,
                                 beginExclKey: 11,
                                 endExclKey: 27});
```

Note that a query spec cannot include both the inclusive and exclusive forms of the same endrange selector; for example, you cannot specify `beginKey` and a `beginExclKey` values in the same query spec. However, you can specify, for example, a `beginKey` value and an `endExclKey` value in the same query spec.

Because the index sorts entries according to key values, a `beginKey` on a soup indexed in descending key order may appear to act like an `endKey` on a soup indexed in ascending order, and vice versa. For more information, see “Queries on Descending Indexes” beginning on page 11-46.

Another way to find all entries having a particular value in a specified slot is to use an `indexValidTest` method, which can test any index key value without reading its corresponding entry into the NewtonScript heap. The system passes index key values to this function as the cursor moves. Your `indexValidTest` must return a non-`nil` value if the entry associated with the key value should be included in the query result. For example, you could use an `indexValidTest` method to select entries that hold even-number values in a specified slot, as shown in the following code fragment:

```
// mySoup indexed on 'number slot
// select entries having a 'number slot that holds
// an even value between 19 and 58
local myCursor :=
      mySoup:Query({ beginKey: 19, endExclKey: 58,
                     indexValidTest: func (key)
                        (key MOD 2 = 0)});
```

A less-preferred way to test entries is to provide a `validTest` function to test entries individually. The use of a `validTest` increases the memory requirements of the search because the system must read soup entries into the NewtonScript heap in order to pass them to the `validTest` function. Whenever possible, you should avoid using `validTest` methods in favor of using `indexValidTest` methods. Generally, you need not use a `validTest` method unless you must read the entry’s data to determine whether to include it in the query result.

The query passes the entire entry to the `validTest` method, rather than just the value of the indexed slot. The next code example reads the entry's `aSlot` and `otherSlot` slots in order to compare their values:

```
// select entries for which aSlot > otherSlot
local myCursor :=
         mySoup:Query({endKey: aKeyValue,
                       validTest: func (entry)
                       begin
                          entry.aSlot > entry.otherSlot
                       end});
```

Querying for Tags

In order to select soup entries according to their associated tag values, you need to include a tags query spec frame in the `tagSpec` slot of the query specification frame passed to the `Query` method. In addition to specifying one or more tags used to select entries, the tags query spec can specify set operators such as `not`, `any`, `equal`, and `all` to create complex filters based on tag values. For a complete description of the tags query spec frame, see "Tags Query Specification Frame" (page 9-13) in *Newton Programmer's Reference.*

You cannot query for tags on a soup that does not have a tags index. This index is usually specified by your soup definition and created along with the soup, but it can be added to an existing soup if necessary. Note that each soup or union soup has only one tags index; if you add a tags index to a soup that already has one, it replaces the original tags index. For more information, see "Tags Index Specification Frame" (page 9-8) in *Newton Programmer's Reference.*

The next several examples presume that the `mySoup` soup has a tags index on the `labels` slot. Note that queries need not specify the path to the slot from which tag values are extracted—in this case, the `labels` slot—because each soup has only one tags index and its index path is specified when the tags index is created. However, because a soup or union soup is allowed to have multiple soup indexes, queries must specify a path to the indexed slot; hence, these examples also presume that the `mySoup` soup has a soup index on the `name` slot.

The presence of any tag specified by the `any` set operator is sufficient to include its entry in the results of the query that uses this operator. For example, the following query selects entries having either the symbol `'flower` or `'tall` in the `labels` slot. Entries not marked with at least one of these symbols are not included in the query result.

```
local myCurs := mySoup:Query({indexPath:'name,
                              tagSpec: {any:['tall, 'flower]}});
```

The `equal` set operator specifies a set of tags an entry must match exactly to be included in the query result. The query in the following example uses the `equal` set operator to select entries marked with only the 'flower and 'tall tags; this query does not select entries missing either tag, nor does it select entries marked with additional tags:

```
local myCurs := mySoup:Query({indexPath:'name,
                              tagSpec: {equal: ['tall, 'flower]}});
```

Like the `equal` set operator, the `all` set operator specifies a set of tags that entries must have to be selected; however, the `all` set operator does not exclude entries marked with additional tags. For example, the query in the following example uses the `all` set operator to select entries marked with both the `'flower` and `'tall` tags. This query excludes entries missing either of these tags but includes entries marked with a superset of the `'flower` and `'tall` tags:

```
local myCurs := mySoup:Query({indexPath:'name,
                              tagSpec: {all: ['tall, 'flower]}});
```

The presence of any tag specified by the `none` set operator is sufficient to exclude that entry from the query result. For example, the following query matches entries having both of the tags `'flower` and `'tall` but excludes any entry marked with the `'thorns` tag:

```
local myCurs := mySoup:Query({indexPath:'name,
                              tagSpec: {all:['flower, 'tall],
                                    none:['thorns]}});
```

The following exceptions may be thrown when attempting to query using a tag spec. If the soup does not have a tags index, a "no tags" 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`, an "invalid tag spec" exception `|evt.ex.fr.store| -48028` is thrown.

Querying for Text

This section describes how to select entries according to the presence of one or more strings in any slot. The current system allows you to search entries for string beginnings, entire strings, or substrings of larger strings.

To select entries according to the presence of one or more specified string beginnings, add to your query spec a `words` slot containing an array of string beginnings to be matched. For example, the following code fragment illustrates a query that returns entries having strings beginning with `"bob"`:

```
// find words beginning with "bob"
local myCurs := mySoup:Query({words: ["bob"]});
```

This query finds entries containing the words "Bob", "Bobby", and so forth, but not words such as "JoeBob". Text queries are not case sensitive—even though the original query spec is all lower case, this query finds entries such as "Bob" or "BOB".

Because the words slot contains an array, it can be used to search for multiple string 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, but an entry missing either of the word beginnings "Bob" or "Apple" is not included.

```
// find entries holding "bob" and "apple" word beginnings
// won't find entries having only one of these beginnings
local myCurs := mySoup:Query({words: ["bob", "apple"]});
```

Because each element in the array is a string, each "word" to be matched 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.":

```
// find word beginnings "bob" and "Apple Computer, Inc."
local myCursor := mySoup: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. ◆

To search for entire strings, rather than string beginnings, the query spec must include an entireWords slot that holds the value true, as shown in the following code fragment:

```
// return entries holding entire words "bob" and "Apple Computer"
local myCursor := mySoup:Query({words: ["bob", "Apple Computer"],
                                entireWords: true });
```

This query returns entries that contain both of the strings "Bob" and "Apple Computer". Because the entireWords slot holds the value true, this query does not match strings such as "Apple Computer, Inc." or "Bobby". Entries containing only one of the specified words are not included in the results of the search.

To conduct a text search that is not constrained by word boundaries, add to your query spec a text slot containing a single string to be matched. For example, the following code fragment illustrates a query that returns entries having strings that contain the substring "Bob":

```
// find strings containing the substring "Bob"
local myCursor := mySoup:Query({text: "bob"});
```

This query finds entries containing words such as "JoeBob", as well as those containing words such as "bob" and "Bobby".

Internationalized Sorting Order for Text Queries

Indexes are not normally sensitive to case, diacritical marks, or ligatures in string values; however, index and query specifications can request this behavior specifically. When internationalized index ordering is used, uppercase letters sort first, followed by lowercase letters, diacritical marks, and ligatures. Thus, the letter A sorts before the letter a, which sorts before the letter å, which sorts before the letter á, which sorts before the ligature æ.

To index string values in internationalized order, include an optional sortID slot holding the value 1 in the index specification frame used to build a soup's index. A cursor subsequently generated against that soup returns entries holding the following strings in the order listed here:

```
"AA", "aa", "åå", "EE", "ÉÉ", "ee"
```

This internationalized indexing order is available only for indexes on string values. When the sortID slot is missing from the index spec or this slot's value is nil, the index generated is not sensitive to case, diacritics, or ligatures; in other words, the index may not necessarily sort "AA" before "aa", and so on.

If an index has internationalized ordering, find operations performed by cursors generated against that index can be made sensitive to case and diacritics. To request this behavior, include a non-nil secOrder slot in the query spec passed to the Query method of an internationally-indexed soup.

The value of the secOrder slot affects the use of the beginKey, beginExclKey, endKey, and endExclKey slots, as well as the GoToKey cursor method. For example, sending the GoToKey("åå") message to the cursor generated by this query returns the first entry found at or after the "åå" index value but does not return entries holding values that vary in case, diacritics, and so on.

When the secOrder slot is missing or holds the value nil, find operations carried out by cursor methods such as GoToKey ignore case and diacritics; that is, they may return entries holding case and diacritic variations on the requested value. For example, sending the myCursor:GoToKey("åå") message returns the first entry found that holds any of the "AA", "aa", or "åå" values. However, the cursor generated by this query still uses the sorting order provided by the internationalized index: cursor methods such as `Next` and `Prev` return entries in the internationally-indexed order.

Queries on Descending Indexes

Even though queries and cursors based on descending order indexes work just like normal queries and cursors, their behavior can seem confusing if you forget that it is a function of index order. It is always helpful to remember the following points when working with queries and cursors—especially when using descending indexes:

- The "beginning" and "end" of a range of index values is a function of index key order.
- The cursor navigates entries in index key order.

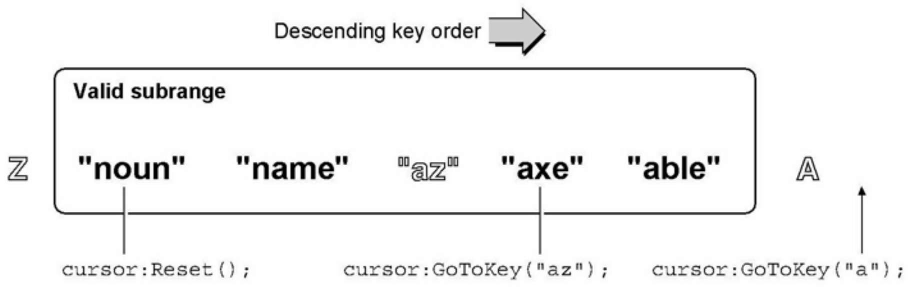
This section provides examples of the behavior of cursors that use descending indexes. These examples are based on a soup containing the entries shown in the following code fragment; although this example uses string values, any kind of index key value may be sorted in descending order.

```
{data: "able", …};
{data: "axe", …};
{data: "name", …};
{data: "noun", …};
```

Soup indexes normally sort string data in ascending alphabetical order; for example, `"able"`, `"axe"`, `"name"`, `"noun"`. A descending index sorts the same data in reverse alphabetical order; for example, `"noun"`, `"name"`, `"axe"`, `"able"`.

Figure 11-6 depicts the reversed ordering that a descending index provides, with examples of cursor behavior that is a function of index ordering.

Figure 11-6 Cursor operations on descending index



Sending the `Reset` message to the cursor positions it at the first valid entry in index order. In this case, the first entry is `"noun"` because the entries are sorted in descending alphabetical order.

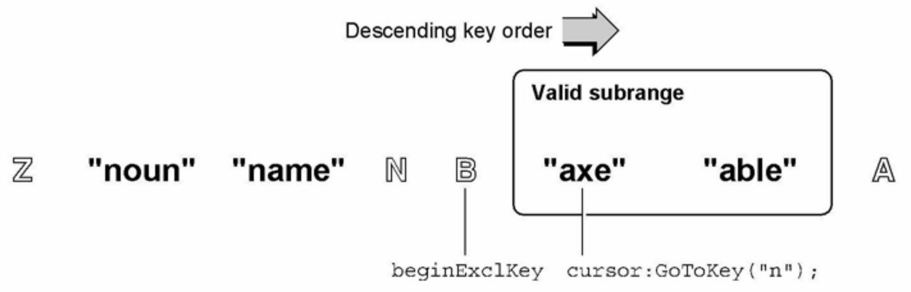
The `GoToKey` cursor method steps through the set of valid entries in index order until it finds the first entry having a specified key value. If the specified key value is not found, this method returns the next valid entry found after the specified index position. Thus, sending the `GotoKey("az")` message to this cursor returns the value `"axe"` because it's the first valid entry that appears in the index after the unoccupied `"az"` position.

Sending the `GotoKey("a")` message to this cursor returns the value `nil` because this message positions the cursor beyond the end of the range of valid entries defined by the query that generated the cursor.

Figure 11-7 illustrates that specifying a `beginExclKey` value of `"b"` excludes from consideration every entry beginning with a letter that comes after `"b"` in the reverse alphabet; that is, this `beginExclKey` value causes the valid range of entries to include only entries beginning with `"a"`. As a result, sending the `GotoKey("n")` message causes this cursor to return the value `"axe"` because it is the first valid entry appearing in the index after the `"n"` position.

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. ◆

Figure 11-7 Specifying ends of a descending index



Querying on Multiple-Slot Indexes

Before reading this section, you should understand the contents of "Querying on Single-Slot Indexes" beginning on page 11-39.

A multiple-slot query can be performed only on a soup that has a multiple-slot index generated against the same set of keys in the same order as the query spec. For information on creating an index, see “Registering and Unregistering Soup Definitions” beginning on page 11-33 and “Adding an Index to an Existing Soup” beginning on page 11-36. For a description of the data structure that defines a multiple-slot index, see “Multiple-Slot Index Specification Frame” (page 9-6) in *Newton Programmer’s Reference.*

In a general sense, queries on multiple-slot indexes are specified like queries on single-slot indexes and behave the same way. The “differences” you’ll encounter are usually the result of misunderstanding how multiple index keys are used to sort and select indexed entries.

For purposes of discussion, assume that you have a soup containing the entries in the following code fragment, and that you want to sort these entries alphabetically by last name and then by first name:

```
// entries used for all examples in this section
{last: "Perry", first: "Bruce", num: 1}
{last: "Perry", first: "Ralph", num: 2}
{last: "Perry", first: "Barbara", num: 3}
{last: "Perry", first: "John", num: 4}
{last: "Bates", first: "Carol", num: 5}
{last: "Perry", first: "Daphne", num: 7}
```

A single-slot index sorts entries according to the value held in a single slot that you specify when the index is created. In contrast, a multiple-slot index may consider the values of multiple slots when sorting entries. It’s important to understand that either kind of index imposes only one sort order on the indexed data, regardless of the number of slots examined to arrive at that order. A query on index values evaluates its associated entries in this order, and the cursor generated by this query iterates over its entries in this order, as well.

The first example illustrates how the entries in the example data could be sorted by a single-slot index. For purposes of discussion, assume that these entries are indexed on the value that each holds in its `last` slot, as specified by the single-slot index spec in the following code fragment:

```
// single-slot index on string data from 'last slot
{structure:'slot, path: 'last, type:'string}
```

Sorting the entries according to the value each holds in its `last` slot isn’t very useful because all of the entries except one hold an identical value in this slot. Unfortunately, sorting the entries on the value of another slot does not produce a useful ordering, either: an index on any other single slot sorts the `"Bates"` entry in the midst of all the `"Perry"` entries.

A multiple-slot index solves this problem by sorting entries according to multiple key values. The key values are extracted from up to six index paths specified by the `path` array of the index specification frame. For example, the following code fragment specifies a multiple-slot index that sorts entries according to the values each holds in its `'last`, `'first`, and `'num` slots:

```
// multiple-slot index on data from three slots
myMultiSlotSpec :=
   {structure:'multiSlot,
   path: ['last,'first,'num],
   type: ['string, 'string, 'int }
```

The first key in the `path` array is called the primary key; subsequent lower-order keys, if they are present, are the secondary key, tertiary key, and so on, up to a total of six keys per array.

The primary key specifies a minimum criterion for inclusion in the index and provides a value used to sort the indexed entries initially. In the example, only entries having a `last` slot are indexed, and the value of the `last` slot is used to impose an initial ordering on the indexed entries. Thus, the multiple-slot index in the previous example sorts the `"Bates"` entry before all of the `"Perry"` entries.

The secondary key, if it is present, is used to sort entries having identical primary keys. In the previous example, the multiple-slot index imposes a secondary ordering on all `"Perry"` entries, according to the value each holds in its `first` slot. Similarly, the tertiary key, if present, is used to sort further any entries having identical secondary key values. Because none of the entries in the example have identical secondary key values (none of the `first` slots hold identical values), the value of each entry's `num` slot has no effect on how the index sorts the entries.

Thus, the multiple-slot index shown previously sorts the set of sample entries in the following order:

```
{last: "Bates", first: "Carol", num: 5}
{last: "Perry", first: "Barbara", num: 3}
{last: "Perry", first: "Bruce", num: 1}
{last: "Perry", first: "Daphne", num: 7}
{last: "Perry", first: "John", num: 4}
{last: "Perry", first: "Ralph", num: 2}
```

Now that you're familiar with the manner in which multiple-slot indexes sort entries, let's look at the way the `Query` method uses a multiple-slot index to select entries.

Missing slots in a multiple-slot query spec are treated as `nil` values, just as they are when querying on single-slot indexes. For example, if the query spec is missing an `endKey` slot, the upper end of the range of entries examined by the query is unbounded, just as it would be for a query on a single-slot index.

Instead of using single values for the `indexPath`, `beginKey`, `beginExclKey`, `endKey`, and `endExclKey` slots in the query spec, the `Query` method accepts arrays of keys or values as these arguments when it works with a soup having a multiple-slot index. The first key in the array is the primary key; subsequent lower-order keys, if they are present, are the secondary key, tertiary key, and so on, up to a total of six keys per array.

To get a better idea of how queries evaluate multiple-slot key selectors, consider how the `beginKey` value in the following code fragment would work with the example data:

```
myQSpec := {indexPath: ['last,'first,'num],
             beginKey:["Perry","Bruce",5]}
```

Querying the example data using this specification returns a cursor that initially references the following entry:

```
{last: "Perry", first: "Daphne", num: 7}
```

First, the query finds the primary key value of `"Perry"` in the index, skipping over the `"Bates"` entry in the process of doing so. Next, the query searches for an index value identical to the secondary key `"Bruce"`, skipping over the `"Barbara"` entry in the process of doing so. Finally, the query searches for an index value identical to the tertiary key value `5`. Because an entry having this value is not found, the cursor is positioned on the next valid entry in index order, which has the tertiary key value `7`.

When specifying strings as bounding values for queries, don't forget that the `beginKey`, `beginExclKey`, `endKey`, and `endExclKey` slots in a query spec specify identical matches only. For example, the key value `"P"` is not identical to the key value `"Perry"`.

When an identical index value cannot be found for a key specification, subordinate key values have no effect. For example, if the primary key value is not matched, the secondary and tertiary key values have no effect.

To demonstrate these points, imagine that you wrote the query spec in the previous example a bit differently. Instead of specifying a value of `"Perry"` for the primary element in the `beginKey` array, assume you specified a value of `"P"`. This change would make the query spec look like the following code fragment:

```
myQSpec := {indexPath: ['last,'first,'num],
             beginKey:["P","Bruce",5]}
```

Querying our example data using this specification returns a cursor that initially references the following entry:

```
{last: "Perry", first: "Barbara", num: 3}
```

This time around, the query again skips over the `"Bates"` entry in the process of positioning the cursor at index value `"P"`. However, because no entry holds a primary index key value of `"P"`, the cursor stops at the next valid entry in index order. Further, because an identical index value was not found for the primary key specification, the secondary and tertiary key selectors have no effect at all. Thus the cursor stops on the first index value found after the position that `["P","Bruce",5]` would occupy if it were present in the index data.

When an element of an array in a query spec is missing or `nil`, the `Query` method does not test subordinate key values specified by the array. For example, the presence of the `nil` value in the `endKey` specification `{endKey : ["bob", nil, 55]}` makes it equivalent to the `{endKey : ["bob"]}` specification.

One result of this behavior is that it is impossible to make a query ignore higher-order sort keys while still testing on lower-order keys. For example, it is meaningless to specify a value such as `[nil,` *validKey, …*`]` for the `beginKey`, `beginExclKey`, `endKey`, or `endExclKey` slot in a query spec—the `nil`-value primary element specifies that the query is to ignore subsequent elements of the array.

If you want to be able to ignore key specifiers in a query spec selectively, you need to define for your entries a default "`nil`-equivalent" value that does have a position in index order. For example, you could use the empty string (`""`) for string key values, either of the values `0` or `MININT` for integer key values, and the null symbol (`'||`) for symbolic key values.

Further, the presence of a `nil`-value index key in an entry suppresses the evaluation of lower-order keys in that entry for sorting in the multiple-slot index. For example, the entries in the following code fragment sort to the same position in the multiple-slot index because as soon as the system encounters the `nil` key value in each entry's `secondary` slot, it does not attempt to sort that entry any further:

```
{primary: "foo", secondary: nil, tertiary: "bar"}
{primary: "foo", secondary: nil, tertiary: "qux"}
```

Querying explicitly for `nil` key values (`nil`-value slots) is not supported. Your entries' indexed slots must hold non-`nil` values to participate in queries.

For cursors generated against multiple-slot indexes, the cursor method `GoToKey` accepts arrays of keys as its argument. You can use this method to experiment with multiple-slot key specifications.

Similarly, for queries on multiple-slot indexes, the input passed to the `indexValidTest` function is an array of key values, with the first key in the array being the primary key, followed by any subordinate key values held by the entry being tested.

▲ **WARNING**
Index keys are limited to a total of 39 unicode characters (80 bytes, 2 of which are used internally) per soup entry. Keys that exceed this limit may be truncated when passed to an `indexValidTest` function. This 80-byte limit applies to the entire key space allocated for an entry, not for individual keys. As a result, subordinate keys in multiple-slot indexes may be truncated or missing when the total key size for the entry is greater than 80 bytes. For more information, see the description of the `indexValidTest` function in "Query Specification Frame" (page 9-9) in *Newton Programmer's Reference*. See also the description of the `MakeKey` method (page 9-45) in *Newton Programmer's Reference*. ▲

Limitations of Index Keys

Under the following conditions, a string may not match its index key exactly:

- Keys of type `'string` are truncated after 39 unicode characters.
- Ink data is stripped from `'string` keys.
- Subkeys in multiple-slot indexes may be truncated or missing when the total key size is greater than 80 bytes.

You can use the `MakeKey` function to determine precisely the index key that the system generates for a particular string. The interface to this function looks like the following code fragment:

soup`:MakeKey(`*string*`, `*indexPath*`)`

The following examples presume that *mySoup* is a valid soup (not a union soup) having the multiple-slot index specified by the following code fragment:

```
myMultiSlotIndexSpec := {structure: ' multislot,
                        path: ['name.first,
                               'cardType,
                               'name.last],
                        type : ['string, 'int, 'string]};
```

Each of the soup's entries has a `name` slot and a `cardType` slot. The `name` slot holds a frame containing the slots `first` and `last`, which contain string data. The `cardType` slot holds integer data.

The first example illustrates the truncation of string keys longer than 39 characters. Evaluating the following code fragment in the Inspector

```
mySoup:MakeKey(["12345678901234567890", 3,
               "ABCDEFGHIJKLMNOPQRSTUVWXYZ"],
               ['name.first, 'cardType, 'name.last] )
```

returns the key value

```
["12345678901234567890", 3, "ABCDEFGHIJKLMNO"]
```

The next example illustrates the truncation of subkeys when the total key size is greater than 80 bytes. In this example, the first string in the *string* array is so long that it uses up the entire 80 bytes allocated for the key, with the result that the first string is truncated and the remaining key values are `nil`. Evaluating the following code fragment in the Inspector

```
mySoup:MakeKey(["12345678901234567890abcdefghijjlmnopqrstuvwxyz",
            3, "ABCDEFGHIJKLMNOPQRSTUVWXYZ12345678901234567890"],
            ['name.first, 'cardType, 'name.last] )
```

returns the key value

```
["12345678901234567890abcdefghijjlmnopqr", NIL, NIL]
```

Missing elements in the *string* array are treated as `nil` values. For example, the following code fragment is missing the second two elements of the *string* array:

```
mySoup:MakeKey(["12345678901234567890],
               ['name.first, 'cardType, 'name.last] )
```

Evaluating this code fragment in the Inspector returns the key value

```
["12345678901234567890", NIL, NIL]
```

On the other hand, missing index paths cause this method to throw an exception. If one of the index paths in a multiple-slot index is missing from the array passed as the value of the *indexPath* parameter, the `MakeKey` method throws a “soup index does not exist” `evt.ex.fr.store -48013` exception.

Using Cursors

This section discusses the functions and methods used to work with cursor objects returned by the `Query` method of soups or union soups. 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
- testing validity of the cursor
- getting the currently referenced soup entry from the cursor
- moving the cursor

- getting the number of entries in cursor data
- getting an index key from the cursor
- copying the cursor

Getting a Cursor

Cursor objects are returned by the `Query` method. For more information, see "Using Queries" beginning on page 11-38.

Testing Validity of the Cursor

When a storage card is inserted or a soup is created, union soups include new soups in the union automatically as is appropriate. A cursor on a union soup may not be able to include a new soup when the new soup's indexes do not match those present for the other soups in the union. In particular, this situation can occur when

- The new soup does not have the index specified in the `indexPath` of the query spec used to generate the cursor.
- The query spec used to generate the cursor included a `tagSpec` and the new soup does not have the correct 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. Note that a valid cursor returns `nil` when it receives a message that positions it outside of the range of valid entries. (For an example, see the text accompanying Figure 11-6 on page 11-46.)

You can test the cursor's validity by invoking the `Status` cursor method. This method 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 Methods" (page 9-60) in *Newton Programmer's Reference*. For a discussion of soup change notification messages, see the section "Callback Functions for Soup Change Notification" (page 9-14) in *Newton Programmer's Reference*.

Getting the Entry Currently Referenced by the Cursor

To obtain the entry currently referenced by the cursor, 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();
```

Moving the Cursor

This section describes various ways to position the cursor within the range of entries it references.

Sometimes the following discussion refers to the "first" entry in a cursor. As you know, the order imposed on cursor data is defined by the soup index used by the query that generated the cursor. When you see mentions of the "first" entry in a cursor, be aware that this phrasing really means "the first entry as defined by index order (ascending or descending order)."

When first returned by a query, the cursor points to the first 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.

You can also position the cursor on the first entry in its data set by sending the `Reset` message. The `Reset` method moves the cursor to the first valid entry in the query result and returns that entry. For example:

```
local cursor := mySoup:Query(nil);
// move the cursor ahead a bit
local anEntry := cursor:Move(3);
// go back to first entry
local firstEntry := cursor:Reset();
```

Note that if the query spec includes a `beginKey` value, the `Reset` method returns the first valid entry at or after the `beginKey` value in index order.

To obtain the last entry in the set of entries referenced by the cursor, send the `ResetToEnd` cursor message, as shown in the following example:

```
local cursor := mySoup: Query({indexPath: 'name,
                              endKey: "ZZ"});
local lastCursorEntry := cursor:ResetToEnd();
```

Note that if the query spec includes an `endKey` value, the `ResetToEnd` method positions the cursor on the last valid entry in index order at or before the specified `endKey` value. For example, if you specify an `endKey` value of `"Z"` but the last valid entry previous to that in index order has the key value `"gardenia"`, the entry associated with the `"gardenia"` key value is returned.

The cursor can be advanced to the next entry in index order 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. If sending either of these messages positions the cursor outside of the range of valid entries, it returns `nil`.

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 5 do myCursor:Next();
```

you can obtain faster results by using the Move method. The following code fragment depicts a typical call to this method. After positioning the cursor, the Move method returns the current entry.

```
// skip next four entries and return the fifth one or nil
local theEntry := myCursor:Move(5);
```

To move the cursor in large increments, it's faster to use the GoTo and GoToKey methods to position 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 example:

```
// index spec for soup that generated myCursor
indxSpec: {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");
```

If the argument to the GoToKey method is not of the type specified by the soup's index spec, this method throws an exception. For example, the index spec in the previous example specifies that the name slot holds string data. If you pass a symbol to the GoToKey method, it signals an error because this soup's index holds string data:

```
// throws exception - arg doesn't match index data type
myCursor:GotoKey('name);
```

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 referenced by 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:

```
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 first valid entry having an index key value equal to or greater than the `beginKey` value. Similarly, if the query that generated the cursor used an `endKey` value, the `CountEntries` method stops counting at the last valid entry having an index key value equal to or less than the `endKey` value.

Note that the use of the `CountEntries` method is somewhat time-consuming and may increase your application's heap space requirements; for performance reasons, use this method only when necessary.

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 the NewtonScript heap. Note, however, that under certain circumstances the value returned by this method does not match the entry's index key data exactly. For more information, see "Limitations of Index Keys" on page 11-52.

Copying Cursors

You can clone a cursor to use for browsing soup entries without disturbing the original cursor. Do not use the global functions `Clone` or `DeepClone` to clone cursors. Instead, use the `Clone` method of the cursor to be copied, as shown in the following code fragment:

```
local namesUSoup:= GetUnionSoupAlways(ROM_CardFileSoupName);
local namesCursor := namesUSoup:Query(nil);
local cursorCopy:= namesCursor:Clone();
```

Using Entries

This section discusses the functions and methods that work with soup entry objects returned by cursors. This section describes

- adding entries to soups
- removing entries from soups
- saving references to entries
- modifying entries
- replacing entries
- sharing entry data
- copying entry data
- using the entry cache effectively

Saving Frames as Soup Entries

To save a frame as a soup entry, pass the frame to either of the union soup methods `AddToDefaultStoreXmit` or `AddToStoreXmit`, or pass it to the `AddXmit` soup method. Each of these methods 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 the `AddToDefaultStoreXmit` method:

```
local myFrame := {text: "Some info", color: 'blue};
// assume mySoupDef is a valid soup definition
local myUSoup := RegUnionSoup(mySoupDef)
myUSoup:AddToDefaultStoreXmit(myFrame,'|MyApp:MySig|);
```

The new soup entry that these methods create consists of the frame presented to the entry-creation method, along with copies of any data structures the frame references, as well as copies of any data structures those structures reference, and so on. Thus, you must be very cautious about making soup entries out of frames that include references to other data structures. In general, this practice is to be avoided—it can result in the creation of extremely large entries or entries missing slots that were present in the original frame.

For example, the presence of a `_parent` slot in the frame presented as an argument to these methods 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 find the parent object in a frame of templates at run time.

Do not include `_proto` slots in frames presented to methods that create soup entries. These slots are not written to the soup entry and are missing when the entry is read from the soup.

Do not save magic pointers in soup entries. Because the objects they reference are always available in ROM, saving magic pointers is unnecessary and may cause the entries to exceed the maximum permissible size.

Circular pointers within an entry are supported, and an entry can refer to another by using an entry alias.

The size of an individual entry is not limited by the NewtonScript language; however, due to various practical limitations, entries larger than about 16 KB can impact application performance significantly. For best performance, it is recommended that you limit the size of individual entries to 8 KB or less. Note that this total does not include data held by virtual binary objects that the entry references; virtual binary objects save their data separately on a store specified when the virtual binary object is created. For more information, see “Virtual Binary Objects” on page 12-2 in Chapter 12, “Special-Purpose Objects for Data Storage and Retrieval.”

No more than 32 KB of text (total of all strings, keeping in mind that one character is 2 bytes) can reside in any soup entry. Another practical limitation is that there must be space in the NewtonScript heap to hold the entire soup entry. You should also be aware that Newton Backup Utility and Newton Connection Kit do not support entries larger than 32K.

Keeping these limitations 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 to index. 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, 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` cause various exceptions. For example, if you should try to add this kind of frame to an indexed soup, the method that attempts to add the frame throws an exception; if you try to add a new index on a slot that varies in data type from entry to entry, the `AddIndex` method throws an exception, and so on. Soup entries can contain `nil`-value slots, but querying for such slots is not supported; that is, you can query only for slots that hold non-`nil` values.

Removing Entries From Soups

To remove an entry, pass it to the `EntryRemoveFromSoupXmit` function, as shown in the following code fragment. If you try to remove an invalid entry, this function throws an exception. An entry can become invalid when, for example, the user ejects the storage card on which it resides.

```
local myCursor := Query(nil);
local theEntry := myCursor:Entry();
if theEntry then
   EntryRemoveFromSoup(theEntry, '|MyApp:MySig|);
```

Modifying Entries

Only one instance of a particular entry exists at any time, regardless of how the entry was obtained. That is, if two cursors from two different queries on a particular soup 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 in the NewtonScript heap. The frame is 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 do 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` function 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. Note that you can use the `FrameDirty` function to determine whether a cached entry has been modified since it was read into the NewtonScript heap; however, this function does not detect changes to individual characters in a string (a common operation for `clParagraphView` views). For more information, see "FrameDirty" (page 9-69) in *Newton Programmer's Reference.*

The following code example gets an entry from the `namesUSoup` union soup, changes it, and writes the changed entry back to the soup:

```
local namesUSoup := GetUnionSoupAlways(ROM_CardFileSoupName);
local namesCursor := namesUSoup:Query(nil);
local theEntry := namesCursor:Entry();
if theEntry then
   begin
      theEntry.cardType := 4;
      EntryChangeXmit(theEntry, '|MyApp:MySig|);
   end;
```

It's not always easy to determine the best time to write a cached entry back to its soup. For example, it would be inappropriate to call a function like `EntryChangeXmit` 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 for 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` would be when the user dismisses an input slip.

Moving Entries

You can use the `MoveTarget` method of the root view to move (not copy) an entry into the same-named soup on another store. For example, you would use this method to move entries from one union soup member to another. For more information, see "System-Supplied Filing Methods" (page 12-11) in *Newton Programmer's Reference.*

Copying Entries

The `EntryCopyXmit` global function and the `CopyEntriesXmit` soup method enable you to copy entries from one soup to another and transmit appropriate change notifications.

The following code fragment uses the `CopyEntriesXmit` soup method to copy all the entries from a specified source soup into a specified destination soup. Note that this method is defined only for soups, not for union soups. The following code fragment uses the `GetMember` union soup method to retrieve the plain soup constituent of a union soup from a specified store. The `GetMember` method never returns `nil`; instead, it creates an empty member soup on the specified store if one does not already exist:

```
// myUsoup member on internal store is the source soup
local myUSoup := GetUnionSoupAlways("myUSoup:mySig");
local sourceSoup := myUSoup:GetMember(GetStores()[0])
// myUsoup member on another store is the destination soup
local destSoup :=  myUSoup:GetMember(GetStores()[1]);
// copy all entries from source soup to dest soup
local cursor := sourceSoup:Query(nil);
if (cursor:CountEntries() <> 0) then
   sourceSoup:CopyEntriesXmit(destSoup, '|MyApp:MySig|);
```

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. Note that this function is defined only for soups, not for union soups. The following code fragment uses the `GetSoup` store method to retrieve a specified soup from its store. Because the `GetSoup` method returns `nil` when the soup to be retrieved is not available, you must at least ensure that this result is non-`nil` before using it. The following code fragment actually goes one step further and uses the `IsValid` soup method to test the validity of the `destSoup` soup in additional ways:

```
local myUSoup := GetUnionSoupAlways("myUSoup:mySig");
// get all entries having 'hot in 'temperature slot
local cursor := myUSoup:Query({indexPath: 'temperature,
                              beginKey: 'hot
                              endKey: 'hot});
local destSoup := GetStores()[0]:GetSoup("mySoup:mySig");
// make sure we actually got a valid soup
if destSoup:IsValid() then
   begin
      // xmit a single notification after all changes are made
      while e := cursor:Entry() do EntryCopyXmit(e,destSoup,nil);
      XmitSoupChange(destSoup, '|MyApp:MySig|, 'whatThe, nil);
   end;
```

Note
The `EntryCopyXmit` method copies the cached entry—not the original soup entry—into the destination soup. ◆

Sharing Entry Data

Shared soups and shared 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 19, "Built-in Applications and System Data." There you will see descriptions of the soups used by the built-in applications on Newton devices produced by Apple Computer, Inc.

Using the Entry Cache Efficiently

Whenever you access a slot in a soup entry, the system reads the entire entry into the NewtonScript heap 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. For best performance, avoid creating cached entries when you don't need them, and flush the entry cache as soon as is appropriate. This section describes how you can avoid unnecessary caching and how you can reclaim cache memory explicitly. Table 11-1 on page 11-63 summarizes the use of the entry cache by the functions and methods described in this discussion.

Reading a soup entry into memory requires more heap space than testing tag or index values does. 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 the NewtonScript heap.
- Use the cursor method `EntryKey` to retrieve an entry's key value without reading the entry into the NewtonScript heap.

Normally, adding or changing a soup entry creates a cached entry. If you do not plan on working further with an entry's data after you've added or modified it, you can reclaim heap space by releasing the memory used by the entry cache. You can use the `AddFlushedXmit` soup method to add a soup entry without creating a cached entry at all; in addition to saving heap space, this method saves you the time normally required to create the cached entry. When working with a cached entry, you can use the `EntryFlushXmit` function to write it back to its soup and clear the entry cache.

In contrast, the `EntryUndoChanges` function clears the entry cache without writing the cached entry to the soup. This function makes references to the entry point to the original, unmodified entry residing in the soup, instead of the cached entry.

Note that reading, printing, or modifying any slot in the entry after calling `EntryFlushXmit`, `EntryUndoChanges`, or `AddFlushedXmit` causes the entire entry to be read back into the NewtonScript heap; thus, use these functions only when you're sure you won't need to access the entry in the near future.

If you do need to work with the entry data after you've written it to the soup, you'll want to use functions and methods that don't clear the entry cache after writing the soup entry. The `AddToDefaultStoreXmit` and `AddToStoreXmit` union soup methods save frames as soup entries without clearing the entry cache afterward. When adding frames to single soups, you can use the `AddXmit` soup method for the same purpose. The `EntryChangeXmit` function also writes the cached entry back to its soup without flushing the cache afterward. Contrast this function with the `EntryFlushXmit` function, which clears the entry cache after writing the cached entry back to its soup. Table 11-1 summarizes the caching behavior of all methods that write entries to soups or union soups.

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

Function or method	**Cached entry**
uSoup:`AddToDefaultStoreXmit`(*frame*, *changeSym*)	Creates and returns
uSoup:`AddToStoreXmit`(*frame*, *changeSym*)	Creates and returns
soup:`AddXmit`(*frame*, *changeSym*)	Creates and returns
soup:`AddFlushedXmit`(*frame*, *changeSym*)	Does not create or return
`EntryFlushXmit`(*entry*)	Returns existing
`EntryChangeXmit`(*entry*)	Returns existing
`EntryUndoChanges`(*entry*)	Throws away existing

Using Soup Change Notification

When your application changes an entry in a shared soup, the system executes callback functions registered by applications using that soup, allowing them to take action in response to the change. The system-supplied soup change notification service allows applications to

- notify each other when they make changes to soup entries
- respond to notifications precisely

- 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. Note that this callback function must not call either of the `RegSoupChange` or `UnRegSoupChange` functions.

If your application needs to respond to changes in more than one soup, you'll need to call the `RegSoupChange` function once on each soup for which your application requires change notification. This approach is valid for any system-supplied soup except that used by the built-in Preferences application. For notification of changes to user preferences, you must call the `RegUserConfigChange` function.

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 callback functions remain registered.

The following code example shows how to register your application for notification of changes to the soup used by the built-in Names application:

```
local myFn := func (soupName, appSym, changeType, changeData)
         begin
            if (changeType) then
               begin
               if (changeType <> 'whatThe) then
                  print (changeType && "in the" && soupName &&
                        "soup by the" && GetAppName(appSym) &&
                        "application.");
               else
                  print ("Unspecified changes occurred in the" &&
                        soupName && "soup.");
               end;
         end;
// register for changes to soup used by built-in "Names" app
RegSoupChange(ROM_CardFileSoupName, '|myFn1:MyApp:MySig|, myFn);
```

▲ **WARNING**
Any callback function registered by the `RegSoupChange` function must not call either of the `RegSoupChange` or `UnRegSoupChange` functions. ▲

The second argument to the `RegSoupChange` function can be any unique symbol that identifies the callback to be registered. If your application registers only one callback function, you can just use your application symbol as the callback identifier (ID). A callback ID need only be unique within the registry that uses it. For example, no two power registry callback functions can share the same callback ID; on the other hand, your application's power registry callback can use the same ID as your application's login screen callback. Thus, if your application only registers one callback function with each of the various registries, all of your callback functions can use your application symbol (with developer signature) as their callback ID.

To generate unique identifiers for multiple callbacks within the same registry, you can prefix an additional identifier to your application symbol. For example, the symbol `'|myFn1:MyApp:MySig|` could be used to identify one of several callback functions registered by the `MyApp:MySig` application.

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.

```
// unregister my app's Names soup callback
UnRegSoupChange(ROM_CardFileSoupName, '|myFn1:MyApp:MySig|);
```

Normally, you can unregister your soup change callbacks in the `viewQuitScript` method of your application's base view.

Responding to 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 a cached entry is not considered a change to the soup until the `EntryChangeXmit` function writes the cached entry back to the soup.

Note
The system-supplied Preferences application sends soup change notifications only if your application uses the `RegUserConfigChange` function to register for such notifications. ◆

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 simple code example, see "Registering Your Application for Change Notification" beginning on page 11-64. For a complete description of the callback function and its parameters, see the section "Callback Functions for Soup Change Notification" (page 9-14) in *Newton Programmer's Reference.*

▲ **WARNING**
The `'soupEnters` and `'soupLeaves` messages are guaranteed to be sent only when a reference to the changed soup exists. These messages may not be sent for soups that are not in use. For example, if no cursor object references the soup, this message may not be sent. ▲

Sending Notifications

When your application alters a 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 transmit 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, such as adding frames to soups as entries, changing entries, and so on.

The auto-transmit (*fnOrMethodName*`Xmit`) routines provide the easiest and best way to send notifications when making a limited number of changes to a soup. For example, to save a frame in a union soup and transmit an appropriate notification message, use the `AddToDefaultStoreXmit` method as shown in the following code fragment:

```
// get soup in which to save the new entry
local myUSoup := GetUnionSoupAlways("myUSoup:mySig");
// frame to add as new entry
local myFrame := {name: Daphne, color: tabby};
// add the entry and transmit change notification
local ent := myUSoup:AddToDefaultStoreXmit(myFrame,'|MyApp:MySig|);
```

The auto-transmit methods and functions accept a *changeSym* parameter identifying the application that 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 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, as shown in the following code example:

```
// assume cursor and destSoup are valid
// xmit a single notification after all changes are made
while e := cursor:Entry() do EntryCopyXmit(e,destSoup,nil);
XmitSoupChange("mySoup:mySig", '|MyApp:MySig|, 'whatThe, nil);
```

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. Because this particular example makes multiple changes to the `destSoup` soup, it passes the `'whatThe` symbol to indicate unspecified changes, and passes `nil` as the change data. For a more detailed discussion of change type and change data, see the section "Callback Functions for Soup Change Notification" (page 9-14) in *Newton Programmer's Reference*.

Soup change notification messages are sent on a deferred basis. In most situations, this implementation detail has no practical impact; however, you should be aware that soup change messages are not sent until after the method that sends them returns. For example, if your `ButtonClickScript` method causes a soup change, the change notification message is not sent until after the `ButtonClickScript` method returns.

Summary of Data Storage

This section summarizes data structures, functions, objects and methods used for data storage on Newton devices.

Data Structures

Soup Definition Frame

```
mySoupDef :=
{  // string that identifies this soup to the system
   name: "appName:appSym",
   // string that is user visible name
   userName: "My Application soup",
   // application symbol
   ownerApp: '|myApp:mySig|,
   // user-visible name of app that owns this soup
   ownerAppName: "My Application",
   // user-visible string describing soup
   userDescr: "This soup is used by
               My Application.",
   // array of indexSpecs - default indexes
   indexes: [anIndexSpec, anotherIndexSpec]
   // optional function used to initialize the soup
   initHook: symbolOrCallBackFn
}
```

Single-Slot Index Specification Frame

```
{
   // must use this value - index keys are slot values
   structure:'slot,
   // entries indexed on this slot
   path: pathExpr,
   // data type found in the indexed slot
   type: symbol,
   // optional. 'ascending or 'descending
   order: symbol,
   // optional. pass 1 to use alternate sort table
   sortID: nil
}
```

Multiple-Slot Index Specification Frame

```
{
   // index keys may be multiple slot values
   structure: 'multiSlot, // must use this value
   // up to six path expressions specifying indexed slots
   path:[pathExpr1, pathExpr2, … , pathExpr6],
   // data type found in each indexed slot
   type:[sym1, sym2, … sym6]
   // optional. 'ascending or 'descending
   order: [sym1, sym2, … sym6]
   // optional. pass 1 to use alternate sort table
   sortID: nil
}
```

Tags Index Specification Frame

```
{
   // must use this value - tags are slot values
   structure:'slot,
   // index values (tags) extracted from this slot
   path:'slotName,
   // must use this value
   type:'tags,
}
```

Query Specification Frame

```
// pass nil instead of a query spec frame
// to retrieve all entries in the soup

// this frame used for queries on single-slot indexes
// see next example for multiple-slot query spec frame
{
// use the specified single-slot index for this query
// required when querying for index values
indexPath : 'pathExpr,
// minimum index key value examined by this query
// for all entries, (beginKey ≤ entry.indexPath)
beginKey : keyValue, // optional
// excluded lower boundary of key range examined by query
// for all entries, (beginExclKey < entry.indexPath)
beginExclKey : keyValue, // optional
```

```
// maximum index key value examined by this query
// for all entries, (entry.indexPath ≤ endKey)
endKey: keyValue, // optional
// excluded upper boundary of key range examined by query
// for all entries, (beginExclKey < entry.indexPath)
endExclKey : keyValue, // optional
// returns non-nil to include entry in result
indexValidTest: func (keyValue) begin … end;, // optional
// returns non-nil to include entry in result
validTest: func (entry) begin … end; // optional
// optional tags query spec frame; see page 11-71
tagSpec: {equal:[t1, t2, ...tN], all:[t1, t2, ...tN],
         any:[t1, t2, ...tN],none:[t1, t2, ...tN]},
// when non-nil, match entire string in 'words slot
entireWords: Boolean, // optional
// string(s) to match w/ word beginnings in entries
words: string|[str1, str2, … , strN], // optional
// string to match w/ any substring in entries
text: string, // optional
}

// this frame used for queries on multiple-slot indexes
// see previous example for single-slot query spec frame
{
// use the specified multiple-slot index for this query
indexPath : ['pathExpr1,'pathExpr2, …'pathExpr6], // required
// minimum index key value examined by this query
// for all entries, (beginKey ≤ entry.indexPath)
beginKey : [keyValue1,keyValue2 … keyValue6], // optional
// excluded lower boundary of key range examined by query
// for all entries, (beginExclKey < entry.indexPath)
beginExclKey : [keyValue1,keyValue2 … keyValue6], // optional
// maximum index key value examined by this query
// for all entries, (entry.indexPath ≤ endKey)
endKey: [keyValue1,keyValue2 … keyValue6], // optional
// excluded upper boundary of key range examined by query
// for all entries, (beginExclKey < entry.indexPath)
endExclKey : [keyValue1,keyValue2 … keyValue6], // optional
// optional; returns non-nil to include entry in result
indexValidTest: func ([keyValue1,keyValue2 … keyValue6])
                     begin … end;,
// optional; returns non-nil to include entry in result
validTest: func (entry) begin … end;
```

```
// optional tags query spec frame; see page 11-71
tagSpec: {equal:[t1, t2, ...tN], all:[t1, t2, ...tN],
          any:[t1, t2, ...tN],none:[t1, t2, ...tN]},
// when non-nil, match entire string in 'words slot
entireWords: Boolean, // optional
// string(s) to match w/ word beginnings in entries
words: string|[str1, str2, ... , strN], // optional
// string to match w/ any substring in entries
text: string, // optional
}
```

Tags Query Specification Frame

```
// this frame resides in tagSpec slot of query spec frame
// at least one of these slots must appear
// select only entries having identical set of tags
{equal:[t1, t2, ...tN],
// select only entries having identical tags or superset
all:[t1, t2, ...tN],
// select entries having any of these tags
any:[t1, t2, ...tN],
// select entries having none of these tags
none:[t1, t2, ...tN]}
```

Callback Functions for Soup Change Notification

```
func(soupNameString, appSymbol, changeTypeSymbol, changeData);
```

Data Storage Functions and Methods

Stores

```
store:AtomicAction(function)
store:BusyAction(appSymbol, appName, action)
store:CheckWriteProtect()
GetDefaultStore()
store:GetInfo(slotSymbol)
store:GetName()
store:GetSoup(soupNameString)
store:GetSoupNames()
GetStores()
store:HasSoup(soupName)
store:IsReadOnly()
```

store:`IsValid()`
`SetDefaultStore(`*newDefaultStore*`)`
store:`SetInfo(`*slotSymbol*, *value*`)`
store:`TotalSize()`
store:`UsedSize()`

Soups

These functions and methods allow you to work with soup-level data such as frames, soup indexes, soup information frames, and soup signatures.

Creating Soups

`RegUnionSoup(`*appSymbol*, *soupDef*`);`
`UnRegUnionSoup(`*name*, *appSymbol*`);`
store:`CreateSoupXmit(`*soupName*, *indexArray*, *changeSym*`)`
`CreateSoupFromSoupDef(`*soupDef*, *store*, *changeSym*`)`
uSoup:`GetMember(`*store*`)`

Adding and Copying Entries

uSoup:`AddToDefaultStoreXmit(`*frame*, *changeSym*`)`
uSoup:`AddToStoreXmit(`*frame*, *store*, *changeSym*`)`
soupOrUsoup:`AddFlushedXmit(`*frameOrEntry*, *changeSym*`)`
soup:`AddXmit(`*frame*, *changeSym*`)`
soup:`CopyEntriesXmit(`*destSoup*, *changeSym*`)`

Retrieving Entries

soupOrUSoup:`Query(`*querySpec*`)`

Change Notification

`RegSoupChange(`*soupName*, *callbackID*, *callBackFn*`)`
`UnRegSoupChange(`*soupName*, *callbackID*`)`
`XmitSoupChange(`*soupName*, *appSymbol*, *changeType*, *changeData*`)`

Manipulating Tags

soup:`HasTags()`
soup:`GetTags()`
soupOrUsoup:`ModifyTagXmit(`*oldTag*, *newTag*, *changeSym*`)`
soupOrUsoup:`RemoveTagsXmit(`*tags*, *changeSym*`)`
soupOrUsoup:`AddTagsXmit(`*tags*, *changeSym*`)`

Additional Functions and Methods

soupOrUsoup:`AddIndexXmit(`*indexSpec*, *changeSym*`)`
soup:`GetIndexes()`
soup:`GetInfo(`*slotSymbol*`)`
soupOrUsoup:`GetName()`
soup:`GetSignature()`

soupOrUsoup:`GetSize()`
uSoup:`GetSoupList()`
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*`)`

Cursors

These functions and methods work with the cursor object returned by the `Query` method.

Cursor Validity
cursor:`Status()`

Retrieving Entries and Manipulating the Cursor
cursor:`Entry()`
cursor:`Next()`
cursor:`Prev()`
cursor:`Move(`*n*`)`
cursor:`EntryKey()`
cursor:`GoToKey(`*key*`)`
cursor:`GoTo(`*entry*`)`
cursor:`Reset()`
cursor:`ResetToEnd()`
cursor:`WhichEnd()`

Additional Functions and Methods
`MapCursor(`*cursor*, *function*`)`
cursor:`CountEntries()`
cursor:`Clone()`

Entries

These functions allow you to work with individual soup entries returned by the cursor object.

`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*`)`
`EntryUndoChangesXmit(`*entry*`,` *changeSym*`)`
`EntryUniqueId(`*entry*`)`
`FrameDirty(`*frame*`)`
`IsSameEntry(`*entryOralias1*`,` *entryOralias2*`)`

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`:GetSignature()`
store`:SetName(`*storeNameString*`)`

soup`:AddWithUniqueIdXmit(`*entry*`,` *changeSym*`)`
soup`:GetAllInfo()`
soup`:GetIndexesModTime()`
soup`:GetInfoModTime()`
soup`:GetNextUid()`
soup`:SetSignature(`*signature*`)`
soup`:SetAllInfoXmit (`*frame*`,` *changeSym*`)`
`EntryChangeWithModTimeXmit(`*entry*`,` *changeSym*`)`
`EntryReplaceWithModTimeXmit (`*original*`,` *replacement*`,` *changeSym*`)`

CHAPTER 12

Special-Purpose Objects for Data Storage and Retrieval

This chapter describes the use of special-purpose objects to augment or replace the behavior of the system-supplied store, soup, cursor, and entry objects. This chapter describes

- the use of entry alias objects to save references to soup entries
- the use of virtual binary objects to store large amounts of binary data
- the use of store parts to build read-only soup data into packages
- the use of mock entry objects to implement your own suite of objects that provide access to nonsoup data in the same manner as the system-provided store, soup, cursor, and entry objects.

Before reading this chapter, you should understand the contents of Chapter 11, "Data Storage and Retrieval," which provides an overview of the Newton data storage system and describes how to use stores, soups, queries, cursors, and entries to meet most applications' data storage needs.

About Special-Purpose Storage Objects

The special-purpose data storage objects described here can be used to augment or replace the behavior of stores, soups, cursors, and entries.

Entry Aliases

An entry alias is an object that provides a standard way to save a reference to a soup entry. Unless it uses an entry alias to do so, a soup entry cannot save a reference to an entry in another soup—the referenced entry is copied into the host entry when the host entry is written back to its soup. However, entry aliases may be saved in soup entries without causing this problem.

Entry aliases are also useful for providing convenient access to entries from multiple soups. For example, the built-in Find service uses entry aliases to present entries from multiple soups in a single overview view.

Virtual Binary Objects

The size of any NewtonScript object is limited by the amount of memory available in the NewtonScript heap. As a result, you cannot create binary objects larger than the amount of available NewtonScript heap space. For similar reasons, the amount of data that can be stored in a single soup entry is limited as well. (See "Saving Frames as Soup Entries" beginning on page 11-58 for details.) You can use virtual binary objects to work around these restrictions.

A **virtual binary object** or **VBO** is a special kind of object that is useful for holding binary data larger than the available space in the NewtonScript heap. VBOs can be used to store large amounts of raw binary data, such as large bitmaps, the samples of large digitized sounds, fax data, packages, or application-specific binary data. A package is actually a special kind of virtual binary object; however, a package is read-only and is created in a slightly different manner than a normal VBO.

In the following ways, VBOs are like normal NewtonScript binary objects:

- The VBO is not persistent until it is written to a soup. As with any soup entry data, if a VBO in a soup entry is modified, the changes are not persistent until the cached entry frame is written back to the soup. If a soup entry containing a VBO is moved to another store, the binary data associated with the VBO is moved to that store as well. For a discussion of the soup entry cache, see "Entries" on page 11-17.
- The space used by the VBO is made available for garbage collection when there are no more references to the VBO.
- Binary data—including VBO data—is not shared between soup entries, even when their respective soups reside on the same store. As a result, you may need to consider space issues when moving or duplicating entries that hold VBO data.

VBOs are different from normal NewtonScript binary objects in the following ways:

- VBO data does not reside in the NewtonScript heap—it resides in store memory.
- Store memory for VBO data is not allocated until it is needed to write data. "Using Virtual Binary Objects" on page 12-8 discusses this important point in detail.
- You cannot use a value stored in a virtual binary object as a soup index key.
- VBOs can be created in compressed or uncompressed form. If the VBO is compressed, the system compresses and decompresses its associated binary data on demand. The fact that a VBO is compressed is normally transparent to your code; however, the time required to compress and uncompress VBO data may affect performance.
- When passed a reference to a VBO residing on a store that is unavailable, methods that write VBO data throw exceptions rather than displaying the "Newton still needs the card" alert.

Normal binary objects encapsulate their data and reside entirely in the NewtonScript heap; thus, creating one of these objects or reading any of its data requires an amount of heap space sufficient to hold all its data. Therefore, the size of a normal binary object is limited by the amount of NewtonScript heap space available at the time it is created. For example, a binary object encapsulating 5 KB of data requires 5 KB of NewtonScript heap space. If sufficient heap space is not available, the binary object cannot be created.

In contrast, VBO data resides on a store specified when the VBO is created. The system manages VBO data automatically, providing NewtonScript objects with transparent access to it on demand. A VBO can hold more data than a normal binary object because it is not limited by the amount of free space available in the NewtonScript heap. Contrasting the previous example, a VBO holding 5 KB of data requires a negligible amount of heap space, because its data resides in store memory, rather than in the NewtonScript heap.

Note
The system does not allocate store memory for VBO data until it is needed to write data to the store. Testing the amount of store memory available when the VBO is created does not guarantee the future availability of this memory. Thus, it is possible to fail due to lack of store space when writing to a VBO, even though the VBO was created successfully. The only practical solution to this problem is to enclose in a `try` block any code that writes VBO data. ◆

Parts

Recall that 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. 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.** Table 12-1 on page 12-4 lists the various kinds of parts and their associated
type identifiers.

Some of the parts described in Table 12-1 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. `Font` parts are used to add new typefaces to Newton devices; for more information about these parts, contact Newton Developer Technical Support. `Auto` parts are described in the *Newton Toolkit User's Guide*.

Table 12-1 Parts and type identifiers

Part	Type	Description
Application	`form`	Application.
Book	`book`	Book created by Newton Book Maker or Newton Press.
Auto part	`auto`	Background application/extension.
Store part	`soup`	Read-only soup.
Dictionary	`dict`	Custom dictionary for Newton recognition subsystem.
Font	`font`	Additional font.

Except for `soup` parts, all the parts listed in Table 12-1 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 composed 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` function that performs 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, soups residing on package stores must also be read-only. Store parts can be used to provide soup-like access to read-only data residing in an application package.

For more information about the characteristics of soups, see “Soups” on page 11-7 and “Using Soups” on page 11-32.

Mock Entries

A **mock entry** is a NewtonScript object that mimics the behavior of a soup entry. The mock entry is a foundation object you can use to build up a suite of objects that acts like the system-supplied store, soup, cursor, and entry objects. For example, you could create a mock entry object that uses a serial communications link to retrieve a record from a remote database; additional objects could implement methods to provide cursor-like access to these mock entries, just as if they resided in a local soup. Your mock entry could reside in a mock soup, which, in turn, could reside on a mock store.

The mock entry counterparts to the system-supplied Entry*Xxx* functions are implemented as the methods of a NewtonScript frame known as the mock entry's **handler.** You supply this frame, which implements these methods as well as any it requires for its own purposes. The handler may also hold information local to a specific mock entry or information required to retrieve the mock entry's data.

Like a normal soup entry, the mock entry caches its data in the NewtonScript heap when the entry is accessed; thus, the data associated with a mock entry is called its **cached frame.** As with normal soup entries, the cached frame appears to be the mock entry itself when accessed by other NewtonScript objects. Your handler provides an `EntryAccess` method that creates this frame in response to messages from the system.

The cached frame must be self-contained, just as a normal soup entry is. Therefore, the cached frame must not use `_proto` and `_parent` inheritance.

To create a mock entry, you call the `NewMockEntry` global function. Depending on your needs, you can create the mock entry with or without its associated cached frame. Either way, the mock entry object returned by this function manages other objects' access to its cached frame.

When the mock entry's cached frame is present, the system forwards entry accesses to it transparently. When the cached frame is not present, the system calls the handler's `EntryAccess` method to generate a cached frame before forwarding the access. You must supply this method, which creates and installs the cached frame in the mock entry.

The handler's `EntryAccess` method is called only when a slot in the mock entry is accessed. Simply referencing the mock entry does not cause the cached entry to be created. For example, in the following code fragment, assigning `m` to `x` does not create a cached entry—it just creates another reference to the mock entry. However, accessing the mock entry's `foo` slot from either of the variables `m` or `x` may cause the `EntryAccess` method of `myHandler` to be invoked.

```
local myHandler := {
         object: {foo: 'bar},
         EntryAccess: func (mockEntry)
            begin
               // install cached obj & notify system
               EntrySetCachedObject(mockEntry, object);
               // return cached obj
               object;
            end,
         // your additional slots and methods here
         …}
```

```
// create new mock entry w/ no cached frame
local m := NewMockEntry(myHandler, nil);
// referencing m doesn't create cached frame
local x := m;
// either statement could invoke myHandler:EntryAccess()
local a := x.foo;
local b := m.foo;
```

To almost all of the system, the mock entry appears to be a normal soup entry; for example:

- `m.foo` evaluates to `'bar`
- `ClassOf(m)` is `'frame`
- `m.baz := 42` adds a slot to the `handler.object` frame and this modified frame is returned the next time the mock entry is accessed.

Only the `IsMockEntry` global function can determine that `m` is a mock entry, rather than a soup entry. Note that the `IsSoupEntry` function returns `true` for both mock entries and normal soup entries.

Mock Stores, Mock Soups, and Mock Cursors

The current implementation of the Newton object system provides only mock entries; you must implement appropriate mock cursors, mock soups, and mock stores as required.

The **mock store** is a frame you supply which responds appropriately to all the messages that might normally be sent to a store object. For example, when the mock store's `GetSoup` method is invoked, it should return a mock soup.

The **mock soup** is a frame you supply which responds appropriately to all the messages that might normally be sent to a soup object. For example, when the mock soup's `Query` method is called, the mock soup should return a mock cursor. Mock soups cannot participate in union soups; however, you can implement your own mock union soup objects that manage the interaction of your mock soups with normal soups or union soups.

A **mock cursor** is a frame you supply that can respond appropriately to all the messages that might normally be sent to a cursor object. For example, when the mock cursor's `Entry` method is invoked, it should return a mock entry.

Using Special-Purpose Data Storage Objects

This section describes how to use entry aliases, virtual binary objects (VBOs), store parts, and mock entries. This section presumes understanding of the conceptual material presented in preceding sections.

Using Entry Aliases

This section describes how to create entry aliases, how to save them, and how to resolve them.

Aliases can be created for any entry that resides in a soup or union soup. Aliases cannot be created for mock entry objects.

You must not assume that an entry alias is valid. When the entry to which it refers is deleted or is moved to another store, an entry alias becomes invalid. Renaming a store renders invalid all aliases to entries residing on that store.

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 myCurs:= namesSoup:Query({ entireWords: true,
                              words:["Bob", "Apple"]});
// keep an alias to bob around
local bobAlias := MakeEntryAlias(myCurs:Entry());
// but get rid of the cursor
myCurs := nil;
```

To save an entry alias, simply save it in a soup entry.

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(bobAlias);
```

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 myCurs:= namesSoup:Query({ entireWords: true,
                              words:["Bob", "Apple"]});
local aBob:= myCurs:Entry();
// keep an alias to bob around
local bobAlias := MakeEntryAlias(aBob);
// the following comparison returns true
IsSameEntry(aBob, bobAlias)
```

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 myCurs:= namesSoup:Query({ entireWords: true,
                              words:["Bob", "Apple"]});
// keep an alias to bob around
local bobAlias := MakeEntryAlias(myCurs:Entry());
// the following test returns true
IsEntryAlias(bobAlias);
```

Using Virtual Binary Objects

This section describes how to use a virtual binary object to store binary data that is too large to fit into the NewtonScript heap. Topics discussed include:

- creating compressed or uncompressed VBOs
- saving VBOs in soup entries
- adding data to VBOs
- undoing changes to VBO data

In addition to the subjects discussed here, see “VBO Functions and Methods” (page 9-74) in *Newton Programmer’s Reference* for descriptions of VBO utility functions.

Creating Virtual Binary Objects

When you create a VBO, you specify whether its associated binary data is to be stored in compressed or uncompressed format. Whether you create compressed or uncompressed VBO objects is a question of space versus speed: uncompressed data provides faster access, but requires more store space than the equivalent compressed data.

The `NewVBO` and `NewCompressedVBO` store methods create virtual binary objects. Both methods require that you specify the class of the binary object to be created, as well as the store on which VBO data is to reside.

The following code fragment uses the store method `NewVBO` to create a new, uncompressed, "blank" virtual binary object on the default store:

```
// create new uncompressed VBO of size 5 KB and class 'samples
local binData := GetDefaultStore():NewVBO('samples,5000);
```

Another way to create an uncompressed VBO is to pass `nil` as the values of the *companderName* and *companderData* parameters to the `NewCompressedVBO` method, as the following code fragment shows:

```
// create new uncompressed VBO of size 5 KB and class 'samples
local binData := GetDefaultStore():NewCompressedVBO('samples, 5000,
                                                       nil, nil);
```

When you create a compressed VBO, you need to specify how the system is to expand and compress data moved to and from the store associated with the VBO. The system provides two compressor-expanders (also known as companders), which compress and expand raw binary data on demand. The *companderName* parameter to the `NewCompressedVBO` method indicates the compander to be used for that particular VBO's data.

The Lempel-Ziv 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 kind of data other than pixel maps. For more information, see the description of the `NewCompressedVBO` method (page 9-75) in *Newton Programmer's Reference*. ▲

Because both of the companders provided by the current implementation of the system initialize themselves automatically, you must always pass `nil` as the value of the *companderArgs* parameter to the `NewCompressedVBO` method.

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 new compressed VBO of size 5 KB and class 'pixMap
local binData := GetDefaultStore():NewCompressedVBO('pixMap,
                                    5000,"TPixelMapCompander", nil);
```

A VBO becomes permanent only when it is written to a soup entry, and its associated binary data always resides on the same store as the entry. Thus, when creating a VBO, it's usually best to specify that it use the same store as the soup entry into which you'll save the VBO. If you try to put the same VBO in two different soup entries, a duplicate VBO is created, even if both entries reside on the same store.

It is recommended that you enclose in a `try` block any code that writes VBO data. Store memory for VBO data is not allocated when the VBO is created; rather, it is allocated as needed to write VBO data. Thus, when writing an entry containing a VBO back to its soup, it is possible to fail due to lack of store space for new or changed VBO data, even though the VBO was created successfully.

Because the system manages store-backed VBO data transparently, calling a function such as `StuffByte` on a VBO does not necessarily cause the system to write new VBO data to the store. For similar reasons, VBOs may raise exceptions at seemingly unusual times, as the system moves VBO data to and from store memory as required to accommodate various objects' needs.

Finally, you may need to consider store space requirements when copying soup entries that hold VBOs. When moving or copying a soup entry containing a VBO, another copy of the VBO data is made by the destination soup's `Add` method because VBO data is not shared between entries.

For a short code example that creates a VBO, saves data in it, and writes the VBO to a soup, see the conclusion of the "Modifying VBO Data" section, immediately following.

Modifying VBO Data

Recall that examining or modifying any slot in a soup entry causes the entire entry to be read into the entry cache. When an entry containing a VBO is read into the entry cache, the VBO data is not read into the entry cache, but made available to the entry transparently.

Subsequently modifying the entry changes the cached data while leaving the original soup entry untouched. The changes to the entry (and any VBOs residing in it) are not saved until the entry is written back to the soup; for example, as the result of an `EntryChangedXmit` call.

Note
Because store space for VBO data is not allocated until the data is actually written, it's recommended that you enclose VBO write operations in exception handling code. ◆

To undo changes to binary data associated with a VBO that resides in a cached soup entry, call the `EntryUndoChanges` function. This function disposes of the cached soup entry and restores references to the original, untouched soup entry; it also undoes changes to VBO data referenced by the entry.

The following code fragment 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:myApp:mySig|,
        {name: "foo:myApp:mySig", 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
try
   EntryChangeXmit(anEntry, nil);
onException |evt.ex.fr.store| do
   :Notify(kNotifyAlert, "My App", "Sorry, can’t save changes.");

// listen to the sound to verify change
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:myApp:mySig") ;
        if  mySoup then
                mySoup:RemoveFromStoreXmit(nil);
end ;
UnregUnionSoup("foo:myApp:mySig", '|foo:myApp:mySig|);
```

VBOs and String Data

In most cases, you should avoid using the `&` and `&&` string-concatenation operators with VBO-based strings. These operators work by allocating a new string in the NewtonScript heap and copying data from its arguments into the new object. You can run out of heap space easily when attempting this operation with large strings.

Instead, use the `StrMunger` global function to concatenate two strings. The following code fragment appends the `str2` string to the `str1` string, increasing the size of `str1` as necessary, regardless of whether `str1` is VBO data or resident in the NewtonScript heap.

```
StrMunger(str1, MAXINT, nil, str2, 0, nil);
```

The value of `MAXINT` is `1<<29-1` or `536870911`; however, any number larger than `StrLen(str1)` works adequately.

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.

Note that other representations may provide better space efficiency or faster access to data. Store parts are useful when you wish to avoid recoding soup data in a more efficient representation, or when you need multiple indexes or some other convenience that soup-based queries provide.

Creating a Store Part

To create a store part, take the following steps using Newton Toolkit version 1.5 or greater:

- Create a new project.
- Select the Store Part radio button in the Output Settings dialog box. NTK disables all 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 file to the project. You'll add to this document the NewtonScript code that creates one or more soups to reside on the store part.

At compile time, NTK provides a global variable named `theStore`, which represents the store part (package store) you are building. Any changes made to this 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) ;

// create the soup but don't xmit at build time
local soup:=theStore:CreateSoupXmit(kSoupName,
                     kSoupIndices, nil);

// add a couple entries
soup:Add({anInteger: 1}) ;
soup:Add({anInteger: 2}) ;
```

When the package is built, NTK incorporates the store part in it.

Getting the Store Part

Store parts (also known as package stores) are made available by the `GetPackageStore` function. Package stores do not appear in the `GetStores` result array, which is reserved for normal store objects.

The `GetPackageStore` function retrieves 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 most of the messages that normal soups do, remember that store parts are read-only. Sending to a store part those messages that would normally change a soup or its store (such as `CreateSoupXmit`, `SetName` and so on) throws 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. Usually, you use the global function `GetPackageStore` and pass the name of the store part you created as its argument. Assuming the example code shown in "Creating a Store Part" on page 12-13 was used to create the store part, you could use code similar to the following example to check for the existence of the read-only soup residing on the store part:

```
local pStore := GetPackageStore(kStoreName) ;
if pStore then
   local pSoup := pStore:GetSoup(kSoupName) ;
```

Using Mock Entries

A mock entry has two parts: one is a cached frame, which the NewtonScript interpreter treats as the entry 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.

Topics discussed in this section include

- implementing the `EntryAccess` method
- creating a mock entry
- testing the validity of a mock entry
- getting entry cache data
- getting and setting mock entry handlers
- implementing additional handler methods

Implementing the EntryAccess Method

Each of your mock entry handler frames must supply an `EntryAccess` method that creates a cached frame containing the mock entry's data, installs the cached frame in the mock entry, and returns the cached frame. This method is called when the system attempts to access a cached frame that is not present.

The system passes the mock entry to your `EntryAccess` method when it is invoked. This method calls the `EntrySetCachedObject` function to install the cached frame in the mock entry and then returns the cached frame.

The following code fragment provides a simple example of an `EntryAccess` method:

```
myHandler := {
   object: {foo: 'bar},
   EntryAccess: func (mockEntry)
      begin
         // install cached frame
         EntrySetCachedObject(mockEntry, object);
         // return cached frame
         object;
      end,
   // your additional slots and methods here
   …}
```

Creating a New Mock Entry

The `NewMockEntry` global function creates a new mock entry object having a specified handler and cached frame. Your application can use this method to create a new mock entry; for example, in response to a *mockSoup*`:Add()` message.

The handler frame you pass to the `NewMockEntry` function must define an `EntryAccess` method, as described in "Implementing the EntryAccess Method" on page 12-15. The handler may also contain supporting methods or data used by the mock entry; for example, it might hold information local to a specific mock entry or information required to retrieve the mock entry's data.

Depending on your needs, you can create new mock entries with or without their corresponding cached frames. To create a mock entry with its cached frame already installed, pass both the handler and the cached frame to this function.

To create a mock entry without a cached frame, pass `nil` as the value of the *cachedObject* parameter to the `NewMockEntry` function. When a slot in the returned mock entry is accessed, the handler's `EntryAccess` method is invoked to create the cached entry if it is not present.

Testing the Validity of a Mock Entry

The `IsMockEntry` global function returns the value `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 Mock Entry Data

The `EntryCachedObject` global function returns the cached frame associated with a specified mock entry. You can call this function to retrieve the cached frame associated with a specified mock entry. For example, your handler frame's `EntryChange` method must retrieve the cached frame in order to write it back to a mock soup.

Changing the Mock Entry's Handler

The `EntrySetHandler` function is a special-purpose function that you can use to replace a mock entry's handler. For example, you can use this function to install a handler that implements debug versions of methods present in the mock entry's original handler frame. Such methods might include breakpoints and print statements that would not be present in the commercial version of an application.

Getting the Mock Entry's Handler

The system supplies the `EntryHandler` function for debugging purposes. The `EntryHandler` function returns a reference to the handler frame associated with the mock entry specified by the value of the *mockEntry* parameter.

Implementing Additional Handler Methods

Your handler needs to provide additional methods that are the mock entry counterparts to system-supplied entry functions, such as `EntryUndoChangesXmit`, and others. For a list of suggested methods that your handler may implement, see "Application-Defined Mock Entry Handler Methods" on page 12-19.

Summary of Special-Purpose Data Storage Objects

This section summarizes data structures, objects, methods and global functions used by Newton devices for specialized data storage purposes.

Data Structures

Package Reference Information Frame

```
{
size: nBytes, // pkg's uncompressed size in bytes
store: aStore, // store on which pkg resides
title: string,// user-visible package name string
version: int, // version number
timeStamp: int,// date and time pkg was loaded
creationDate: int, // date pkg created
copyProtection: value, Non-nil means protected.
dispatchOnly: value, // Non-nil means dispatch-only pkg.
copyright: string, // copyright information string
compressed:value, // Non-nil value means pkg is compressed
cmprsdSz: int,// compressed size of pkg in bytes
numParts: int, // number of parts in pkg
parts: [p1, p2, … pN], // parts comprising this package.
partTypes:[sym1, sym2, … symN] // parallel to parts array.
```

Functions and Methods

Packages

```
GetPackageNames(store)
GetPackages()
GetPkgRef(name, store)
GetPkgRefInfo(pkgRef)
IsValid(obj)
IsPackage(obj)
IsPackageActive(pkgRef)
MarkPackageBusy(pkgRef, appName, reason)
MarkPackageNotBusy(pkgRef)
ObjectPkgRef(obj)
```

`SafeFreezePackage(`*pkgRef*`)`
`SafeMovePackage(`*pkgRef*`, `*destStore*`)`
`SafeRemovePackage(`*pkgRef*`)`
store`:SuckPackageFromBinary(`*binary, paramFrame*`)`
store`:SuckPackageFromEndpoint(`*endPoint, paramFrame*`)`
`ThawPackage(`*pkgRef*`)`

Store Parts (Package Stores)

`GetPackageStore(`*name*`)`
`GetPackageStores()`

Entry Aliases

`IsEntryAlias(`*object*`)`
`MakeEntryAlias(`*entry*`)`
`ResolveEntryAlias(`*alias*`)`
`IsSameEntry(`*entryOralias1*`, `*entryOralias2*`)`

Virtual Binary Objects (VBOs)

store`:NewVBO(`*class*`, `*size*`)`
store`:NewCompressedVBO(`*class*`, `*size*`, `*companderName*`, `*companderArgs*`)`
`IsVBO(`*vbo*`)`
`GetVBOStore(`*vbo*`)`
`GetVBOStoredSize(`*vbo*`)`
`GetVBOCompander(`*vbo*`)`

Mock Entries

`EntryCachedObject(`*mockEntry*`)`
`EntryHandler(`*mockEntry*`)`
`EntrySetCachedObject(`*mockEntry*`, `*newCachedObj*`)`
`EntrySetHandler(`*mockEntry*`, `*newHandler*`)`
`IsMockEntry(`*object*`)`
`NewMockEntry(`*handler*`, `*cachedObj*`)`
`NewWeakArray(`*length*`)`

Application-Defined Mock Entry Handler Methods

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*)

C H A P T E R 1 3

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 them
- drawing methods and functions used to perform specific tasks
- drawing classes and protos that operate on graphics and drawing methods and functions

About Drawing

The drawing interface provides a number of functions, methods, and protos that allow you to create graphic 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" (page 3-26) 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 Newton OS 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 the bottom of the screen from the origin. For additional information on the Newton coordinate system see "Coordinate System" (page 3-6).

Shape-Based Graphics

Newton system software provides functions for drawing primitive graphic 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:

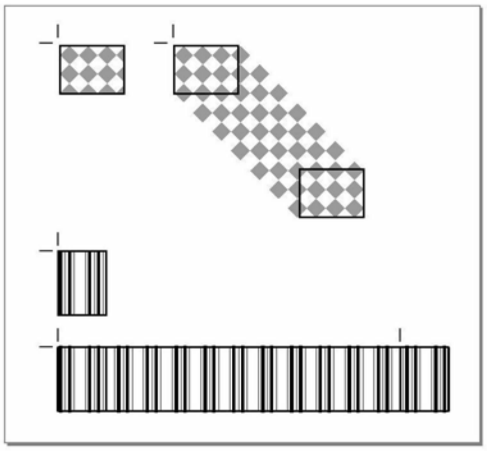
- lines
- rectangles
- rounded rectangles
- ovals (including circles)
- polygons
- wedges and arcs
- regions
- text
- pictures
- bitmaps

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" (page 13-12) and "Transforming a Shape" (page 13-13) for more information.

Each type of shape is described in the following pages.

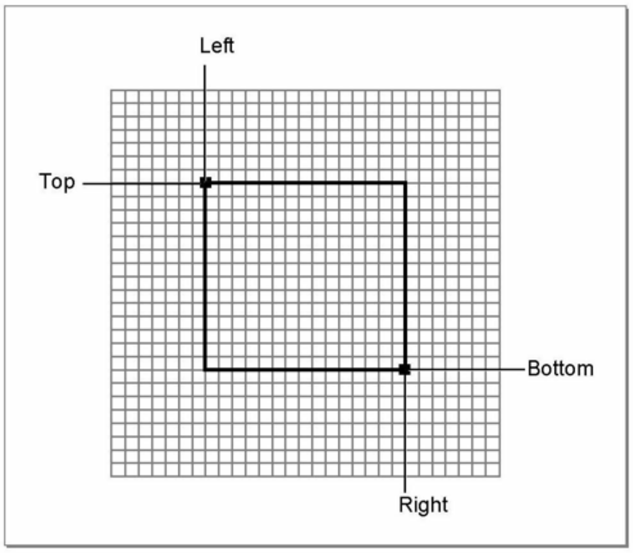
A **line** is defined by two points: the current *x* and *y* location of the graphics pen and the *x* and *y* location of its destination. The pen hangs below the right of the defining points, as shown in Figure 13-1, where two lines are drawn with two different pen sizes.

Figure 13-1 A line drawn with different bit patterns and pen sizes



A **rectangle** can be defined by two points—its top-left and bottom-right corners, as shown in Figure 13-2, 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 graphic entities, and to specify the locations and sizes for various graphics operations.

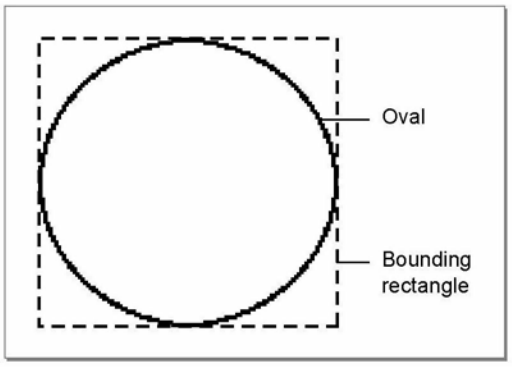
Figure 13-2 A rectangle



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.

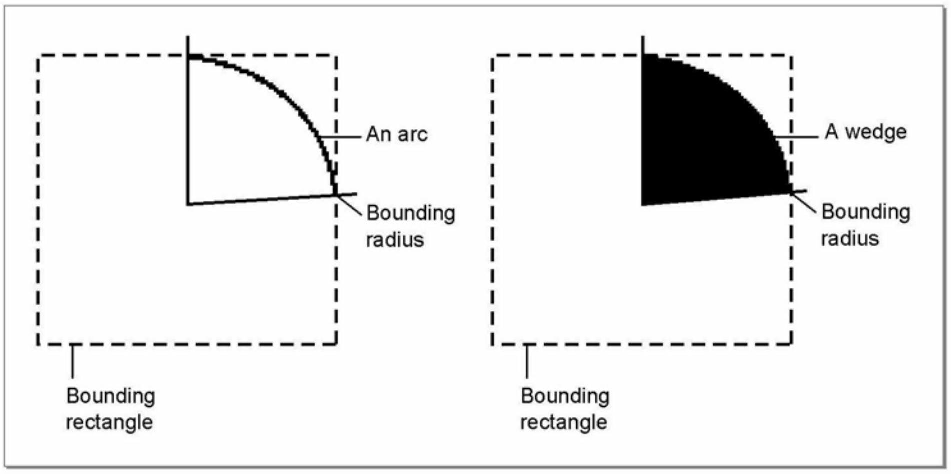
An **oval** is a circular or elliptical shape defined by the bounding rectangle that encloses it. If the bounding rectangle is a square (that is, has equal width and height), the oval is a circle, as shown in Figure 13-3.

Figure 13-3 An oval



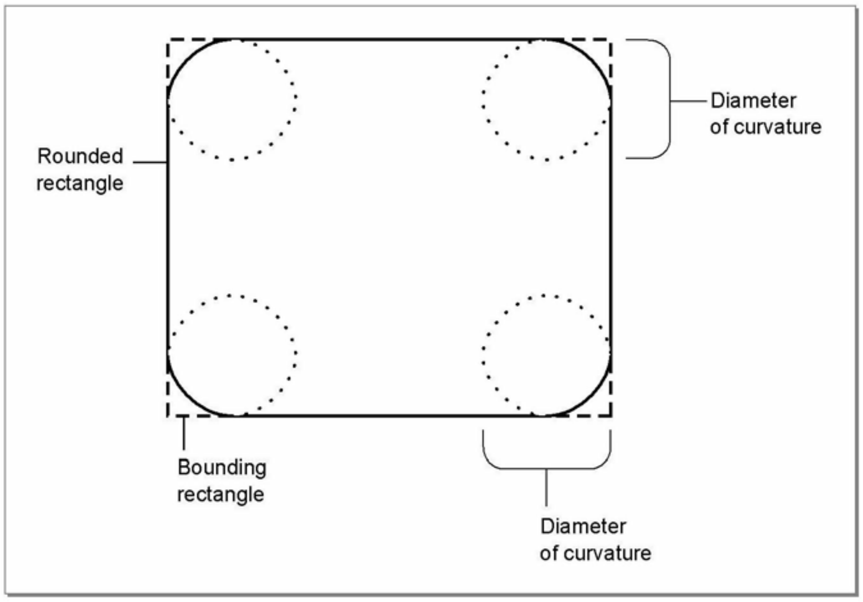
An **arc** is a portion of the circumference of an oval bounded by a pair of radii joining at the oval's center; a wedge includes part of the oval's interior. Arcs and wedges are defined by the bounding rectangle that encloses the oval, along with a pair of angles marking the positions of the bounding radii, as shown in Figure 13-4.

Figure 13-4 An arc and a wedge



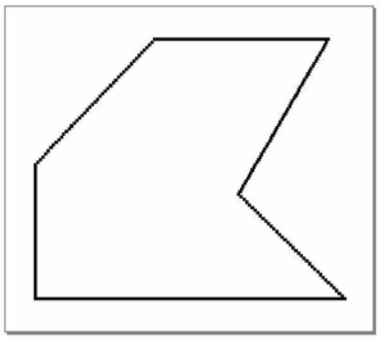
A **rounded rectangle** is a rectangle with rounded corners. The figure is defined by the rectangle itself, along with the width and height of the ovals forming the corners (called the diameters of curvature), as shown in Figure 13-5. The corner width and corner height are limited to the width and height of the rectangle itself; if they are larger, the rounded rectangle becomes an oval.

Figure 13-5 A rounded rectangle



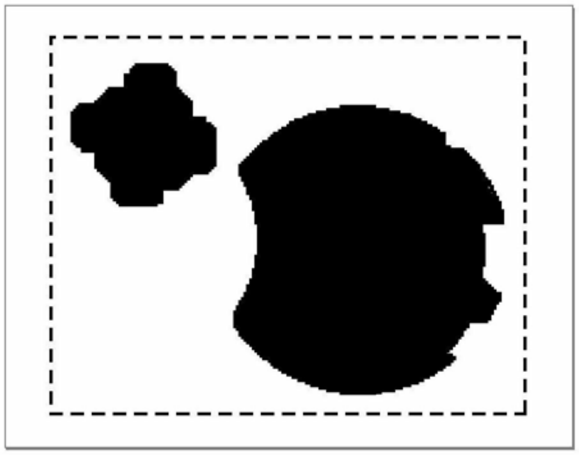
A **polygon** is defined by a sequence of points representing the polygon's vertices, connected by straight lines from one point to the next. You define a polygon by specifying an array of *x* and *y* locations in which to draw lines and passing it as a parameter to `MakePolygon`. Figure 13-6 shows an example of a polygon.

Figure 13-6 A polygon



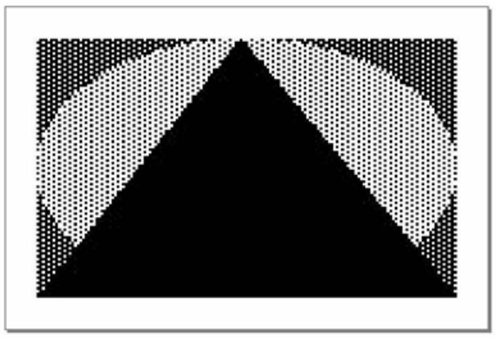
A **region** is an arbitrary area or set of areas, the outline of which is one or more closed loops. One of drawing's most powerful capabilities is the ability to work with regions of arbitrary size, shape, and complexity. You define a region by drawing its boundary with drawing functions. The boundary 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 13-7, the region consists of two unconnected areas.

Figure 13-7 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 13-8 shows an example of a picture containing a rectangle, an oval, and a triangle.

Figure 13-8 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" (page 10-1) in the *Newton Programmer's Reference* for complete details.

Drawing Compatibility

The following new functionality has been added for Newton OS 2.0. For complete details on the new drawing functions, see the “Drawing and Graphics Reference” in the *Newton Programmer’s Reference.*

New Functions

The following functions have been added:

- `GetShapeInfo`—returns a frame containing slots of interest for the shape.
- `DrawIntoBitmap`—draws shapes into a bitmap in the same way that the `DrawShape` method draws shapes into a view.
- `MakeBitmap`—returns a blank (white) bitmap shape of the specified size.
- `MungeBitmap`—performs various destructive bitmap operations such as rotating or flipping the bitmap.
- `ViewIntoBitmap`—provides a screen-capture capability, writing a portion of the specified view into the specified bitmap.

New Style Attribute Slots

Version 2.0 of Newton system software supports two new slots in the style frame: `clipping` and the `transform`.

Changes to Bitmaps

Previous versions of Newton system software treated bitmaps 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 such operations 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.

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.

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.

Using the Drawing Interface

This section describes how to use the drawing interface to perform specific tasks. See "Drawing and Graphics Reference" (page 10-1) in the *Newton Programmer's Reference* for descriptions of the functions and methods discussed in this section.

How to Draw

Drawing on the Newton screen is a two-part process. You first create a shape object by calling one or more graphics functions, such as `MakeRect`, `MakeLine`, 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 with style frames to the `DrawShape` 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. "Using Nested Arrays of Shapes" (page 13-10), discusses the use of arrays of shapes in more detail.

Responding to the ViewDrawScript Message

When the system draws a view, it sends a `ViewDrawScript` message 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 it the `Dirty` message. 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 make the update area redraw before the next event loop cycle, you must call the `RefreshViews` function after sending the `Dirty` message.

Drawing Immediately

If you want to draw in a view at times other than when the view is being opened or redrawn automatically, you can execute drawing code outside of the `ViewDrawScript` method by using `DoDrawing`. 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 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

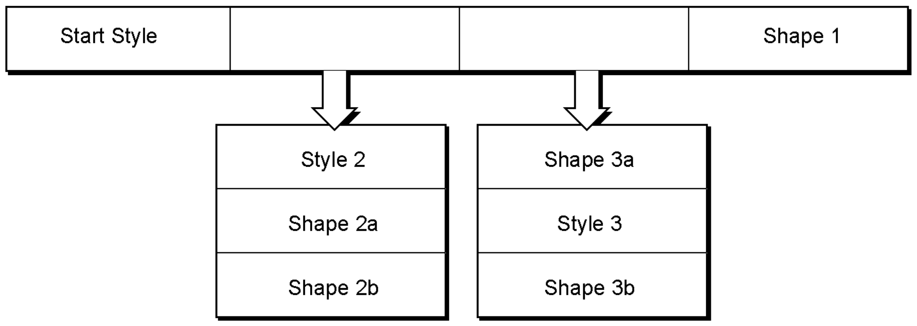
The `DrawShape` method can draw multiple shapes when passed an array of shapes as its argument. Style frames may be included in the shape array to change the drawing style used to image subsequent elements of the array. Each element of the array can itself be an array as well; this section refers to 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 topmost 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 13-9.

Figure 13-9 Example of nested shape arrays



If the nested shape array depicted in Figure 13-9 were passed to the `DrawShape` function, the results summarized in Table 13-1 would occur.

Table 13-1 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 13-9 *startStyle* has a transform of 10,10 and Style 3 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` 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` 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 Protos

Four view classes and three protos, which you can use to create your own templates, are built into the system. The view classes include:

- `clPolygonView` —displays polygons or ink, or accepts graphic or ink input.
- `clPictureView`—displays a bitmap or picture object shape.
- `clEditView`—edits views that can accept both text and graphic user input.
- `clRemoteView`—displays a scaled image of another view.

The protos include:

- `protoImageView`—provides a view in which you can display, magnify, scroll, and annotate images.
- `protoThumbnail`—is 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`—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 and Ink

Use the `clPolygonView` class to display polygons and 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 Bitmaps, Pictures, and Graphics Shapes

You can use a view of the `clPictureView` class to display a bitmap, picture, or graphic shape (polygon). The icon slot in this view can contain a bitmap, a picture object, or a graphic shape.

Displaying Pictures in a clEditView

Use the `clEditView` view class to display and accept text and graphic data in a view. Views of the `clEditView` class contain no data directly; instead, they have child views that contain the individual data items. Pictures are contained in child views of the class `clPictureView`. For details on displaying text, see "Using Views and Protos for Text Input and Display" (page 8-6).

To add a picture to a `clEditView`, you need to create an appropriate template of the `clPictureView` class, add the template to the `viewChildren` array, and then open the view or call `RedoChildren`. You can also use the `AddView` method to add the picture to an existing view, and then mark the view as dirty so that it will be redrawn.

The template holding the PICT items must contain the following slots:

- `viewStationery`—which must have the symbol `'pict`
- `viewBounds`—which is a bounds frame; for example,

  ```
  RelBounds(0,0,40,40)
  ```
- `icon`—which is a bitmap frame, a picture object, or a graphic shape

Displaying Scaled Images of Other Views

Use the `clRemoteView` view class to display a scaled image of another view. This class 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 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` and `ArrayToPoints` to translate points data between a polygon shape (`'polygonShape`) and a NewtonScript array.

Finding Points Within a Shape

Use the `HitShape` function 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 in *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. This flexible treatment of bitmaps allows you to use them as offscreen buffers and 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. Then create a shape with the `MakeShape` function. `DrawIntoBitmap` 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.

The following example shows how to draw a bitmap. It creates a bitmap by drawing a shape and 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: vfBlack}, 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` 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 make this method scale the bitmap it draws. If you want to scale the output bitmap without changing the bounds of the original, call `ScaleShape` 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), all items in your package are compressed in small (approximately 1 KB) pages, rather than object by object.

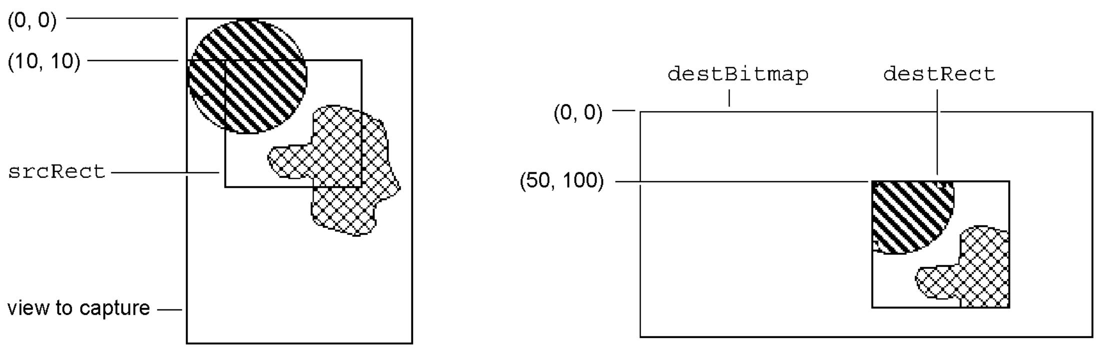
You can use the NTK compile-time function `GetNamedResource` to get a Macintosh PICT resource that can be drawn on the Newton in a view of the `clPictureView` class. PICT resources are generally smaller than bitmap frames because each bitmap within the PICT resource contains compressed bitmap data.

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 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 13-10 shows the relationships between the view to be captured, the source rectangle, the destination bitmap, and the destination rectangle.

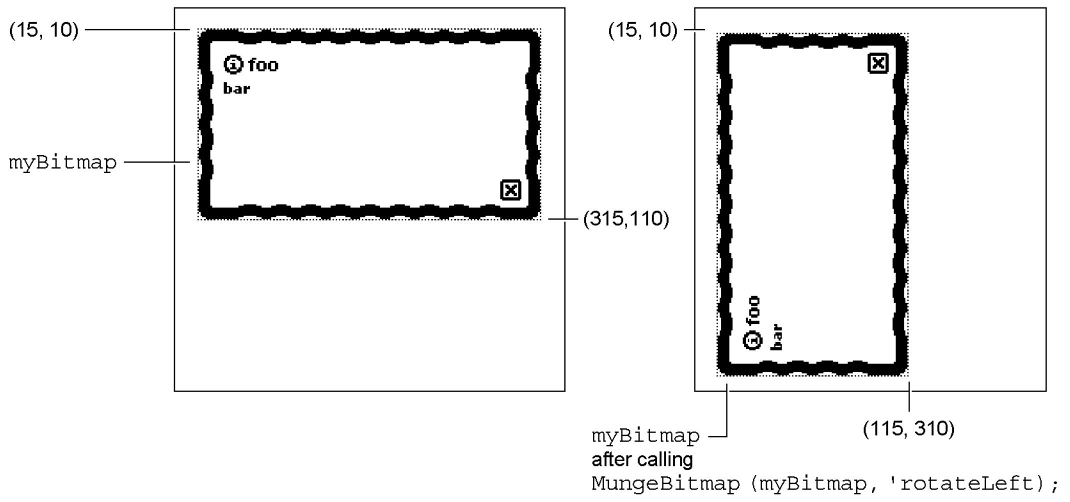
Figure 13-10 Example of `ViewIntoBitmap` method



Rotating or Flipping a Bitmap

Use the `MungeBitmap` function (page 10-22) 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 13-11 illustrates how the `MungeBitmap` function works. See "Using Bitmaps" (page 13-17) for a code example.

Figure 13-11 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. 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` or `MakePict`.

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 embed a style frame in an array of shapes:

```
:DrawShape ([{font:        ...}, myText, shape ], nil);
```

You can also use MakePict:

```
myText := MakePict([{penpattern: 0, font: ...}, rect,
   {font: ...}, txtshape], {font: ...});
```

You can set the font in locations 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 that there are two types of pictures: bitmaps (a frame with `bits`, a `bounds`, and `mask` slots) and Format 1 PICTs (binary objects of class picture). `clPictureView` 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);
or
   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 a `ViewDrawScript` or a `ViewClickScript` rather than using smaller views with an individual `ViewClickScript`. This is especially true of a view that consists of regular smaller views.

Summary of Drawing

Data Structure

Style Frame

```
aStyle := {
transferMode : constant, // transfer mode for the pen
penSize : integer, // size of the pen in pixels
penPattern : constant, // the pen pattern
fillPattern : constant, // the fill pattern
font : string, // font to use for drawing text
justification : symbol, // alignment of text
clipping : shape, region, or array of shapes, // specifies clipping
transform : array, // offsets or scales the shape
```

View Classes

clPolygonView

```
clPolygonView := {
viewbounds : int, // size of view and location
points : struct, // binary data structure containing
                  polygon data
ink : struct, // binary data structure containing ink data
viewFlags : const, // controls the recognition behavior of
                  the view
viewFormat : const, // controls appearance of the view
```

clPictureView

```
clPictureView := {
icon : bitmap, graphic shape, picture, // icon to display
viewBounds: int, // size and location of the view
viewFlags : const, // controls the recognition behavior of
                  the view
viewFormat : const, // controls appearance of the view
```

clRemoteView

```
clRemoteView := {
stepChildren : int, // specifies a single view
viewBounds: int, // size and location of the view
viewFlags : const, // controls the recognition behavior of
                  the view
viewFormat : const, // controls appearance of the view
```

Protos

protoImageView

```
aProtoImageView := {
_proto: ProtoImageView,
Image : shape,
Annotations : array,
scalingInfo : frame,
viewBounds : boundsFrame,
viewJustify: justificationFlags,
viewFormat : formatFlags,
zoomStops : array,
dragCorridor : integer,
grabbyHand : shape,
myImageView:penDown : function, // drags image
myImageView:ScalingInfoChanged : function, // called when
                                  scaling changes
myImageView:Setup : function, // initializes the image
myImageView:OpenImage : function, // opens image
myImageView:ToggleImage : function, // closes image
myImageView:GetScalingInfo : function, // returns scaling
                                information
myImageView:HasAnnotations : function, // returns annotation
                                information
myImageView:GetAnnotations : function, // returns an array of
                                 views
myImageView:SetAnnotationMode : function, // sets display
                                   behavior
myImageView:GetAnnotationMode : function, // returns a symbol
myImageView:TargetChanged : function, // called when
                                annotation is changed
myImageView:CanScroll : function, // returns scrolling
                           information
myImageView:ScrollTo : function, // scrolls an image
myImageView:ScrollBy : function, // scrolls an image
myImageView:ZoomBy : function, // makes an image larger
                                 or smaller
myImageView:ZoomTo : function, // changes the size of
                                 the image
myImageView:CanZoomBy : function, // changes the size of
                                    the image
myImageView: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
myThumbnail:Setup : function, // prepares thumbnail
myThumbnail:OpenThumbnail : function, // opens thumbnail
myThumbnail:ToggleThumbnail : function, // opens or closes
                                          thumbnail
myThumbnail:Update : function, // renders thumbnail view
myThumbnail:GetScalingInfo : function, // returns scaling
                                         information
myThumbnail:PrepareToScroll : function, // prepares for
                                          scrolling
myThumbnail:ScrollTo : function, // scrolls a view
myThumbnail: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
...
}
```

Functions and Methods

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*)
`MakeTextLines`(*string, bounds, lineheight, font*)
`TextBox`(*text, fontFrame, bounds*)

Shape Operation Functions and Methods

`GetShapeInfo`(*shape*)
view:`DrawShape` (*shape, styleFrame*)
`OffsetShape` (*shape, deltaH, deltaV*)
`ScaleShape` (*shape, srcRect, dstRect*)
`ShapeBounds` (*shape*)
`InvertRect`(*left, top, right, bottom*)
`InsetRect`(*rect, deltax, deltay*)

`IsPtInRect(`*x*`, `*y*`, `*bounds*`)`
`FitToBox(`*sourceBox*`, `*boundingBox*`, `*justify*`)`
`OffsetRect(`*rect*`, `*deltaX*`, `*deltaY*`)`
`SectRect(`*rect1*`, `*rect2*`)`
`UnionRect(`*rect1*`, `*rect2*`)`
`RectsOverlap(`*rect1*`, `*rect2*`)`

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 14

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, describing:

- sounds, sound channels, and sound frames
- 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. See “Sound Resources” (page 11-10) in the *Newton Programmer’s Reference* for complete details. You can also use the Newton Toolkit (NTK) to create custom sounds on desktop computers.

A Newton sound is represented as a sound frame. You can easily associate any sound with a certain events or play sound frames programmatically. The system allows you to play sound frames 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 sampled sounds starting from any point within the data. For more advanced uses of sound you can open a sound channel which allows multiple channels to play simultaneously, or multiple sounds to be queued in a single channel. You use a sound channel by sending messages to a sound channel frame. 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.

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 "Sound Resources" (page 11-10) in the *Newton Programmer's Reference* for complete details.

Sounds for Predefined Events

All views recognize a set of predefined slot names that specify sounds to accompany certain system events. To add a ROM-based sound to one of these events, 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`	The sound is played when the view is shown.
`hideSound`	The sound is played when the view is hidden.
`scrollUpSound`	The sound is played when the view receives a `ViewScrollUpScript` message.
`scrollDownSound`	The 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 Data Structures

Three data structures are related to sounds: a sound frame, a sound result frame, and a `protoSoundChannel`.

A sound frame stores sound sample data and 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 32 KB in size. See "Sound Data Structures" (page 11-1) in the *Newton Programmer's Reference*, for a complete list of slots required by for both types of frames.

The `protoSoundChannel` provides methods that implement pause and playback of sounds and completion callbacks. It also provides query methods that return whether the sound is running or paused.

If you are providing custom sounds, you can store them as virtual binary objects. An example of storing a sound as a VBO is given in Chapter 11, "Data Storage and Retrieval.".

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 Newton 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.

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 and Using Custom Sound Frames

The following information applies to the Mac 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 Mac OS `'snd '` resource data. This section summarizes the main steps required to create custom sound frames from Mac OS `'snd '` resources in NTK; for a complete discussion of this material, see the *Newton Toolkit User's Guide*.

Follow these 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 `PlaySoundSync` function. These run-time functions play sound from anywhere in your code.

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 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.

Playing Sound

Newton system software plays sound in two ways. The first is to use the global sound functions `PlaySoundAtVolume` or `PlaySoundIrregardless`. 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 having many sounds available on short notice or playing multiple sounds at the same time.

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` 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();
```

Playing Sounds

Once you create the sound channel, you can use any of the following methods to control the sound.

`Schedule`—queues the sound for play.

`Start`—starts playing the sounds in the order that they were scheduled.

`Stop`—stops all scheduled sounds including currently playing sounds, if any.

`Pause`—temporarily stops the current playback process in the specified sound channel.

`IsPaused`—checks to see if the sound channel is paused.

`IsActive`—checks to see if the sound channel is playing.

Deleting the Sound Channel

When finished with the sound channel, you need to dispose of it by sending the `Close` 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 Programmatically

You can use any of the global functions to play sound programmatically. 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 asynchronously, the playback can be intermixed with other tasks because the system does not wait for the sound to finish before beginning another task (such as updating the user interface, allowing user feedback; for example with buttons, or playing a subsequent sound).

When playback must be allowed to complete, use the `PlaySoundSync`, `PlaySoundAtVolume`, or `PlaySoundIrregardless` 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 NewtonScript interpreter can do nothing else until it completes synchronous playback.

Both approaches have benefits and drawbacks: synchronous playback can block other NewtonScript code from running 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.

Differences Between Synchronous Asynchronous Playback

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
      PlaySoundSync(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 can call the sound chip so fast 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 `PlaySoundSync` function. In fact, it's likely that you won't hear twenty sounds in the asynchronous playback demo; the calls occur faster than the Newton sound chip can respond.

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 interested developers, along with the 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` view method and the `RawDial` global function to generate telephone dialing tones from NewtonScript. It is strongly recommended that you use these functions rather than attempt 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.

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 generally works best is to take an 11 kHz sound and play it at some higher rate. Of course, 22 kHz sound resources cannot be played at any higher sampling rate.

You can experiment with pitch shifting by playing sounds in the Inspector using the PlaySoundSync function. You can use any of the ROM sounds or your own custom sounds. The following example shows 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);
PlaySoundSync(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 want to manipulate sample data on the Newton only 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, 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.

```
// 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;
```

A better solution is to provide the backwards sound as a resource that can be played just like any other sound; a number of sound editors are available to create such a resource on a desktop computer.

Summary of Sound

Data Structures

SndFrame Structure

```
mySndFrame := {
_proto: mySndFrame,
sndFrameType : symbol, // specifies format
samples : frame, // contains sampled binary data
samplingRate : integer/floating point, // specifies playback rate
compressionType : integer, // indicates no compression
dataType : integer, // indicates size of sample in bits
start : integer, // index of first sample to play
count : integer, // number of samples to play
loops : integer, // time to repeat sound
Callback : function, // indicates the state of the sound
```

SndResult Structure

```
mySndResult := {
_proto: mySndResult,
sound : integer, // reference to soundFrame that was paused
index : function, // index of sample that was paused/stopped
```

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, // pauses playback
IsPaused : function, // checks if sound channel is paused
IsActive : function, // checks if  sound channel is active
...
}
```

Functions and Methods

```
view:Dial(numberString, where)
GetVolume()
PlaySoundSync(soundFrameRef)
RawDial(number, where)
SetVolume(volume)
PlaySoundAtVolume(soundFrameRef, volume)
PlaySoundIrregardless(soundFrameRef)
PlaySoundIrregardlessAtVolume(soundFrameRef, volume)
PlaySoundEffect(soundFrameRef, volume, type)
Clicker()
```

Sound Resources

```
ROM_alarmWakeup // alarm sound
ROM_click // click sound
ROM_crumple // paper crumpling sound
ROM_drawerClose // drawer closing sound
ROM_drawerOpen // drawer opening sound
ROM_flip // page flipping sound
ROM_funBeep // trill sound
ROM_hiliteSound // squeek sound
ROM_plinkBeep // xylo sound
ROM_simpleBeep // bell sound
ROM_wakeupBeep // power on sound
ROM_plunk // paper hitting trash sound
ROM_poof // puff of air sound
```

C H A P T E R 1 5

Filing

This chapter describes how your application can support the Filing service. This service allows the user to

- associate data items with folders displayed by the user interface
- create, edit, or delete folders at will
- specify the store on which a soup entry is to reside when it is filed

Before reading this chapter, you should understand the use of views to image data, as explained in Chapter 3, "Views." You should also understand the contents of Chapter 11, "Data Storage and Retrieval," which describes the soup-based storage model on which the Filing service is based. If your application does not save data as soup entries, you need to implement mock entries and related objects to provide soup-like access to your data, as described in Chapter 12, "Special-Purpose Objects for Data Storage and Retrieval."

A related service called the Soupervisor allows the user to file or move all entries in a specified soup at once. For more information, see the description of this service in Chapter 19, "Built-in Applications and System Data."

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 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 Filing service also allows the user to move entries to a specified store when they are filed.

The currently displayed application data to be filed is referred to as the **target** of the filing action. The target may consist of multiple data items; for example, most applications provide an overview view from which the user can file and move multiple items simultaneously.

Your application must provide a **target view** that can manipulate the target. The target view is usually the same view that images the target data. Although the application base view is often an appropriate target view, it may not be under all circumstances. For example, each of these common situations has specialized targeting needs:

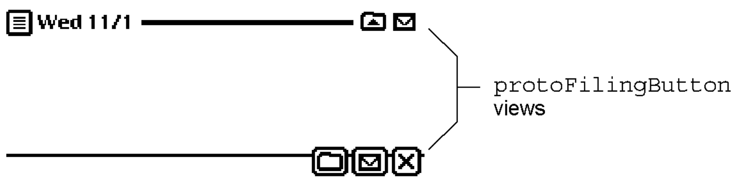
- Most applications allow the user to file and move multiple data items from within an overview view. In this situation, the target may consist of multiple items, and the overview view is usually the appropriate target view.
- Applications that display more than one data item at a time, such as the built-in Notes application, may need to specify which of several equal child views is actually the target.
- You might want the target view to be a floating window when one is present, and the application's base view at all other times.

You can override the system-supplied `GetTargetInfo` method to vary the target and target view according to circumstances.

Applications with less-elaborate targeting needs can use the default `GetTargetInfo` method supplied by the system. To use the default `GetTargetInfo` 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.

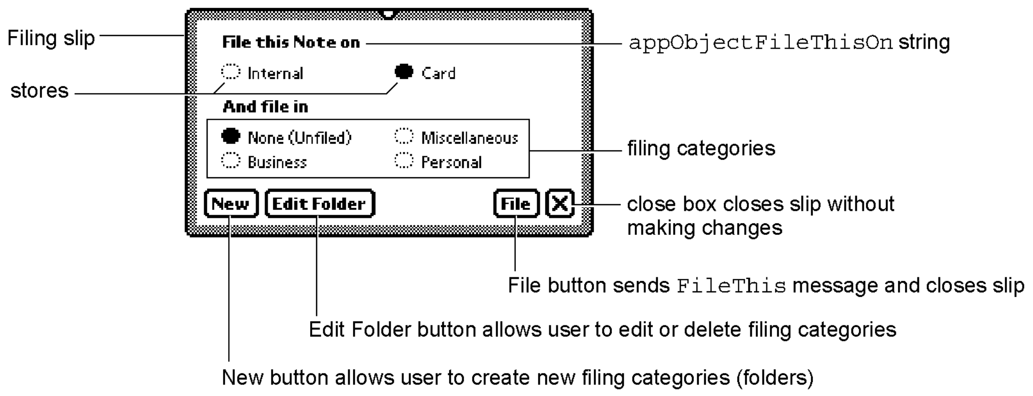
To file the target, the user taps a file folder button you provide. This view, which is based on the `protoFilingButton` system prototype, looks like a button with a picture of a paper file folder on it. Figure 15-1 provides two examples of views based on the `protoFilingButton` system prototype.

Figure 15-1 Two examples of filing button views



When the user taps the `protoFilingButton` view, it displays the Filing slip shown in Figure 15-2.

Figure 15-2 Filing slip


Filing slip
stores
File this Note on
Internal
Card
And file in
None (Unfiled)
Miscellaneous
Business
Personal
New
Edit Folder
File
appObjectFileThisOn string
filing categories
close box closes slip without making changes
File button sends FileThis message and closes slip
Edit Folder button allows user to edit or delete filing categories
New button allows user to create new filing categories (folders)

The Filing slip displays a set of categories in which the target can be filed. These filing categories include all folders available to the application that displayed the Filing slip, as well as the Unfiled category. This slip also provides a close box that dismisses it without making any changes.

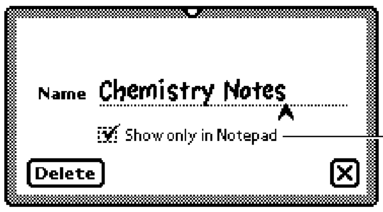
The user can create new folders and edit the names of existing ones by means of buttons the Filing slip provides for this purpose. When a new folder is created, it may be designated as visible only from within a specified application; such a folder is said to be a **local folder** belonging to the application that created it. Any folder not created as a local folder is visible from all applications, and is called a **global folder.** The system permits the creation of a maximum of twelve local folders per application and twelve global folders system-wide. The system does not permit the creation of local and global folders having the same name.

Most applications allow the user to create and view any combination of local and global folders; however, you can suppress the display of either kind of folder if necessary. For example, the Extras Drawer displays only its own filing categories because those created by other applications are not likely to be useful for organizing the display of application packages, soups, and so on.

When the user adds, removes, or edits filing categories, the folder change notification service executes your previously registered callback function to respond appropriately to these changes. You use the `RegFolderChanged` global function to register a callback function with this service. The companion function `UnRegFolderChanged` unregisters a specified callback function.

Filing and other system services display user messages containing a string that is the user-visible name of your application. For example, this string is used to complete the text displayed when the user creates a local folder. You need to create in your application's base view an `appName` slot that holds this string. Figure 15-3 depicts the text displayed when the user creates a folder that is local to the Notepad application.

Figure 15-3 Creating a local folder



The system determines whether to display global or local folders by testing the values of optional slots 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.

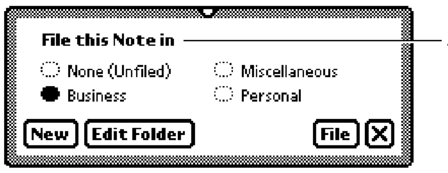
▲ **WARNING**
The `localFoldersOnly` and `globalFoldersOnly` must not both hold non-`nil` values at the same time. ▲

Your target view can provide an optional `doCardRouting` slot to control the display of the buttons that specify the store on which to file the target. When an external store is available and the value of the `doCardRouting` slot is `true`, the Filing slip includes 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 15-2 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 15-4.

Figure 15-4 Filing slip without external store

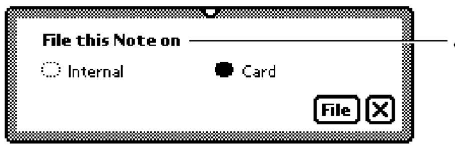


This simplified version of the Filing slip does not include the buttons that allow the user to choose a store. Note that the string labelling the group of buttons representing filing categories differs slightly in this version of the Filing slip. This string is provided by an `appObjectFileThisIn` slot that your application's base view supplies.

Regardless of 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 15-4. 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 15-5 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 15-5 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 the following:

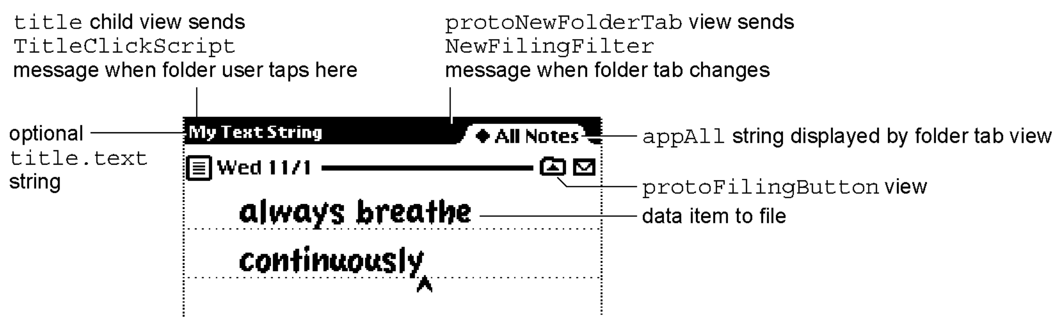
- moving its soup entry to a different store
- redrawing the current view
- setting the target's `labels` slot to its new value
- performing any additional tasks that are appropriate

Your application must provide a folder tab view that

- indicates the filing category of currently displayed data.
- allows the user to choose a new filing category to display

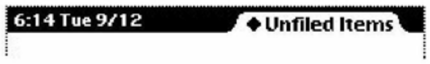
The system provides `protoNewFolderTab` and `protoClockFolderTab` system prototypes you can use to create your folder tab view. Views based on either proto can display a title text string in the area to the left of the folder tab. The `protoNewFolderTab` view displays a text string that you may supply optionally, as shown in Figure 15-6.

Figure 15-6 A `protoNewFolderTab` view


title child view sends TitleClickScript message when folder user taps here
protoNewFolderTab view sends NewFilingFilter message when folder tab changes
optional title.text string
appAll string displayed by folder tab view
protoFilingButton view
data item to file
My Text String
All Notes
Wed 11/1
always breathe
continuously

The protoClockFolderTab is a variation on protoNewFolderTab that displays the current time as its title text. Do not attempt to replace this text; if you want to display your own title text in a folder tab view, use a protoNewFolderTab view rather than a protoClockFolderTab view. Figure 15-7 depicts a typical protoClockFolderTab view.

Figure 15-7 A protoClockFolderTab view



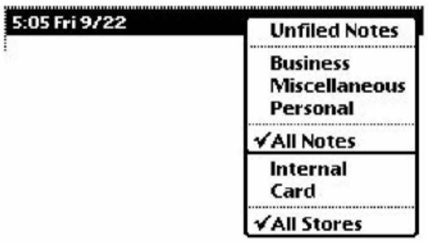
Either kind of folder tab view sends a TitleClickScript message to your application when the user taps its title text. The protoClockFolderTab view's default TitleClickScript method opens the built-in Clock application. The protoNewFolderTab view provides no default TitleClickScript method. Your folder tab view can provide its own TitleClickScript method to customize the action it takes in response to a tap on its title text. Your titleClickScript method accepts no arguments.

Both kinds of 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." This string appears in the folder tab view when the application displays data items not associated with any filing category. This string is also displayed in the picker that opens when the user taps the filing tab.

Both kinds of folder tab views also rely on the use of an appAll slot that you provide in your application's base view. This slot contains the full text of the string "All *items*" in which *items* is the plural form of the target your application manipulates; for example, "All Notes." This string appears in the folder tab view when the application displays all its data items (including those that are not filed). This string is also displayed in the picker that opens when the user taps the folder tab.

Tapping the folder tab displays a picker 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 check mark 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 picker, 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 15-8 shows the folder tab picker in a view based on the `protoClockFolderTab` proto.

Figure 15-8 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 picker, 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 necessary to display the appropriate data, such as redrawing views affected by the new filter value.

You can use the `RegFolderChanged` function to register your own callback function to be executed when the user adds, deletes, or edits folder names. You cannot respond adequately to these kinds of changes from within your `NewFilingFilter` or `FileThis` methods alone.

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 to 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 visible only to 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 are filed 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 string 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. They are replaced by the `FileThis` method and the folder change notification mechanism. If your application supplies 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; your application should supply a `NewFilingFilter` method instead. 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 of filing filter. If your application supplies a `NewFilingFilter` method, the system does not send `FilterChanged` messages to your application.

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. However, 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

To support the Filing service, your application must

- provide three views (a folder tab view, a filing button view, and a view that images the filing target)
- respond to two messages (`FileThis` and `NewFilingFilter`)
- register a callback function with the folder change notification service

Additionally, you can

- support the use of multiple target items
- customize the Filing slip and folder set that your application uses

The remainder of this section describes these tasks in detail.

Overview of Filing Support

You need to take the following steps to support the Filing service:

- 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; 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.

- Create a `storesFilter` 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
- implement support for local or global folders only
- customize the title text in your `protoNewFolderTab` view
- provide a `TitleClickScript` method to customize the action your folder tab view takes when the user taps 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 a default soup entry created by a frame-constructor function you supply. (For information on the use of frame-constructor functions, see "Programmer's Overview" on page 11-25 in Chapter 11, "Data Storage and Retrieval.")

When the user files a soup entry, the system stores a value in its `labels` 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` and `NewFilingFilter` methods provide the user-interface illusion that the data has been put in a folder.

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, when the user taps the folder tab view in the built-in Notes application, the last item in the picker is "All Notes."

The following code fragment defines a typical `appAll` slot:

```
myAppBase := {… 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 15-2 (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 15-5 (page 15-5).

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 picker is "Unfiled Notes."

The following code fragment defines a typical `appObjectUnfiled` slot:

```
myAppBase := {… 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.

If you use the default version, your application's base view must supply `target` and `targetView` slots that you update whenever the target or target view changes. If you override this method, you provide these slots in the result frame that it returns, rather than in your application's base view. These slots provide the same information regardless of whether they reside in the `GetTargetInfo` method's result frame or in the application's base view.

Creating the Target Slot

The `target` slot contains the data item with which the user is working, such as the soup entry that represents the currently displayed note to file. 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 is appropriate for every application; however, it is common to update the value of this slot from the `ViewClickScript` method 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 a time, you probably need to supply a `GetTargetInfo` method that can return an appropriate target and target view in those situations.

You must override the `GetTargetInfo` method in order to move an item to another store when it is filed. The result frame returned by your `GetTargetInfo` override can include an optional `targetStore` slot that specifies the store on which an item is to reside when it is filed.

To override this method, create in your application base view a slot named `GetTargetInfo` and implement this method as specified in the description of the `GetTargetInfo` method (page 12-11) in *`Newton Programmer's Reference`*.

Creating the labelsFilter slot

Your application's target view must supply a `labelsFilter` slot. This slot holds a value indicating the current filing filter selected by the user in the picker displayed by the folder tab view. This slot can store either a symbol indicating the currently displayed filing category or the value `nil` (which specifies that the Unfiled category is currently displayed).

The system sets the value of the `labelsFilter` slot for you. Your `NewFilingFilter` method must update the display of your application's data according to the value of this slot.

To display a predetermined filing category when your application opens, you can set an initial value for the `labelsFilter` slot from within the application base view's `ViewSetupFormScript` method.

Creating the storesFilter slot

Your application's target view must supply a `storesFilter` slot. This slot stores a value indicating the current store filter selected by the user from the picker displayed by the folder tab view. This slot can store either a symbol or the value `nil`.

The system sets the value of the `storesFilter` slot for you. Your `NewFilingFilter` method must update the display of your application's data according to the value of this slot.

To display items on a particular store when your application opens, you can set an initial value for the `storesFilter` slot from within the application base view's `ViewSetupFormScript` method.

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.

Adding the Folder Tab View

Your application's base view must provide a view that displays the currently selected filing category and allows the user to select a new filing category. This view is based on either the `protoNewFolderTab` or `protoClockFolderTab` system proto.

Adding the folder tab view to your application is easy. In NTK, sketch the folder tab in your application's base view using the `protoNewFolderTab` proto and declare your folder tab view to the application's base view. The system sets the folder tab view's bounds for you at run time, positioning the folder tab relative to its parent, near the top of the screen.

Customizing Folder Tab Views

The `protoNewFolderTab` proto supplies a child view named `title` that images a string that you may supply optionally. To display your own string as the title text in a `protoNewFolderTab` view, use the global function `SetValue` to set the value of the `text` slot in the `title` view child of your folder tab view.

For example,

```
SetValue(myNewFolderTab.title, 'text, "My text");
```

▲ **WARNING**
Do not create a `title` slot in any folder tab view. Do not replace the title text in a `protoClockFolderTab` view. ▲

Defining a TitleClickScript Method

The folder tab view's `TitleClickScript` method is invoked when the user taps 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 provided by the `protoClockFolderTab` view displays the built-in Clock application.

You can provide your own `TitleClickScript` method to customize the action your folder tab views take when the user taps 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 to file it (the value to which you set the target's `labels` slot), and the store on which to file it.

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 `storesChanged` parameter is `true`, your `FileThis` method must move the target to the new store. However, if the value of the `storesChanged` parameter is `nil`, the value of the `destStore` parameter is undefined.

The following code example shows the implementation of a typical `FileThis` method. Remember to call `EntryChangeXmit` from this method so your changes to filed entries are saved!

```
FileThis: // example code - your mileage may vary
func(target, labelsChanged, newLabels, storesChanged, destStore)
   begin
      if labelsChanged AND target.labels <> newLabels then
      begin
         target.labels := newLabels;
         EntryChangeXmit(target, kAppSymbol);
      end // labelsChanged
      if storesChanged and (EntryStore(target) <> destStore) and
         not destStore:IsReadOnly() then
         begin
            // move the entry to the new store & xmit change
            // make sure you handle locked stores too
            if EntryStore(target):IsReadOnly() then
               EntryCopyXmit(target, destStore, kAppSymbol);
            else
               EntryMoveXmit(target, destStore, kAppSymbol);
         end; //storesChanged
   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 passed 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.

The following code example shows the implementation of a typical `NewFilingFilter` method, which queries the application soup for the entries that match the current filing category and then redraws views affected by the change in filing category.

```
NewFilingFilter: func(newFilterPath)
begin
   // first figure out if query should be done on
   // a union soup or on a specific store soup
   // this is to make filter by store more efficient

   local querySoup := GetUnionSoupAlways(kSoupName) ;

   if storesFilter and storesFilter:IsValid() then
      querySoup := querySoup:GetMember(storesFilter) ;

   // now construct the query based on the labelsFilter
   // and set my application cursor (called myCursor)
   // to the new query

   // the default is to show all items, i.e.,
   // labelsFilter is '_all
   local theQuery := nil ;

   if NOT labelsFilter then
      // labelsFilter is NIL, so show only those entries
      // that do not have a valid tag in the labels
      // slot
      theQuery := {none: GetFolderList(appSymbol, nil)};
   else if labelsFilter <> '_all then
      // labelsFilter is some specific folder
      theQuery := {all: labelsFilter} ;

   myCursor := querySoup:Query(theQuery) ;
   // now redraw views affected by the change
   // NOTE: You could keep track of the original
   //       labelsFilter and storesFilter to see
   //       whether you need to actually do work.
end
```

Using the Folder Change Notification Service

You can use the `RegFolderChanged` global function to register callback functions that are executed when the user adds, removes, or edits folders. Most applications register these functions only while the application is actually open, so the application base view's `ViewSetupFormScript` is an ideal place from which to call the `RegFolderChanged` function. For example,

```
myCallback1 := func(oldFolder, newFolder);
begin
   // retag entries
end;
myAppBase.viewSetupFormScript := func ()begin
RegFolderChanged('|myFnId1:myApp:mySig|, myCallback1);
end;
```

The `UnRegFolderChanged` function removes a specified callback from use by the folder change mechanism. Most applications unregister their folder change callback functions when they close, making the application base view's `ViewQuitScript` method an appropriate place to unregister folder change callback functions. For example,

```
myAppBase.viewQuitScript := func ()begin
UnRegFolderChanged('|myFnId1:myApp:mySig|);
end;
```

Creating the doCardRouting slot

If you want to move items between stores from within 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.

Using Local or Global Folders Only

To suppress the display of either local or global folders in the Filing slip and the folder tab views, you can set the values of optional `localFoldersOnly` and `globalFoldersOnly` slots that you supply in your application's base view. Note that the use of local or global folders only 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.

▲ **WARNING**
The `localFoldersOnly` and `globalFoldersOnly` must not both hold non-`nil` values at the same time. ▲

Adding and Removing Filing Categories Programmatically

You can use the `AddFolder` and `RemoveFolder` global functions to modify the set of folders (filing categories) available to your application. Note that the `RemoveFolder` function does not remove any folder that is also used by other applications. For more information, see the descriptions of these methods in the *Newton Programmer's Reference.*

Interface to User-Visible Folder Names

Symbols that represent folders are not tied to the strings that represent those folders 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.

Summary

This section summarizes the data structures, protos, functions, and methods used by the Filing service.

Data Structures for Filing

Application Base View Slots

```
myAppBaseView :=
            {_parent: {…},// root view
            _proto:  {// myAppBaseViewTemplate
                      _proto: {…}, // protoApp,
                     // slots you supply in myAppBaseViewTemplate
                     appObjectUnfiled: "Unfiled Items",
                     appAll: "All Items",
                     appObjectFileThisOn: "File this item on",
                     storesFilter: NIL,
                     doCardRouting: 1,
                     GetTargetInfo: <function, 1 arg(s) #6000F951>,
                     labelsFilter: NIL,
                     appObjectFileThisIn: "File this item in",
                     appSymbol: |myApp:mySig|,
                     …},
            // my filing button template, defined in app base view
            myfilingButton: {_parent: <2> // myAppBaseView,
                             _proto: protoFilingButton, …},
            // my new folder tab template, defined in app base view
            myNewFolderTab: {…}, // see summary on page 15-21
            …}
```

Target Information Frame

```
// 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
// this frame may also include your own slots
…}
```

Filing Protos

protoFilingButton

```
myFilingButtonView :=
// do not override ViewClickScript; use ButtonClickScript instead
   {  _parent:{ // MyAppBaseView
              _parent: {…}, // root view
              _proto: {…}, // myAppBaseViewTemplate
             …},
   _proto:  {// myFilingButtonTemplate
              // set your own viewBounds in NTK view editor
              viewBounds: {left: 10, top: 250,
                          right: 27, bottom: 263},
              _proto: {// protoFilingButton
                       _proto: {…}, // protoPictureButton
                      // your ButtonClickScript must call inherited
                      ButtonClickScript:<function, 0 arg(s) …>,
                      // your Update must call inherited
                       Update: <function, 0 arg(s) …>,
                      // your viewSetupFormScript must call inherited
                       viewSetupFormScript:<function, 0 arg(s)…>
                      …},
              …},
   …}
```

protoNewFolderTab

```
myNewFolderTabView := {
   {_parent: myAppBaseView, // see summary on page 15-20
   _proto: { protoNewFolderTab,
             // your folder tab's viewSetupFormScript must
             // call this inherited method using conditional send
             viewSetupFormScript: <function, 0 arg(s) …>,
             …},
   // do not alter this slot; set only the text slot
   title: {_parent: <2> // myNewFolderTabView,
             _proto: {viewClass: clTextView, …},
             // string displayed at left of newFolderTab view
             text: "My Text",
             …},
   }
```

protoClockFolderTab

```
myClockFolderTabView := {
   {_parent: myAppBaseView, // see page 15-20
   _proto: { protoClockFolderTab,
            // your folder tab's viewSetupFormScript must
            // call inherited:?viewSetupFormScript()
            viewSetupFormScript: <function, 0 arg(s) …>, …},
   // do not attempt to alter the time display text
   …}
```

Filing Functions and Methods

```
view:GetTargetInfo(reason) // override for multiple targets
view:MoveTarget (target, destStore) // move target between stores
RegFolderChanged(callbackID, callBackFn)// register folder change callback
UnRegFolderChanged(callbackID) // unregister folder change callback
AddFolder(newFolderStr, appSymbol) // add local folder
RemoveFolder(folderSym, appSymbol) // remove local folder
GetFolderStr(folderSym) // get user-visible folder name from app sym
RemoveAppFolders(appSym) // remove specified app's local folders
GetFolderList(appSymbol, localOnly) // list the app's local folders
```

Application-Defined Filing Functions and Methods

```
// Optional. Specify filing target, target view, target store
GetTargetInfo(reason)
// Required. Respond to changes in filing filter or store filter
targetView:NewFilingFilter(newFilter)
// Required. File the item as specified
targetView:FileThis (target, labelsChanged, newLabels, storesChanged, newStore)
```

C H A P T E R 1 6

Find

This chapter describes how your application can support finding text, dates, or your own data types in its data. If you want users to be able to use the system's Find service to locate data in your application, you should be familiar with the material discussed in this chapter.

Before reading this chapter, you should understand the concept of the target of an action, explained in Chapter 15, "Filing." Familiarity with using views to image data, covered in Chapter 3, "Views," is also helpful. 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 two main parts:

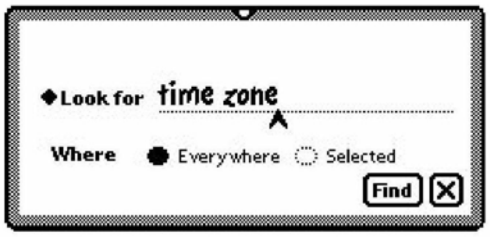
- "About the Find Service" describes the core user interface to the Find service, along with variations and optional features. A compatibility section covers differences between the current version of the Find service and previous ones.
- "Using the Find Service" provides a technical overview of Find operations, with code examples to show how to implement support for this service in your application.

In addition, the "Find Reference" (page 13-1) in *Newton Programmer's Reference* provides complete descriptions of all Find service data structures, functions, and methods.

About the Find Service

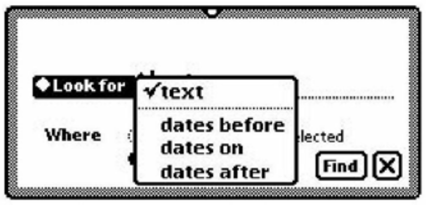
The Find service searches for occurrences of data items the user specifies on a Find slip. The Find slip may be supplied by the system or by the developer. Figure 16-1 illustrates the system-supplied Find slip.

Figure 16-1 The system-supplied Find slip



The system-supplied Find slip contains an input line that specifies a search string, several buttons indicate the scope of the search, and a Look For picker (pop-up menu) that specifies the kind of search to perform. By choosing from the Look For picker (pop-up menu) you may specify whether the search string is a text item or a date, as shown in Figure 16-2.

Figure 16-2 Specifying text or date searches in the Find slip



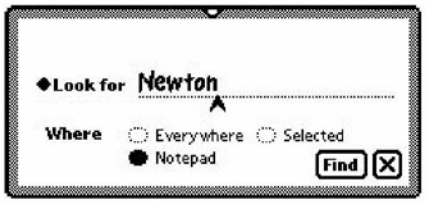
Text searches are case insensitive and find only string beginnings. That is, a search for the string "smith" may return the items "Smith" and "smithers," but not "blacksmith." Date searches find items dated before, after, or on the date specified by the search string.

From the application developer's perspective, text finds and date finds are nearly identical. The only significant difference between them is the test an item must pass to be included in the result of the search.

The system-supplied Find slip always contains an Everywhere button and Selected button. If the current application supports the Find service, a button with the application's name appears in this slip as well.

Searching for data in the current application only is called a **Local find** operation. Figure 16-3 depicts a Local find in the Notepad application.

Figure 16-3 A local Find operation



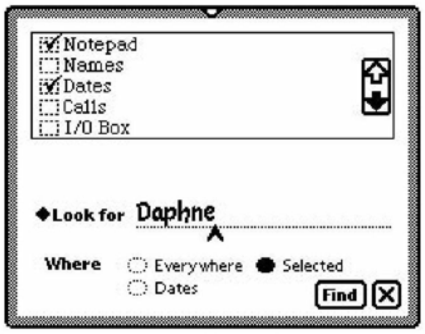
The Everywhere and Selected buttons specify that the system perform searches in applications other than the currently active one. Applications must register with the Find service to participate in such searches.

Tapping the Everywhere button tells the system 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. The list includes all currently available applications registered with the Find service. Tapping items in the list places a check mark next to applications in which the system should conduct a Local find. This kind of search is called a **Selected find.** The slip in Figure 16-4 depicts a search for the string `"Daphne"` in the Notes and Dates applications.

Figure 16-4 Searching selected applications



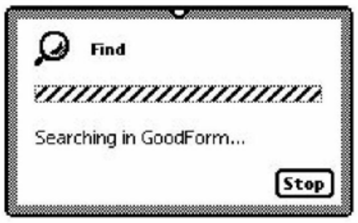
In addition, an application can support searches of multiple data sets. For example, a personal finance program might allow you to search the check ledger, the account list, and the credit charges list as separate searches, even though all the data resides in a single application. For more information on how to implement this in your application see "Adding Application Data Sets to Selected Finds" beginning on page 16-19.

In addition, you can replace the Find slip in the currently active application. Typically, you would do this to provide a customized user interface for specialized searches. For more information, see "Replacing the Built-in Find Slip" beginning on page 16-24.

After setting the parameters of the search with the Find slip, 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.

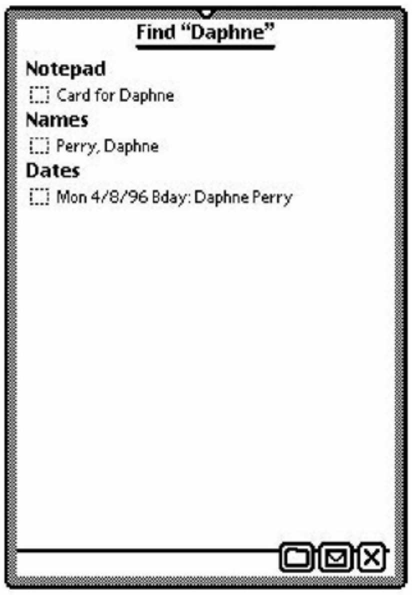
While the search executes, the system provides user feedback through a Progress slip. This slip provides a Stop button that allows the user to cancel a search in progress. Figure 16-5 shows a typical Progress slip.

Figure 16-5 Progress slip



When the search is completed, the Find service displays an overview list of items that match the search criteria. Figure 16-6 shows the Find overview as it might appear after searching all applications for the string `"Daphne"`.

Figure 16-6 The Find overview

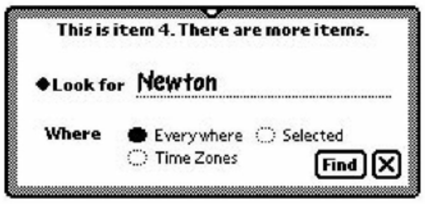


Find "Daphne"

Notepad
Card for Daphne
Names
Perry, Daphne
Dates
Mon 4/8/96 Bday: Daphne Perry

The user can tap 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 16-7 depicts this status message.

Figure 16-7 Find status message



This is item 4. There are more items.

Look for Newton

Where [✓ Everywhere] Selected Time Zones

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 picker. The next time this service is used, it reopens in the most recently set find mode. Note that in order to conserve memory, the list of found items is not saved between finds.

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 Newton 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 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."

The Find service now offers routines that allow you to include multiple data sets from a single application in Selected find operations. Three new methods support this functionality: `AppFindTargets`, `FindTargeted`, and `DateFindTargeted`. You use the `AppFindTargets` method to add identifying strings for the data sets to the Selected picker. The two new Find methods with which you implement targeted finds are `FindTargeted` and `DateFindTargeted`. They are identical to their nontargeted counterparts, except the last parameter, *`indexPath`*, is a path to a data set within an application.

Do not modify any system data structures directly to register or unregister an application 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 `ShowFoundItem` method now has two parameters, a data item and a finder frame. However, the old `ShowFoundItem` method, with one parameter (*`item`*) is still supported.

The `SetStatus` method is obsolete; use the `SetMessage` method instead. In addition, the `FileAs` and `MoveTo` methods are also obsolete; you should use `FileAndMove` instead.

Using the Find Service

This section includes a technical overview of Find operations and describes how to implement Find support in your application.

Technical Overview

When the user taps the Find button, the system invokes your application's search method. This can be a date find method (`DateFind`) or a text find method (`Find`).

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 support any of these search methods independently of one another; for example, you can implement the `Find` method without implementing the `DateFind` method.

You may also customize searches by adding a subset of data items from one application to the Selected picker menu in the Find slip. Items added here may be, for instance, a checkbook and ledger from a personal finance program.

A **finder** is a frame that enumerates items resulting from a Find operation. The general characteristics of your finder are defined by the proto it's based on. The system supplies two protos on which to base your finder: the `ROM_SoupFinder` or the `ROM_CompatibleFinder`.

The `ROM_SoupFinder` proto supports searching soup data. The `ROM_CompatibleFinder` proto provides a framework, which you should override, that supports searching data that is not soup based. When a finder based on the `ROM_SoupFinder` proto completes a search, it returns with a cursor which is used to retrieve the found items from the application soup. When a finder based on the `ROM_CompatibleFinder` proto completes a search, it returns with the actual found items in an array (the `items` array).

If you store data in soups, there are standard find methods defined for the `ROM_SoupFinder` proto that you can use. When you devise a different scheme, you must use the `ROM_CompatibleFinder` proto and define versions of the finder methods that are tailored to your type of data storage.

After a search method scans your application's data and returns a finder frame, you must append it to the system-supplied `results` array. Global and Selected finds usually append more than one frame to this array, as multiple applications complete their searches.

While a search continues, the system automatically provides user feedback on its progress. When the search method completes, the system displays an overview list of the items that were found.

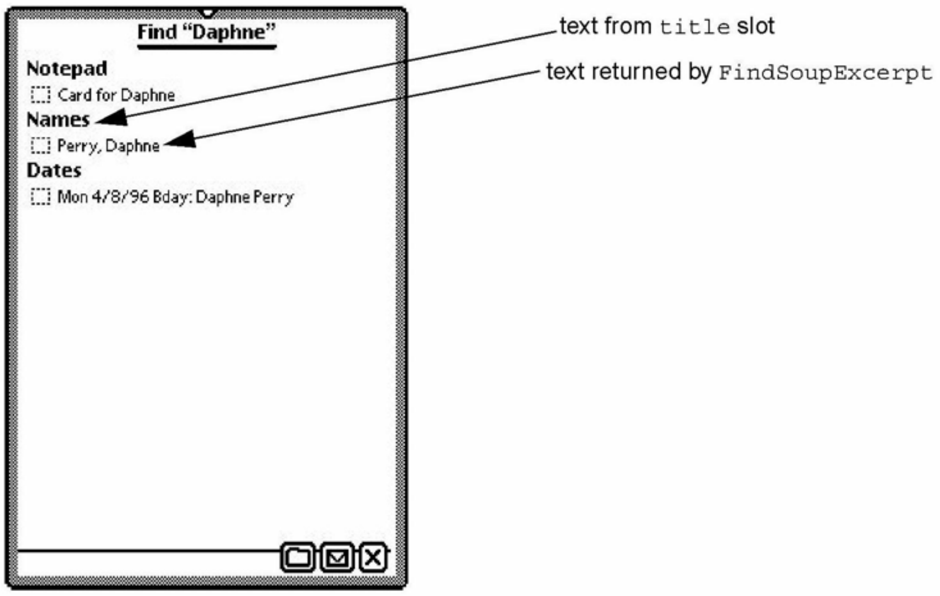
For Global or Selected finds, each application (or data set, for a targeted data set find) in which items were found is identified by a heading, with the found items listed under it. The application name that appears in this heading is supplied by the `title` slot each application provides in its base view.

The system sends a `FindSoupExcerpt` message to your application, which must have a `FindSoupExcerpt` method to respond to it. This method must extract and return a string for the Find overview to display. If no items are found, the `FindSoupExcerpt` message is not sent. If you are using the

ROM_CompatibleFinder proto, the string to display for each found item is contained in the title slot of each of the items in the items array in your finder.

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. Figure 16-8 depicts the strings from both the title slot and the FindSoupExcerpt method as they are used in a typical Find overview.

Figure 16-8 Strings used in a Find overview



When the user taps an item in the overview, the system sends a ShowFoundItem message to the view specified by the owner slot in your finder frame (which you appended to the system's results array). In the template of the specified owner view, you define a ShowFoundItem method that must locate the found item in your application's data and perform 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 your finder is based on, you can often use the same method to display the results of both text and date searches. If you are using a finder based on ROM_CompatibleFinder, you must override its ShowFakeEntry method to call your ShowFoundItem method.

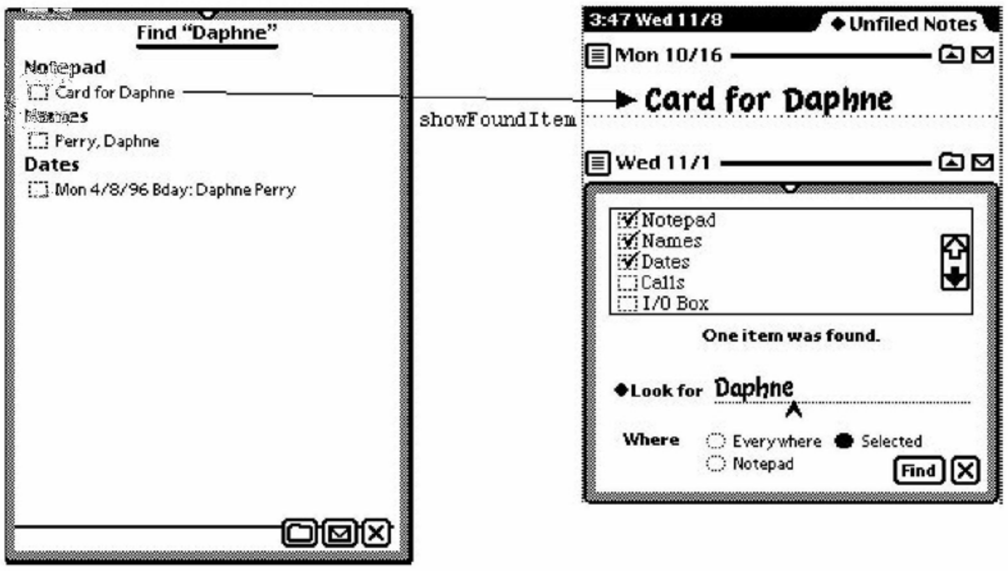


Figure 16-9 The `ShowFoundItem` method displays the view of an overview item

The Find overview provides Routing and Filing buttons. If you are using the `ROM_SoupFinder` the system will file, move, and delete your entries in the overview of found items. In such an occurrence, the soup-change notification mechanism notifies your application. (The soup-change notification mechanism is described in Chapter 11, "Data Storage and Retrieval.")

If you are using the `ROM_CompatibleFinder` you may either not allow your found item to be routed or override the relevant methods.

Note that if the system routes your soup-based data, your application is notified via the soup-change notification mechanism. For a complete description of this mechanism, see Chapter 11, "Data Storage and Retrieval."

Global and Selected Finds

When the user taps the Find button, the system invokes find methods in the appropriate applications. For a Local find, only the currently active application is sent a message. For a Global find, all applications registered with the Find service are sent messages. Selected finds send messages to a user-specified subset of all applications registered for Global finds. In terms of the messages sent, Global finds and Selected finds are 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 must test to see that the application soup is available before searching it.

The system informs your search method of the scope of the search through the *scope* parameter. 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 or Selected find. The `'localFind` symbol is passed for Local find operations.

Checklist for Adding Find Support

To add application support for the Find service, you need to do the following:

- Create a `title` slot, in the view referenced by the `owner` slot of your finder frame, that contains the user-visible name of the application.
- Create the `appName` slot in your application's base view that contains the user-visible name of the application.
- Choose a finder proto on which to base your application's frame. You should use `ROM_SoupFinder` if your data is stored in a single soup, and `ROM_CompatibleFinder` otherwise.
- Supply at least one search method (`Find`, `DateFind`).
- Append the resultant finder frame to the system-supplied `results` array at the end of your search method(s).
- 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 use the `ROM_CompatibleFinder` proto you must add a `title` slot with a string defining each found item to the frame representing the item.
- Supply a `ShowFoundItem` method that displays an individual entry from the found items.
- When using a `ROM_CompatibleFinder` proto, write a `ShowFakeEntry` method to call your `ShowFoundItem` method.
- When using the `ROM_CompatibleFinder`, you should either not allow your found items to be selected (and thus not routed), or override the relevant routing methods.

Optionally, you may also do the following:

- Register and unregister for participation in Global and Selected finds.

- Employ multiple data sets from one application in a Selected find by adding the method `AppFindTargets`, and one or both of the search methods `FindTargeted` and `DateFindTargeted`.
- Replace the system-supplied Find slip with one of your own by supplying a `CustomFind` method in your application's base view. This method will be called when the user taps Find and your application is frontmost.

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 finder 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. The value of this slot is used to name the Find slip button that represents your application when it is the current application. It is also employed by other system services to obtain a user-visible name for your application.

Using the Finder Protos

You use a finder proto as the basis from which to construct the finder frame returned by your search method. The two system-supplied finder protos are employed according to the data type you use for your application's data storage. You can create your own customizations at compile time by creating an item like the following example:

```
kMySoupFinder:= {
               _proto: ROM_SoupFinder,

               Delete: func()
               begin
                  print("About to delete " &
                        Length(selected) && "items");
                  inherited:Delete();
               end
               }
```