

EXHIBIT 64 PART 2

C H A P T E R 6

Pickers, Pop-up Views, and Overviews

This chapter describes how to use pickers and pop-up views to present information and choices to the user. You should read this chapter if you are

- creating your own pickers and pop-up views
- taking advantage of built-in picker and pop-up protos
- presenting outlines and overviews of data

Before reading this chapter, you should be familiar with the information in Chapter 3, “Views.”

This chapter contains:

- an overview of pickers and pop-up views
- descriptions of the pickers and pop-up views used to perform specific tasks
- a summary of picker and pop-up view reference information

About Pickers and Pop-up Views

A **picker** or **pop-up view** is a view that pops up and presents a list of items from which the user can make selections. The view pops up in response to a user action such as a pen tap.

The distinction between a picker and a pop-up view is not important and has not been maintained in naming the protos, so the terms are used somewhat interchangeably. In the discussion that follows, picker is used for both terms.

The simplest picker protos handle the triggering and closing of the picker; for these protos, all you need to do is provide the items in the list. When the user taps a button, a label, or a hot spot in a picture, the picker view opens automatically. When the user makes a selection, the view closes automatically and sends a message with the index of the chosen item. If the user taps outside the picker, the view closes, with no selection having been made.

CHAPTER 6

Pickers, Pop-up Views, and Overviews

More sophisticated picker protos allow multiple selections and use a close box to dispatch the view.

With some picker protos, you must determine when and how the picker is displayed. You open a picker view by sending the `Open` message to the view, or by calling the `PopupMenu` function.

Your picker views can display

- simple text
- bitmaps
- icons with strings
- separator lines
- two-dimensional grids

The most sophisticated picker protos let you access built-in system soups as well as your own soups. Much of the behavior of these protos is provided by data definitions that iterate through soup entries, display a list, allow the user to see and modify the data, and add new entries to the soup.

Pickers and Pop-up View Compatibility

The 2.0 release of Newton system software contains a number of new picker protos and a replacement for the `DoPopup` global function.

New Pickers and Pop-up Views

Two new picker protos, `protoPopupButton` and `protoPopInPlace`, define text buttons that display pickers.

A new set of map pickers allows you to display various maps from which a user can select a location and receive information about it. The map pickers include the following:

- `protoCountryPicker`
- `protoProvincePicker`
- `protoStatePicker`
- `protoWorldPicker`

A set of new text pickers lets you display pop-up views that show text that the user can change by tapping the string and entering a new string. The `protoDateTextPicker`, for example, lets the user change a date. The text-picker protos include the following:

- `protoTextPicker`
- `protoDateTextPicker`

CHAPTER 6

Pickers, Pop-up Views, and Overviews

- protoDateDurationTextPicker
- protoRepeatDateDurationTextPicker
- protoDateNTimeTextPicker
- protoTimeTextPicker
- protoDurationTextPicker
- protoTimeDeltaTimePicker
- protoMapTextPicker
- protoCountryTextPicker
- protoUSstatesTextPicker
- protoCitiesTextPicker
- protoLongLatTextPicker

New date, time, and location pop-up views let the user specify new information in a graphical view—changing the date on a calendar, for example. These protos include the following:

- protoDatePopup
- protoDatePicker
- protoDateNTimePopup
- protoDateIntervalPopup
- protoMultiDatePopup
- protoYearPopup
- protoTimePopup
- protoAnalogTimePopup
- protoTimeDeltaPopup
- protoTimeIntervalPopup

A new number picker displays pickers from which a user can select a number. The new number picker is

- protoNumberPicker

A set of new overview protos allows you to create overviews of data; some of the protos are designed to display data from the Names soup. The data picker protos include the following:

- protoOverview
- protoSoupOverview
- protoListPicker
- protoPeoplePicker

About Pickers and Pop-up Views

6-3

CHAPTER 6

Pickers, Pop-up Views, and Overviews

- `protoPeoplePopup`

The following two protos are data types that support the `protoListPicker`:

- `protoNameRefDataDef`
- `protoPeopleDataDef`

Obsolete Function

The `DoPopup` global function used in system software version 1.x is obsolete; it is supported in version 2.0, but support is not guaranteed in future releases. Use the new `PopupMenu` function instead.

Picker Categories

The remainder of this chapter divides the pickers into a number of categories. The protos within each category operate in a related manner. General-purpose protos are used to create simple, general-purpose pickers and pop-up views. The remaining protos in the list are triggered by specific user actions or by events that you define:

- general-purpose pickers
- map pickers
- text pickers
- date, time, and location pickers
- number pickers
- picture picker
- overview protos
- roll protos

There is also a section discussing the view classes used with pickers.

General-Purpose Pickers

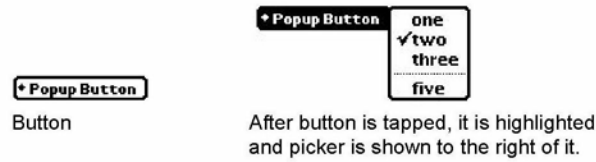
You use the protos described in this section to create simple, general-purpose pickers and pop-up views. Some of the following protos are triggered by specific user actions, while others are triggered by events that you define:

- The `protoPopupButton` picker is a text button that displays a picker when tapped. The button is highlighted while the picker is open. For information about the slots and methods for this picker, see “`protoPopupButton`” (page 5-4) in *Newton Programmer’s Reference*. Figure 6-1 shows an example of a `protoPopupButton`.

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Figure 6-1 A `protoPopupButton` example



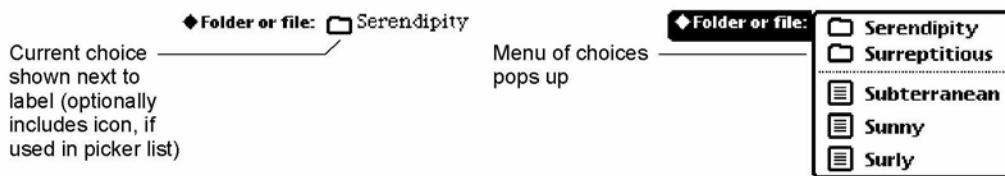
- The `protoPopInPlace` picker is a text button that displays a picker when tapped. When the user chooses an item from the picker, the text of the chosen item appears in the button. For information about the slots and methods for this picker, see “`protoPopInPlace`” (page 5-6) in *Newton Programmer’s Reference*. Figure 6-2 shows an example of a `protoPopInPlace`.

Figure 6-2 A `protoPopInPlace` example



- The `protoLabelPicker` is a label that displays a picker when tapped. The currently selected item in the list is displayed next to the label. For information about the slots and methods for this picker, see “`protoLabelPicker`” (page 5-8) in *Newton Programmer’s Reference*. Figure 6-3 shows an example of a `protoLabelPicker`.

Figure 6-3 A `protoLabelPicker` example



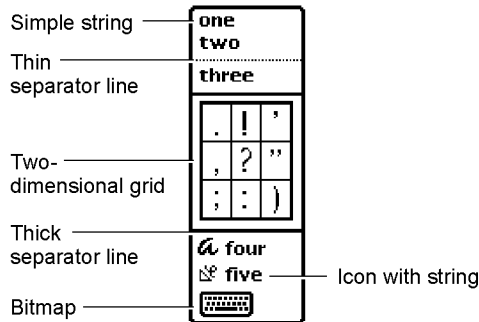
- The `protoPicker` is a picker that displays anything from a simple text list to a two-dimensional grid containing shapes and text. For information about the slots and methods for this picker, see “`protoPicker`” (page 5-13) in *Newton*

CHAPTER 6

Pickers, Pop-up Views, and Overviews

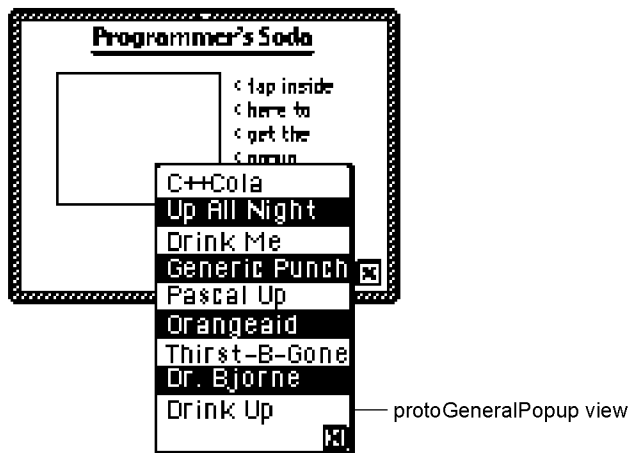
Programmer's Reference. Figure 6-4 shows the types of objects you can display in a `protoPicker`.

Figure 6-4 A `protoPicker` example



- The `protoGeneralPopup` is a pop-up view that has a close box. The view cancels if the user taps outside it. This can use this proto to construct more complex pickers. It is used, for example, as the basis for the duration pickers. For information about the slots and methods for this proto, see “`protoGeneralPopup`” (page 5-19) in *Newton Programmer's Reference*. Figure 6-5 shows an example of a `protoGeneralPopup`.

Figure 6-5 A `protoGeneralPopup` example

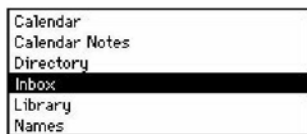


CHAPTER 6

Pickers, Pop-up Views, and Overviews

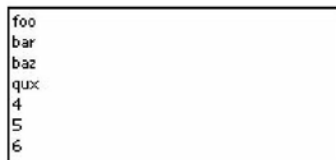
- The `protoTextList` picker is a scrollable list of items. The user can scroll the list by dragging or scrolling with the optional scroll arrows and can choose one or more items in the list by tapping them. The scrollable list can include shapes or text. For information about the slots and methods for this picker, see “`protoTextList`” (page 5-20) in *Newton Programmer’s Reference*. Figure 6-6 shows an example of a `protoTextList`.

Figure 6-6 A `protoTextList` example



- The `protoTable` picker is a simple one-column table of text. The user can tap any item in the list to select it. For information about the slots and methods for this picker, see “`protoTable`” (page 5-24) in *Newton Programmer’s Reference*. Figure 6-7 shows an example of a `protoTableList` picker.

Figure 6-7 A `protoTable` example



You define the format of the table using a `protoTableDef` object; see “`protoTableDef`” (page 5-27) in *Newton Programmer’s Reference* for information. You define the format of each row using a `protoTableEntry` object; see “`protoTableEntry`” (page 5-29) in *Newton Programmer’s Reference* for information.

Using `protoGeneralPopup`

As with most protos, you create a `protoGeneralPopup` object by using the NTK palette to draw one in your layout. After creating the object, you should remove the context and cancelled slots. The `viewBounds` should be `(0, 0, width, height)` for the box. The `New` method tries to set the bounds correctly, based on the recommended bounds passed to the call.

CHAPTER 6

Pickers, Pop-up Views, and Overviews

The `protoGeneralPopup` sends a `pickCancelledScript` to the `callbackContext` specified in the `New` method. However, it does not send a `pickActionScript` back; instead, it sends an Affirmative message to itself. You supply the method and decide what call to make to the context and what information to send back.

To put other objects in the `protoGeneralPopup`, just drag them out in NTK. For example, if you want a checkbox in your pop-up view, drag out a `protoCheckbox`. You can put anything in the pop-up view, including your own protos.

Since you have to assemble the information to send on an affirmative, you will likely end up declaring your content to the general pop-up.

The only slots you really need to set are `Affirmative` and `viewBounds`.

`Affirmative` is a function. Here's an example:

```
func ()
begin
// Notify the context that the user has accepted the
// changes made in the popup
if context then
    context:?pickActionScript(changeData) ;
end
```

Map Pickers

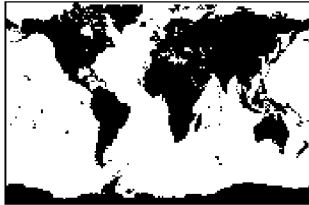
You can use the pickers described in this section to display maps and allow the user to select countries, U.S. states, Canadian provinces, and cities. The Newton system software provides the following map picker protos:

- The `protoCountryPicker` displays a map of the world. When the user taps a country, the `PickWorld` message is sent to your view. For information about the slots and methods for this picker, see “`protoCountryPicker`” (page 5-30) in *Newton Programmer's Reference*. Figure 6-8 shows an example of a `protoCountryPicker`.

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Figure 6-8 A `protoCountryPicker` example



- The `protoProvincePicker` displays a map of North America. When the user taps a province, the `PickWorld` message is sent to your view. For information about the slots and methods for this picker, see “`protoProvincePicker`” (page 5-31) in *Newton Programmer’s Reference*. Figure 6-9 shows an example of a `protoProvincePicker`.

Figure 6-9 A `protoProvincePicker` example



- The `protoStatePicker` displays a map of North America. When the user taps a state, the `PickWorld` message is sent to your view. For information about the slots and methods for this picker, see “`protoStatePicker`” (page 5-32) in *Newton Programmer’s Reference*. Figure 6-10 shows an example of a `protoStatePicker`.

Figure 6-10 A `protoStatePicker` example



- The `protoWorldPicker` displays a map of the world. When the user taps a continent, the `PickWorld` message is sent to your view. For information about

CHAPTER 6

Pickers, Pop-up Views, and Overviews

the slots and methods for this picker, see “protoWorldPicker” (page 5-34) in *Newton Programmer’s Reference*. Figure 6-11 shows an example of a protoWorldPicker.

Figure 6-11 A protoWorldPicker example



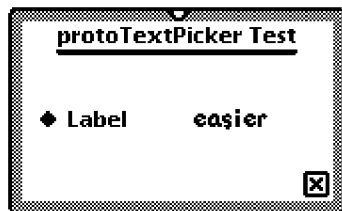
Text Pickers

Text picker protos allow the user to specify various kinds of information by picking text representations. Each of these protos displays a label picker with a string that shows the currently selected data value. For example, protoDurationTextPicker, which lets the user set a duration, might have a label of “When” followed by a duration in the form “8:26 A.M. – 10:36 P.M.”

When the user taps a text picker, the picker displays a pop-up view in which the user can enter new information. The Newton system software provides the following text picker protos:

- The protoTextPicker is a label picker with a text representation of an entry. When the user taps the picker, a customized picker is displayed. For information about the slots and methods for this picker, see “protoTextPicker” (page 5-35) in *Newton Programmer’s Reference*. Figure 6-12 shows an example of a protoTextPicker.

Figure 6-12 A protoTextPicker example



CHAPTER 6

Pickers, Pop-up Views, and Overviews

- The `protoDateTextPicker` is a label picker with a text representation of a date. When the user taps the picker, a `protoDatePopup` is displayed, which allows the user to specify a different date. For information about the slots and methods for this picker, see “`protoDateTextPicker`” (page 5-37) in *Newton Programmer’s Reference*. Figure 6-13 shows an example of a `protoDateTextPicker`.

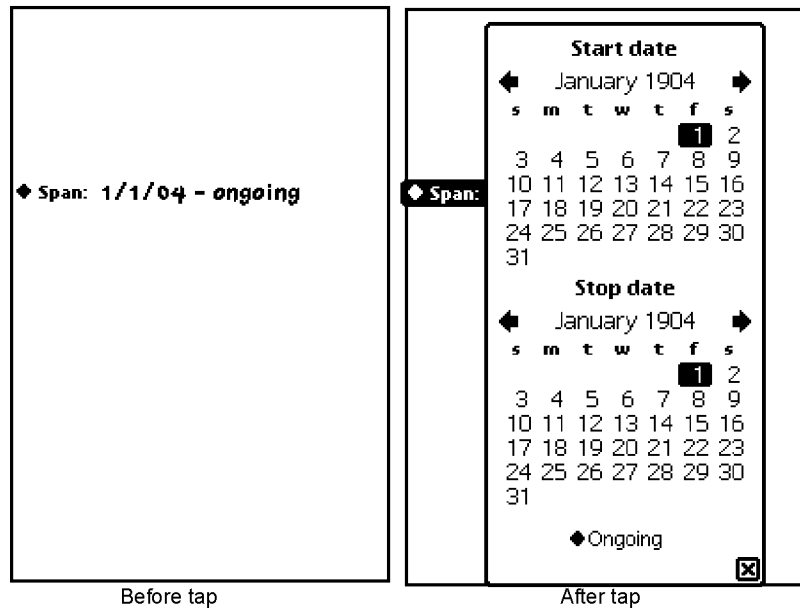
Figure 6-13 A `protoDateTextPicker` example



- The `protoDateDurationTextPicker` is a label picker with a text representation of a range of dates. When the user taps the picker, a `protoDateIntervalPopup` is displayed, which allows the user to specify a different range. For information about the slots and methods for this picker, see “`protoDateDurationTextPicker`” (page 5-40) in *Newton Programmer’s Reference*. Figure 6-14 shows an example of a `protoDateDurationTextPicker`.

CHAPTER 6

Pickers, Pop-up Views, and Overviews

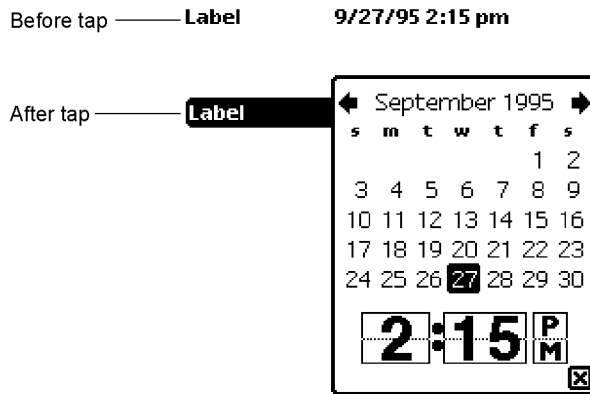
Figure 6-14 A protoDateDurationTextPicker example

- The protoRepeatDateDurationTextPicker is a label picker with a text representation of a range of dates. When the user taps the picker, a protoDateIntervalPopup is displayed, which allows the user to specify a different range. This proto differs from the protoDateDurationTextPicker in that the protoRepeatDateDurationDatePicker presents choices that are appropriate for the repeatType slot, and the duration displayed when the user taps a duration or stop date is given in units of the repeatType. Otherwise, it looks like the protoDateDurationTextPicker and popup shown in Appendix Figure 6-14. For information about the slots and methods for this picker, see “protoRepeatDateDurationTextPicker” (page 5-43) in *Newton Programmer’s Reference*.
- The protoDateNTimeTextPicker is a label picker with a text representation of a date and time. When the user taps the picker, a protoDateNTimePopup is displayed, which allows the user to specify a different date and time. For information about the slots and methods for this picker, see “protoDateNTimeTextPicker” (page 5-46) in *Newton Programmer’s Reference*. Figure 6-15 shows an example of a protoDateNTimeTextPicker.

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Figure 6-15 A `protoDateNTimeTextPicker` example



- The `protoTimeTextPicker` is a label picker with a text representation of a time. When the user taps the picker, a `protoTimePopup` is displayed, which allows the user to specify a different time. For information about the slots and methods for this picker, see “A `protoTimeTextPicker` example” (page 6-13) in *Newton Programmer’s Reference*. Figure 6-16 shows an example of a `protoTimeTextPicker`.

Figure 6-16 A `protoTimeTextPicker` example

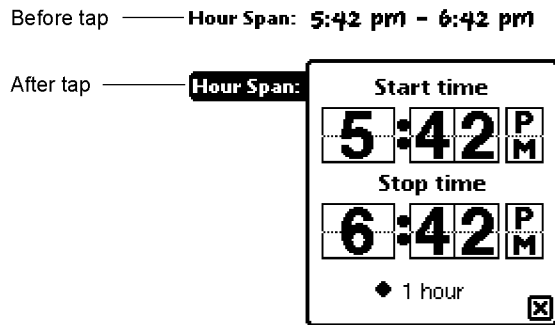


- The `protoDurationTextPicker` is a label picker with a text representation of a time range. When the user taps the picker, a `protoTimeIntervalPopup` is displayed, which allows the user to specify a different time range. For information about the slots and methods for this picker, see “`protoDurationTextPicker`” (page 5-51) in *Newton Programmer’s Reference*. Figure 6-17 shows an example of a `protoDurationTextPicker`.

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Figure 6-17 A protoDurationTextPicker example



- The protoTimeDeltaTextPicker is a label picker with a text representation of a time delta. When the user taps the picker, a protoTimeDeltaPopup is displayed, which allows the user to specify a different time delta. For information about the slots and methods for this picker, see “protoTimeDeltaTextPicker” (page 5-53) in *Newton Programmer’s Reference*. Figure 6-18 shows an example of a protoTimeDeltaTextPicker.

Figure 6-18 A protoTimeDeltaTextPicker example

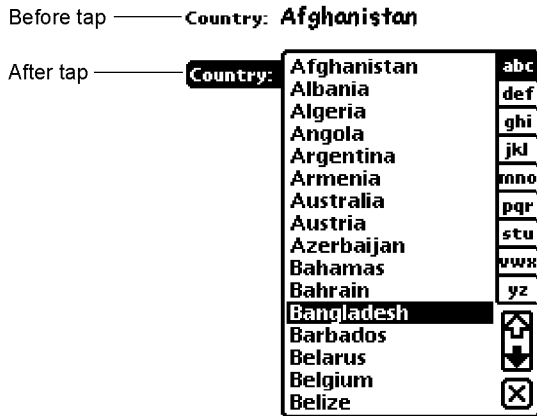


- The protoMapTextPicker is a label picker with a text representation of a country. When the user taps the picker, a popup displays that allows the user to select a new country from an alphabetical list. For information about the slots and methods for this picker, see “protoMapTextPicker” (page 5-54) in *Newton Programmer’s Reference*. Figure 6-19 shows an example of a protoMapTextPicker.

CHAPTER 6

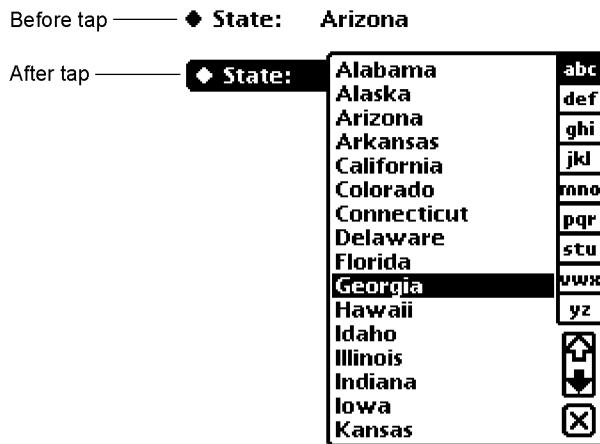
Pickers, Pop-up Views, and Overviews

Figure 6-19 A protoMapTextPicker example



- The protoCountryTextPicker is the same as protoMapTextPicker.
- The protoUSstatesTextPicker is a label picker with a text representation of a U.S. state. When the user taps the picker, a popup displays that allows the user to select a new state from an alphabetical list. For information about the slots and methods for this picker, see “protoUSstatesTextPicker” (page 5-56) in *Newton Programmer’s Reference*. Figure 6-20 shows an example of a protoUSstatesTextPicker.

Figure 6-20 A protoUSstatesTextPicker example

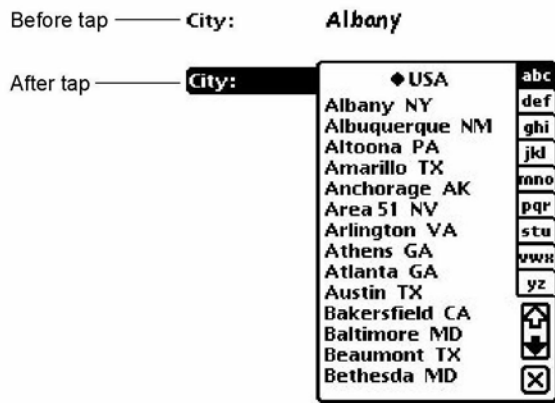


CHAPTER 6

Pickers, Pop-up Views, and Overviews

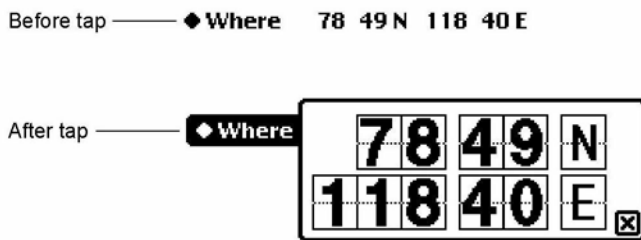
- The `protoCitiesTextPicker` is a label picker with a text representation of a city. When the user taps the picker, a popup displays that allows the user to select a new city from an alphabetical list. For information about the slots and methods for this picker, see “`protoCitiesTextPicker`” (page 5-58) in *Newton Programmer’s Reference*. Figure 6-21 shows an example of a `protoCitiesTextPicker`.

Figure 6-21 A `protoCitiesTextPicker` example



- The `protoLongLatTextPicker` is a label picker with a text representation of longitude and latitude values. When the user taps the picker, a `longLatPicker` is displayed, which allows the user to select new longitude and latitude values. For information about the slots and methods for this picker, see “`protoLongLatTextPicker`” (page 5-61) in *Newton Programmer’s Reference*. Figure 6-22 shows an example of a `protoLongLatTextPicker`.

Figure 6-22 A `protoLongLatTextPicker` example



CHAPTER 6

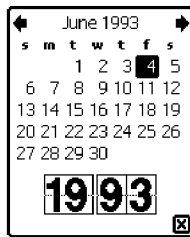
Pickers, Pop-up Views, and Overviews

Date, Time, and Location Pop-up Views

You can use the protos described in this section to present pop-up views to the user for setting or choosing specific types of values. The Newton System Software provides the following pop-up protos for date, time, and location values:

- The `protoDatePopup` allows the user to choose a single date. For information about the slots and methods for this proto, see “`protoDatePopup`” (page 5-63) in *Newton Programmer’s Reference*. Figure 6-23 shows an example of a `protoDatePopup`.

Figure 6-23 A `protoDatePopup` example



- The `protoDatePicker` allows the user to choose a single date when the date is likely to be relatively close to the current date. Changing the year is not easily done with this proto. For information about the slots and methods for this proto, see “`protoDatePicker`” (page 5-64) in *Newton Programmer’s Reference*. Figure 6-24 shows an example of a `protoDatePicker`.

Figure 6-24 A `protoDatePicker` example

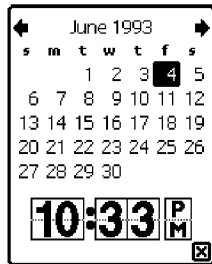


CHAPTER 6

Pickers, Pop-up Views, and Overviews

- The `protoDateNTimePopup` allows the user to choose a single date and time. For information about the slots and methods for this proto, see “`protoDateNTimePopup`” (page 5-67) in *Newton Programmer’s Reference*. Figure 6-25 shows an example of a `protoDateNTimePopup`.

Figure 6-25 A `protoDateNTimePopup` example



- The `protoDateIntervalPopup` allows the user to choose an interval of dates by specifying the start and stop dates. For information about the slots and methods for this proto, see “`protoDateIntervalPopup`” (page 5-69) in *Newton Programmer’s Reference*. Figure 6-26 shows an example of a `protoDateIntervalPopup`.

Figure 6-26 A `protoDateIntervalPopup` example



CHAPTER 6

Pickers, Pop-up Views, and Overviews

- The `protoMultiDatePopup` allows the user to specify a range of dates. For information about the slots and methods for this proto, see “`protoMultiDatePopup`” (page 5-72) in *Newton Programmer’s Reference*. Figure 6-27 shows an example of a `protoMultiDatePopup`.

Figure 6-27 A `protoMultiDatePopup` example



- The `protoYearPopup` allows the user to choose a year. For information about the slots and methods for this proto, see “`protoYearPopup`” (page 5-73) in *Newton Programmer’s Reference*. Figure 6-28 shows an example of a `protoYearPopup`.

Figure 6-28 A `protoYearPopup` example



- The `protoTimePopup` allows the user to choose a time with a digital clock. For information about the slots and methods for this proto, see “`protoTimePopup`” (page 5-74) in *Newton Programmer’s Reference*. Figure 6-29 shows an example of a `protoTimePopup`.

Figure 6-29 A `protoTimePopup` example

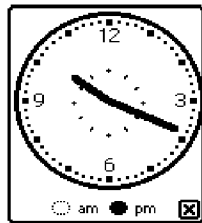


CHAPTER 6

Pickers, Pop-up Views, and Overviews

- The `protoAnalogTimePopup` allows the user to choose a time with an analog clock. For information about the slots and methods for this proto, see “`protoAnalogTimePopup`” (page 5-76) in *Newton Programmer’s Reference*. Figure 6-30 shows an example of a `protoAnalogTimePopup`.

Figure 6-30 A `protoAnalogTimePopup` example



- The `protoTimeDeltaPopup` allows the user to choose a time period (a delta). For information about the slots and methods for this proto, see “`protoTimeDeltaPopup`” (page 5-78) in *Newton Programmer’s Reference*. Figure 6-31 shows an example of a `protoTimeDeltaPopup`.

Figure 6-31 A `protoTimeDeltaPopup` example



- The `protoTimeIntervalPopup` allows the user to choose a time interval by specifying the start and stop times. For information about the slots and methods for this proto, see “`protoTimeIntervalPopup`” (page 5-79) in *Newton Programmer’s Reference*. Figure 6-32 shows an example of a `protoTimeIntervalPopup`.

Figure 6-32 A `protoTimeIntervalPopup` example



CHAPTER 6

Pickers, Pop-up Views, and Overviews

Number Pickers

This section describes the protos available to allow users to pick numbers. The Newton system software provides the following protos for picking numbers:

- The `protoNumberPicker` displays a picker from which the user can select a number. For information about the slots and methods for this picker, see “`protoNumberPicker`” (page 5-81) in *Newton Programmer’s Reference*. Figure 6-33 shows an example of a `protoNumberPicker`.

Figure 6-33 A `protoNumberPicker` example

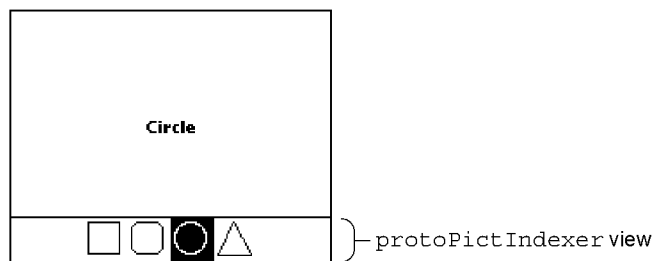


Picture Picker

This section describes the proto you can use to create a picture as a picker.

- The `protoPictIndexer` picker displays a horizontal array of pictures, from which the user can choose. For information about the slots and methods for this picker, see “`protoPictIndexer`” (page 5-82) in *Newton Programmer’s Reference*. Figure 6-34 shows an example of a `protoPictIndexer`.

Figure 6-34 A `protoPictIndexer` example



CHAPTER 6

Pickers, Pop-up Views, and Overviews

Overview Protos

You can use the protos described in this section to create overviews of data. An overview allows the user to see all of data in a soup or an array scrolling list. The user can select individual items and open them to see the detail. Overview protos include:

- The `protoOverview` provides a framework for displaying an overview of the data in your application. Each overview item occupies one line, and the user can scroll the list and pick individual or multiple items. “Using `protoOverview`” (page 6-24) has information on using this proto. For further information about the slots and methods of `protoOverview`, see “`protoOverview`” (page 5-85) in *Newton Programmer’s Reference*. Figure 6-35 shows an example of a `protoOverview`.

Figure 6-35 A `protoOverview` example

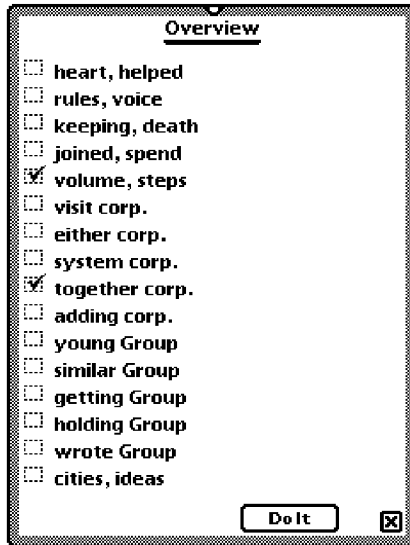


- The `protoSoupOverview` provides a framework for displaying an overview of soup entries in your application. For information about the slots and methods for this proto, see “`protoSoupOverview`” (page 5-90) in *Newton Programmer’s Reference*. Figure 6-36 shows an example of a `protoSoupOverview`.

CHAPTER 6

Pickers, Pop-up Views, and Overviews

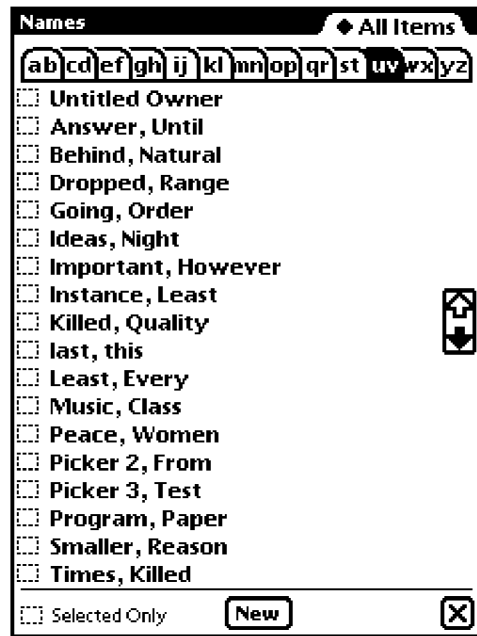
Figure 6-36 A protoSoupOverview example



- The `protoListPicker` provides a scrollable list of items. Items can be from a soup, an array, or both. The user can select any number of items in the list. For information about the slots and methods for this proto, see “`protoListPicker`” (page 5-93) in *Newton Programmer’s Reference*. “Using `protoListPicker`” (page 6-26) has a more extensive example and discusses how to use this proto. Figure 6-37 shows an example of a `protoListPicker`.

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Figure 6-37 A `protoListPicker` example

- The `protoPeoplePicker` displays a list of names and associated information from the Names application. For information about the slots and methods for this proto, see “`protoPeoplePicker`” (page 5-110) in *Newton Programmer’s Reference*.
- The `protoPeoplePopup` is similar to the `protoPeoplePicker`, except that `protoPeoplePopup` displays the picker in a pop-up view. For information about the slots and methods for this proto, see “`protoPeoplePopup`” (page 5-111) in *Newton Programmer’s Reference*.

Using `protoOverview`

The `protoOverview` was set up primarily to be the basis for `protoSoupOverview`. Because of that, you need to do some extra work to use just the `protoOverview`.

You need to define `Abstract`, `HitItem`, `IsSelected`, `SelectItem`, and `viewSetupChildrenScript` methods in your `protoOverview`. See “`protoOverview`” (page 5-85) in *Newton Programmer’s Reference* for details.

CHAPTER 6

Pickers, Pop-up Views, and Overviews

You also need to define the following slot in your `protoOverview`:

`cursor` This should be a cursor-like object.

You use the object stored in this slot to encapsulate your data. The cursor-like object must support the methods `Entry`, `Next`, `Move`, and `Clone`. An example is given below.

In addition, you must provide a mechanism to find an actual data item given an index of a displayed item. In general, you need some sort of saved index that corresponds to the first displayed item. See the example code in “HitItem” (page 5-88) in *Newton Programmer’s Reference* for an example of how this is used.

You also should provide a mechanism to track the currently highlighted item, which is distinct from a selected item.

Since your data is probably in an array, you can use a “cursor” object like this:

```
{ items: nil,
  index: 0,

  Entry:func()
  begin
    if index < Length(items) then
      items[index];
    end,

  Next: func()
  if index < Length(items)-1 then
  begin
    index := index + 1;
    items[index];
  end,

  Move: func(delta)
  begin
    index := Min(Max(index + delta, 0),
                 kNumItems-1) ;
    items[index];
  end,

  Clone:func()
  Clone(self) }
```

The methods that you need to have in the cursor-like object are:

- `Entry`, which returns the item pointed to by the “cursor.”
- `Next`, which moves the “cursor” to the next item and returns that item or, if there is no next item, `nil`.

CHAPTER 6

Pickers, Pop-up Views, and Overviews

- Move, which moves the “cursor” a given number of entries and returns that entry or, if there is no item in that place, nil.
- Clone, which returns a copy of the “cursor” that is modifiable independent of the original “cursor.”

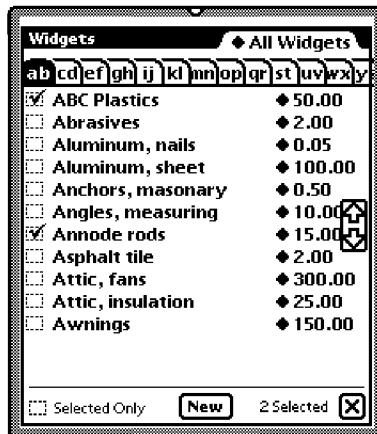
Using protoListPicker

The `protoListPicker` proto—documented in *Newton Programmer’s Reference* (page 5-93)—provides a number of controls for finding specific entries, including folder tabs, alphabet tabs (`azTabs`), and scrolling arrows; any of these controls can be suppressed.

Like `protoOverview`, this proto manages an array of selected items. Any soup that can be queried by a cursor can be displayed, or elements from an array can be displayed.

Figure 6-38 shows a full-featured example of `protoListPicker` that displays a two-column list. The first column is used to select or deselect members, and the second column provides additional information that can be edited in place.

Figure 6-38 A `ProtoListPicker` example

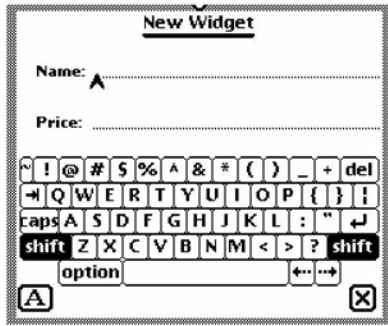


The checkbox at the bottom-left of the slip is used to either show every eligible item or to trim all unselected elements from the list. The New button at the bottom allows the immediate creation of another entry to be displayed. See Figure 6-39.

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Figure 6-39 Creating a new name entry



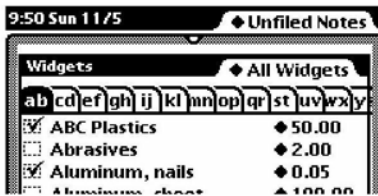
When the pen comes down in any column, the row/column cell inverts as shown in Figure 6-40.

Figure 6-40 Highlighted row



When the pen is released, if it is within the first column, the item is either checked to show that it is selected or unchecked to show that it is not. See Figure 6-41.

Figure 6-41 Selected row



When the pen tap is released within the second column, what happens next depends on the underlying data. If there are many options already available, a

CHAPTER 6

Pickers, Pop-up Views, and Overviews

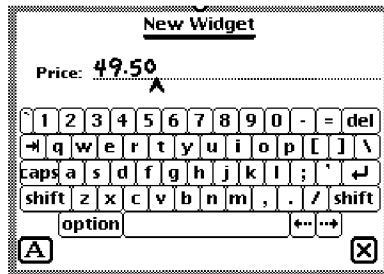
pop-up view is displayed to allow the user to select any option or enter a new one. See Figure 6-42.

Figure 6-42 Pop-up view displayed over list



If the user selects “Add new price” (or if there were one or no options already available to them), the user can enter a new price as shown in Figure 6-43.

Figure 6-43 Slip displayed for gathering input



The proto is driven by a frame contained in the `pickerDef` slot. This **picker definition** frame may or may not come from the data definition registry. The functionality it provides, however, is similar to that of any data definition: it offers all the hooks the proto needs to interpret and display the data without the proto itself knowing what the data is.

The chosen items are collected into an array, as described in “Name References” (page 5-1) in *Newton Programmer’s Reference*, which can be stored separately from the original entries. Each selection is represented in the array by a name reference that contains all information needed to display or operate on the entries. The name reference is stored as part of the selection, along with an entry alias that refers to the original entry, if there is an original entry. (See “Entry Aliases” beginning on page 12-1 for basic information on these objects.)

CHAPTER 6

Pickers, Pop-up Views, and Overviews

The picker definition (described in the next section) is a data definition frame that provides the routines to create a name reference from an entry, an entry alias, another name reference, a straight frame, or just to create a canonical empty name reference (if no data is provided). It also retrieves the data from a name reference. Finally, it provides some information about the name reference to support actions like tapping and highlighting.

You also need to define the soup to query. Both this and the query specification can be defined either in the data definition or in the list picker.

Using the Data Definitions Frame in a List Picker

The `pickerDef` slot of the list picker holds a data definition frame that determines the overall behavior of the list picker. This frame should be based on `protoNameRefDataDef` or `protoPeopleDataDef`, or at should least support the required slots.

Here is an example:

```
pickerDef:= {
  _proto:      protoNameRefDataDef,
  name:        "Widgets",
  class:       '|nameRef.widget|',
  entryType:   'widget',
  soupToQuery: "Widgets",
  querySpec:   {indexPath: 'name'},
  columns:     kColumns,
};
```

Specifying Columns

The `columns` slot hold an array that determines how the columns in the list picker are displayed. Here's an example of column specification array:

```
columns:= [{
  fieldPath:'name,// path for field to display in column
  optional:true,// not required -- unnamed widget

  tapWidth:155},// width for checkbox & name combined

{
  fieldPath:'price,// path for field to display
                in column
  optional:nil,// price is required

  tapWidth:0}];// width -- to right end of view
```

CHAPTER 6

Pickers, Pop-up Views, and Overviews

See “Column Specifications” (page 5-3) in *Newton Programmer’s Reference* for details of the slots.

Having a Single Selection in a List Picker

The key to getting single selection is that single selection is part of the picker definition and not an option of `protoListPicker`. That means the particular class of `nameRef` you use must include single selection. In general, this requires creating your own subclass of the particular name reference class.

The basic solution is to create a data definition that is a subclass of the particular class your `protoListPicker` variant will view. That data definition will include the `singleSelect` slot. As an example, suppose you want to use a `protoPeoplePopup` that just picks individual people. You could use the following code to bring up a `protoPeoplePopup` that allows selecting only one individual at a time:

```
// register the modified data definition
RegDataDef('|nameref.people.single:SIG|',
  {_proto: GetDataDefs('|nameRef.people|), singleSelect:
true});

// then pop the thing
protoPeoplePopup:New('|nameref.people.single:SIG|, [],self, [
]);

// sometime later
UnRegDataDef('|nameref.people.single:SIG|);
```

For other types of `protoListPickers` and classes, create the appropriate subclass. For example, a transport that uses `protoAddressPicker` for e-mail messages might create a subclass of `|nameRef.email|` and put that subclass symbol in the class slot of the `protoAddressPicker`.

Since many applications are likely to do this, you may cut down on code in your `installScript` and `removeScript` by registering your `dataDef` only for the duration of the picker. That would mean registering the class just before you pop the picker and unregistering after the picker has closed. You can use the `pickActionScript` and `pickCanceledScript` methods to be notified when to unregister the `dataDef`.

Having Preselected Items in a List Picker

If you want to have items that are initially selected in a list picker, use the `viewSetupDoneScript` to set up the selected array, rather than setting up the selected array in your `viewSetupFormScript` or `viewSetupChildrenScript`, then send the `Update` message to `protoListPicker` to tell it to update the display.

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Validation and Editing in protoListPicker

The built-in validation mechanism is not designed to deal with nested soup information. In general, you gain more flexibility by not using a `validationFrame` in your `pickerDef`, even if you have no nested entries. Instead, you can provide your own validation mechanism and editors:

- define a `Validate` method in your picker definition
- define an `OpenEditor` method in your picker definition
- draw a layout for each editor you require

Here is how your `Validate` method should work. The following example assumes that `pickerDef.ValidateName` and `pickerDef.ValidatePager` have been implemented:

```
pickerDef.Validate := func(nameRef, pathArray)
begin
  // keep track of any paths that fail
  local failedPaths := []

  for each index, path in pathArray do
  begin
    if path = 'name then
    begin
      // check if name validation fails
      if NOT :ValidateName(nameRef) then
        // if so, add it to array of failures
        AddArraySlot(failedPaths, path)
      end;
    else begin
      if NOT :ValidatePager(nameRef) then
        AddArraySlot(failedPaths, path)
      end;
    end;
  end;
  // return failed paths or empty array
  failedPaths;
end;
```

Here is how your `OpenEditor` method should work:

```
pickerDef.OpenEditor := func(tapInfo, context, why)
begin
  local valid = :Validate(tapInfo.nameRef,
tapInfo.editPaths) ;
  if (Length(valid) > 0) then
    // if not valid, open the editor
```

Overview Protos

6-31

CHAPTER 6

Pickers, Pop-up Views, and Overviews

```

// NOTE: returns the edit slip that is opened
GetLayout("editor.t"):new(tapInfo.nameRef,
    tapInfo.editPaths, why, self, 'EditDone, context);
else
begin
    // the item is valid, so just toggle the selection
    context:Tapped('toggle');
    nil; // Return <nil>.
end;..
end;

```

The example above assumes that the base view of the layout `editor.t` has a `New` method that opens the editor and returns the associated view.

The editor can be designed to fit your data. However, we suggest that you use a `protoFloatNGo` that is attached to the root view using `BuildContext`. You are also likely to need a callback to the `pickerDef` so it can appropriately update the edited or new item. Finally, your editor needs to update your data soup using an `Xmit` soup method so that the list picker updates.

In the `OpenEditor` example above, the last three arguments are used by the editor to send a callback to the `pickerDef` from the `viewQuitScript`. The design of the callback function is up to you. Here is an example:

```

pickerDef.EditDone := func(nameRef, context)
begin
    local valid = :Validate(tapInfo.nameRef, tapInfo.editPaths) ;
    if (Length(valid) > 0) then
    begin
        // Something failed. Try and revert back to original
        if NOT :ValidatePager(nameRef) AND
            self.('[pathExpr: savedPagerValue, nameRef]) = nameRef then
            nameRef.pager := savedPagerValue.pager;

        context:Tapped(nil); // Remove the checkmark
    end;
    else
        // The nameRef is valid, so select it.
        context:Tapped('select');

        // Clear the saved value for next time.
        savedPagerValue := nil;
    end;
end;

```

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Changing the Font of protoListPicker

The mechanism described here will probably change in the future. Eventually you may be able to set a `viewFont` slot in the list picker itself, just as you can set `viewLineSpacing` now. In the meantime, you need a piece of workaround code. You must set the `viewFont` of the list picker and also include this workaround code.

Give the list picker the following `viewSetupDoneScript`:

```
func()
begin
  if listBase then
    SetValue(listBase, 'viewFont', viewFont) ;

    inherited:?viewSetupDoneScript();
  end;
```

This sets the `viewFont` of the `listbase` view to the view font of the list picker. You cannot rely on the `listbase` view always being there (hence the test).

Using protoSoupOverview

For the most part, you use this proto like `protoOverview`, except that it is set up to use a soup cursor, and, so, is easier to use. See “Using `protoOverview`” (page 6-24) for information.

Determining Which protoSoupOverview Item Is Hit

There is a method of `protoSoupOverview` called `HitItem` that is called whenever an item is tapped. The method is defined by the overview and you should call the inherited method. Also note that `HitItem` gets called regardless of where in the line a tap occurs. If the tap occurs in the checkbox (that is, if `x` is less than `selectIndent`), you should do nothing other than calling the inherited functions, because the inherited function will handle the tap, otherwise you should do something appropriate.

The method is passed the index of the item that is hit. The index is relative to the item displayed at the top of the displayed list. This item is always the current entry of the cursor used by `protoSoupOverview`, so you can find the actual soup entry by cloning the cursor and moving it.

```
func(itemIndex, x, y)
begin
  // MUST call the inherited method for bookkeeping
  inherited:HitItem(itemIndex, x, y);
```

CHAPTER 6

Pickers, Pop-up Views, and Overviews

```

    if x > selectIndent then
    begin
    // get a temporary cursor based on the cursor used
    // by soup overview
        local tCursor := cursor:Clone();

    // move it to the selected item
        tCursor:Move(itemIndex) ;

    // move the application's detail cursor to the
    // selected entry
        myBaseApp.detailCursor:Goto(tCursor:Entry());

    // usually you will close the overview and switch to
    // some other view
        self:Close();
    end;
    // otherwise, just let them check/uncheck
    // which is the default behavior
end

```

Displaying the protoSoupOverview Vertical Divider

The mechanism for bringing up the vertical divider line was not correctly implemented in `protoSoupOverview`. You can draw one in as follows:

```

// set up a cached shape for efficiency
mySoupOverview.cachedLine := nil;

mySoupOverview.viewSetupDoneScript := func()
begin
    inherited:?viewSetupDoneScript();

    local bounds := :LocalBox();
    cachedLine := MakeRect(selectIndent - 2, 0,
        selectIndent - 1, bounds.bottom);
end;

mySoupOverview.viewDrawScript := func()
begin
    // MUST call inherited script
    inherited:?viewDrawScript();

    :DrawShape(cachedLine,
        {penPattern: vfNone, fillPattern: vfGray});
end;

```

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Roll Protos

You can use the protos described in this section to present roll views in your applications. A roll view is one that contains several discrete subviews that are arranged vertically. The roll can be viewed in overview mode, in which each subview is represented by a one-line description. Any or all of the subviews can be expanded to full size. The individual subviews are contained in objects based on `protoRollItem`.

The Newton system software provides the following roll protos:

- The `protoRoll` provides a roll-like view that includes a series of individual items. The user can see the items either as a collapsed list of one-line overviews or as full-size views. When the user taps an overview line, all the full-size views are displayed, with the tapped view shown at the top of the roll. For information about the slots and methods for this proto, see “`protoRoll`” (page 5-112) in *Newton Programmer’s Reference*. Figure 6-44 shows an example of a `protoRoll`.

Figure 6-44 A `protoRoll` example

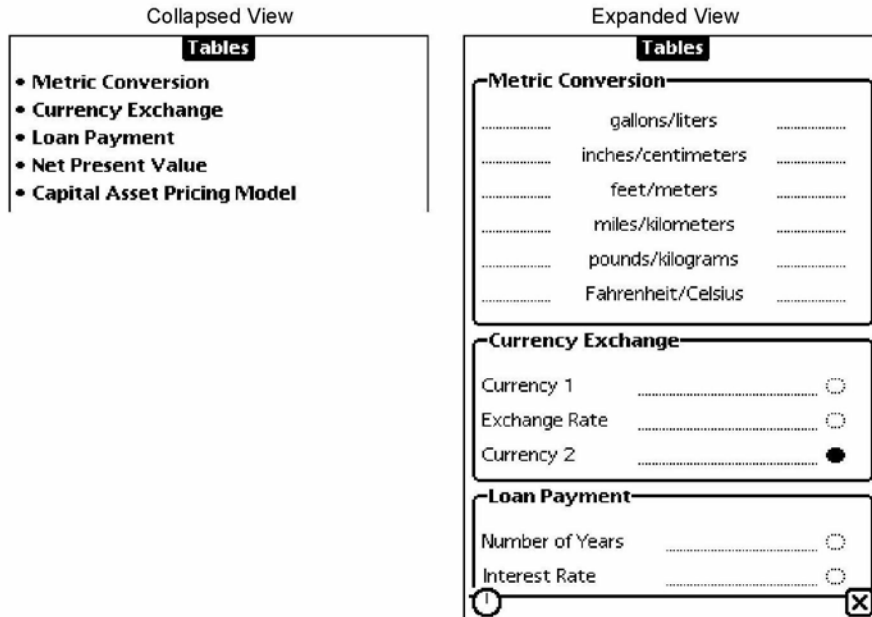
- Overview of item 1
- Overview of item 2
- Overview of item 3
- Overview of item 4
- Overview of item 5

- The `protoRollBrowser` is similar to `protoRoll`, except that `protoRollBrowser` creates a self-contained application based on the `protoApp`, described in “`protoApp`” (page 1-2) in *Newton Programmer’s Reference*. See “`protoRollBrowser`” (page 5-116) in *Newton Programmer’s Reference* for information about the slots and methods for this proto. Figure 6-45 shows an example of a `protoRollBrowser`:

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Figure 6-45 A protoRollBrowser example



View Classes

There are two view classes that you use for pickers:

- The `c1Outline` view class displays an expandable text outline. Figure 6-46 shows an example.

Figure 6-46 Example of an expandable text outline

```

My First Heading
  First level 2 head
    Another level 2 head
      Wow—a third level!
Second main heading
Third main heading
    
```

CHAPTER 6

Pickers, Pop-up Views, and Overviews

- The `clMonthView` view class displays a monthly calendar. Figure 6-47 shows an example.

Figure 6-47 Example of a month view

```

S M T W T F S
Selected day — 1 2 3 4 5
                6 7 8 9 10 11 12
Current day — 13 14 15 16 17 18 19
                20 21 22 23 24 25 26
                27 28 29 30 31

```

Specifying the List of Items for a Popup

You specify the item list for `protoPicker`, `protoTextList`, `protoPopUpButton`, `proptoPopupInPlace`, and `PopUpMenu` in an array. In the simplest case, this is an array of strings, but it can contain different kinds of items:

simple string	A string. You can control the pickability of a text item or add a mark to the display by specifying the text in a frame, as described in Table 6-1 (page 6-38).
bitmap	A bitmap frame or a NewtonScript frame, as returned from the <code>GetPictAsBits</code> compile-time function. You can control the pickability of the item or add a mark to the display by placing the bitmap in a frame, as described in Table 6-1 (page 6-38).
icon with string	A frame that specifies both a string and an icon, as described in Table 6-2 (page 6-38).
separator line	An instruction to display a line that runs the width of the picker. To display a dashed gray line, specify the symbol <code>'pickSeparator</code> . For a solid black line, specify the symbol <code>'pickSolidSeparator</code> .
two-dimensional grid	A frame describing the grid item, as described in Table 6-3 (page 6-39).

If all the items in the picker list cannot fit into the view, the user can scroll the list to see more items.

Table 6-1 describes the frame used to specify simple string and bitmap items in the picker list.

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Table 6-1 Item frame for strings and bitmaps

Slot name	Description
<code>item</code>	The item string or bitmap reference.
<code>pickable</code>	A flag that determines whether the item is pickable. Specify <code>non-nil</code> if you want the item to be pickable, or <code>nil</code> if you don't want the item pickable. Not-pickable items appear in the list but are not highlighted and can't be selected.
<code>mark</code>	A character displayed next to an item when it's chosen. Specify a dollar sign followed by the character you want to use to mark this item if it is chosen. For example, <code>\$_\uFC0B</code> specifies the check mark symbol. (You can use the constant <code>kCheckMarkChar</code> to specify the check mark character.)
<code>fixedHeight</code>	When you give a bitmap, you can give this slot for the first item in order to force all items to be the same size. If you use bitmaps in a list that can become large enough to scroll, you should specify the <code>fixedHeight</code> slot for every item. You can also use slot this for any item to specify a height different from other items.

Table 6-2 describes the frame used to specify a string with an icon in the picker list.

Table 6-2 Item frame for string with icon

Slot name	Description
<code>item</code>	The item string.
<code>icon</code>	A bitmap frame, as returned from the compile-time function <code>GetPictAsBits</code> . The bitmap is displayed to the left of the text, and the text is drawn flush against it, unless the <code>indent</code> slot is specified.

continued

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Table 6-2 Item frame for string with icon (continued)

Slot name	Description
<code>indent</code>	An integer that defines a text indent to use for this item and subsequent icon/string items. This integer specifies the number of pixels to indent the text from the left side of the picker view. You can use it to line up a number of text items that may have icons of varying width. Specify <code>-1</code> to cancel the indent effect for the current and subsequent text items. The icon is always centered within the indent width.
<code>fixedHeight</code>	You can give this slot for the first item in order to force all items to be the same size. If you use icons in a list that can become large enough to scroll, you should specify the <code>fixedHeight</code> slot for every item. You can also use this slot for any item to specify a height different from other items. (When you use <code>PopupMenu</code> , you must specify a <code>fixedHeight</code> slot for the first item, because <code>PopupMenu</code> ignores the height of the icon.)

Table 6-3 describes the frame required to specify a two-dimensional grid item in the picker list.

Table 6-3 Item frame for two-dimensional grid

Slot Name	Description
<code>bits</code>	A binary object representing the bitmap of the grid item. A bitmap is returned in the <code>bits</code> slot in the frame returned by the compile-time function <code>GetPictAsBits</code> . The bitmap is a complete picture of the grid item, including the lines between cells and the border around the outside of the cells. There must be no extra white space outside the border. Each cell must be the same size and must be symmetrical.
<code>bounds</code>	The bitmap bounds frame, from the <code>bounds</code> slot in the frame returned by <code>GetPictAsBits</code> .
<code>width</code>	The number of columns in the grid (must be non-zero).
<code>height</code>	The number of rows in the grid (must be non-zero).

continued

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Table 6-3 Item frame for two-dimensional grid (continued)

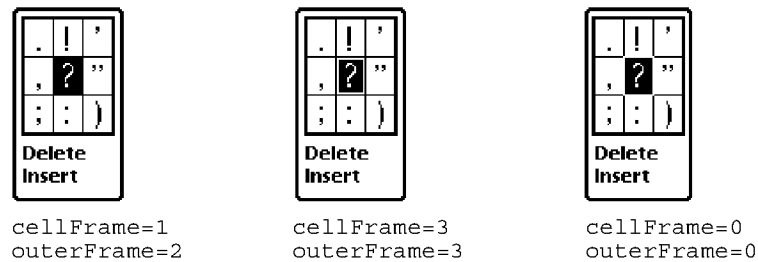
Slot Name	Description
cellFrame	Optional. The width of the separator line between cells, used for highlighting purposes. If you don't specify this slot, the default is 1 pixel.
outerFrame	Optional. The width of the border line around the cells, used for highlighting purposes. If you don't specify this slot, the default is 2 pixels.
mask	Optional. A binary object representing the bits for a bitmap mask. This mask is used to restrict highlighting, or for special hit-testing. The mask must be exactly the same size as the bitmap. Cells in the grid are highlighted only if the position tapped is "black" in the mask.

Note

Picker items can include 1.x bitmaps but not 2.0 shapes. ♦

When a cell is highlighted in a two-dimensional picker item, only the part of the cell inside the cell frame lines is inverted. You can vary the highlighting effect by changing the values of the `cellFrame` and `outerFrame` slots, which control how much unhighlighted space to leave for the cell frame lines. An example of how these values affect cell highlighting is shown in Figure 6-48.

Figure 6-48 Cell highlighting example for `protoPicker`



CHAPTER 6

Pickers, Pop-up Views, and Overviews

Summary

The following sections summarize the reference information in this chapter.

General Picker Protos

protoPopupButton

```

aProtoPopupButton := {
  _proto:          protoPopupButton,
  viewFlags:       flags,
  viewBounds:      boundsFrame,
  viewJustify:     justificationFlags,
  text:            string, // text inside button
  popup:           array,  // items in list
  ButtonClickScript: function, // called on button tap
  PickActionScript: function, // returns item selected
  PickCancelledScript: function, // user cancelled
  ...
}

```

protoPopInPlace

```

aProtoPopInPlace := {
  _proto:          protoPopInPlace,
  viewBounds:      boundsFrame,
  viewFlags:       constant,
  viewJustify:     justificationFlags,
  text:            string, // text inside button
  popup:           array,  // items in list
  PickActionScript: function, // returns selected item
  PickCancelledScript: function, // user cancelled
  ...
}

```

protoLabelPicker

```

aProtoLabelPicker := {
  _proto:          protoLabelPicker,
  viewBounds:      boundsFrame,
  viewFont:        fontSpec,
}

```

Summary

6-41

CHAPTER 6

Pickers, Pop-up Views, and Overviews

```

iconSetup:          icon frame,
labelCommands:     array, // items in list
iconBounds:        boundsFrame, // bounds of largest icon
iconIndent:        integer, // indent of text from icon
checkCurrentItem: Boolean, // true to check selected item
indent:            integer, // indent of picker from label
textIndent:        integer, // indent of text
LabelActionScript: function, // returns selected item
TextSetup:         function, // gets initial item
TextChanged:       function, // called upon item value change
UpdateText:        function, // call to change selected item
PickerSetup:       function, // called when user taps label
Popit:            function, // call to programmatically
                  // pop up picker
...
}

```

protoPicker

```

aProtoPicker := {
  _proto:          protoPicker,
  bounds:          boundsFrame,
  viewBounds:     boundsFrame, // ignored
  viewFlags:      constant,
  viewFormat:     formatFlags,
  viewJustify:    justificationFlags,
  viewFont:       fontSpec,
  viewEffect:     effectFlag,
  pickItems:      array, // items in list
  pickTextItemHeight: integer, // height reserved for items
  pickLeftMargin: integer, // margin from left of view
  pickRightMargin: integer, // margin from right of view
  pickTopMargin: integer, // margin above each item in
                        // list
  pickAutoClose: Boolean, // true to close list after pick
  pickItemsMarkable: Boolean, // true to reserve space for
                        // check mark before item
  pickMarkWidth: integer, // space to reserve for marks
  callbackContext: view, // view with pick scripts
  PickActionScript: function, // returns selected item
  PickCancelledScript: function, // user cancelled
  SetItemMark:    function, // sets char for check marks
  GetItemMark:    function, // gets char for check marks
  ...
}

```

CHAPTER 6

Pickers, Pop-up Views, and Overviews

protoGeneralPopup

```

aProtoGeneralPopup := {
  _proto:          protoGeneralPopup,
  viewBounds:     boundsFrame,
  viewFlags:      constant,
  cancelled:      Boolean, // true if user cancelled
                    // pop-up view
  context:        view,    // view with pick scripts
  New:            // open pop-up view
  Affirmative:    function, // user taps pop-up view
  PickCancelledScript: function, // called in pop-up view
                    // cancelled
  ...
}

```

protoTextList

```

aProtoTextList := {
  _proto:          protoTextList,
  viewBounds:     boundsFrame,
  viewFont:       fontSpec,
  viewFormat:     formatFlags,
  viewLines:      integer, // number of lines to show
  selection:      integer, // index of selected item
  selectedItems:  array,   // items in list
  listItems:      array,   // strings or shapes in list
  lineHeight:     array,   // height of lines in list
  isShapeList:    Boolean, // true if pict's instead of text
  useMultipleSelections:
                    Boolean, // true for multiple select
  useScroller:    Boolean, // true to include scrollers
  scrollAmounts:  array,   // units to scroll
  DoScrollScript: function, // scrolls list by offset
  ViewSetupFormScript: function, // set up list
  ButtonClickScript: function, // returns selected item
  ...
}

```

protoTable

```

aProtoTable := {
  _proto:          protoTable,
  viewBounds:     boundsFrame,

```

Summary

6-43

CHAPTER 6

Pickers, Pop-up Views, and Overviews

```

viewFormat:      formatFlags,
def:             frame,      // protoTableDef table
                  // definition frame
scrollAmount:   integer,    // number of rows to scroll
currentSelection: string,   // text of selected item
selectedCells:  array,      // selected cell indexes
declareSelf:    symbol,     // 'tabbase; do not change
ViewSetupFormScript: function, // set up table
SelectThisCell: function,   // called when cell is
selected
...
}

```

protoTableDef

```

aProtoTableDef := {
  _proto: protoTableDef,
  tabAcross:    integer,    // number of columns - must be 1
  tabDown:      integer,    // number of rows in table
  tabWidths:    integer,    // width of table
  tabHeight:    integer,    // height of rows
  tabProtos:    frame,      // references to row templates
  tabValues:    integer/array, // value/array of values for
                          // rows
  tabValueSlot: symbol,     // slot to store tabValues in
  tabUniqueSelection: Boolean, // true for single selection
  indentX:      integer,    // do not change: used internally
  TabSetUp:     function,   // called before each row set up
  ...
}

```

protoTableEntry

```

aProtoTableEntry := {
  _proto:      protoTableEntry,
  viewClass:   ciTextView,
  viewFlags:   flags,
  viewJustify: justificationFlags,
  viewTransferMode: modeOr,
  text:        string,      // text inside table
  ViewClickScript: function, // sets current selection
  ViewHiliteScript: function, // highlights selection
  ...
}

```

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Map Pickers

protoCountryPicker

```

aProtoCountryPicker := {
  _proto:      protoCountryPicker,
  viewBounds:  boundsFrame,
  autoClose:   Boolean, // true to close picker on selection
  listLimit:   integer, // maximum items listed
  PickWorld:   function, // called when selection is made
  ...
}

```

protoProvincePicker

```

aProtoProvincePicker := {
  _proto:      protoProvincePicker,
  viewFlags:   constant,
  autoClose:   Boolean, // true to close picker on selection
  listLimit:   integer, // maximum items listed
  PickWorld:   function, // called when selection is made
  ...
}

```

protoStatePicker

```

aProtoStatePicker := {
  _proto:      protoStatePicker,
  viewFlags:   constant,
  autoClose:   Boolean, // true to close picker on selection
  PickWorld:   function, // called when selection is made
  listLimit:   integer, // maximum items listed
  ...
}

```

protoWorldPicker

```

aProtoWorldPicker := {
  _proto:      protoWorldPicker,
  viewBounds:  boundsFrame,
  autoClose:   Boolean, // true to close picker on selection
  listLimit:   integer, // maximum items listed
  PickWorld:   function, // called when selection is made
  ...
}

```

Summary

6-45

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Text Picker Protos

protoTextPicker

```

aProtoTextPicker := {
  _proto:          protoTextPicker,
  label:           string,          // picker label
  indent:          integer,         // indent
  labelFont:       fontSpec,       // font for label
  entryFont:       fontSpec,       // font for picker line
  Popit:           function,       // user tapped picker
  PickActionScript: function,     // returns selected item
  PickCancelledScript: function, // user cancelled picker
  TextSetup:       function,       // returns text string
  ...
}

```

protoDateTextPicker

```

aProtoDateTextPicker := {
  _proto:          protoDateTextPicker,
  label:           string,          // picker label
  date:            integer,         // initial and currently
                                     // selected date
  longFormat:      symbol,         // format to display date
  shortFormat:     symbol,         // format to display date
  PickActionScript: function,     // returns selected item
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

protoDateDurationTextPicker

```

aProtoDateDurationTextPicker := {
  _proto:          protoDateDurationTextPicker,
  label:           string,          // picker label
  labelFont:       fontSpec,       // display font
  entryFont:       fontSpec,       // picked entry font
  startTime:       integer,         // initial start date
  stopTime:        integer,         // initial end date
  longFormat:      symbol,         // format to display date
  shortFormat:     symbol,         // format to display date
}

```


CHAPTER 6

Pickers, Pop-up Views, and Overviews

```

PickActionScript:    function, // returns selected item
PickCancelledScript: function, // user cancelled picker
...
}

```

protoRepeatDateDurationTextPicker

```

aProtoRepeatDateDurationTextPicker := {
  _proto:          protoRepeatDateDurationTextPicker,
  label:           string, // picker label
  startTime:       integer, // initial start date
  stopTime:        integer, // initial end date
  longFormat:      symbol, // format to display date
  shortFormat:     symbol, // format to display date
  repeatType:      constant, // how often meeting meets
  mtgInfo:         constant, // repeating meetings
  PickActionScript: function, // returns selected item
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

protoDateNTimeTextPicker

```

aProtoDateNTimeTextPicker := {
  _proto:          protoDateNTimeTextPicker,
  label:           string, // picker label
  date:            integer, // initial date/time
  format:          symbol, // format to display time
  longFormat:      symbol, // format to display date
  shortFormat:     symbol, // format to display date
  increment:       integer // amount to change time
  PickActionScript: function, // returns selected item
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

protoTimeTextPicker

```

aProtoTimeTextPicker := {
  _proto:          protoTimeTextPicker,
  label:           string, // picker label
  labelFont:       fontSpec, // label display font
  entryFont:       fontSpec, // picked entry font
  indent:          integer, // amount to indent text

```

Summary

6-47

CHAPTER 6

Pickers, Pop-up Views, and Overviews

```

time:                integer,    // initial start time
format:              symbol,    // format to display time
increment:           integer,   // increment to change
                        // time for taps
PickActionScript:   function,  // returns selected item
PickCancelledScript: function, // user cancelled picker
...
}

```

protoDurationTextPicker

```

aProtoDurationTextPicker := {
  _proto:             protoDurationTextPicker,
  label:              string,    // picker label
  startTime:          integer,   // initial start time
  stopTime:           integer,   // initial end time
  format:             symbol,    // format to display time
  increment:          integer,   // increment to change
                        // time for taps
  PickActionScript:   function,  // returns selected item
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

protoTimeDeltaTextPicker

```

aProtoTimeDeltaTextPicker := {
  _proto:             protoTimeDeltaTextPicker,
  label:              string,    // picker label
  time:               integer,   // initial time
  labelFont:          fontSpec,  // label display font
  entryFont:          fontSpec,  // picked entry font
  indent:             integer,   // amount to indent text
  increment:          integer,   // increment to change
                        // time for taps
  minValue:           integer,   // minimum delta value
  PickActionScript:   function,  // returns selected item
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

CHAPTER 6

Pickers, Pop-up Views, and Overviews

protoMapTextPicker

```

aProtoMapTextPicker := {
  _proto:          protoMapTextPicker,
  label:           string,          // picker label
  labelFont:       fontSpec,       // label display font
  entryFont:       fontSpec,       // picked entry font
  indent:          integer,        // amount to indent text
  params:          frame,
  PickActionScript: function,     // returns selected item
  PickCancelledScript: function,  // user cancelled picker
  ...
}

```

protoCountryTextPicker

```

aProtoCountryTextPicker := {
  _proto:          protoCountryTextPicker,
  label:           string,          // picker label
  labelFont:       fontSpec,       // label display font
  entryFont:       fontSpec,       // picked entry font
  indent:          integer,        // amount to indent text
  params:          frame,
  PickActionScript: function,     // returns selected item
  PickCancelledScript: function,  // user cancelled picker
  ...
}

```

protoUSstatesTextPicker

```

aProtoUSstatesTextPicker := {
  _proto:          protoUSstatesTextPicker,
  label:           string,          // picker label
  labelFont:       fontSpec,       // label display font
  entryFont:       fontSpec,       // picked entry font
  indent:          integer,        // amount to indent text
  params:          frame,
  PickActionScript: function,     // returns selected item
  PickCancelledScript: function,  // user cancelled picker
  ...
}

```

Summary

6-49

CHAPTER 6

Pickers, Pop-up Views, and Overviews

protoCitiesTextPicker

```

aProtoCitiesTextPicker := {
  _proto:          protoCitiesTextPicker,
  label:           string,      // picker label
  labelFont:      fontSpec,    // label display font
  entryFont:      fontSpec,    // picked entry font
  indent:         integer,     // amount to indent text
  params:         frame,
  PickActionScript: function, // returns selected item
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

protoLongLatTextPicker

```

aProtoLongLatTextPicker := {
  _proto:          protoLongLatTextPicker,
  label:           string,      // picker label
  latitude:        integer,     // initial latitude
  longitude:       integer,     // initial longitude
  labelFont:      fontSpec,    // label display font
  entryFont:      fontSpec,    // picked entry font
  indent:         integer,     // amount to indent text
  PickActionScript: function, // returns selected item
  PickCancelledScript: function, // user cancelled picker
  worldClock:     boolean     // do not change
  ...
}

```

Date, Time, and Location Pop-up Views

protoDatePopup

```

aProtoDatePopup := {
  _proto:          protoDatePopup,
  New:            function,     // creates pop-up view
  PickActionScript: function, // returns selected item
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

CHAPTER 6

Pickers, Pop-up Views, and Overviews

protoDatePicker

```

aProtoDatePicker := {
  _proto:      protoDatePicker,
  selectedDates: array,    // selected date
  DateChanged: function,  // called when date is selected
  Refresh:     function,  // update view with new dates
  ...
}

```

protoDateNTimePopup

```

protoDateNTimePopup := {
  _proto:      protoDateNTimePopup,
  New:         function,    // creates pop-up view
  NewTime:     function,    // called when time changes
  PickActionScript: function, // returns selected item
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

protoDateIntervalPopup

```

protoDateIntervalPopup := {
  _proto:      protoDateIntervalPopup,
  New:         function,    // creates pop-up view
  NewTime:     function,    // called when time changes
  PickActionScript: function, // returns selected item
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

protoMultiDatePopup

```

protoMultiDatePopup := {
  _proto:      protoMultiDatePopup,
  New:         function,    // creates pop-up view
  PickActionScript: function, // returns selected item
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

Summary

6-51

CHAPTER 6

Pickers, Pop-up Views, and Overviews

protoYearPopup

```

protoYearPopup := {
  _proto:      protoYearPopup,
  New:         function, // creates pop-up view
  NewYear:     function, // called when year changes
  DoneYear:    function, // called on close box tap
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

protoTimePopup

```

protoTimePopup := {
  _proto:      protoTimePopup,
  New:         function, // creates pop-up view
  NewTime:     function, // called when time changes
  PickActionScript: function, // returns selected item
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

protoAnalogTimePopup

```

protoAnalogTimePopup := {
  _proto:      protoAnalogTimePopup,
  New:         function, // creates pop-up view
  NewTime:     function, // called when time changes
  PickActionScript: function, // returns selected item
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

protoTimeDeltaPopup

```

protoTimeDeltaPopup := {
  _proto:      protoTimeDeltaPopup,
  New:         function, // creates pop-up view
  PickActionScript: function, // returns selected item
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

CHAPTER 6

Pickers, Pop-up Views, and Overviews

protoTimeIntervalPopup

```

protoTimeIntervalPopup := {
  _proto:      protoTimeIntervalPopup,
  New:         function, // creates pop-up view
  PickActionScript: function, // returns selected item
  PickCancelledScript: function, // user cancelled picker
  ...
}

```

Number Pickers

protoNumberPicker

```

aProtoNumberPicker := {
  _proto:      protoNumberPicker,
  minValue:    integer, // minimum value in list
  maxValue:    integer, // maximum value in list
  value:       integer, // currently selected value
  showLeadingZeros: Boolean, // true to show leading zeros
  prepareForClick: function, // called after click is
                          // processed
  ClickDone:   function, // called after click is
                          // processed
  ...
}

```

Picture Picker

protoPictIndexer

```

aProtoPictIndexer := {
  _proto:      protoPictIndexer,
  viewBounds : boundsFrame,
  viewJustify: justificationFlags,
  viewFormat : formatFlags,
  icon:        bitmap, // bitmap with objects
                          // arranged vertically
  iconBBox:    boundsFrame, // bitmap bounds within view
  numIndices:  integer, // # of objects in bitmap
  curIndex:    integer, // index of current item
  IndexClickScript: function, // user taps bitmap
  ...
}

```

Summary

6-53

CHAPTER 6

Pickers, Pop-up Views, and Overviews

Overview Protos

protoOverview

```

aProtoOverview := {
  _proto:          protoOverview,
  viewBounds :    boundsFrame,
  viewFlags :     constant,
  viewFont :      fontSpec,
  lineHeight:     integer, // height of items in pixels
  selectIndent:   integer, // specifies left margin
  nothingCheckable: Boolean, // true for no checkboxes
  SelectItem:     function, // to record selected items
  SetupAbstracts: function, // set up entry
  Abstract:       function, // return shape given entry
  HitItem:        function, // called when item is tapped
  IsSelected:     function // Return true if the item is
                      // selected
  cursor:         cursor, // cursor for the items
  CheckState:     function, // determines if selectable
  Scroller:       function, // implement scrolling here
  SelectItem:     function, // records selected items
  viewSetupChildrenScript:
                  function, // Calls SetupAbstracts
  ...
}

```

protoSoupOverview

```

aProtoSoupOverview := {
  _proto:          protoSoupOverview,
  autoDeselect:   Boolean, // whether to deselect when
                      // the pen leaves an item
  cursor:         cursor, // cursor for the entries
  Scroller:       function, // implement scrolling here
  SelectItem:     function, // records selected items
  Abstract:       function, // return shape given entry
  IsSelected:     function, // returns true if selected
  ForEachSelected: function, // called for each selected
                      // item
  ...
}

```


CHAPTER 6

Pickers, Pop-up Views, and Overviews

protoListPicker

```

aProtoListPicker := {
  _proto:          protoListPicker,
  declareSelf :   symbol,    // Set to 'pickBase
  defaultJustification :constant,
  viewFlags :     constant,
  viewBounds :    boundsFrame,
  lineHeight:     integer,    // height of items in pixels
  listFormat:     formatFlags,
  pickerDef:      frame,      // defines list behavior
  selected:       array,      // references to selected items
  soupToQuery:    string,     // union soup to query
  querySpec:      frame,      // query to use
  selected:       array,      // modified as user selects
                                // and deselects item
  singleSelect:   Boolean,    // single selection if non-nil
  suppressNew:    Boolean,    // suppress New button if non-nil
  suppressScrollers: Boolean, // suppress scroller if
                                // non-nil
  suppressAZTabs: Boolean,    // suppress tabs if non-nil
  suppressFolderTabs: Boolean, // suppress if non-nil
  suppressSelOnlyCheckbox: Boolean, // suppress if non-nil
  suppressCloseBox: Boolean,  // suppress if non-nil
  suppressCounter: Boolean,   // suppress if non-nil
  reviewSelections: Boolean,  // Selected Only if non-nil
  readOnly:       Boolean,    // items are read-only if
                                // non-nil
  dontPurge:      Boolean,    // keep unselected refs if
                                // non-nil
  soupChangeSymbol: symbol,    // for RegSoupChange method
  SoupEnters:     function,    // syncs up changed soup
  SoupLeaves:     function,    // syncs up changed soup
  SetNowShowing: function,    // set Selected Only
  AddFakeItem:    function,    // add item to array; update
                                // screen
  GetSelected:    function,    // returns clone of selected
                                // array
  ...
}

```

Summary

6-55

CHAPTER 6

Pickers, Pop-up Views, and Overviews

protoNameRefDataDef

```

aProtoNameRefDataDef := {
  _proto:      protoNameRefDataDef,
  name:        string,      // name to identify picker in
                                // top left corner
  class:       symbol,      // specify class for new name
                                // references
  entryType:   symbol,      // class for new soup entries
  columns:     array,       // column specifications
  singleSelect: Boolean,    // single selection if non-nil
  soupToQuery: string,      // union soup to query
  querySpec:   frame,       // query to use
  validationFrame: frame,   // checks validity of entry
  MakeCanonicalNameRef: function, // make blank name ref
  MakeNameRef:  function,   // make name reference
  Get:          function,   // returns data from specified
                                // object
  GetPrimaryValue: function, // retrieves data from object
  HitItem:      function,   // called when item tapped
  MakePopup:    function,   // called before making pop-up
                                // view
  Tapped:       function,   // called when tap has been
                                // handled
  New:          function,   // called when tap on New button
  DefaultOpenEditor: function, // open an edit view
  OpenEditor:   function,   // open an custom edit view
  NewEntry:     function,   // returns a new soup entry
  ModifyEntry:  function,   // returns a modified soup entry
  Validate:     function,   // validates paths
  ...
}

```

protoPeopleDataDef

```

aProtoPeopleDataDef := {
  _proto:      protoPeopleDataDef,
  entryType:   symbol,      // class for new soup entries
  soupToQuery: string,      // union soup to query
  primaryPath: symbol,      // the primary path column
  primaryPathMapper: frame, // maps entry class to data
  Equivalent:  function,   // compares two name refs
}

```

CHAPTER 6

Pickers, Pop-up Views, and Overviews

```

Validate:          function, // returns array of invalid
                    // refs
ModifyEntryPath:  function, // entry modification of Names
GetRoutingInfo:   function, // retrieves routing info
GetItemRoutingFrame: function, // converts routing info
GetRoutingTitle:  function, // creates target string
PrepareForRouting: function, // strips extra info
...
}

```

protoPeoplePicker

```

aProtoPeoplePicker := {
  _proto:          protoPeoplePicker,
  class:           symbol, // type of data to display
  selected:        array, // references to selected items
  ...
}

```

protoPeoplePopup

```

aProtoPeoplePicker := {
  _proto:          protoPeoplePicker,
  class:           symbol, // type of data to display
  selected:        array, // references to selected items
  context:         symbol, // view with PickActionScript
                    // method
  options:         array, // options for protoListPicker
  PickActionScript: function,
                    // called when pop-up is closed
  ...
}

```

Roll Protos

protoRoll

```

aProtoRoll := {
  _proto:          protoRoll,
  viewFlags:       constant,
  viewBounds:      boundsFrame,
  items:           array, // templates for roll items
}

```

Summary

6-57

CHAPTER 6

Pickers, Pop-up Views, and Overviews

```

allCollapsed: Boolean, // roll collapsed if non-nil
index:        integer, // index of item to start
               // display at
declareSelf:  symbol,  // 'roll – do not change
...
}

```

protoRollBrowser

```

aProtoRollBrowser := {
  _proto:      protoRollBrowser,
  viewBounds:  boundsFrame,
  viewJustify: justificationFlags,
  viewFormat:  formatFlags,
  title:       string, // text for title at top of roll
  rollItems:   array,  // templates for roll items
  rollCollapsed: Boolean, // roll collapsed if non-nil
  rollIndex:   integer, // index of item to start
               // display at
declareSelf:  symbol,  // 'base – do not change
...
}

```

protoRollItem

```

aProtoRollItem := {
  _proto:      protoRollItem,
  viewBounds:  boundsFrame,
  viewJustify: justificationFlags,
  viewFormat:  formatFlags,
  overview:    string, // text for one-line overview
  height:      integer, // height of the view in pixels
  stepChildren: Boolean, // child views for this roll item
...
}

```

View Classes

clOutlineView

```

myOutline:= {...
  viewClass:  clOutline,
  viewBounds: boundsFrame,

```

CHAPTER 6

Pickers, Pop-up Views, and Overviews

```

browsers:      array,          // frame with array of outline
                // items
viewFont:      fontSpec,
viewFlags :    constant,
viewFormat:    formatFlags,
clickSound:    frame,          // sound frame for taps
OutlineClickScript: function, //called when user taps item
...
}

```

clMonthView

```

theMonth := {...
viewclass: clMonthView,
viewBounds: boundsFrame,
viewflags:  constant,
labelFont:  fontSpec,
dateFont:   fontSpec,
selectedDates: array,
viewSetupFormScript: function,
...
}

```

Functions

```

PopupMenu (list, options)
IsNameRef (item)
AliasFromObj (item)
EntryFromObj (item)
ObjEntryClass (item)

```


C H A P T E R 7

Controls and Other Protos

Controls are software objects that provide various user interface capabilities, including scrolling, selection buttons, and sliders. You use the controls and other protos described in this chapter to add these features to your NewtonScript applications.

This chapter gives a general description of the controls and related protos provided in Newton System Software. For a detailed description of these protos, including the slots that you use to set to implement each, see “Controls Reference” (page 6-1) in *Newton Programmer’s Reference*.

This chapter provides information about the following controls and protos:

- horizontal and vertical scrollers
- boxes and buttons
- alphabetical selection tabs
- gauges and sliders
- time-setting displays
- special views
- view appearance enhancements
- status bars

Controls Compatibility

The 2.0 release of Newton System Software includes a number of new protos, including:

- four new scroller protos: `protoHorizontal2DScroller`, `protoLeftRightScroller`, `protoUpDownScroller`, and `protoHorizontalUpDownScroller`
- two new buttons: `protoInfoButton` and `protoOrientation`
- two selection tab protos: `protoAZTabs` and `protoAZVertTabs`

CHAPTER 7

Controls and Other Protos

- four new date and time protos: `protoDigitalClock`, `protoSetClock`, `protoNewSetClock`, and `protoAMPMCluster`
- two special view protos: `protoDragger` and `protoDragNGo`

Scroller Protos

Scrollers allow the user to move vertically or horizontally through a display that is bigger than the view. The Newton System Software provides a number of scrollers to allow users to scroll their views.

All scroller protos are implemented in the same way; that is, they use the same methods and slots. These scrollers are not linked or related to the scroll arrows on the built-in button bar. For individual descriptions of the scroller protos, see “Scroller Protos” (page 7-2) in *Newton Programmer’s Reference*. This section describes how to implement scrollers in your applications.

The scroller protos do not perform the actual scrolling of data in a view; they simply display and maintain the arrows as the user taps them. To scroll data in a view, you can use the following protos in your applications:

- The `protoHorizontal2DScroller` is centered at the bottom of a view and provides both horizontal and vertical scroll arrows. For more information about the slots and methods for this scroller, see “`protoHorizontal2DScroller`” (page 6-2) in *Newton Programmer’s Reference*. Figure 7-1 shows an example of a `protoHorizontal2DScroller` view.

Figure 7-1 A `protoHorizontal2DScroller` view



- The `protoLeftRightScroller` is centered at the bottom of a view and provides horizontal scroll arrows. For more information about the slots and methods for this scroller, see “`protoLeftRightScroller`” (page 6-5) in *Newton Programmer’s Reference*. Figure 7-2 shows an example of a `protoLeftRightScroller` view.

Figure 7-2 A `protoLeftRightScroller` view



CHAPTER 7

Controls and Other Protos

- The `protoUpDownScroller` is centered on the right side of a view and provides vertical scroll arrows. For more information about the slots and methods for this scroller, see “`protoUpDownScroller`” (page 6-5) in *Newton Programmer’s Reference*. Figure 7-3 shows an example of a `protoHorizontal2DScroller` view.

Figure 7-3 A `protoUpDownScroller` view



- The `protoHorizontalUpDownScroller` is centered at the bottom of a view and provides vertical scroll arrows. For more information about the slots and methods for this scroller, see “`protoHorizontalUpDownScroller`” (page 6-6) in *Newton Programmer’s Reference*. Figure 7-4 shows an example of a `protoHorizontalUpDownScroller` view.

Figure 7-4 A `protoHorizontalUpDownScroller` view



Implementing a Minimal Scroller

To implement a minimal scroller, all that you have to define is a `ViewScroll2DScript` method in your scroller template. This method is called whenever the user taps one of the scroll arrows in the scroller view. Your `ViewScroll2DScript` method must perform the actual scrolling of the contents of some other view, which you usually do by calling the `SetOrigin` method.

For more information on the `ViewScroll2DScript` method, see “`ViewScroll2DScript`” (page 6-3) in *Newton Programmer’s Reference*. For more information on the `SetOrigin` method, see “`SetOrigin`” (page 2-48) in *Newton Programmer’s Reference*.

Automatic Arrow Feedback

All of the scroller protos can provide visual feedback to the user indicating that there is more information to see. This feedback is handled automatically for you if you provide three additional slots in your scroller template: `scrollRect`,

CHAPTER 7

Controls and Other Protos

`viewRect`, and `dataRect`. Each of these slots is a bounds frame with the following form:

```
{left: 0, top: 0, right: 10, bottom: 10}
```

You usually create these bounds frame slots with the utility function `SetBounds`, which is described in “SetBounds” (page 2-34) in *Newton Programmer’s Reference*.

When you use these slots, the scroller protos highlight the scrolling arrows automatically to indicate to the user that more data can be viewed by tapping on the highlighted arrows.

Each of the bounds frame slots serves a specific purpose in the scroller, as shown in Table 7-1. The next section provides several examples of setting the values of these slots for different scrolling effects.

Table 7-1 Scroller bounds frame slots

Slot name	Description
<code>scrollRect</code>	Specifies the scrollable area, which is the total area that the user can see, or scroll over, with the scroller.
<code>viewRect</code>	Specifies the part of the scrollable area that the user can see at any one time. This is usually smaller than the area specified by <code>scrollRect</code> .
<code>dataRect</code>	Specifies the portion of the <code>scrollRect</code> that contains data. In simple cases, this is the same as <code>scrollRect</code> .

Scrolling Examples

This section presents several simple examples of setting the bounds frame slots in your scroller to allow scrolling.

Scrolling Lines of Text

To scroll lines of text, you set the values of the three scroller bounds frames as required for your application. For example, if you have 20 text items in a vertical list and you want to show 6 of the items at a time, you need to set the slot values as follows:

```
scrollRect: SetBounds(0, 0, 0, 20) // 20 possible lines
viewRect:   SetBounds(0, 0, 0, 6)  // show 6 at a time
dataRect:   SetBounds(0, 0, 0, 20)
```

CHAPTER 7

Controls and Other Protos

Scrolling in the Dates Application

Scrolling in the Dates application allows the user to see the 24 hours in a day, 7 hours at a time. When there is only interesting data in a certain range of the day, the application sets the `dataRect` for that time frame. This tells the scroller to blacken a scroll arrow when the data time frame is not displayed in the `viewRect`, providing additional visual feedback to the user.

```
scrollRect: SetBounds(0, 0, 0, 24) // 24 hours per day
viewRect:   SetBounds(0, 0, 0, 7)  // show 7 at a time
dataRect:   SetBounds(0, 0, 0, 10) // meeting from 9-10
```

Scrolling In a Graphics Application

A final example shows scrolling in a graphics application. This example shows a total scrollable area of 200 pixels by 200 pixels, of which a 50 pixel by 50 pixel area is shown at any one time. In this example, an object of interest (data) is located at (100,100).

```
                // total area is 200 by 200
scrollRect: SetBounds( 0, 0, 200, 200)
                // show a 50 by 50 area at a time
viewRect:   SetBounds( 0, 0, 50, 50)
                // there's something at location (100,100)
dataRect:   SetBounds(100, 100, 110, 110)
```

Scroll Amounts

Whenever the `ViewScroll2DScript` method is called, the scroller proto increments the `viewRect` by 1. For example, in the Dates application example, each time the user taps an arrow, the `viewRect` is moved up or down by 1 hour.

In the graphics application example, each time the user taps an arrow, the `viewRect` is moved up or down by 1 pixel. Since scrolling by 1 pixel at a time is too slow, you need to be able to adjust the scrolling amount for certain applications. To do so, you change the value of the `scrollAmounts` slot, which is an array of three values. The default value of this slot is:

```
[1, 1, 1]
```

The first value in the `scrollAmounts` array specifies the amount to scroll for a single tap. The second value specifies the amount to scroll when the user holds down on the arrow (accelerated scrolling), and the third value specifies the amount to scroll for a double tap. For a typical graphics application, you can use values like the following:

```
[10, 50, 50]
```

Scroller Protos

7-5

CHAPTER 7

Controls and Other Protos

Keep in mind that if you set `scrollAmounts` to values other than the default, your method must check the value passed to it and scroll that amount.

Note

In general, you should discourage double-tapping, since inadvertently tapping twice can cause a double-tap action to occur. ♦

Advanced Usage

If you want more control over the arrow feedback, don't use the `scrollRect`, `viewRect`, or `dataRect` slots at all; instead, use the `SetArrow` and `GetArrow` methods.

For more information about the `SetArrow` method, see “SetArrow” (page 6-4) in *Newton Programmer's Reference*; for more on the `GetArrow` method, see “GetArrow” (page 6-4) in *Newton Programmer's Reference*.

Button and Box Protos

You use the protos described in this section to display text and picture buttons, checkboxes, and radio buttons. The Newton System Software provides a variety of button and box types for use in your applications.

Each of these protos uses specific methods to control its behavior. For many of the protos, the Newton System Software calls the `ButtonClickScript` when the button is tapped. You can define or redefine this method to generate the actions that you want associated with the button.

The Newton System Software calls certain methods for each of the protos described here. For information about which methods you need to define for each proto, see “Button and Box Protos” (page 6-6) in *Newton Programmer's Reference*.

For information about sizing and placement recommendations for your button and box protos, see *Newton 2.0 User Interface Guidelines*.

The following are the button and box protos that you can use in your applications:

- The `protoTextButton` creates a rounded text button with text centered vertically and horizontally inside it. For more information about the slots and methods for this button, see “protoTextButton” (page 6-7) in *Newton Programmer's Reference*. Figure 7-5 shows an example of a `protoTextButton` view.

Figure 7-5 A `protoTextButton` view

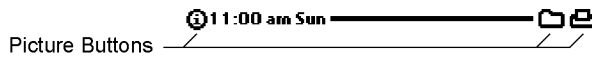


CHAPTER 7

Controls and Other Protos

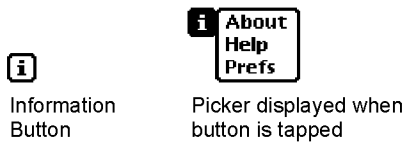
- The `protoPictureButton` creates a picture that is a button. For more information about the slots and methods for this button, see “`protoPictureButton`” (page 6-9) in *Newton Programmer’s Reference*. Figure 7-6 shows an example of a `protoPictureButton` view.

Figure 7-6 A `protoPictureButton` view



- The `protoInfoButton` includes an information button in a view. When the user taps this button, a picker containing information items appears. The picker includes the About, Help, and Prefs items. For more information about the slots and methods for this button, see “`protoInfoButton`” (page 6-10) in *Newton Programmer’s Reference*. Figure 7-7 shows an example of a `protoInfoButton` view.

Figure 7-7 A `protoInfoButton` view



- The `protoOrientation` is a text button that changes the screen orientation so that data on the screen can be displayed facing different directions. This proto is available only on Newton platforms that support changing the screen orientation. For more information about the slots and methods for this button, see “`protoOrientation`” (page 6-13) in *Newton Programmer’s Reference*. Figure 7-8 shows an example of a `protoOrientation` view.

Figure 7-8 A `protoOrientation` view



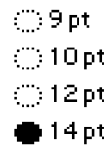
- The `protoRadioCluster` groups a series of radio buttons into a cluster in which only one can be “on” at a time. For more information about the slots and methods for this proto, see “`protoRadioCluster`” (page 6-14) in *Newton Programmer’s Reference*. This proto has no visual representation.

CHAPTER 7

Controls and Other Protos

- The `protoRadioButton` creates a radio button child view of a radio button cluster (based on `protoRadioCluster`). Each radio button is a small oval bitmap that is labeled with text. For more information about the slots and methods for this button, see “`protoPictRadioButton`” (page 6-18) in *Newton Programmer’s Reference*. Figure 7-9 shows an example of several radio buttons in a cluster.

Figure 7-9 A cluster of `protoRadioButtons`



- The `protoPictRadioButton` creates a child view of a radio button cluster (based on `protoRadioCluster`). For more information about the slots and methods for this button, see “`protoPictureButton`” (page 6-9) in *Newton Programmer’s Reference*. Figure 7-10 shows a cluster of `protoPictRadioButtons`.

Figure 7-10 A cluster of `protoPictRadioButtons`



- The `protoCloseBox` allows the user to close the view. For more information about the slots and methods for this box, see “`protoCloseBox`” (page 6-20) in *Newton Programmer’s Reference*. Figure 7-11 shows an example of a `protoCloseBox` view.

Figure 7-11 A `protoCloseBox` view

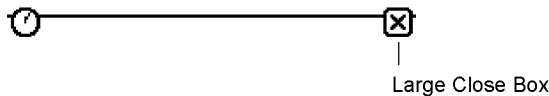


CHAPTER 7

Controls and Other Protos

- The `protoLargeCloseBox` creates a picture button with an “X” icon that is used to close the view. For more information about the slots and methods for this box, see “`protoLargeCloseBox`” (page 6-22) in *Newton Programmer’s Reference*. Figure 7-12 shows an example of a `protoLargeCloseBox` view.

Figure 7-12 A `protoLargeCloseBox` view

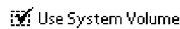


Note

See *Newton 2.0 User Interface Guidelines* for information about when to use `protoCloseBox` and when to use `protoLargeCloseBox`. ♦

- The `protoCheckBox` creates a labeled checkbox with the label text to the right of the box. When the user taps the checkbox, a checkmark is drawn in it. For more information about the slots and methods for this box, see “`protoCheckBox`” (page 6-24) in *Newton Programmer’s Reference*. Figure 7-13 shows an example of a `protoCheckBox` view.

Figure 7-13 A `protoCheckBox` view



- The `protoRCheckBox` creates a labeled checkbox with the text to the left of the checkbox. When the user taps the checkbox, a checkmark is drawn in it. For more information about the slots and methods for this box, see “`protoRCheckBox`” (page 6-26) in *Newton Programmer’s Reference*. Figure 7-14 shows an example of a `protoRCheckBox` view.

Figure 7-14 A `protoRCheckBox` view



CHAPTER 7

Controls and Other Protos

Implementing a Simple Button

To provide a simple button in your application, pick a button proto to use, set the appropriate slots in the button object, and (in most cases) define one or more scripts for the button.

The following is an example of a template that includes `protoTextButton`:

```
aButton := {...
  _proto: protoTextButton,
  viewFont: ROM_fontSystem12Bold,
  text: "My Button",

  ButtonClickScript: func()
    Print("ouch!");

    // a handy way to fit a button around a string
  ViewSetupFormScript: func()
    viewbounds := RelBounds(10, 60,
                          StdButtonWidth(self.text), 13);
  ...}
```

The above example creates the following button on the Newton screen:



When the user taps this button in the Inspector, “ouch” is printed to the Inspector.

You implement a picture button with a similar template, as shown in the following example:

```
pictButton := {...
  _proto: protoPictureButton,
  icon: namesBitmap,
  viewBounds: SetBounds( 2, 8, 34, 40 ),

  ButtonClickScript: func()
    cardfile:Toggle()
  ...}
```

For more information on implementing specific button and box protos, see “Button and Box Protos” (page 7-6) in *Newton Programmer’s Reference*.

CHAPTER 7

Controls and Other Protos

Selection Tab Protos

You can use the protos described in this section to display alphabetic selection tabs on the screen. There are two tab protos that you can use:

- The `protoAZTabs` displays alphabetical tabs arranged horizontally in a view. For more information about the slots and methods for this proto, see “`protoAZTabs`” (page 6-28) in *Newton Programmer’s Reference*. Figure 7-15 shows an example of a `protoAZTabs` view.

Figure 7-15 A `protoAZTabs` view



- The `protoAZVertTabs` displays alphabetical tabs arranged vertically in a view. For more information about the slots and methods for this proto, see “`protoAZVertTabs`” (page 6-29) in *Newton Programmer’s Reference*. Figure 7-16 shows an example of a `protoAZVertTabs` view.

Figure 7-16 A `protoAZVertTabs` view



When the user taps in either of the tab protos, the proto calls the `PickLetterScript` method, passing in the letter that was tapped. The tabs protos and the `PickLetterScript` method are described in “Selection Tab Protos” (page 6-28) in *Newton Programmer’s Reference*.

CHAPTER 7

Controls and Other Protos

Gauge and Slider Protos

You can use the gauge and slider protos described in this section to display gauges. Each slider presents a gauge view that indicates the current progress in relation to the entire operation. There are three protos and one view class available for defining sliders:

- The `protoSlider` creates a user-settable gauge view, which looks like an analog bar gauge with a draggable diamond-shaped knob. For more information about the slots and methods for this proto, see “`protoSlider`” (page 6-33) in *Newton Programmer’s Reference*. Figure 7-17 shows an example of a `protoSlider` view.

Figure 7-17 A `protoSlider` view



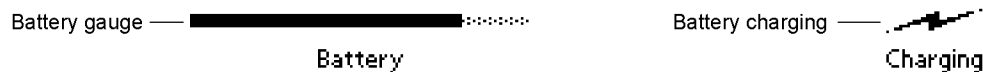
- The `protoGauge` creates a read-only gauge view. For more information about the slots and methods for this proto, see “`protoGauge`” (page 6-35) in *Newton Programmer’s Reference*. Figure 7-18 shows an example of a `protoGauge` view.

Figure 7-18 A `protoGauge` view



- The `protoLabeledBatteryGauge` creates a read-only gauge view that periodically samples the system battery and graphically shows the amount of power left. For more information about the slots and methods for this proto, see “`protoLabeledBatteryGauge`” (page 6-37) in *Newton Programmer’s Reference*. Figure 7-19 shows an example of a `protoLabeledBatteryGauge` view.

Figure 7-19 A `protoLabeledBatteryGauge` view

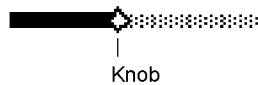


CHAPTER 7

Controls and Other Protos

- The `clGaugeView` class is used to display objects that look like analog bar gauges. Although the `clGaugeView` class is available, you should use the `protoGauge` to display bar gauges. `purpose` as is the `protoGauge` proto. For more information about the slots and methods for the `protoGauge` proto, see “`protoGauge`” (page 6-35) in *Newton Programmer’s Reference*.

Figure 7-20 A `clGaugeView` view



Implementing a Simple Slider

The `clGaugeView` class and the slider protos all have several slots to define the appearance and range of the slider:

- The `viewBounds` slot specifies the size and location of the slider.
- The `viewValue` slot specifies the current value of the slider.
- The `minValue` slot specifies the minimum value of the slider, with a default value of 0.
- The `maxValue` slot specifies the maximum value of the slider, with a default value of 100.

You can specify the initial value of a slider in the `viewValue` slot. However, you often need to look up the initial value; when this is the case, set the initial value of the slider in the `ViewSetupFormScript` method.

To implement a slider, define your template with the proto that you want to use, specify the appearance and range slots, and (optionally) assign an initial value in the `ViewSetupFormScript` method of the proto. For some protos, you need to define additional methods that respond to the user modifying the slider.

The following example is a template that uses `protoSlider` to allow adjustment of the current system volume:

```
SoundSetter := {...
  _proto: protoSlider,
  viewBounds: RelBounds( 12, -21, 65, 9),
  viewJustify: vjParentBottomV,
  maxValue: 4,

  ViewSetupFormScript: func()
    self.viewValue := GetUserConfig('soundVolume');
```

CHAPTER 7

Controls and Other Protos

```

ChangedSlider: func()
    begin
        SetVolume(viewValue);
        :SysBeep();
    end,
    ...}

```

The example above initializes the slider gauge to indicate the current system volume, which it retrieves from the user configuration that is maintained by the Newton System Software. The range of allowable volume values is from 0 (the default for `minValue`) to 4.

Whenever the user moves the slider and lifts the pen, the `viewValue` slot is updated and the `ChangedSlider` method is called. In the example, the `ChangedSlider` method resets the system volume to the new value chosen by the user and sounds a beep to provide the user with audible feedback.

For more information on the `protoSlider` and the `ChangedSlider` method, see “`protoSlider`” (page 6-33) in *Newton Programmer’s Reference*.

Time Protos

You can use the time protos to allow the user to set time and date values. There are four time protos:

- The `protoDigitalClock` time proto displays a digital clock with which the user can set a time value. For more information about the slots and methods for this proto, see “`protoDigitalClock`” (page 6-38) in *Newton Programmer’s Reference*. Figure 7-21 shows an example of a `protoDigitalClock` view.

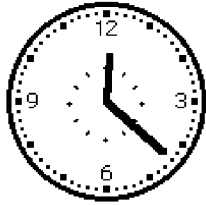
Figure 7-21 A `protoDigitalClock` view



- The `protoNewSetClock` time proto displays an analog clock with which the user can set a time value. For more information about the slots and methods for this proto, see “`protoNewSetClock`” (page 6-40) in *Newton Programmer’s Reference*. Figure 7-22 shows an example of a `protoNewSetClock` view.

CHAPTER 7

Controls and Other Protos

Figure 7-22 A protoNewSetClock view

- The protoSetClock time proto also displays an analog clock with which the user can set a time value. Although this proto is still available for use, it has been updated to the protoNewSetClock, which you should use instead.
- The protoAMPMCluster time proto displays A.M. and P.M. radio buttons in a protoNewSetClock view. For more information about the slots and methods for this proto, see “protoAMPMCluster” (page 6-44) in *Newton Programmer’s Reference*. Figure 7-23 shows an example of a protoAMPMCluster view.

Figure 7-23 A protoAMPMCluster view

Implementing a Simple Time Setter

To implement a time setter, define your template with the proto that you want to use, specify the initial time value to show in the clock, and define the TimeChanged method. You might also need to define additional slots or messages, as described in “Time Protos” (page 6-38) in *Newton Programmer’s Reference*.

The following example is a template that uses protoDigitalClock to allow the user to specify a time:

```
clock := { ...
  _proto: protoDigitalClock,
  time: 0,

  TimeChanged: func()
    begin
      // add your code to respond to time change
      print(time);
    end,
```

CHAPTER 7

Controls and Other Protos

```

// initialize with current time
ViewSetupFormScript: func()
begin
time := time();
end,
...};

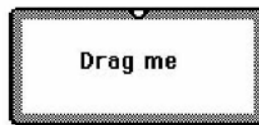
```

Special View Protos

You can use the protos described in this section to provide special-purpose views in your applications. There are seven special view protos:

- The `protoDragger` creates a view that can be dragged around the screen with the pen. For more information about the slots and methods for this proto, see “`protoDragger`” (page 6-45) in *Newton Programmer’s Reference*. Figure 7-22 shows an example of a `protoDragger` view.

Figure 7-24 A `protoDragger` view



- The `protoDragNGo` creates a view that can be dragged around the screen with the pen. This is identical to `protoDragger`, except that `protoDragNGo` includes a close box in the lower-right corner of the view. For more information about the slots and methods for this proto, see “`protoDragNGo`” (page 6-47) in *Newton Programmer’s Reference*. Figure 7-25 shows an example of a `protoDragNGo` view.

Figure 7-25 A `protoDragNGo` view



CHAPTER 7

Controls and Other Protos

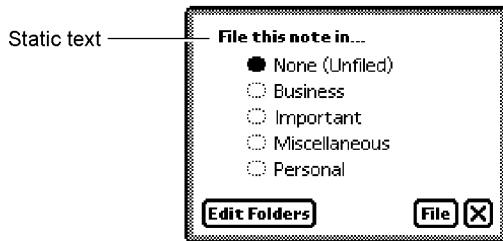
- The `protoDrawer` creates a view that looks and acts like the base view of the Extras Drawer. For more information about the slots and methods for this proto, see “`protoDrawer`” (page 6-49) in *Newton Programmer’s Reference*.
- The `protoFloater` creates a draggable view that is horizontally centered within its parent view and floats above all other nonfloating sibling views within an application. For more information about the slots and methods for this proto, see “`protoFloater`” (page 6-49) in *Newton Programmer’s Reference*.
- The `protoFloatNGo` creates a draggable view that is horizontally centered within its parent view and floats above all other nonfloating sibling views within an application. This is identical to `protoFloater`, except that `protoFloatNGo` includes a close box in the lower-right corner of the view. For more information about the slots and methods for this proto, see “`protoFloatNGo`” (page 6-51) in *Newton Programmer’s Reference*.
- The `protoGlance` creates a text view that automatically closes itself after displaying for a brief period. For more information about the slots and methods for this proto, see “`protoGlance`” (page 6-52) in *Newton Programmer’s Reference*. Figure 7-26 shows an example of a `protoGlance` view.

Figure 7-26 A `protoGlance` view



- The `protoStaticText` creates a one-line paragraph view that is read-only and left-justified. For more information about the slots and methods for this, see “`protoStaticText`” (page 6-54) in *Newton Programmer’s Reference*. Figure 7-22 shows an example of a `protoStaticText` view.

Figure 7-27 A `protoStaticText` view



CHAPTER 7

Controls and Other Protos

View Appearance Protos

You can use the protos described in this section to add to the appearance of your views in certain ways. There are three view appearance protos:

- The `protoBorder` is a view filled with black. You can use this proto as a border, a line, or a black rectangle. For more information about the slots and methods for this proto, see “`protoBorder`” (page 6-56) in *Newton Programmer’s Reference*. Figure 7-28 shows an example of a `protoBorder` view.

Figure 7-28 A `protoBorder` view



- The `protoDivider` creates a divider bar that extends the whole width of its parent view. This proto also includes a text label. For more information about the slots and methods for this proto, see “`protoDivider`” (page 6-56) in *Newton Programmer’s Reference*. Figure 7-29 shows an example of a `protoDivider` view.

Figure 7-29 A `protoDivider` view



- The `protoTitle` creates a title centered above a heavy black line at the top of a view. You can optionally include an icon that appears to the left of the title text. For more information about the slots and methods for this proto, see “`protoTitle`” (page 6-58) in *Newton Programmer’s Reference*. Figure 7-30 shows an example of a `protoTitle` view.

Figure 7-30 A `protoTitle` view



CHAPTER 7

Controls and Other Protos

Status Bar Protos

You can use the protos described in this section to display a status bar at the bottom of a view. There are two status bar protos:

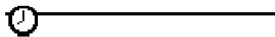
- The `protoStatus` creates a status bar, which includes a close button and an analog clock, at the bottom of a view. For more information about the slots and methods for this proto, see “`protoStatus`” (page 6-59) in *Newton Programmer’s Reference*. Figure 7-31 shows an example of a `protoStatus` view.

Figure 7-31 A `protoStatus` view



- The `protoStatusBar` creates a status bar, which includes an analog clock, at the bottom of a view. This is identical to `protoStatus`, except that `protoStatusBar` does not include a close button. For more information about the slots and methods for this proto, see “`protoStatusBar`” (page 6-60) in *Newton Programmer’s Reference*. Figure 7-32 shows an example of a `protoStatusBar` view.

Figure 7-32 A `protoStatusBar` view



Note

The new status bar protos `newtStatusBarNoClose` and `newtStatusBar`, are the preferred way to add a status bar to your applications. These protos, which are described in “`NewtApp Applications`” (page 4-1), simplify adding buttons and automate hiding the close box when your application is moved into the background. ♦

CHAPTER 7

Controls and Other Protos

Summary

Scroller Protos

protoLeftRightScroller

```

aProtoLeftRightScroller := {
  _proto: protoLeftRightScroller,
  scrollView:    viewTemplate,
  scrollRect:    boundsFrame, // extent of scrollable area
  dataRect:     boundsFrame, // extent of data in the view
  viewRect:     boundsFrame, // extent of visible area
  xPos:         integer,     // initial x-coord in scrollRect
  yPos:         integer,     // initial y-coord in scrollRect
  scrollAmounts: array,      // line, page, dbl-click values
  pageThreshold: integer,   // lines before page scrolling
  ViewScroll2DScript: function, // called when arrows tapped
  ViewScrollDoneScript: function, // called when scroll done
  SetArrow:      function, // set scroll direction
  GetArrow:      function, // returns scroll direction
  ...
}

```

protoUpDownScroller

```

aProtoUpDownScroller := {
  _proto: protoUpDownScroller,
  scrollView:    viewTemplate,
  scrollRect:    boundsFrame, // extent of scrollable area
  dataRect:     boundsFrame, // extent of data in the view
  viewRect:     boundsFrame, // extent of visible area
  xPos:         integer,     // initial x-coord in scrollRect
  yPos:         integer,     // initial y-coord in scrollRect
  scrollAmounts: array,      // line, page, dbl-click values
  pageThreshold: integer,   // lines before page scrolling
  ViewScroll2DScript: function, // called when arrows tapped
  ViewScrollDoneScript: function, // called when scroll done
  SetArrow:      function, // set scroll direction
  GetArrow:      function, // returns scroll direction
  ...
}

```

CHAPTER 7

Controls and Other Protos

protoHorizontal2DScroller

```

aProtoHorizontal2DScroller := {
  _proto: protoHorizontal2DScroller,
  scrollView:    viewTemplate,
  scrollRect:    boundsFrame, // extent of scrollable area
  dataRect:     boundsFrame, // extent of data in the view
  viewRect:     boundsFrame, // extent of visible area
  xPos:         integer,     // initial x-coord in scrollRect
  yPos:         integer,     // initial y-coord in scrollRect
  scrollAmounts: array,      // line, page, dbl-click values
  pageThreshold: integer,   // lines before page scrolling
  ViewScroll2DScript: function, // called when arrows tapped
  ViewScrollDoneScript: function, // called when scroll done
  SetArrow:      function, // set scroll direction
  GetArrow:      function, // returns scroll direction
  ...
}

```

protoHorizontalUpDownScroller

```

aProtoHorizontalUpDownScroller := {
  _proto: protoHorizontalUpDownScroller,
  scrollView:    viewTemplate,
  scrollRect:    boundsFrame, // extent of scrollable area
  dataRect:     boundsFrame, // extent of data in the view
  viewRect:     boundsFrame, // extent of visible area
  xPos:         integer,     // initial x-coord in scrollRect
  yPos:         integer,     // initial y-coord in scrollRect
  scrollAmounts: array,      // line, page, dbl-click values
  pageThreshold: integer,   // lines before page scrolling
  ViewScroll2DScript: function, // called when arrows tapped
  ViewScrollDoneScript: function, // called when scroll done
  SetArrow:      function, // set scroll direction
  GetArrow:      function, // returns scroll direction
  ...
}

```

CHAPTER 7

Controls and Other Protos

Button and Box Protos

protoTextButton

```

aProtoTextButton := {
  _proto: protoTextButton,
  viewBounds:      boundsFrame,
  viewFlags:       integer,    // viewFlags constants
  text:            string,     // text inside the button
  viewFont:        fontFlags,
  viewFormat:      formatFlags,
  viewJustify:     justificationFlags,
  viewTransferMode: integer,    // view transfer constants
  ButtonClickScript: function, // when button is tapped
  ButtonPressedScript: function, // while button is pressed
  ...
}

```

protoPictureButton

```

aProtoTextButton := {
  _proto: protoPictureButton,
  viewBounds:      boundsFrame,
  viewFlags:       integer,    // viewFlags constants
  icon:            bitmap,     // bitmap to use for button
  viewFormat:      formatFlags,
  viewJustify:     justificationFlags,
  ButtonClickScript: function, // when button is tapped
  ButtonPressedScript: function, // while button is pressed
  ...
}

```

protoInfoButton

```

aProtoInfoButton := {
  _proto: protoInfoButton,
  viewFlags:       integer,    // viewFlags constants
  viewBounds:      boundsFrame,
  viewJustify:     justificationFlags,
  ...
}

```

CHAPTER 7

Controls and Other Protos

protoOrientation

```

aProtoOrientation := {
  _proto: protoOrientation,
  viewFlags:      integer, // viewFlags constants
  viewBounds:    boundsFrame,
  viewJustify:   justificationFlags,
  ...
}

```

protoRadioCluster

```

aProtoRadioCluster := {
  _proto: protoRadioCluster,
  viewBounds:      boundsFrame,
  clusterValue:    integer, // value of selected button
  InitClusterValue: function, // initialize cluster
  ViewSetupFormScript: function, // set initial button
  ClusterChanged:  function, // called upon value change
  SetClusterValue: function, // change selected button
  ...
}

```

protoRadioButton

```

aProtoRadioButton := {
  _proto: protoRadioButton,
  viewBounds:      boundsFrame,
  viewFormat:      formatFlags,
  text:            string, // radio button text label
  buttonValue:     integer, // identifies button
  viewValue:       integer, // current value of radio button
  ...
}

```

protoPictRadioButton

```

aProtoPictRadioButton := {
  _proto: protoPictRadioButton,
  viewBounds:      boundsFrame,
  viewFormat:      formatFlags,
  viewJustify:     justificationFlags,
  icon:            bitmap, // bitmap for picture button
  buttonValue:     integer, // identifies button
}

```

Summary

7-23

CHAPTER 7

Controls and Other Protos

```
viewValue:      integer,    // current value of radio button
ViewDrawScript: function,  // to highlight button
...
}
```

protoCloseBox

```
aProtoCloseBox := {
  _proto: protoCloseBox,
viewFlags:      integer,    // viewFlags constants
viewBounds:     boundsFrame,
viewJustify:    justificationFlags,
viewFormat:     formatFlags,
ButtonClickScript: function, // called before closing
...
}
```

protoLargeCloseBox

```
aProtoLargeCloseBox := {
  _proto: protoLargeCloseBox,
viewFlags:      integer,    // viewFlags constants
viewBounds:     boundsFrame,
viewJustify:    justificationFlags,
viewFormat:     formatFlags,
ButtonClickScript: function, // called before closing
...
}
```

protoCheckbox

```
aProtoCheckbox := {
  _proto: protoCheckbox,
viewBounds:     boundsFrame,
viewFormat:     formatFlags,
viewFont:       fontFlags,  // font for text label
text:           string,     // the checkbox label
buttonValue:    value,      // value when box is checked
viewValue:      value,      // current value (nil=unchecked)
ValueChanged:   function,   // checkbox value changed
ToggleCheck:    function,   // toggles checkbox state
...
}
```

CHAPTER 7

Controls and Other Protos

protoRCheckbox

```

aProtoRCheckbox := {
  _proto: protoRCheckbox,
  viewBounds:    boundsFrame,
  viewFormat:    formatFlags,
  viewFont:      fontFlags, // font for text label
  text:          string,    // the checkbox label
  indent:        integer,   // pixels to indent box
  buttonValue:   value,     // value when box is checked
  viewValue:     value,     // current value (nil=unchecked)
  ValueChanged: function,  // checkbox value changed
  ToggleCheck:  function,  // toggles checkbox state
  ...
}

```

Selection Tab Protos

protoAZTabs

```

aProtoAZTabs := {
  _proto: protoAZTabs,
  PickLetterScript: function, // tab is tapped
  SetLetter:        function, // sets tab letter
  ...
}

```

protoAZVertTabs

```

aProtoAZVertTabs := {
  _proto: protoAZVertTabs,
  PickLetterScript: function, // tab is tapped
  SetLetter:        function, // sets tab letter
  ...
}

```

Gauges and Slider Protos

protoSlider

```

aProtoSlider := {
  _proto: protoSlider,
  viewBounds:    boundsFrame,

```

CHAPTER 7

Controls and Other Protos

```

viewValue:          integer,    // gauge value
minValue:           integer,    // minimum gauge value
maxValue:           integer,    // maximum gauge value
ViewSetupFormScript: function, // set initial gauge value
ChangedSlider:      function,   // slider moved
TrackSlider:        function,   // viewValue changed
...
}

```

protoGauge

```

aProtoGauge := {
  _proto: protoGauge,
  viewBounds: boundsFrame,
  viewValue:  integer,    // gauge value
  minValue:   integer,    // minimum gauge value
  maxValue:   integer,    // maximum gauge value
  gaugeDrawLimits: Boolean, // non-nil for gray bg
  ViewSetupFormScript: function, // set initial gauge value
  ...
}

```

protoLabeledBatteryGauge

```

aProtoLabeledBatteryGauge := {
  _proto: protoLabeledBatteryGauge,
  viewBounds: boundsFrame,
  ...
}

```

clGaugeView

```

aClGaugeView := {
  viewBounds: boundsFrame,
  viewClass:  clGaugeView,
  viewValue:  integer,    // value of gauge
  viewFlags:  integer,    // viewFlags constants
  viewFormat: formatFlags,
  minValue:   integer,    // min value of gauge
  maxValue:   integer,    // max value of gauge
  gaugeDrawLimits: Boolean, // non-nil for gray bg
  ViewChangedScript: function, // gauge dragged
  ViewFinalChangeScript: function, // gauge changed
  ...
}

```


CHAPTER 7

Controls and Other Protos

Time Protos

protoDigitalClock

```

aProtoDigitalClock := {
  _proto: protoDigitalClock,
  viewFlags:      integer,    // viewFlags constants
  viewBounds:     boundsFrame,
  viewJustify:    justificationFlags,
  increment:      integer,    // minutes to change on tap
  time:           integer,    // initial or current time
  wrapping:       Boolean,    // non-nil to wrap around day
                               // boundaries
  midnite:        Boolean,    // non-nil if 0 means midnight
                               // tomorrow
  Refresh:        function,   // update clock
  TimeChanged:    function,   // called when time is changed
  ...
}

```

protoSetClock

```

aProtoSetClock := {
  _proto: protoSetClock,
  viewBounds:     boundsFrame,
  viewFlags:      integer,    // viewFlags constants
  viewFormat:     formatFlags,
  hours:          integer,    // value set by hour hand
  minutes:        integer,    // value set by minute hand
  TimeChanged:    function,   // called when time is changed
  ...
}

```

protoNewSetClock

```

aProtoNewSetClock := {
  _proto: protoNewSetClock,
  viewBounds:     boundsFrame,
  viewJustify:    justificationFlags,
  time:           integer,    // initial or current time
  annotations:    array,      // four strings to annotate
                               // the clock face
  supressAnnotations: Boolean, // if slot exists, suppress
  exactHour:      Boolean,    // adjust hour markers
}

```

Summary

7-27

CHAPTER 7

Controls and Other Protos

```
Refresh:           function,    // update clock
TimeChanged:      function,    // called when time is changed
...
}
```

protoAMPMCluster

```
aProtoAMPMCluster := {
  _proto: protoAMPMCluster,
  viewBounds:      boundsFrame,
  viewJustify:     justificationFlags,
  time:            integer,    // specify time--required
  ...
}
```

Special View Protos

protoDragger

```
aProtoDragger := {
  _proto: protoDragger,
  viewBounds :    boundsFrame,
  viewFlags:      integer,    // viewFlags constants
  viewFormat:     formatFlags,
  noScroll:       string,    // msg to display if no scrolling
  noOverview:     string,    // msg to display if no overview
  ...
}
```

protoDragNGo

```
aProtoDragNGo := {
  _proto: protoDragNGo,
  viewBounds:     boundsFrame,
  viewFlags:      integer,    // viewFlags constants
  viewJustify:    justificationFlags,
  viewFormat:     formatFlags,
  noScroll:       string,    // msg to display if no scrolling
  noOverview:     string,    // msg to display if no overview
  ...
}
```

CHAPTER 7

Controls and Other Protos

protoDrawer

```

aProtoDrawer := {
  _proto: protoDrawer,
  viewFlags: integer, // viewFlags constants
  viewBounds: boundsFrame,
  viewFormat: formatFlags,
  viewEffect: effectFlags,
  showSound: soundFrame, // sound when drawer opens
  hideSound: soundFrame, // sound when drawer closes
  ...
}

```

protoFloater

```

aProtoFloater := {
  _proto: protoFloater,
  viewBounds: boundsFrame,
  viewFlags: integer, // viewFlags constants
  viewJustify: justificationFlags,
  viewFormat: formatFlags,
  viewEffect: effectFlags,
  noScroll: string, // msg to display if no scrolling
  noOverview: string, // msg to display if no overview
  ...
}

```

protoFloatNGo

```

aProtoFloatNGo := {
  _proto: protoFloatNGo,
  viewFlags: integer, // viewFlags constants
  viewBounds: boundsFrame,
  viewJustify: justificationFlags,
  viewFormat: formatFlags,
  viewEffect: effectFlags,
  noScroll: string, // msg to display if no scrolling
  noOverview: string, // msg to display if no overview
  ...
}

```

CHAPTER 7

Controls and Other Protos

protoGlance

```

aProtoGlance := {
  _proto: protoGlance,
  viewBounds:      boundsFrame,
  viewJustify:     justificationFlags,
  viewFormat:      formatFlags,
  viewFont:        fontFlags, // font for text
  viewEffect:      effectFlags,
  viewIdleFrequency: integer, // time view to remain open
  text:            string,    // text to appear in view
  ...
}

```

protoStaticText

```

aProtoStaticText := {
  _proto: protoStaticText,
  viewBounds:      boundsFrame,
  viewFlags:       integer, // viewFlags constants
  text:            string,    // text to display
  viewFont:        fontFlags,
  viewJustify:     justificationFlags,
  viewFormat:      formatFlags,
  viewTransferMode: integer, // view transfer constants
  tabs:            array, // up to eight tab-stop positions
  styles:          array, // font information
  ...
}

```

View Appearance ProtosprotoBorder

```

aProtoBorder := {
  _proto: protoBorder,
  viewBounds:      boundsFrame,
  viewFlags:       integer, // viewFlags constants
  viewFormat:      formatFlags,
  ...
}

```

CHAPTER 7

Controls and Other Protos

protoDivider

```

aProtoDivider := {
  _proto: protoDivider,
  viewBounds:    boundsFrame,
  viewFlags:     integer, // viewFlags constants
  viewFont:      fontFlags, // font for text
  viewJustify:   justificationFlags,
  viewFormat:    formatFlags,
  title:         string, // text on divider bar
  titleHeight:   integer, // height of divider
  ...
}

```

protoTitle

```

aProtoTitle := {
  _proto: protoTitle,
  viewJustify:   justificationFlags,
  viewFormat:    formatFlags,
  viewFont:      fontFlags,
  title:         string, // text of title
  titleIcon:     bitMapFrame,
  titleHeight:   integer, // height of title
  viewTransferMode: integer, // view transfer constants
  ...
}

```

Status Bar Protos

protoStatus

```

aProtoStatus := {
  _proto: protoStatus,
  ...
}

```

protoStatusBar

```

aProtoStatusBar := {
  _proto: protoStatusBar,
  ...
}

```


C H A P T E R 8

Text and Ink Input and Display

This chapter describes how the Newton system handles text and presents interfaces you can use to work with text in NewtonScript applications.

The material covers the following components of Newton text handling:

- handwritten text input
- keyboard text input
- views for text display
- fonts for text display

The first section of this chapter, “About Text,” describes the basic terms and concepts needed to understand text processing on the Newton.

The second section, “Using Text,” describes how to use the various input and display components to handle text in your applications.

For comprehensive reference information about the text-related constants, data structures, views, methods, and functions, see “Text and Ink Input and Display Reference” (page 7-1) in *Newton Programmer’s Reference*.

About Text

This section describes the basic concepts, terms, and processes you need to understand to work with text in your applications.

About Text and Ink

The Newton allows you to process two forms of text input: **ink text** and **recognized text**. This section describes both forms of text input.

CHAPTER 8

Text and Ink Input and Display

When the user writes a line of text on the Newton screen, the Newton system software performs a series of operations, as follows:

- The raw data for the input is captured as ink, which is also known as **sketch ink** or **raw ink**.
- Raw ink is stored as a sequence of **strokes** or stroke data.
- If the view in which the ink was drawn is configured for **ink text**, the recognition system groups the stroke data into a series of **ink words**, based on the timing and spacing of the user's handwriting. A user can insert, delete, and move ink words in the same way as recognized text. Ink words can be scaled to various sizes for display and printing. They can also be recognized at a later time, by a process known as **deferred recognition**.
- If the view in which the ink was drawn supports or is configured for text recognition, the ink words are processed by the recognition system into recognized text and displayed in a typeface.

The data describing the handwriting strokes of the ink word are stored as compressed data in a binary object. This **stroke data** can be accessed programmatically, using the stroke bundle methods described in "Recognition" (page 9-1) in *Newton Programmer's Guide*.

The recognition system and deferred recognition are described in "Recognition" (page 9-1).

Note

To provide maximum user flexibility for your applications, you are encouraged to allow ink text in all of your input views. ♦

Written Input Formats

Ink words can be intermixed with recognized text. This data, normally represented as **rich strings**, can be used anywhere that you might expect a standard string. Each ink word is encoded as a single character in a rich string, as described in "Rich Strings" (page 8-22).

You should use the **rich string format** to store data in a soup, because of its compact representation. You can safely use rich strings with all functions, including the string functions, which are documented in "Utility Functions" (page 26-1). Another data format, described in "Text and Styles" (page 8-25), pairs text strings with style data for viewing in text views.

CHAPTER 8

Text and Ink Input and Display

Caret Insertion Writing Mode

Caret insertion writing mode is a text input mode that the user can enable or disable. When caret insertion mode is disabled, handwritten text is inserted into the view at the location where it is written. When caret insertion writing mode is enabled, handwritten text is inserted at the location indicated by the insertion caret, regardless of where on the screen it is drawn. Caret insertion writing mode is used automatically for keyboard text entry.

To enable or disable caret insertion writing mode, the user selects or deselects the “Insert new words at caret” option from the Text Editing Settings slip. You can display this slip by tapping the Options button in the Recognition Preferences slip.

Applications do not normally need to be aware of whether caret insertion writing mode is enabled or disabled. The one exception to this is at application startup time, when you might want to set the initial location of the insertion point. This is described in “Setting the Caret Insertion Point” (page 8-26).

There are a few caret insertion writing mode routines you can use to implement your own version of this mode. They are described in “Caret Insertion Writing Mode Functions and Methods” (page 7-47) in *Newton Programmer’s Reference*.

Fonts for Text and Ink Display

The Newton system software allows you to specify the font characteristics for displaying text and ink in a paragraph view on the screen. The font information is stored in a font specification structure known as a **font spec**. The font specification for built-in ROM fonts can also be represented in a frame as a packed integer. Both of these representations are described in “Using Fonts for Text and Ink Display” (page 8-17).

The system provides a number of functions you can use to access and modify font attributes. These are described in “Text and Styles” (page 8-25).

About Text Views and Protos

There are a number of views and protos to use for displaying text and for receiving text input. For basic information about views, see “Views” (page 3-1).

CHAPTER 8

Text and Ink Input and Display

The views and protos that you use for text are listed in Table 8-1.

Table 8-1 Views and protos for text input and display

View or Proto	Description
edit view	Use the <code>clEditView</code> class for basic text input and display. Objects of this class can display and/or accept text and graphic data. The <code>clEditView</code> automatically creates child <code>clParagraphView</code> views for text input and display and <code>clPolygonView</code> views for graphic input and display. You can also include <code>clPictureView</code> views in your <code>clEditViews</code> . For more information about this class, see “General Input Views” (page 8-6).
paragraph views	Use the <code>clParagraphView</code> class to display text or to accept text input. For more information about this class, see “Paragraph Views” (page 8-10).
lightweight paragraph views	If your paragraph view template meets certain criteria, the Newton system automatically creates a lightweight paragraph view for you. Lightweight paragraph views are read-only and use only one font, although they can contain ink. These views require significantly less memory than do standard paragraph views. For more information about lightweight paragraph views, see “Lightweight Paragraph Views” (page 8-11).
input line protos	You can use one of the input line protos to allow the user to enter a single line of text, as described in “Using Input Line Protos” (page 8-12).

About Keyboard Text Input

Your application can provide keyboards and keypads for user text input by creating an object from one of the keyboard view classes or protos:

- The `clKeyboardView` class provides a keyboard-like array of buttons that the user can tap with the pen to perform an action. This class is described in “Keyboard Views” (page 8-26).

CHAPTER 8

Text and Ink Input and Display

- Use one of the keyboard protos to create keyboard views in your applications. These protos include the `protoKeyboard`, which creates a keyboard view that floats above all other views. The keyboard protos are also described in “Keyboard Views.”

The Keyboard Registry

You need to register any custom keyboards or keypads that you create with the Newton system’s keyboard registry. Caret insertion writing mode is used whenever the user enters text from a keyboard or keypad. When a registered keyboard or keypad is opened, the system knows to display the insertion caret at the proper location.

The Newton system also allows you to customize the behavior of the insertion caret and key presses by calling your application-defined methods whenever an action occurs in a registered keyboard or keypad.

For more information about the keyboard registry, see “Using the Keyboard Registry” (page 8-36).

The Punctuation Pop-up Menu

The user can tap the insertion caret to display a Punctuation pop-up menu. This menu, shown in Figure 8-1, provides an easy way to add punctuation when writing with the stylus.

Figure 8-1 The Punctuation pop-up menu



Choosing any item on the Punctuation pop-up menu inserts the appropriate character into the text, at the insertion caret. The bent arrow, at the top left, is a carriage return, and the blank box at the bottom indicates a space.

You can override this menu with your own caret pop-up menu, as described in “The Caret Pop-up Menu” (page 8-38).

CHAPTER 8

Text and Ink Input and Display

Compatibility

One of the significant advances in software functionality in the Newton 2.0 system is the capacity to process ink in most views, which includes deferred recognition and the ability to mix text and ink together in rich string. This expands the behavior provided by Newton 1.x machines, which generally process written input immediately for recognition and display the resulting word in a typeface.

These additional capabilities made it necessary to extend the Recognition menu. The Newton 2.0 Recognition menu adds more input options and replaces the toggling Recognizer buttons of the Newton 1.x status bar.

The Newton 2.0 system also behaves slightly differently when merging text into paragraph views. When caret insertion writing mode is disabled, paragraphs no longer automatically insert carriage returns or tabs. This is true regardless of whether the user is entering text or ink words.

With Newton System 2.0, you can include images in your edit views. Edit views (`clEditView`) can now contain picture views (`clPictureView`) as child views

Any ink written on a 1.x machine can be dragged into a Newton System 2.0 paragraph and automatically converted into an ink word.

Notes, text, or ink moved from a Newton 1.x to a Newton with the 2.0 system works correctly without any intervention. However, the reverse is not true: you cannot insert a card with 2.0 or later data into a 1.x machine.

The expando protos have become obsolete. These are `protoExpandoShell`, `protoDateExpando`, `protoPhoneExpando`, and `protoTextExpando`. These protos are still supported for 1.x application compatibility, but should not be used in new applications.

Using Text

This section describes how to use various features of text input and display on the Newton and provides examples of some of these features.

Using Views and Protos for Text Input and Display

This section describes the different views and protos to use in your applications for text input and display.

General Input Views

The `clEditView` view class is used for a view that can display and/or accept text and graphic data. Views of the `clEditView` class contain no data directly;

CHAPTER 8

Text and Ink Input and Display

instead, they have child views that contain the individual data items. Text items are contained in child views of the class `clParagraphView` and graphics are contained in child views of the class `clPolygonView`.

A view of the `clEditView` class includes the following features:

- Automatic creation of `clParagraphView` or `clPolygonView` children as the user writes or draws in the view. These child views hold the data the user writes.
- Support for inclusion of `clPictureView` views, which are used for images.
- Text and shape recognition, selection, and gestures such as scrubbing, copying to clipboard, pasting from clipboard, duplicating, and others, as controlled by the setting of the `viewFlags` slot. The initial recognition is handled by the `clEditView`. A child `clParagraphView` or `clPolygonView` is created and that child view handles subsequent editing of the data.
- Drag and drop handling. A child view can be dragged (moved or copied) out of the `clEditView` and dropped into another `clEditView`, whose child it then becomes. Other views can be configured to handle data dragged from a `clEditView`, as described in “Views” (page 3-1).
- Clipboard support. A `clParagraphView` or `clPolygonView` child view can be dragged (moved or copied) to the clipboard, from which it can be pasted into another `clEditView` or `clView`, whose child it becomes.
- Automatic resizing of `clParagraphView` child views to accommodate added input. This feature is controlled by the `vCalculateBounds` flag in the `viewFlags` slot of those child views.
- Automatic addition of new words to existing paragraphs when caret insertion writing mode is disabled.

Views of the class `clEditView` are intended for user input of text, shape, image, and ink data. Consequently, views of this class expect that any child views have been defined and created at run time, not predefined by templates created in NTK.

If you need to include predefined child views in a `clEditView`, use the `ViewSetupChildrenScript` method of the `clEditView` to define the child views and set up the `stepChildren` array. You might need to do this, for example, if you store the data for child views in a soup, and you need to retrieve the data and rebuild the child views at run time. For more information, see “Including Editable Child Views in an Input View” (page 8-9).

The default font for a `clParagraphView` created by a `clEditView` is the font selected by the user on the Styles palette in the system.

The default pen width for a `clPolygonView` created by a `clEditView` is the width set by the user on the Styles palette.

The slots of `clEditView` are described in “General Input View (`clEditView`)” (page 7-12) in *Newton Programmer’s Reference*.

CHAPTER 8

Text and Ink Input and Display

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

```
editor := {...
  viewClass: clEditView,
  viewBounds: {left:0, top:0, right:200, bottom:200},
  viewFlags: vVisible+vAnythingAllowed,
  viewFormat: vfFillWhite+vfFrameBlack+vfPen(1)+
              vfLinesLtGray,
  viewLineSpacing: 20,
  // methods and other view-specific slots
  viewSetupFormScript: func()...
...}
```

System Messages in Automatically Created Views

When a child view is automatically created by a `clEditView`, the `vNoScripts` flag is set in the `viewFlags` slot of the child view. This flag prevents system messages from being sent to a view.

This behavior is normally desirable for automatically created views, because they have no system message-handling methods and the system saves time by not sending the messages to them.

If you want to use one of these views in a manner that requires it to receive system messages, you need to remove the `vNoScripts` flag from the `viewFlags` slot of the view.

Creating the Lined Paper Effect in a Text View

A view of the `clEditView` class can appear simply as a blank area in which the user writes information. However, this type of view usually contains a series of horizontal dotted lines, like lined writing paper. The lines indicate to the user that the view accepts input. To create the lined paper effect, you must set the following slots appropriately:

`viewFormat` Must include one of the `vfLines...` flags. This activates the line display.

`viewLineSpacing` Sets the spacing between the lines, in pixels.

`viewLinePattern` Optional. Sets a custom pattern that is used to draw the lines in the view. In the `viewFormat` slot editor in NTK, you must also set the Lines item to Custom to signal that you are using a custom pattern. (This sets the `vfCustom<<vfLinesShift` flag in the `viewFormat` slot.)

Patterns are binary data structures, which are described in the next section.

CHAPTER 8

Text and Ink Input and Display

Defining a Line Pattern

You can define a custom line pattern for drawing the horizontal lines in a paragraph view. A line pattern is an eight-byte binary data structure with the class 'pattern.

To create a binary pattern data structure on the fly, use the following NewtonScript trick:

```
myPattern := SetClass( Clone("\uAAAAAAAAAAAAAAAA"),
                        'pattern );
```

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

Including Editable Child Views in an Input View

For a child view of a clEditView to be editable, you need to follow certain rules:

- The child view must reside in the viewChildren slot of the clEditView. You cannot store a child view's template in the stepChildren slot, as NTK normally does.
- The child view must contain a viewStationery slot with an appropriate value, depending on the view class and data type. The acceptable values are shown in Table 8-2:

Table 8-2 viewStationery slot value for clEditView children

View class	View data type	Value of viewStationery slot
clParagraphView	text	'para
clPolygonView	recognized graphics	'poly
clPolygonView	ink	'ink
clPictureView	bitmap or picture object	'pict

- Add the child view templates to the viewChildren array of the edit view and open the view or send it the RedoChildren message. Alternatively, you can add the child view with the AddView method and then send the Dirty message to the edit view.

CHAPTER 8

Text and Ink Input and Display

IMPORTANT

You store view templates (not view objects) in the `viewChildren` array of an edit view. ▲

Paragraph Views

The `clParagraphView` class displays text or accepts text input. It includes the following features:

- Text recognition
- Text correction
- Text editing, including scrubbing, selection, copying to the clipboard, pasting from the clipboard, and other gestures, including duplicating, as controlled by the setting of the `viewFlags` slot.
- Automatic word-wrapping.
- Support for the caret gesture, which adds a space or splits a word.
- Clipping of text that won't fit in the view. (An ellipsis is shown to indicate text beyond what is visible.)
- Use of ink and different text fonts (styles) within the same paragraph.
- Tab-stop alignment of text.
- Automatic resizing to accommodate added text (when this view is enclosed in a `clEditView`). This feature is controlled by the `vCalculateBounds` flag in the `viewFlags` slot.
- Automatic addition of new words written near the view when this view is enclosed in a `clEditView` and caret insertion writing mode is disabled.

The slots of `clParagraphView` are described in “Paragraph View (`clParagraphView`)” (page 7-15) in *Newton Programmer's Reference*.

Note that you don't need to create paragraph views yourself if you are accepting user input inside a `clEditView`. Just provide a `clEditView` and when the user writes in it, the view automatically creates paragraph views to hold text.

The following is an example of a template defining a view of the `clParagraphView` class:

```
dateSample := { ...
  viewClass: clParagraphView,
  viewBounds: {left:50, top:50, right:200, bottom:70},
  viewFlags: vVisible+vReadOnly,
  viewFormat: vFillWhite,
  viewJustify: oneLineOnly,
  text: "January 24, 1994",
```


CHAPTER 8

Text and Ink Input and Display

```
// 8 chars of one font, 3 chars of another, 5 chars
// of another
styles: [8, 18434, 3, 12290, 5, 1060865],
...}
```

Paragraph views are normally lined to convey to the user that the view accepts text input. To add the lined paper effect to paragraph views, see “Creating the Lined Paper Effect in a Text View” (page 8-8).

Lightweight Paragraph Views

When you create a template using the `clParagraphView` class, and that template is instantiated into a view at run time, the system may create a specialized kind of paragraph view object, called a lightweight paragraph view. Lightweight paragraph views have the advantage of requiring much less memory than do standard paragraph views.

The system automatically creates a lightweight paragraph view instead of a standard paragraph view if your template meets the following conditions:

- The view must be read-only, which means that its `viewFlags` slot contains the `vReadOnly` flag.
- The view must not include any tabs, which means that the template does not contain the `tabs` slot.
- The view must not include multiple font styles, which means that the template does not contain the `styles` slot; however, the view can contain a rich string in its `text` slot. For information about rich strings, see “Rich Strings” (page 8-22).
- The `viewFlags` slot of the view must not contain the following flags: `vGesturesAllowed`, `vCalculateBounds`.

Note

Lightweight paragraph views can contain ink. ◆

Most paragraph views look the same after they are instantiated; that is, there is not normally a way to tell whether a particular paragraph view is a standard or a lightweight view. However, ink displayed in a lightweight paragraph view is displayed in a fixed font size.

Note

When laying out text in a lightweight paragraph view, the `viewLineSpacing` value is ignored. This is not generally a problem, since the line spacing dotted lines are normally used to indicate that the text can be edited, and text in a lightweight paragraph cannot be edited. ◆

CHAPTER 8

Text and Ink Input and Display

Once a lightweight paragraph view has been instantiated, you cannot dynamically change the view flags to make it an editable view, or add multistyled text by providing a `styles` slot, since this type of view object doesn't support these features. If you need this functionality for an existing lightweight paragraph view, you'll have to copy the text out of it into an editable paragraph view.

Using Input Line Protos

Input line protos provide the user with single lines in which to enter data. There are four input line protos available:

- `protoInputLine` is a one-line input field that defines a simple paragraph view in which the text input is left-justified.
- `protoRichInputLine` is the text and ink equivalent of `protoInputLine`.
- `protoLabelInputLine` is a one-line input field that includes a text label and can optionally include a pop-up menu known as a **picker**.
- `protoRichLabelInputLine` is the text and ink equivalent of `protoLabelInputLine`.

`protoInputLine`

This proto defines a view that accepts any kind of text input and is left-justified. Below is an example of what a `protoInputLine` looks like on the Newton screen:

.....

The `protoInputLine` is based on a view of the `clParagraphView` class. It has no child views.

The following is an example of a template using `protoInputLine`:

```
myInput := { ...
  _proto: protoInputLine,
  viewJustify: vjParentRightH+vjParentBottomV,
  viewLineSpacing: 24,
  viewBounds: SetBounds( -55, -33, -3, -3),
  ... }
```

The slots of the `protoInputLine` are described in “`protoInputLine`” (page 7-17) in *Newton Programmer's Reference*.

`protoRichInputLine`

This proto works exactly like `protoInputLine`. The only difference is that `protoRichInputLine` allows mixed ink and text input, as determined by the current user recognition preferences.

CHAPTER 8

Text and Ink Input and Display

The slots of `protoRichInputLine` are described in “`protoRichInputLine`” (page 7-19) in *Newton Programmer’s Reference*.

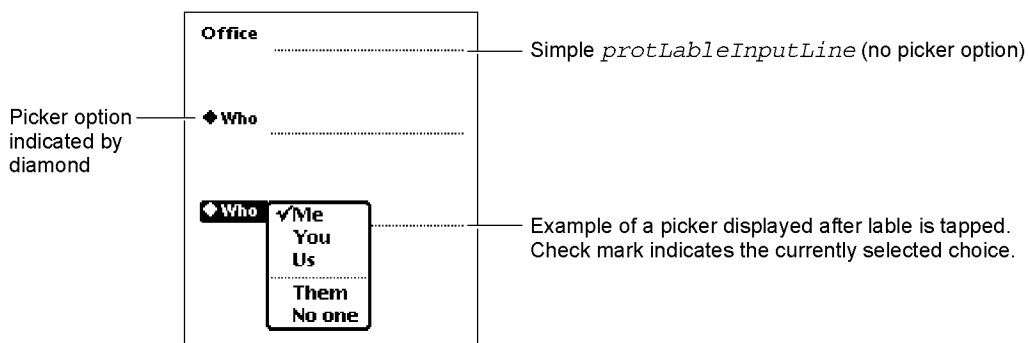
protoLabelInputLine

This proto defines a view that features a label, accepts any kind of input, and is left-justified. The `protoLabelInputLine` can optionally include a picker.

When the `protoLabelInputLine` does include a picker, the user selects a choice from the picker. That choice is entered as the text in the input line, and is marked with a check mark in the picker.

Figure 8-2 shows an example of a `protoLabelInputLine` with and without the picker option:

Figure 8-2 An example of a `protoLabelInputLine`



The `protoLabelInputLine` is based on a view of the `clParagraphView` class. It has two child views:

- The `labelLine` child view uses the `protoStaticText` proto to create the static text label and to activate the picker if the proto includes one.
- The `entryLine` child view uses the `protoInputLine` proto to create the input field into which the user writes text. The text value entered into this field is stored in the `text` slot of this view.

You can have your label input line protos remember a list of recent items. To do this, assign a symbol that incorporates your developer signature to the `'memory` slot of your prototype. The system automatically maintains the list of recent items for your input line. To access the list, use the same symbol with the `AddMemoryItem`, `AddMemoryItemUnique`, `GetMemoryItems`, and `GetMemorySlot` functions, which are described in “Utility Functions” (page 26-1).

CHAPTER 8

Text and Ink Input and Display

IMPORTANT

You can programmatically access the value of the text slot for the `protoLabelInputLine` with the expression `entryLine.text`. If you update the text slot programmatically, you need to call the `SetValue` function to ensure that the view is updated. Below is an example:

```
SetValue(entryLine, 'text', "new text")]
```



The following is an example of a template using `protoLabelInputLine`:

```
labelLine := {...
  _proto: protoLabelInputLine,
  viewBounds: {top: 90, left: 42, right: 194, bottom: 114},
  label: "Who",
  labelCommands: ["Me", "You", "Us", 'pickseparator',
                  "Them", "No one"],
  curLabelCommand: 0,
  ...}
```

The slots of the `protoLabelInputLine` are described in “`protoLabelInputLine`” (page 7-19) in *Newton Programmer’s Reference*.

protoRichLabelInputLine

This proto works exactly like `protoLabelInputLine`. The only difference is that `protoRichLabelInputLine` allows mixed ink and text input, as determined by the current user recognition preferences.

The slots of the `protoRichLabelInputLine` are described in “`protoRichLabelInputLine`” (page 7-22) in *Newton Programmer’s Reference*.

Displaying Text and Ink

In addition to knowing about the views and protos that you can use for displaying text and ink, you should understand how text and ink are displayed. This involves the use of fonts, text styles, and rich strings. This section describes these objects and how you can use them in your applications to control the display of text and ink.

Text and Ink in Views

When the user draws with the pen on the Newton screen, pen input data is captured as ink, which is also known as sketch ink or raw ink.

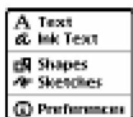
What happens with the raw ink depends upon the configuration of the view in which the input action was performed and the choices that the user made in the

CHAPTER 8

Text and Ink Input and Display

Recognition menu. The view configuration is defined by the view flags and the (optional) recognition configuration (`recConfig`) frame of the view. The Recognition menu is shown in Figure 8-3.

Figure 8-3 The Recognition menu



When the `viewFlags` input flags and the `recConfig` frame of the view are set to accept both text and ink, the Recognition menu choices control what kind of data is inserted into the paragraph view.

Note that you can limit the choices that are available in the Recognition menu of your application, though this is rarely necessary or advisable.

The Recognition menu, recognition view flags, and the recognition configuration frame are described in “Recognition” (page 9-1).

Mixing Text and Ink in Views

Some views require textual input and cannot accept ink words. The recognition controls are not used by these text-only views, in which writing is always recognized and inserted as text. If the user drops an ink word into a text-only field, the ink word is automatically recognized before control returns to the user.

Edit views can handle both ink words and sketch ink. If an edit view receives an ink word, it either merges that word into an existing paragraph view or creates a new view for the ink word. If an edit view receives sketch ink, it creates a polygon view for the ink drawing.

You can also create fields that accepts only ink words. However, if the user types or drops recognized text into such a field, the recognized text remains recognized text.

You can set a paragraph view to accept either text or ink input with the following settings:

```
viewClass: clParagraphView,
  viewFlags: vVisible + vClipping + vClickable +
             vGesturesAllowed + vCharsAllowed +
             vNumbersAllowed,
  recConfig: rcInkOrText
```

CHAPTER 8

Text and Ink Input and Display

Note

The view flags are described in “Views” (page 3-1). The recognition view flags are described in “Recognition” (page 9-1). ♦

Although raw ink is intended mostly for drawing, the user can still write with raw ink by choosing “Sketches” from the Recognition menu. The recognizer automatically segments raw ink into ink words. The raw ink can subsequently be recognized, using deferred recognition. Unlike ink text, raw ink is not moved or resized after it is written.

When raw ink from a 1.x system is dragged into a paragraph view, each piece of ink is automatically converted into an ink word. This conversion is not reversible.

Note

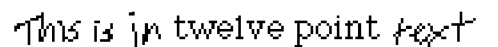
You can use one of two representations for text and ink that are mixed together. The first and more common representation is as a rich string, as described in “Rich Strings” (page 8-22). The second representation, used in paragraph views, is as a text string with a corresponding series of matching style runs. This representation, which is used for editing operations in paragraph views, is described in “Text and Styles” (page 8-25). ♦

Ink Word Scaling and Styling

Ink words are drawn using the pen thickness that the user specifies in the Styles menu. After the ink words are drawn, they are scaled by the system software. The scaling value is specified in the Text Editing Settings menu, which the user can access by choosing Preferences from the Recognition menu.

The standard values for scaling ink words are 50 percent, 75 percent, and 100 percent. After the system performs scaling, it assigns a font style and size to the ink word. The initial style is plain. The initial size is proportional to the x-height of the ink word, as estimated by the recognizer. This size is defined so that an ink word of a certain size will be roughly the same size as a text word displayed in a font of that size. For example, an ink word of size 12 is drawn at roughly the same size as a text word in a typical 12-point font, as shown in Figure 8-4. The ink words in Figure 8-4 were first scaled to 50 percent of their written size.

Figure 8-4 Resized and recognized ink



This is 12 twelve point text

CHAPTER 8

Text and Ink Input and Display

You can modify the size at which ink words are displayed in two ways: by changing the scaling percentage or the font size. For example, suppose that you draw an ink word and the system calculates its font size, as written, at 36 point. If your ink text scaling is set to 50 percent, the ink word is displayed at half of the written size, which makes its font size 18 point. If you subsequently change the scaling of that ink word to 100 percent, its font size changes to 36 point.

If the user applies deferred recognition to the ink words, the recognized text is displayed in the current font family, size, and style, as specified in the Styles menu.

Note

There is a maximum ink word size. Ink words are scaled to the smaller of what would be produced by the selected scaling percentage or the maximum size. ♦

Constraining Font Style in Views

You can override the use of styles in a paragraph view so that all of the text in the paragraph is displayed with a certain font specification. To do this, use the `viewFont` slot of the paragraph view along with two of the text view flags.

If you include `vFixedTextStyle` in the text flags for a paragraph view, all recognized text in the view is displayed using the font family, point size, and character style specified for `viewFont`. This is the normal behavior for input fields.

If you include `vFixedInkTextStyle` in the text flags for a paragraph view, all ink words in the view are displayed using the point size and character style specified for `viewFont`. Note that the font family does not affect the display of ink words.

Note

Using the `vFixedTextStyle` or `vFixedInkTextStyle` flags does not modify the `'styles` slot of the view. However, if you use either of these flags, the system does not allow the user to change the text style for your paragraph view. ♦

The text view flags are described in “Text Flags” (page 7-2) in *Newton Programmer’s Reference*.

Using Fonts for Text and Ink Display

Whenever recognized text is drawn on the Newton screen, the system software examines the font specification associated with the text to determine how to draw the text. The font specification includes the font family name, the font style, and the point size for the text. You can specify a font with a font frame or with a packed integer; both of these formats are described in this section.

CHAPTER 8

Text and Ink Input and Display

The constants you can use in font specifications are shown in “Font Constants for Packed Font Integer Specifications” (page 7-4) in *Newton Programmer’s Reference*.

The Font Frame

A font frame has the following format:

```
{family: familyName, face: faceNumber, size: pointSize}
```

For *familyName*, you can specify a symbol corresponding to one of the available built-in fonts, which are shown in Table 8-3.

Table 8-3 Font family symbols

Symbol	Font Family
'espy	Espy (system) font
'geneva	Geneva font
'newYork	New York font
'handwriting	Casual (handwriting) font

For *faceNumber*, you can specify a combination of the values shown in Table 8-4:

Table 8-4 Font style (face) values

Constant	Value	Font face
kFaceNormal	0x000	Normal font
kFaceBold	0x001	Bold font
kFaceItalic	0x002	Italic font
kFaceUnderline	0x004	Underline font
kFaceOutline	0x008	Outline font
kFaceSuperScript	0x080	Superscript font
kFaceSubscript	0x100	Subscript font

CHAPTER 8

Text and Ink Input and Display

Note

Apple recommending using the normal, bold, and underline font styles. The other styles do not necessarily display well on Newton screens. ♦

For *pointSize*, use an integer that specifies the point size value.

The Packed Integer Font Specification

You can specify a font in one 30-bit integer. A packed integer font specification uses the lower 10 bits for the font family, the middle 10 bits for the font size, and the upper 10 bits for the font style. Since only the ROM fonts have predefined font family number constants, you can only specify ROM fonts in a packed value.

Using the Built-in Fonts

The system provides several constants you can use to specify one of the built-in fonts. These constants are listed in Table 8-5. The fonts shown in the table can be specified by the constant (usable at compile time only), by their font frame, or by an integer value that packs all of the font information into an integer (sometimes this is what you see at run time if you examine a `viewFont` slot in the NTK Inspector).

Table 8-5 Built-in font constants

Constant	Font frame	Integer value
ROM_fontsystem9	{family:'espy, face:0, size:9}	9216
ROM_fontsystem9bold	{family:'espy, face:1, size:9}	1057792
ROM_fontsystem9underline	{family:'espy, face:4, size:9}	4203520
ROM_fontsystem10	{family:'espy, face:0, size:10}	10240
ROM_fontsystem10bold	{family:'espy, face:1, size:10}	1058816
ROM_fontsystem10underline	{family:'espy, face:4, size:10}	4204544
ROM_fontsystem12	{family:'espy, face:0, size:12}	12288
ROM_fontsystem12bold	{family:'espy, face:1, size:12}	1060864

continued

CHAPTER 8

Text and Ink Input and Display

Table 8-5 Built-in font constants (continued)

Constant	Font frame	Integer value
ROM_fontsystem12underline	{family:'espy, face:4, size:12}	4206592
ROM_fontsystem14	{family:'espy, face:0, size:14}	14336
ROM_fontsystem14bold	{family:'espy, face:1, size:14}	1062912
ROM_fontsystem14underline	{family:'espy, face:4, size:14}	4208640
ROM_fontsystem18	{family:'espy, face:0, size:18}	18432
ROM_fontsystem18bold	{family:'espy, face:1, size:18}	1067008
ROM_fontsystem18underline	{family:'espy, face:4, size:18}	4212736
simpleFont9	{family:'geneva, face:0, size:9}	9218
simpleFont10	{family:'geneva, face:0, size:10}	10242
simpleFont12	{family:'geneva, face:0, size:12}	12290
simpleFont18	{family:'geneva, face:0, size:18}	18434
fancyFont9 or userFont9	{family:'newYork, face:0, size:9}	9217
fancyFont10 or userFont10	{family:'newYork, face:0, size:10}	10241
fancyFont12 or userFont12	{family:'newYork, face:0, size:12}	12289
fancyFont18 or userFont18	{family:'newYork, face:0, size:18}	18433

continued

CHAPTER 8

Text and Ink Input and Display

Table 8-5 Built-in font constants (continued)

Constant	Font frame	Integer value
editFont10	{family:'handwriting, face:0, size:10}	10243
editFont12	{family:'handwriting, face:0, size:12}	12291
editFont18	{family:'handwriting, face:0, size:18}	18435

The integers in Table 8-5 are derived by packing font family, face, and size information into a single integer value. Each NewtonScript integer is 30 bits in length. In packed font specifications, the lower 10 bits hold the font family, the middle 10 bits hold the font size, and the upper 10 bits hold the font style.

These three parts added together specify a single font in one integer value. You can use the constants listed in Table 8-6 at compile time to specify all of the needed information. To do this, add one constant from each category together to yield a complete font specification. At run time, of course, you'll need to use the integer values.

Table 8-6 Font packing constants

Constant	Value	Description
Font Family		
(none defined)	0	Identifies the System font (Espy)
tsFancy	1	Identifies the New York font
tsSimple	2	Identifies the Geneva font
tsHWFont	3	Identifies the Casual (Handwriting) font
Font Size		
tsSize (<i>pointSize</i>)	<i>pointSize</i> << 10	Specify the point size of the font in <i>pointSize</i>
Font Face		
tsPlain	0	Normal font
tsBold	1048576	Bold font

continued

CHAPTER 8

Text and Ink Input and Display

Table 8-6 Font packing constants (continued)

Constant	Value	Description
<code>tsItalic</code>	2097152	Italic font
<code>tsUnderline</code>	4194304	Underlined normal font
<code>tsOutline</code>	8388608	Outline font
<code>tsSuperScript</code>	134217728	Superscript font
<code>tsSubScript</code>	268435456	Subscript font

Note that the “Casual” font uses the symbol `'handwriting` for its font family.

You can use the `MakeCompactFont` function at runtime to create a packed integer value from a specification of the font family, font size, and font face. You can only specify ROM fonts with the packed integer format. Here is an example:

```
fontValue := MakeCompactFont('tsSimple, 12, tsItalic)
```

If the font specified by the three parameters does not belong to a ROM font family, `MakeCompactFont` returns a font frame instead.

The `MakeCompactFont` function is described in “`MakeCompactFont`” (page 7-28) in *Newton Programmer's Reference*.

Rich Strings

Rich strings store text strings and ink in a single string. If your application supports user-input text or ink, you can use rich strings to represent all user data. You can convert between the text and styles pairs in paragraph views and rich strings. Text and styles pair are described in “Text and Styles” (page 8-25).

Rich strings are especially useful for storing text with embedded ink in a soup. You can use the rich string functions, described in “Rich String Functions” (page 8-24), to work with rich strings.

The system software automatically handles rich strings properly, including their use in performing the following operations:

- screen display
- sorting and indexing
- concatenation with standard functions such as `StrConcat` and `ParamStr`, described in “Utility Functions” (page 26-1)
- measuring

CHAPTER 8

Text and Ink Input and Display

Important Rich String Considerations

Although the Newton system software allows you to use rich strings anywhere that plain strings are used, there are certain considerations to be aware of when using rich strings. These include:

- Do not use functions that are not rich-string-aware. These include the `Length`, `SetLength`, `BinaryMunger`, and `StuffXXX` functions.
- Use the `StrLen` function to find the length of a string.
- Use the `StrMunger` function to perform operations that modify the length of a string, such as appending or deleting characters.
- Do not assume that the rich string terminator character is the last character in a rich string object.
- Do not truncate a rich string by inserting a string terminator character into the string.
- Do not assign characters into a rich string, due to the presence of ink placeholder characters. Use the `SetChar` function instead of direct assignment.
- Do not use undocumented string functions, which are not guaranteed to work with rich strings.

Using the Rich String Storage Format

Ink data is embedded in rich strings by inserting a placeholder character in the string for each ink word. Data for each ink word is stored following the string terminator character.

Each ink word is represented in the text portion of the rich string by the special character `kInkChar` (0xF700), which is a reserved Unicode character value.

The ink data for all ink words in the string follows the string terminator character. The final 32 bits in a rich string encode information about the rich string.

Note

The string in the `'text` slot of a paragraph view uses the `kParaInkChar` (0xF701) character as a placeholder character instead of the `kInkChar` code. The `'text` slot string is not a rich string but might contain ink word placeholders. See “Text and Styles” (page 8-25) for more information. ♦

Automatic Conversion of Rich Strings

Text is automatically converted from the rich string format to a text/styles pair whenever a paragraph is opened and the `SetValue` function is called with a rich string.

When a paragraph view is opened, the `'text` slot is first examined to determine whether or not the text contains any embedded ink. If so, new versions of the

CHAPTER 8

Text and Ink Input and Display

view's `'text` and `'styles` slots are generated and placed in the context frame of the view.

When `SetValue` is called with a string parameter that is a rich string, it is automatically decoded into a text and style pair. The result is stored in the view frame of the paragraph view.

Rich String Functions

You can use the rich string functions to convert and work with rich strings. Each of these functions, shown in Table 8-7, is described in “Rich String Functions and Methods” (page 7-31) in *Newton Programmer's Reference*.

Table 8-7 Rich string functions

Function or method name	Description
<code>MakeRichString</code>	Converts the data from two slots into a rich string. <code>MakeRichString</code> uses the text from the <code>'text</code> slot of the view and the <code>styles</code> array from the <code>'styles</code> slot of the view.
<code>DecodeRichString</code>	Converts a rich string into a frame containing a <code>'text</code> slot and a <code>'styles</code> slot. These slots can be placed in a paragraph view for editing or viewing.
<code>ExtractRangeAsRichString</code>	Returns a rich string for a range of text from a paragraph view.
<code>IsRichString</code>	Determines if a string is a rich string (i.e., contains ink).
<code>view:GetRichString</code>	Returns the text from a paragraph view as a rich string or plain string, depending on whether the paragraph view contains any ink.
<code>StripInk</code>	Strips any ink from a rich string. Either removes the ink words or replaces each with a specified replacement character or string.

CHAPTER 8

Text and Ink Input and Display

Text and Styles

Within a paragraph view, text is represented in two slots: the 'text slot and the 'styles slot. The 'text slot contains the sequence of text characters in the paragraph, including an instance of the kParaInkChar placeholder character (0xF701) for each ink word.

The 'styles slot specifies how each **text run** is displayed in the paragraph. A text run is a sequence of characters that are all displayed with the same font specification. The 'styles slot consists of an array of alternating length and style information: one length value and one style specification for each text run. For ink words, the length value is always 1, and the style specification is a binary object that contains the ink data.

For example, consider the paragraph text shown in Figure 8-5.

Figure 8-5 A paragraph view containing an ink word and text



Try *this* one

In the paragraph view shown in Figure 8-5, the 'text slot contains the following sequence of Unicode characters:

```
'T' 'r' 'y' ' ' 0xF701 'o' 'n' 'e'
```

The 'styles slot for this paragraph consists of the following array:

```
styles: [4, 12289, 1, <inkData, length 42>, 4, 12289]
```

The first pair of values in the array, (4, 12289), covers the word “Try” and the space that follows it. The length value, 4, specifies that the text run consists of four characters. The packed integer font specification value 12289 specifies plain, 12-point, New York.

The second pair of values in the array, (1, inkData), covers the ink word. The length value is 1, which is always the case for ink words. The value inkData is a binary object that contains the compressed data for the handwritten “this” that is part of the text in the paragraph view. The data is automatically extracted from the tablet data as part of a preliminary recognition process that precedes word recognition.

The third and final pair of values in the 'styles slot array, (4, 12289), covers the word “one” and the space that precedes it. This text run is 4 characters long and is displayed 12 points high in the plain version of the New York font family.

Note

The packed integer font specification values are shown in Table 8-6 (page 8-21). ♦

CHAPTER 8

Text and Ink Input and Display

Setting the Caret Insertion Point

When your application starts up, you might want to establish the insertion point for keyboard entry in caret insertion writing mode. There are three functions that you can use for this purpose:

- to establish the insertion point in an input field, use the `SetKeyView` function, which is described in “SetKeyView” (page 7-43) in *Newton Programmer’s Reference*.
- to establish the insertion point in an edit view, use the `PositionCaret` function, which is described in “PositionCaret” (page 7-49) in *Newton Programmer’s Reference*.
- to establish the insertion point in an edit view or paragraph, you can use the `SetCaretInfo` function, which is described in “SetCaretInfo” (page 7-50) in *Newton Programmer’s Reference*.

Using Keyboards

You can provide the user with on-screen keyboard input in your applications using the built-in keyboard views. You can also define new keyboard views and register them with the system, which will activate caret input when these views are opened.

Keyboard Views

There are four different floating keyboards built into the system root view. Each of the built-in keyboards can be accessed as a child of the root with a symbol.

To use the full alphanumeric keyboard, which is shown in Figure 8-6, use the symbol `'alphaKeyboard`.

Figure 8-6 The built-in alphanumeric keyboard



CHAPTER 8

Text and Ink Input and Display

To use the numeric keyboard, which is shown in Figure 8-7, use the symbol 'numericKeyboard'.

Figure 8-7 The built-in numeric keyboard



To use the phone keyboard, which is shown in Figure 8-8, use the symbol 'phoneKeyboard'.

Figure 8-8 The built-in phone keyboard



To use the time and date keyboard, which is shown in Figure 8-9, use the symbol 'dateKeyboard'.

Figure 8-9 The built-in time and date keyboard



An on-screen keyboard can be opened by the user with a double tap on an input field. The kind of keyboard displayed is determined by what type of input field is recognized. For example, a field in which only numbers are recognized would use the numeric keyboard. The user can also open a keyboard from the corrector pop-up list, which appears when you correct a recognized word.

CHAPTER 8

Text and Ink Input and Display

If you want to open one of these keyboards programmatically, use code like the following to send it the `Open` message:

```
Getroot().alphaKeyboard.Open()
```

The keystrokes entered by the user are sent to the current key receiver view. There can be only one key receiver at a time, and only views of the classes `clParagraphView` and `clEditView` can be key receiver views. When a keyboard is open, a caret is shown in the key receiver view at the location where characters will be inserted.

The keyboard views are based on `clKeyboardView`, which is described in “Keyboard View (`clKeyboardView`)” (page 7-35) in *Newton Programmer’s Reference*.

Using Keyboard Protos

The keyboard protos to provide users of your applications with on-screen keyboards with which to enter text. The following keyboard protos are available:

- `protoKeyboard` provides a standard keyboard view that floats above all other views.
- `protoKeypad` allows you to define a customized floating keyboard.
- `protoKeyboardButton` includes a keyboard button in a view.
- `protoSmallKeyboardButton` includes a small keyboard button in a view.
- `protoAlphaKeyboard` provides an alphanumeric keyboard that you can include in a view.
- `protoNumericKeyboard` provides a numeric keyboard that you can include in a view.
- `protoPhoneKeyboard` provides a phone keyboard that you can include in a view.
- `protoDateKeyboardButton` provides a time and date keyboard that you can include in a view.

protoKeyboard

This proto creates a keyboard view that floats above all other views. It is centered within its parent view and appears in a location that won’t obscure the key-receiving view (normally, the view to which the keystrokes from the keyboard are to be sent). The user can drag the keyboard view by its drag-dot to a different location, if desired. Figure 8-10 shows an example of what a `protoKeyboard` looks like on the screen.

CHAPTER 8

Text and Ink Input and Display

Figure 8-10 An example of a protoKeyboard

This proto enables the caret (if it is not already visible) in the key-receiving view while the keyboard is displayed. Characters corresponding to tapped keys are inserted in the key-receiving view at the insertion bar location. The caret is disabled when the keyboard view is closed.

This proto is used in conjunction with `protoKeypad` to implement a floating keyboard. The `protoKeyboard` proto defines the parent view, and `protoKeypad` is a child view that defines the key characteristics.

protoKeypad

This proto defines key characteristics for a keyboard view (`clKeyboardView` class). It also contains functionality that automatically registers an open keyboard view with the system. If you want to get this behavior in your custom keyboard, use `protoKeypad`.

You use this proto along with `protoKeyboard` to implement a floating keyboard. The view using the `protoKeypad` proto should be a child of the view using the `protoKeyboard` proto.

protoKeyboardButton

This proto is used to include the keyboard button in a view. This is the same keyboard button shown on the status bar in the notepad. Tapping the button causes the on-screen keyboard to appear. If the keyboard is already displayed, a picker listing available keyboard types is displayed. The user can tap one to open that keyboard.

Figure 8-11 shows an example of the keyboard button.

Figure 8-11 The keyboard button

CHAPTER 8

Text and Ink Input and Display

protoSmallKeyboardButton

This proto is used to include a small keyboard button in a view. Tapping the button causes the on-screen keyboard to appear. If the keyboard is already displayed, a picker listing available keyboard types is displayed. The user can tap one to open that keyboard.

Figure 8-12 shows an example of the small keyboard button.

Figure 8-12 The small keyboard button

**protoAlphaKeyboard**

This proto is used to include an alphanumeric keyboard in a view. This is the same as the 'alphaKeyboard keyboard view provided in the root view, as described in “Keyboard Views” (page 8-26). An example of protoAlphaKeyboard is shown in Figure 8-6 (page 8-26).

protoNumericKeyboard

This proto is used to include a numeric keyboard in a view. This is the same as the 'numericKeyboard keyboard view provided in the root view, as described in “Keyboard Views” (page 8-26). An example of protoNumericKeyboard is shown in Figure 8-7 (page 8-27).

protoPhoneKeyboard

This proto is used to include a phone keyboard in a view. This is the same as the 'phoneKeyboard keyboard view provided in the root view, as described in “Keyboard Views” (page 8-26). An example of protoPhoneKeyboard is shown in Figure 8-8 (page 8-27).

protoDateKeyboard

This proto is used to include a time and date keyboard in a view. This is the same as the 'dateKeyboard keyboard view provided in the root view, as described in “Keyboard Views” (page 8-26). An example of protoDateKeyboard is shown in Figure 8-9 (page 8-27).

Defining Keys in a Keyboard View

When you define a keyboard view, you need to specify the appearance and behavior of each key in the keyboard. This section presents the definition of an example keyboard view, which is shown in Figure 8-13 (page 8-31).

CHAPTER 8

Text and Ink Input and Display

The Key Definitions Array

Each keyboard view contains a key definitions array, which determines the layout of the individual keys in the keyboard. The key definitions array is an array of rows. Each row is an array of values that looks like this:

```
row0 := [ rowHeight, rowMaxKeyHeight,
          key0Legend, key0result, key0Descriptor,
          key1Legend, key1result, key1Descriptor,
          key2Legend, key2result, key2Descriptor,
          ...
        ]
```

The first two elements describe the height to allot for the row (*rowHeight*) and the height of the tallest key in the row (*rowMaxKeyHeight*), in key units. These two measurements are often the same, but they may differ. Key units are described in “Key Dimensions” (page 8-35).

Next in the row array is a series of three elements for each key in the row:

- *keyLegend*
- *keyResult*
- *keyDescriptor*

These values are described in the following sections.

Figure 8-13 shows the example keyboard view that is used to explain key definition in this section.

Figure 8-13 A generic keyboard view

1	2	3
4	5	6
7	8	9
*	0	#

The following is the view definition of the keyboard shown in Figure 8-13. The values in the row arrays are explained in the remainder of this section.

```
row0 := [ keyVUnit, keyVUnit,
          "1",1, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite,
          "2",2, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite
          "3",3, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite ];
```

CHAPTER 8

Text and Ink Input and Display

```

row1 := [ keyVUnit, keyVUnit,
  "4",4, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite,
  "5",5, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite,
  "6",6, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite ];

row2 := [ keyVUnit, keyVUnit,
  "7",7, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite,
  "8",8, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite,
  "9",9, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite ];

row3 := [ keyVUnit, keyVUnit,
  "*",$, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite,
  "0",0, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite,
  "#",$, keyHUnit+keyVUnit+keyFramed+2*keyInsetUnit+keyAutoHilite ];

keypad := { ...
  viewClass: clKeyboardView,
  viewBounds: {left:65, top:65, right:153, bottom:145},
  viewFlags: vVisible+vClickable+vFloating,
  viewFormat: vfFrameBlack+vfFillWhite+vfPen(1),
  keyDefinitions: [ row0, row1, row2, row3 ], // defined above
  keyPressScript: func (key)
    begin
      Print("You pressed " & key);
    end,
  ... }

```

The Key Legend

The key legend specifies what appears on the keycap. It can be one of the following types of data:

- `nil`, in which case the key result is used as the legend.
- A string, which is displayed centered in the keycap.
- A character constant, which is displayed centered in the keycap.
- A bitmap object, which is displayed centered in the keycap.
- An integer. The number is displayed centered in the keycap and is used directly as the key result, unless the `keyResultsAreKeycodes` slot is set to `true`, as described in the next section.
- A method. The method is evaluated and its result is treated as if it had been specified as the legend.

CHAPTER 8

Text and Ink Input and Display

- An array. An element of the array is selected and treated as one of the above data types. The index of the array element is determined by the value of the `keyArrayIndex` slot (which can be changed dynamically). Note that arrays of arrays are not allowed here, but an array can include any combination of other data types.

The Key Result

The key result is the value returned when the key is pressed. This value is passed as a parameter to the `keyPressScript` method. If this method doesn't exist, the result is converted (if possible) into a sequence of characters that are posted as key events to the key receiver view.

The key result element can be one of the following types of data:

- A string, character constant, or bitmap object, which is simply returned.
- An integer, which is returned. Alternately, if the `keyResultsAreKeycodes` slot is set to `true`, the integer is treated as a key code. In this case, the character corresponding to the specified key code is returned. If you are using keycodes, make sure to register your keyboard by including the `kKbdUsesKeycodes` view flag.

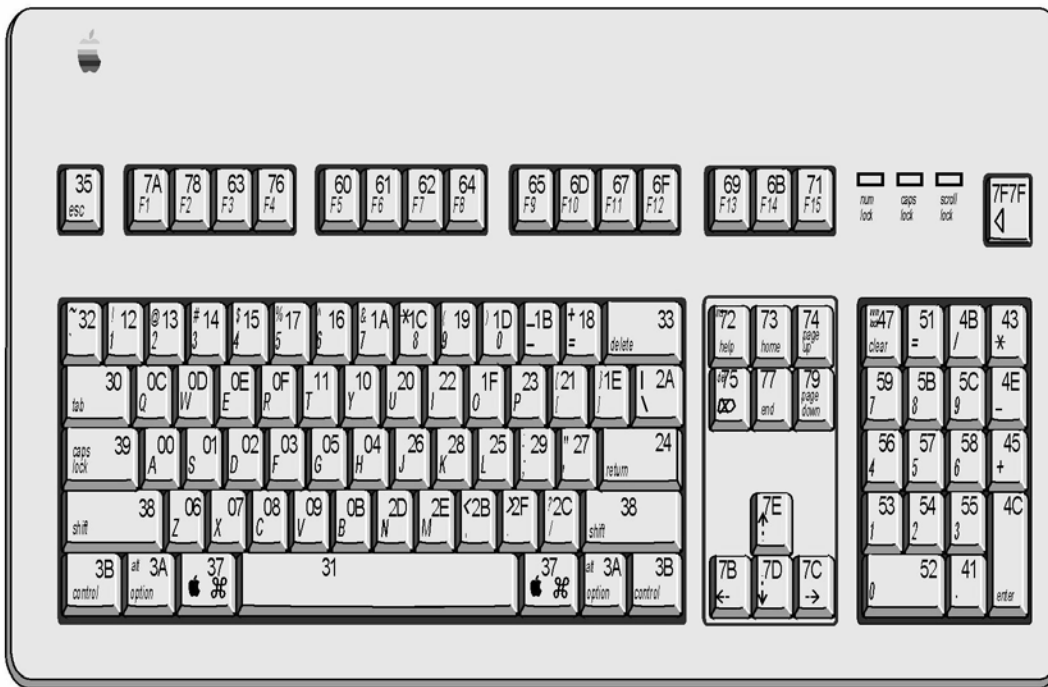
See Figure 8-14 (page 8-34) for the numeric key codes returned by each of the keys on a keyboard.

- A method. The method is evaluated and its result is treated as if it had been specified as the result.
- An array. An element of the array is selected and treated as one of the above data types. The index of the array element is determined by the value of the `keyArrayIndex` slot (which can be changed dynamically). Note that arrays of arrays are not allowed, but an array can include any combination of other data types.

CHAPTER 8

Text and Ink Input and Display

Figure 8-14 Keyboard codes



The Key Descriptor

The appearance of each key in a keyboard is determined by its key descriptor. The key descriptor is a 30-bit value that determines the key size, framing, and other characteristics. The descriptor is specified by combining any of the constants shown in Table 8-8.

Table 8-8 Key descriptor constants

keySpacer	Nothing is drawn in this space; it is a spacer, not a key.
keyAutoHilite	Highlight this key when it is pressed.
keyInsetUnit	Inset this key's frame a certain number of pixels within its space. Multiply this constant by the number of pixels you want to inset, from 0-7 (for example, keyInsetUnit*3).

continued

CHAPTER 8

Text and Ink Input and Display

Table 8-8 Key descriptor constants (continued)

keyFramed	Specify the thickness of the frame around the key. Multiply this constant by the number of pixels that you want to use for the frame thickness, from 0-3.
keyRoundingUnit	Specify the roundedness of the frame corners. Multiply this constant by the number of pixels that you want to use for the corner radius, from 0-15, zero being square.
keyLeftOpen	No frame line is drawn along the left side of this key.
keyTopOpen	No frame line is drawn along the top side of this key.
keyRightOpen	No frame line is drawn along the right side of this key.
keyBottomOpen	No frame line is drawn along the bottom side of this key.
keyHUnit keyHHalf keyHQuarter keyHEighth	A combination of these four constants specifies the horizontal dimension of the key in units. For details, see the next section.
keyVUnit keyVHalf keyVQuarter keyVEighth	A combination of these four constants specifies the vertical dimension of the key in units. For details, see the next section.

Key Dimensions

The width and height of keys are specified in units, not pixels. A key unit is not a fixed size, but is used to specify the size of a key relative to other keys. The width of a unit depends on the total width of all keys in the view and on the width of the view itself. Key widths and heights can be specified in whole units, half units, quarter units, and eighth units.

When it is displayed, the whole keyboard is scaled to fit entirely within whatever size view bounds you specify for it.

To fit the whole keyboard within the width of a view, the total unit widths are summed for each row, and the scaling is determined based on the widest row. This row is scaled to fit within the view width, giving an equal pixel width to each whole key unit. A similar process is used to scale keys vertically to fit within the height of a view.

Fractional key units (half, quarter, eighth), when scaled, must be rounded to an integer number of pixels, and thus may not be exactly the indicated fraction of a whole key unit. For example, if the keys are scaled to fit in the view bounds, a whole key unit ends up to be 13 pixels wide. This means that a key specified to have a width of $1 \frac{3}{8}$ units ($\text{keyHUnit} + \text{keyHEighth} * 3$) is rounded to $13 + 5$, or 18 pixels, which is not exactly $1 \frac{3}{8} * 13$.

CHAPTER 8

Text and Ink Input and Display

Key dimensions are specified by summing a combination of horizontal and vertical key unit constants within the `keyDescriptor`. For example, to specify a key that is $2\frac{3}{4}$ units wide by $1\frac{1}{2}$ units high, specify these constants for `keyDescriptor`:

```
keyHUnit*2 + keyHQuarter*3 + keyVUnit + keyVHalf
```

Using the Keyboard Registry

If your application includes its own keyboard, you need to register it with the system keyboard registry. This makes it possible for the system to call any keyboard-related functions that you have defined and to handle the insertion caret properly.

The `RegisterOpenKeyboard` method of a view is for registering a keyboard for use with that view.

Use the `UnregisterOpenKeyboard` method of a view to remove the keyboard view from the registry. If the insertion caret is visible, calling this method hides it.

Note

The system automatically unregisters the keyboard when the registered view is hidden or closed. The `protokeypad` proto also automatically handles registration for you in its `viewSetupDoneScript`. You do not need to call the `UnregisterOpenKeyboard` method in these cases. ♦

You can use the `OpenKeypadFor` function to open a context-sensitive keyboard for a view. This function first attempts to open the keyboard defined in the view's `_keyboard` slot. If the view does not define a keyboard in that slot, `OpenKeypadFor` determines if the view allows only a single type of input, such as date, time, phone number, or numbers. If so, `OpenKeypadFor` opens the appropriate built-in keyboard for that input type. If none of these other conditions is met, `OpenKeypadFor` opens the `alphaKeyboard` keyboard for the view.

Note

The Newton System Software uses the `OpenKeypadFor` function to open a context-sensitive keyboard when the user double-taps on a view in which a `_keyboard` slot is defined. ♦

These methods and functions, as well as several others you can use with the keyboard registry in your applications, are described in “Keyboard Registry Functions and Methods” (page 7-44) in *Newton Programmer's Reference*.

Defining Tabbing Orders

You can define the tabbing order for an input view with the `_tabChildren` slot, which contains an array of view paths.

CHAPTER 8

Text and Ink Input and Display

Each view path must specify the actual view that accepts the input. An example of a suitable path is shown here:

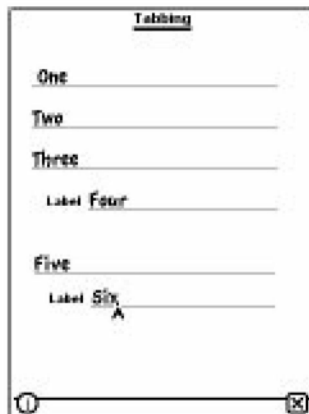
```
'myInputLine, 'myLabelInputLine.entryLine
```

When the user tabs through this list, it loops from end to beginning and, with reverse-tabbing, from beginning to end.

You can use the `_tabParent` slot to inform the system that you want tabbing in a view restricted to that view. Each view in which `_tabParent` is non-nil defines a tabbing context. This makes it possible to have several views on the screen at once with independent tabbing within each view. In this case, the user must tap in another view to access the tabbing order in that view.

For example, in Figure 8-15, there are two independent tabbing orders. The first consists of the input lines that contain the text “One,” “Two,” “Three,” and “Four.” The second tabbing order consists of the input lines that contain the text “Five” and “Six.”

Figure 8-15 Independent tabbing orders within a parent view



The user taps in any of the top four slots; thereafter, pressing the tab key on a keypad or external keyboard moves among the four slots in that tabbing order. If the user taps one of the bottom two slots, the tab key jumps between those two slots.

The slots `_tabParent` and `_tabChildren` can coexist in a view, but the `_tabChildren` slot takes precedence in specifying the next key view. If the current view does not define the `_tabParent` slot, the search moves upward from the current view until one of the following conditions is met:

- a view descended from `protoInputLine` with a `_tabParent` slot is found.
- a `protoFloater` view is found

Using Text

8-37

CHAPTER 8

Text and Ink Input and Display

- a view descended from `protoInputLine` with the `vApplication` flag set in the `viewFlags` slot

The Caret Pop-up Menu

Normally, when the user taps the insertion caret, the system-provided Punctuation pop-up Menu opens. However, you can override this with a pop-up menu of your own creation.

When the user taps the insertion caret, the system starts searching for a slot named `_caretPopup`. The search begins in the view owning the caret, and follows both the proto and parent inheritance paths. The default Punctuation pop-up is stored in the root view.

The `_caretPopup` slot must hold a frame containing two slots. The first slot, `pop`, defines a list of pop-up items suitable for passing to `DoPopup`. The second slot must contain a `pickActionScript`. If not, control passes to the punctuation pop-up, which has its own version of the `pickActionScript`. This routine then inserts a string, corresponding to the selected character at the caret, by using the function `PostKeyString`.

Handling Input Events

You sometimes need to respond to input events that occur in text views. This section describes how to test for a selection hit and respond to keystrokes and insertion events.

Testing for a Selection Hit

After the user taps the screen, you can determine if the point “hits” a specific character or word in a paragraph view.

The `view:PointToCharOffset` method returns the offset within the paragraph that is closest to the point (x, y) . This method is described in “PointToCharOffset” (page 7-51) in *Newton Programmer’s Reference*.

The `view:PointToWord` method returns a frame that indicates the position of the word within the paragraph that is closest to the point (x, y) . This method is described in “PointToWord” (page 7-52) in *Newton Programmer’s Reference*.

Note

Both of these functions return `nil` if the view is not a paragraph view. Also, the point you are testing must correspond to a visible position within the paragraph view; you cannot hit-test on off-screen portions of a view. ♦

CHAPTER 8

Text and Ink Input and Display

Summary of Text

Text Constants and Data Structures

Text Flags

vWidthIsParentWidth	(1 << 0)
vNoSpaces	(1 << 1)
vWidthGrowsWithText	(1 << 2)
vFixedTextStyle	(1 << 3)
vFixedInkTextSTyle	(1 << 4)
vExpectingNumbers	(1 << 9)

Font Family Constants for Use in Frames

'espy
'geneva
'newYork
'handwriting

Font Face Constants for Use in Frames

kFaceNormal	0x000
kFaceBold	0x001
kFaceItalic	0x002
kFaceUnderline	0x004
kFaceOutline	0x008
kFaceSuperScript	0x0080
kFaceSubScript	0x100

Built-in Font Constants

ROM_fontsystem9	9216
ROM_fontsystem9bold	1057792
ROM_fontsystem9underline	4203520
ROM_fontsystem10	10240
ROM_fontsystem10bold	1058816

Summary of Text

8-39

CHAPTER 8

Text and Ink Input and Display

ROM_fontsystem10underline	4204544
ROM_fontsystem12	12288
ROM_fontsystem12bold	1060864
ROM_fontsystem12underline	4206592
ROM_fontsystem14	14336
ROM_fontsystem14bold	1062912
ROM_fontsystem14underline	4208640
ROM_fontsystem18	18432
ROM_fontsystem18bold	1067008
ROM_fontsystem18underline	4212736

simpleFont9	9218
simpleFont10	10242
simpleFont12	12290
simpleFont18	18434

fancyFont9 or userFont9	9217
fancyFont10 or userFont10	10241
fancyFont12 or userFont12	12289
fancyFont18 or userFont18	18433

editFont10	10243
editFont12	12291
editFont18	18435

Font Family Constants for Packed Integer Font Specifications

tsFancy	1
tsSimple	2
tsHWFont	3

CHAPTER 8

Text and Ink Input and Display

Font Face Constants for Packed Integer Font Specifications

tsPlain	0
tsBold	1048576
tsItalic	2097152
tsUnderline	4194304
tsOutline	8388608
tsSuperScript	134217728
tsSubScript	268435456

Keyboard Registration Constants

kKbdUsesKeyCodes	1
kKbdTracksCaret	2
kKbdforInput	4

Key Descriptor Constants

keySpacer	(1 << 29)
keyAutoHilite	(1 << 28)
keyInsetUnit	(1 << 25)
keyFramed	(1 << 23)
keyRoundingUnit	(1 << 20)
keyLeftOpen	(1 << 19)
keyTopOpen	(1 << 18)
keyRightOpen	(1 << 17)
keyBottomOpen	(1 << 16)
keyHUnit	(1 << 11)
keyHHalf	(1 << 10)
keyHQuarter	(1 << 9)
keyHEighth	(1 << 8)
keyVUnit	(1 << 3)
keyVHalf	(1 << 2)
keyVQuarter	(1 << 1)
keyVEighth	(1 << 0)

CHAPTER 8

Text and Ink Input and Display

Keyboard Modifier Keys

kIsSoftKeyboard	(1 << 24)
kCommandModifier	(1 << 25)
kShiftModifier	(1 << 26)
kCapsLockModifier	(1 << 27)
kOptionsModifier	(1 << 28)
kControlModifier	(1 << 29)

Views

clEditView

```

aClEditView:= {
viewBounds:           boundsFrame,
viewFlags:            constant,
viewFormat:           formatFlags,
viewLineSpacing:     integer,
viewLinePattern:     integer,

view>EditAddWordScript (form, bounds)
NotesText (childArray)
...
}

```

clParagraphView

```

aClEditView:= {
viewBounds:           boundsFrame,
viewFont:             fontFrame,
text:                 string,
viewFlags:            constant,
viewFormat:           formatFlags,
viewJustify:          constant,
tabs:                 array,           // tab stops
styles:               array,           // style runs
textFlags:            constant,
copyProtection:      constant,
...
}

```


CHAPTER 8

Text and Ink Input and Display

clKeyboardView

```

aClEditView:= {
  _noRepeat:           constant,
  viewBounds:         boundsFrame,
  keyDefinitions:     array,      // defines key layout
  viewFlags:          constant,
  viewFormat:         constant,
  keyArrayIndex:      array,      // key legends
  keyHighlightKeys:   array,      // keys to highlight
  keyResultsAreKeycodes: Boolean,
  keyReceiverView:   view,       // view for keystrokes
  keySound:           soundFrame,
  keyPressScript:     function
  ...
}

```

Protos

protoInputLine

```

aprotoInputLine:= {
  _proto : protoInputLine,
  viewBounds:           boundsFrame,
  viewFlags:           constant,
  text:                 string,
  viewFont:            constant,
  viewJustify:         constant,
  viewFormat:          constant,
  viewTransferMode:    constant,
  viewLineSpacing:     integer,
  viewLinePattern:     binary,    // 8-byte pattern
  memory:              symbol,

  viewChangedScript:   function.
  ...
}

```

protoRichInputLine

```

aprotoRichInputLine:= {
  _proto : protoRichInputLine,
  viewBounds:           boundsFrame,
  viewFlags:           constant,

```

CHAPTER 8

Text and Ink Input and Display

```

text:                string,
viewFont:            constant,
viewJustify:         constant,
viewFormat:          constant,
viewTransferMode:    constant,
viewLineSpacing:     integer,
viewLinePattern:     binary, // 8-byte pattern
memory:              symbol,

viewChangedScript:  function,
...
}

```

protoLabelInputLine

```

aprotoLabelInputLine:= {
  _proto : protoLabelInputLine,
viewBounds:          boundsFrame,
entryFlags:          constant,
label:               string,
labelFont:           constant,
labelCommands:       array, // strings for list
curLabelCommand:     integer,
indent:              integer,
viewLineSpacing:     integer,
viewLinePattern:     binary, // 8-byte pattern

textSetup:           function,
updateText:          function,
textChanged:         function,
setLabelText:        function,
setLabelCommands:    function,
labelClick:          function,
labelActionScript:   function,
...
}

```

protoRichLabelInputLine

```

aprotoRichLabelInputLine:= {
  _proto : protoRichLabelInputLine,
viewBounds:          boundsFrame,
entryFlags:          constant,
label:               string,

```

CHAPTER 8

Text and Ink Input and Display

```

labelFont:           constant,
labelCommands:      array,      // strings for list
curLabelCommand:    integer,
indent:             integer,
viewLineSpacing:    integer,
viewLinePattern:    binary,     // 8-byte pattern

textSetup:          function,
updateText:         function,
textChanged:        function,
setLabelText:       function,
setLabelCommands:   function,
labelClick:         function,
labelActionScript: function,
...
}

```

protoKeyboard

```

aprotoKeyboard:= {
  _proto : protoKeyboard,
  saveBounds: boundsFrame,
  freeze: Boolean,
  ...
}

```

protoKeypad

```

aprotoKeypad:= {
  _proto : protoKeypad,
  keyDefinitions: array,      // defines key layout
  viewFont: constant,
  viewFormat: constant,
  keyArrayIndex: integer,
  keyHighlightKeys: Boolean,
  keyResultsAreKeycodes: Boolean,
  keyReceiverView: constant,
  keySound: constant,
  keyPressScript: function,
  ...
}

```

CHAPTER 8

Text and Ink Input and Display

protoKeyboardButton

```

aprotoKeyboardButton:= {
  _proto : protoKeyboardButton,
  viewFlags:          constant,
  viewBounds:         boundsFrame,
  viewJustify:        constant,
  defaultKeyboard     symbol,
  ...
}

```

protoSmallKeyboardButton

```

aprotoSmallKeyboardButton:= {
  _proto : protoSmallKeyboardButton,
  viewFlags:          constant,
  viewBounds:         boundsFrame,
  viewJustify:        constant,
  current:            symbol,
  ...
}

```

protoAlphaKeyboard

```

aprotoAlphaKeyboard:= {
  _proto : protoAlphaKeyboard,
  viewBounds:          boundsFrame,
  viewJustify:         constant,
  ...
}

```

protoNumericKeyboard

```

aprotoNumericKeyboard:= {
  _proto : protoNumericKeyboard,
  viewBounds:          boundsFrame,
  viewJustify:         constant,
  ...
}

```

CHAPTER 8

Text and Ink Input and Display

protoPhoneKeyboard

```

aprotoPhoneKeyboard:= {
  _proto : protoPhoneKeyboard,
  viewBounds:          boundsFrame,
  viewJustify:         constant,
  ...
}

```

protoDateKeyboard

```

aprotoDateKeyboard:= {
  _proto : protoDateKeyboard,
  viewBounds:          boundsFrame,
  viewJustify:         constant,
  ...
}

```

Text and Ink Display Functions and Methods

This section summarizes the functions and methods you can use to work with text and ink in your applications.

Functions and Methods for Edit Views

```

view>EditAddWordScript (form, bounds)
NotesText (childArray)

```

Functions and Methods for Measuring Text Views

```

TextBounds (rStr, fontSpec, viewBounds)
TotalTextBounds (paraSpec, editSpec)

```

Functions and Methods for Determining View Ink Types

```

AddInk (edit, poly)
ViewAllowsInk (view)
ViewAllowsInkWords (view)

```

CHAPTER 8

Text and Ink Input and Display

Font Attribute Functions and Methods

FontAscent (*fontSpec*)
FontDescent (*fontSpec*)
FontHeight (*fontSpec*)
FontLeading (*fontSpec*)
GetFontFace (*fontSpec*)
GetFontFamilyNum (*fontSpec*)
GetFontFamilySym (*fontSpec*)
GetFontSize (*fontSpec*)
MakeCompactFont (*family, size, face*)
SetFontFace (*fontSpec, newFace*)
SetFontFamily (*fontSpec, newFamily*)
SetFontParms (*fontSpec, whichParms*)
SetFontSize (*fontSpec, newSize*)

Rich String Functions and Methods

DecodeRichString (*richString, defaultFontSpec*)
view:ExtractRangeAsRichString (*offset, length*)
view:GetRichString ()
IsRichString (*testString*)
MakeRichString (*text, styleArray*)
StripInk (*richString, replaceString*)

Functions and Methods for Accessing Ink in Views

GetInkAt (*para, index*)
NextInkIndex (*para, index*)
ParaContainsInk (*para*)
PolyContainsInk (*poly*)

CHAPTER 8

Text and Ink Input and Display

Keyboard Functions and Methods

This section summarizes the functions and methods that you can use to work with keyboards in your applications.

General Keyboard Functions and Methods

GetCaretBox()
view: KeyboardInput()
 KeyIn(*keyCode*, *down*)
 PostKeyString(*view*, *keyString*)
 SetKeyView(*view*, *offset*)

Keyboard Registry Functions and Methods

KeyboardConnected()
 OpenKeyPadFor(*view*)
 RegGlobalKeyboard(*kbdSymbol*, *kbdTemplate*)
view: RegisterOpenKeyboard(*flags*)
 UnRegGlobalKeyboard(*kbdSymbol*)
view: UnregisterOpenKeyboard()

Caret Insertion Writing Mode Functions and Methods

GetRemoteWriting()
 SetRemoteWriting(*newSetting*)

Insertion Caret Functions and Methods

GetCaretInfo()
 GetKeyView()
view: PositionCaret(*x*, *y*, *playSound*)
 SetCaretInfo(*view*, *info*)

Application-Defined Methods for Keyboards

ViewCaretChangedScript(*view*, *offset*, *length*)

CHAPTER 8

Text and Ink Input and Display

Input Event Functions and Methods

This section summarizes the functions and methods that you can use to work with input events in your applications.

Functions and Methods for Hit-Testing

view:PointToCharOffset(x,y)

view:PointToWord(x,y)

Functions and Methods for Handling Insertions

view:HandleInsertItems(insertSpec)

InsertItemsAtCaret(insertSpec)

Functions and Methods for Handling Ink Words

GetInkWordInfo(inkWord)

view:HandleInkWord(strokeBundle)

view:HandleRawInk(strokeBundle)

Application-Defined Methods for Handling Ink in a View

view:ViewInkWordScript(strokeBundle)

view:ViewRawInkScript(strokeBundle)

CHAPTER 9

Recognition

This chapter and Chapter 10, “Recognition: Advanced Topics,” describe the use of the Newton recognition system. The recognition system accepts written input from views and returns text, ink text, graphical objects, or sketch ink to them.

This chapter describes how to use view flags to enable the recognition of text, shapes and gestures in views. If you are developing an application that must derive text or graphical data from pen input, you should become familiar with the contents of this chapter. Before reading this chapter, you should be familiar with NewtonScript message passing among views and the use of view flags to specify the characteristics of views, as described in Chapter 3, “Views.”

You need not read Chapter 10, “Recognition: Advanced Topics,” unless you need to provide unusual input views or specialized recognition behavior. (See that chapter’s first page for a complete list of its topics.)

About the Recognition System

The Newton recognition system enables views to convert handwritten input into text or graphical shapes, and to take action in response to system-defined gestures such as taps and scrubs.

Any type of view can accept pen input, and different types of views provide different amounts of system-supplied behavior. Views based on the system-supplied `clEditView` and `clParagraphView` classes handle most forms of pen input automatically. Applications need not handle recognition events in these types of views explicitly unless they need to do something unusual. The `clView` class, on the other hand, provides no built-in recognition behavior. Views of this type must provide all recognition behavior themselves.

The system provides recognizer engines (also called **recognizers**) that classify pen input as clicks, strokes, gestures, shapes, or words. Each view can specify independently which recognizers it uses and how the recognition system is to process pen input that occurs within its boundaries. For example, you could configure a view to recognize text and shapes, or you might enable only text recognition in a view not intended to accept graphical input.

CHAPTER 9

Recognition

Although no recognizers are associated with clicks and strokes, they do pass through the recognition system, allowing your view to respond to them by means of optional `ViewClickScript` and `ViewStrokeScript` methods that you supply as necessary. The `ViewClickScript` method of a view that accepts pen input takes application-specific action when the pen contacts or leaves the surface of the screen within the view's boundaries. The `ViewStrokeScript` method performs application-specific processing of input strokes before they are passed on to the gesture, shape, or text recognizers.

The gesture recognizer identifies system-defined gestures such as scrubbing items on the screen, adding spaces to words, selecting items on the screen, and so on. Views based on the `clEditView` and `clParagraphView` classes (edit views and paragraph views, respectively) respond automatically to standard system-defined gestures; other kinds of views do not. Your view can provide an optional `ViewGestureScript` method to perform application-specific processing of system-defined gestures. You cannot define new gestures to the system.

Only views based on the `clEditView` class can recognize shapes. The shape recognizer uses symmetry found in input strokes to classify them as shapes. The shape recognizer may make the original shape more symmetrical, straighten its curves, or close the shape. There is no developer interface to shape recognition.

The system provides two text recognizers—one optimized for a printed handwriting style and another optimized for a cursive handwriting style. The printed text recognizer (also called the **printed recognizer**) requires that the user lift the pen from the screen between letters. The cursive text recognizer (also called the **cursive recognizer**) accepts cursive input (letters connected within a single word), printed input (disconnected letters within a single word), or any combination of these two kinds of input.

In views that recognize text, the system enables the printed recognizer by default unless the cursive recognizer is enabled explicitly. The user can specify the use of a particular text recognizer from within the Handwriting Recognition preferences slip. This user preference slip and others that affect recognition behavior are discussed in “User Preferences for Recognition” beginning on page 9-14.

Only one text recognizer can be active at a time—all views on the screen share the same text recognizer—but individual views can specify options that customize its behavior for a particular view. Individual views can also use any combination of other recognizers in addition to the specified text recognizer. Regardless of which text recognizer is in use, the recognition system limits the size of individual input strings to 32 characters—longer words may not be recognized correctly.

Although the Newton platform currently supports only its built-in recognizers, future versions of the system may permit the use of third-party recognizer engines.

The next section describes how the recognition system classifies input as text, shapes, or gestures.

CHAPTER 9

Recognition

Classifying Strokes

Recognition is an iterative process that compares raw input strokes with various system-defined models to identify the best matches for further processing. When the user writes or draws in an edit view or paragraph view that accepts user input, the system

- notifies the view that a pen event occurred within its boundaries.
- provides user feedback, in the form of electronic ink drawn on the screen as the pen moves across its surface.
- attempts to group strokes meaningfully according to temporal and spatial data.

A view that accepts pen input is notified of pen events within its boundaries by `ViewClickScript` messages that are sent when the pen contacts the screen and when it is lifted from the screen. Views based on the `clEditView` and `clParagraphView` classes handle these events automatically; other views may not, depending on the type of view in which the pen event occurred. Your view can supply an optional `ViewClickScript` method to take application-specific action in response to these events as necessary.

The electronic ink displayed as the pen moves across the screen is called **raw ink**. Raw ink is drawn in the same place on the screen as the original input strokes. Views based on the `clParagraphView` view class can be configured to replace the raw ink with another representation of the input strokes called ink text. **Ink text** is a version of the original strokes that has been scaled for display and formatted into paragraphs: spaces between groups of strokes are made uniform and groups of strokes are wrapped to the margins of the screen. The size to which ink text is scaled is specified by the user from the Text Editing Settings user preference slip. This user preference slip and others that affect recognition behavior are discussed in “User Preferences for Recognition” beginning on page 9-14.

The recognition system encapsulates raw input strokes in an object called a **stroke unit**. Stroke units cannot be examined directly from `NewtonScript`; however, you can pass them to functions that construct useful objects from them or perform recognition using the stroke data they contain.

Views configured to image input as ink text display a scaled representation of the original input strokes without performing any further processing; that is, they circumvent the remainder of the recognition process described here.

When stroke units are made available to a view that performs recognition, all of the recognizers enabled for the view compete equally to classify the input. Each recognizer compares the input to a system-defined model; if there is a match, the recognizer involved claims the stroke unit as its own.

Once a stroke unit is claimed by one of the recognizers, it is not returned to the other recognizers for additional classification; however, recognizers may combine

CHAPTER 9

Recognition

multiple stroke units into meaningful groups. For example, certain letters (such as an uppercase *E*) might be composed of multiple strokes. The process of grouping input strokes is influenced by the user preference settings for handwriting style and letter styles.

The recognizer that claimed one or more stroke units returns to the view one or more interpretations of the strokes. The gesture and shape recognizers return only one interpretation to the view. The text recognizer usually returns multiple interpretations to the view.

Associated with each interpretation is a value, called the **score**, which indicates how well the input matched the system-defined model used by the recognizer that interpreted it. When multiple recognizers are enabled, the system selects the best interpretations based on their scores and the application of appropriate heuristics. For example, the text recognizer might choose between interpreting a stroke as a zero or as the letter *O* based on whether you have specified that the view accepts numeric or alphabetic input.

The recognizer that claimed the strokes places its best interpretations in another kind of unit that is returned to the view. The text recognizer returns *word units*, the shape recognizer returns *shape units*, and the gesture recognizer returns *gesture units*. Each of these units contains data representing one or more strokes. A **word unit** represents a single recognized word, a **shape unit** represents a single recognized shape, and a **gesture unit** represents a single recognized gesture, as shown in Figure 9-1. The next several sections describe how the system handles each of these units.

Gestures

When the recognition system returns a gesture unit to the view, the view performs the action associated with that gesture automatically. The action taken is dependent on the kind of view that received the gesture unit.

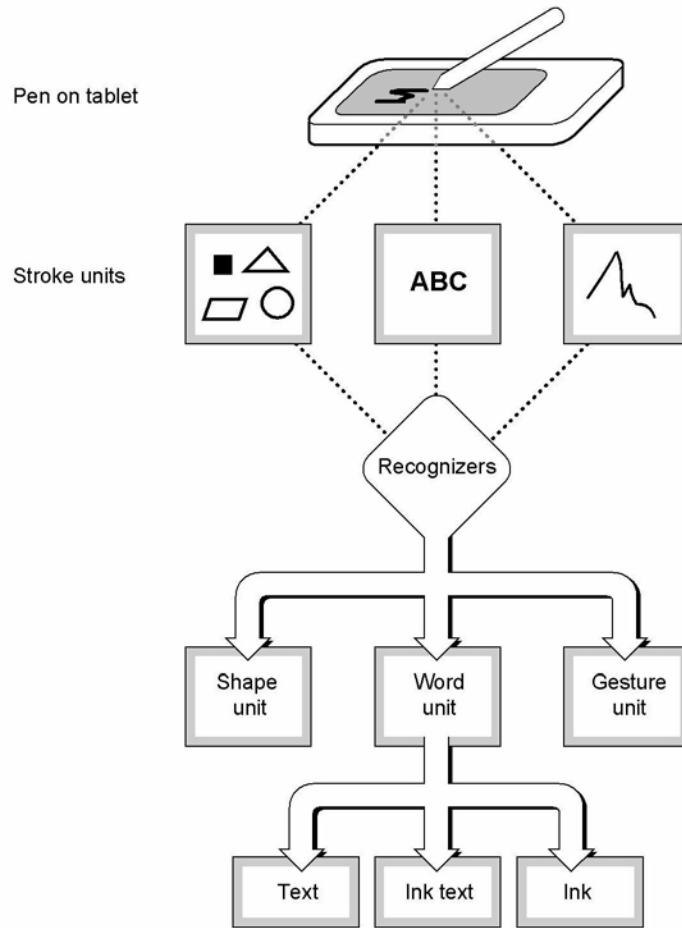
Edit views and paragraph views respond automatically to system-defined gestures such as scrubbing items on the screen, adding spaces to words, selecting items on the screen, and so on. Other kinds of views may do nothing in response to a particular gesture.

You can provide an optional `ViewGestureScript` method to take action in response to any standard gesture. For example, you can use this method to respond to gestures in views that are not paragraph views or edit views. You can also use this method to override or augment the standard behavior of a particular view in response to system-defined gestures. At present, you cannot define custom gestures to the system.

CHAPTER 9

Recognition

Figure 9-1 Recognizers create units from input strokes



Shapes

When the recognition system returns a shape unit to the view, the shape is displayed as the `clPolygonView` child view of a `clEditView` view. The shape unit contains a single, cleaned-up version of the original strokes. The shape recognizer may make the original shape more symmetrical, straighten its curves, or close the shape.

There is no developer interface to shape recognition. To manipulate shapes returned by the recognition system, you must extract polygon view children from edit views yourself. You can do so from within an optional `ViewAddChildScript` method that you supply. The system invokes this method for each `clPolygonView` or `clParagraphView` child added to an edit view.

CHAPTER 9

Recognition

Text

When the recognition system returns a word unit to a view based on the `clParagraphView` or `clEditView` classes, the view displays or uses the best interpretation of the original input strokes. Paragraph views display words directly; edit views create a `clParagraphView` child automatically to display text that the recognition system returns. Additionally, the recognition system constructs a correction information frame from the word unit and saves learning data as appropriate. For more information, see “Correction and Learning” (page 9-13) and “Accessing Correction Information” (page 10-23). Your view can provide an optional `ViewWordScript` method to perform application-specific processing of the word unit.

The set of possible interpretations that the text recognizer returns to a view is affected by

- the text recognizer that the view uses to interpret the input strokes
- options you have specified for the text recognizer in use
- the dictionaries that are available to the view for recognition use

A **dictionary** is a system construct against which the user’s input strings are matched, as a means of ensuring the validity of the text recognizer’s output. The system supplies dictionaries that define names, places, dates, times, phone numbers, and commonly used words to the text recognizers. The user can expand the system’s built-in vocabulary by adding new words to a RAM-based user dictionary accessed from the Personal Word List slip. In addition, you can provide custom dictionaries for the recognition system’s use. For example, you might create a custom dictionary to supply specialized vocabulary, such as medical or legal terminology. The section “System Dictionaries” beginning on page 9-11 describes the system-supplied dictionaries in more detail. The use of custom dictionaries for recognition is described in “Using Custom Dictionaries” beginning on page 10-24.

Although the interpretations returned by the printed recognizer are never limited to dictionary words, its output is influenced strongly by the set of dictionaries available for its use. The interpretations returned by the cursive recognizer can be restricted to those words appearing in the set of dictionaries available for its use; however its default behavior is to return non-dictionary words in addition to words appearing in available dictionaries.

Options specified for the currently enabled recognizer may also influence the interpretations it returns to the view. For example, the cursive recognizer’s default settings enable its letter-by-letter recognition option, to increase the likelihood of its returning strings not in the currently available set of dictionaries. The user can control this option and others from within the Handwriting Settings preferences slip.

Note that even when the cursive and printed recognizers are configured similarly, the results they return for the same input may differ. For example, using the cursive

CHAPTER 9

Recognition

recognizer's letter-by-letter option may produce different results than using the printed recognizer (which always provides letter-by-letter recognition.) Options for both recognizers are described throughout this chapter and in Chapter 10, "Recognition: Advanced Topics."

Unrecognized Strokes

If the input strokes are not recognized, the system encapsulates them in an object known as a stroke bundle. A **stroke bundle** is a `NewtonScript` object that encapsulates stroke data for multiple strokes. The strokes in the bundle have been grouped by the system according to temporal and spatial data gathered when the user first entered them on the screen. You can access the information in stroke bundles to provide your own form of deferred recognition, or to examine or modify stroke data before it is recognized. For information on using stroke bundles, see Chapter 10, "Recognition: Advanced Topics."

Stroke bundles may be returned to the view under any of the following circumstances:

- No recognizers are enabled for the view.
- Recognizers are enabled for the view but recognition fails.
- The view is configured to image input as ink text.
- The view's `vStrokesAllowed` flag is set and a `ViewStrokeScript` method is provided.

When the system passes a stroke bundle to a `clEditView` view, the view images the strokes in the bundle as ink text or sketch ink. Other kinds of views may require you to provide code that displays the strokes.

When no recognizers are enabled for a `clEditView` view, it displays input as sketch ink. Input views for which no recognizers are enabled are not as unusual as they might seem at first; for example, you might provide a view that accepts stroke input without performing recognition as a means of capturing the user's handwritten signature. And some views, such as those used in the built-in Notepad application, allow the user to enable and disable recognizers at will.

When recognizers are enabled for the view but recognition fails, the view may return ink text or sketch ink. Recognition may fail if input strokes are too sloppy to classify or if the view is not configured correctly for the intended input. For more information, see "Recognition Failure" beginning on page 9-11.

When the view is configured to display input as ink text, the system skips the remainder of the recognition process—it does not attempt to further classify the input strokes as letters or words. Instead, the view simply images the strokes as ink text.

The most important difference between ink text and sketch ink has to do with how these two forms of ink are represented. Ink text is inserted into existing text in paragraph views in the same way as recognized words are: as the contents of a

CHAPTER 9

Recognition

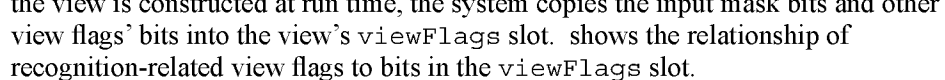
`clParagraphView` view child. Ink text automatically wraps to the paragraph boundaries, just as recognized text does. Ink text is also usually reduced in size when it is drawn, according to the user preference specified by the Ink Text Scaling item in the Text Editing preferences slip. Sketch ink, on the other hand, is treated as a graphic: it is inserted into the view as a `clPolygonView` view child. Sketch ink is always drawn at full size, and in the position at which it was written on the screen.

Thus, stroke bundles are normally returned only to views that do not perform recognition. To cause the system to always return stroke bundles to the view (in addition to any word units, gesture units or shape units that may be passed to the view), set the view's `vStrokesAllowed` flag and provide a `ViewStrokeScript` method, as described in “Customized Processing of Input Strokes” beginning on page 10-40.

The recognition system's classification of user input is essentially a process of elimination. Enabling and configuring only the recognizers and dictionaries appropriate to a particular context is the primary means by which you optimize the recognition system's performance within your application.

Enabling Recognizers

Each view has a `viewFlags` slot that contains a bit field. The bits in this field specify characteristics that the view does not inherit from its view class, such as its recognition behavior. When you set a view flag, it sets bits in this field to enable combinations of recognizers and dictionaries suited to the input you anticipate the view to receive.

Not all of the bits in this field affect recognition; some are used to set other characteristics, such as the view's placement on the screen. The bits in this field that affect the recognition system are referred to as the view's **input mask**. When the view is constructed at run time, the system copies the input mask bits and other view flags' bits into the view's `viewFlags` slot.  shows the relationship of recognition-related view flags to bits in the `viewFlags` slot.

You can set bits in the `viewFlags` slot from within the graphical view editor in Newton Toolkit or you can set them programmatically from within your own NewtonScript code. Either approach allows you to set combinations of bits to produce a variety of behaviors.

This book uses the NewtonScript approach for all examples. For information on using the graphical view editor in Newton Toolkit, see *Newton Toolkit User's Guide*.

CHAPTER 9

Recognition

multiple bits in the input mask to produce a particular behavior. You can use a `recConfig` frame to set individual bits in the input mask, allowing you to control aspects of recognition behavior that view flags do not.

Some features of the recognition system require the use of a `recConfig` frame. For example, to create a view that provides single-letter input areas suitable for accepting pen input in a crossword puzzle application, you must supply a `recConfig` frame that provides an `rcGridInfo` frame. The system-supplied `rcGridInfo` frame is used to specify the location of one or more single-letter input views.

This chapter focuses on the use of view flags to configure recognition. The use of `recConfig` frames is described in Chapter 10, “Recognition: Advanced Topics.” System-supplied `recConfig` frames are described in “System-Supplied `recConfig` Frames” (page 8-18) in *Newton Programmer’s Reference*.

View Flags vs. RecConfig Frames

In most cases, view flags provide the easiest and most efficient way to configure the recognition system. Although `recConfig` frames provide more flexible and precise control over the configuration of recognition behavior, they require more effort to use correctly.

It is recommended that you use view flags to configure recognition unless you need some special recognition behavior that they cannot provide. Examples of such behavior include constraining recognition on a character-by-character basis, implementing customized forms of deferred recognition, and defining baseline or grid information.

The rest of this chapter discusses configuration of the recognition system only in terms of the view flag model. You need to read this material even if you plan to use `recConfig` frames in your application, because the description of `recConfig` frames in Chapter 10, “Recognition: Advanced Topics,” assumes an understanding of the view flag model upon which these frames are based.

Where to Go From Here

If you’re anxious to begin experimenting with view flags, you can skip ahead to “Using the Recognition System” beginning on page 9-21 and test the effects of various flags using the `viewFlags` sample application provided with Newton Toolkit. However, it is recommended that you read the rest of this section before attempting to work with the recognition system.

CHAPTER 9

Recognition

Recognition Failure

Recognition may fail when the handwritten input is too sloppy for the system to make a good match against its internal handwriting model, when the view is not configured correctly for the intended input, or (in the case of dictionary-based recognition only) when none of the interpretations of the input strokes match a dictionary entry. In such cases, the recognition system may return sketch ink or ink text.

Ink text looks similar to sketch ink; however, ink text is scaled and placed in a `clParagraphView` view as text. Sketch ink is not placed in a paragraph but drawn in a `clPolygonView` view on top of anything else that appears in the polygon view's `clEditView` parent. Both ink text and sketch ink hold stroke data that can be used to recognize the strokes at another time. Deferred recognition—the process of recognizing saved ink at a later time—is described in more detail in “Deferred Recognition” (page 10-5), in Chapter 10, “Recognition: Advanced Topics.”

System Dictionaries

The system supplies a variety of dictionaries against which names, places, dates, times, phone numbers, and commonly used words are matched. There are two kinds of dictionaries used for text recognition: enumerated and lexical. An **enumerated dictionary** is simply a list of strings that can be matched. A **lexical dictionary** specifies a grammar or syntax that is used to classify user input. The kind of dictionary used for a particular task is dependent upon task-specific requirements. For example, it would be impractical to create an enumerated dictionary of phone numbers; however, the clearly defined format imposed on these numbers makes them ideal candidates for definition in a lexical dictionary.

The specific set of dictionaries that the system provides for a particular purpose generally varies according to the user's locale. For example, because currency formats vary from country to country, the particular lexical dictionary that the system uses for matching monetary values may change according to the current locale. However, you usually need not be concerned with the specific set of dictionaries used by a particular locale. For more information, see Chapter 20, “Localizing Newton Applications.”

Dictionaries can be in ROM or in RAM (internal or card-based). Most of the system-supplied dictionaries are in ROM; however, the user dictionary resides in RAM.

Applications must never add items to the user dictionary without the user's consent. The user dictionary is intended to be solely in the user's control—adding items to it is akin to changing the user's handwriting preferences or Names entries. It's also important to leave room for users to store their own items.

CHAPTER 9

Recognition

IMPORTANT

An excessively large user dictionary can slow the system when performing searches that are not related to your application. It is therefore recommended that applications do not add items to the user dictionary at all. ▲

The system supports a total of about 1,000 items in the RAM-based user dictionary (also known as the review dictionary). Note that this number may change in future Newton devices. A persistent copy of the user word list is kept on the internal store in the system soup. The user dictionary is loaded into system memory (not the NewtonScript heap or store memory) when the system restarts and saved when the user closes the Personal Word List slip. For more information, see “Working With the Review Dictionary” (page 10-30).

A separate dictionary called the **expand dictionary** allows you or the user to define word expansions that are substituted for abbreviations automatically. The substitution takes place after the abbreviation has been recognized, but before it has been displayed. For example, you could specify that the string *w/* be expanded to the string *with*, or the string *appt* expand to *appointment*. In addition to permitting the substitution of an entirely different string for the one recognized, the expand dictionary can be used to correct recurring recognition mistakes or misspellings automatically.

The expand dictionary is not used directly by the recognition system. Instead, each word to be expanded is added to both the user dictionary and the expand dictionary. Then the user dictionary and any appropriate additional dictionaries are used to perform stroke recognition. Before the recognizer returns the list of recognized words to the view, it determines whether any of the items in the list are present in the expand dictionary. If so, the expanded version of the word is inserted into the list of recognized words before the original version of the word. The original version is also included in the list, just in case the user doesn't want to expand the word.

As words not present in any of the currently enabled dictionaries are recognized, the auto-add mechanism may add them to the user dictionary automatically. This feature is enabled when the cursive recognizer is active, but not when the printed recognizer is active. (Although both recognizers use dictionaries to improve accuracy, the use of dictionaries does not benefit the printed recognizer enough to justify default use of the auto-add mechanism.) You can improve the printed recognizer's treatment of problematic words by making them available from a dictionary, but it is recommended that you create a custom dictionary that provides those words; the user dictionary is intended to be under the user's control.

The **auto-add dictionary** is a list of words that have been added to the user dictionary automatically. If the auto-add dictionary is not empty, the Recently Written Words slip displays its contents when the user opens the Personal Word List slip. The Recently Written Words slip prompts the user to indicate whether each of the words it displays should remain in the user dictionary. To encourage the

CHAPTER 9

Recognition

user to make individual decisions about each word in the list, this slip does not permit selection.

Although the Recently Written Words slip asks the user whether to add words to the Personal Word List, the words have already been added to both the user dictionary and the auto-add dictionary by the time they are displayed in this slip if the cursive recognizer is in use. Rather than actually adding words to any dictionaries, this slip actually removes those words that the user does not confirm as candidates for addition to the user and auto-add dictionaries.

Note

When the printed text recognizer is in use, the automatic addition of words to the user dictionary and the auto-add dictionary is disabled. ♦

The size of the auto-add dictionary is limited to 100 words. A persistent copy of the auto-add dictionary is kept on the internal store in the system soup. The auto-add dictionary is loaded in system memory (not the NewtonScript heap or store memory) when the system restarts and saved when the user opens or edits the Recently Written Words slip. For more information, see “Working With the Review Dictionary” beginning on page 10-30.

Another dictionary, the symbols dictionary, is always enabled for any view that performs text recognition. This dictionary includes alphabetic characters, numerals, and some punctuation marks. Use of this dictionary permits the user to correct single characters by writing over them on the screen.

Correction and Learning

When the recognition system returns a word unit to the view, it constructs a correction information frame from the word unit and may save learning data as well. The correction information frame holds information used to correct misrecognized words. Learning data is used by the system to improve the cursive recognizer’s accuracy.

The system provides a developer interface to the information in the correction information frame, as well as a user interface to a subset of this data. For complete descriptions of the `protoCorrectInfo`, `protoWordInfo` and `protoWordInterp` system prototypes that provide access to correction information, see “Recognition System Prototypes” (page 8-31) in *Newton Programmer’s Reference*

The picker (popup menu) shown in Figure 9-3 provides the user interface to correction information. This picker is displayed automatically when the user double-taps a previously recognized word. This picker’s items include

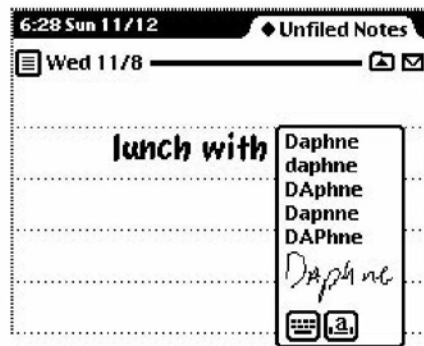
- the five best interpretations returned by the recognizer.
- the alternative capitalization of the most highly scored interpretation.

CHAPTER 9

Recognition

- the expansions of words that match entries in the expansion dictionary.
- a graphical representation of the original input strokes as ink.
- buttons for the soft keyboard and text-corrector views.
- a Try Letters button when the cursive recognizer is active.

Figure 9-3 Text-corrector picker



The words in this list are one example of correction information stored by the system as it recognizes words. In addition to word lists, correction information includes the original stroke data and information known as *learning data*.

Learning data is information gathered as the user corrects misrecognized words. It is used to modify the system's internal handwriting model to more closely match the way the user actually writes. This information is called *learning data* because the system can be said to learn various characteristics of the user's handwriting style, with a resulting increase in recognition accuracy. Not all recognizers return learning data.

User Preferences for Recognition

The user can specify several preferences that affect the overall configuration of the recognition system. This information is provided for reference purposes only; generally, you should not change the user's recognition preferences settings.

CHAPTER 9

Recognition

This section describes only those user preferences for which the system provides a NewtonScript interface. It does not provide a comprehensive summary of the user interface to recognition, which may vary on different Newton devices. For a description of the user interface to a particular Newton device, see the user manual for that device.

The user preference settings for recognition that this section describes are stored as the values of slots in a system-maintained frame that holds user configuration data. These slots are described in “System-Wide Settings” (page 8-2) in *Newton Programmer’s Reference*.

The user preference settings described here may be affected by the setting of a `protoRecToggle` view associated with the view performing recognition. For a description of this view, see “RecToggle Views” beginning on page 9-18.

Recognition-oriented user preference settings may also be overridden by a `recConfig` frame associated with the view performing recognition. For complete information on `recConfig` frames, see Chapter 10, “Recognition: Advanced Topics.”

Handwriting Recognition Preferences

The Handwriting Recognition preferences slip shown in Figure 9-4 specifies the overall characteristics of the user’s handwriting. In general, you should not override the user settings specified in this slip.

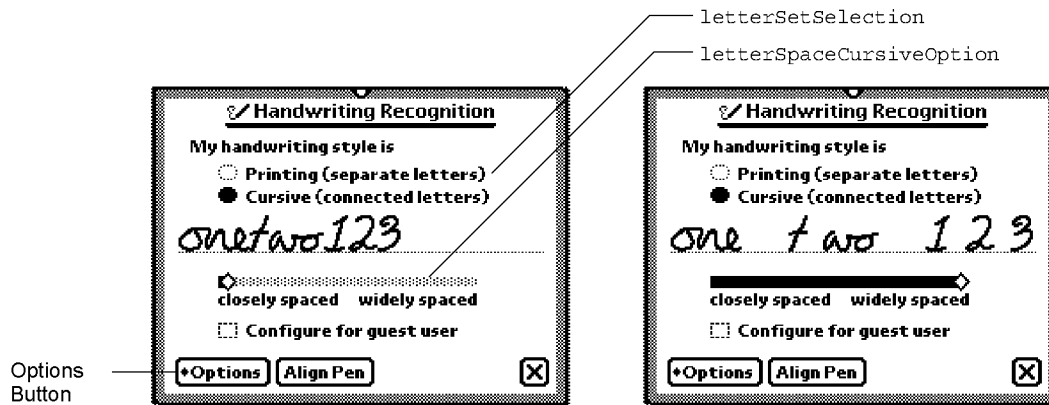
The Printing and Cursive radio buttons specify whether a printed or cursive style of lettering is used. This system-wide setting enables either the printed or cursive recognizer by setting the value of the `letterSetSelection` slot in the system’s user configuration data. It is strongly recommended that you do not change this setting.

The user can also specify the amount of blank space the recognizer may find between words; this setting influences the recognition system’s initial grouping of stroke data. The value returned by the slider control in this slip is kept in the `letterSpaceCursiveOption` slot in the system’s user configuration data. This value may be overridden by views that perform recognition.

CHAPTER 9

Recognition

Figure 9-4 Handwriting Recognition preferences



Checking the “Configure for guest user” checkbox causes the system to

- save all current recognition system settings.
- save the owner’s learning data.
- temporarily reset all recognition system preferences to their default values.
- learn the guest user’s writing style as misrecognized words are corrected if the cursive recognizer is in use. (The printed recognizer does not use learning data.)

When the user deselects the “Configure for guest user” checkbox, the guest user’s learning data is discarded and the original user’s learning data, preferences, and other settings are restored. Note that the system’s use of the auto-add mechanism is not affected by the setting of this checkbox—when the cursive recognizer is enabled, the system always adds new words to the auto-add dictionary.

The Options button displays a picker from which the user can access options for various preferences. The items included in this picker vary according to whether the printed or cursive recognizer is enabled. When the cursive recognizer is enabled, this picker provides the Text Editing Settings, Handwriting Settings, Letter Shapes, and Fine Tuning items. When the printed recognizer is enabled, this picker provides only the Text Editing Settings and Fine Tuning items. Because the system provides no developer interface to the Letter Shapes slip, it is not discussed here.

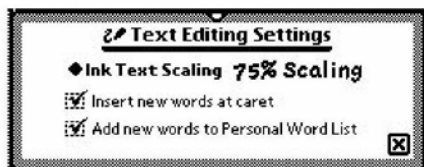
Figure 9-5 shows the Text Editing Settings slip that is displayed for both the printed and cursive recognizers. Of the adjustments available from the Text Editing Settings slip, the “Add new words to Personal Word List” checkbox is of interest to developers. The cursive recognizer adds new words to the RAM-based user dictionary automatically when this checkbox is selected. The printed recognizer never adds new words automatically, regardless of the setting of this checkbox. You

CHAPTER 9

Recognition

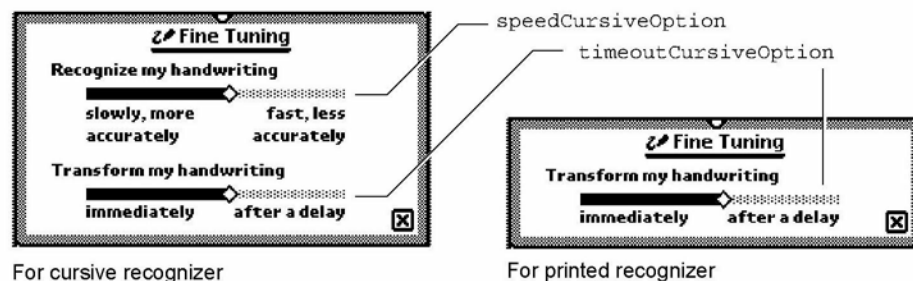
can always add new words to the user dictionary programmatically, regardless of which recognizer is enabled. To display or edit the personal word list, the user taps the book icon on the soft keyboard.

Figure 9-5 Text Editing Settings slip



The system provides two versions of the Fine Tuning slip, one for each of the cursive and printed text recognizers, as shown in Figure 9-6. Both slips provide a “Transform my handwriting” slider control that allows the user to fine-tune the system’s use of temporal cues to determine when a group of strokes is complete. This slider sets the value of the `timeoutCursiveOption` slot in the system’s user configuration data.

Figure 9-6 Fine Tuning handwriting preferences slips

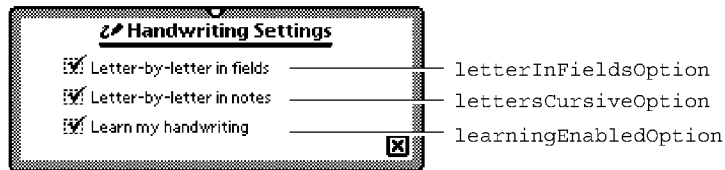


The Fine Tuning slip used by the cursive recognizer includes an additional slider that allows the user to trade some measure of accuracy for a faster response from the recognizer. The “Recognize my handwriting” slider sets the value of the `speedCursiveOption` slot in the system’s user configuration data.

When the cursive recognizer is enabled, the Options button in the Handwriting Recognition preferences slip provides access to the Handwriting Settings slip shown in Figure 9-7.

CHAPTER 9

Recognition

Figure 9-7 Handwriting Settings slip

When the “Learn my handwriting” checkbox is selected, the system sets the value of the `learningEnabledOption` slot in its user configuration data to `true`. When this slot holds the value `true`, the system modifies its internal handwriting model as the user corrects misrecognized words when the cursive recognizer is enabled. The printed recognizer does not provide or use learning data.

The user can cause the cursive recognizer to perform character-based recognition (rather than dictionary-based recognition) in certain kinds of views by selecting the “Letter-by-letter in fields” or “Letter-by-letter in notes” checkboxes in the Handwriting Settings slip. (The printed recognizer can always return character combinations that do not appear in dictionaries.)

The “Letter-by-letter in fields” checkbox enables the cursive recognizer’s letter-by-letter option in `protoLabelInputLine` views that use this recognizer. The intended use of this flag is to permit the user to enable letter-by-letter recognition automatically for views that are unlikely to find user input in dictionaries. For example, an application that restricts the cursive recognizer to returning dictionary words might enable this recognizer’s letter-by-letter option selectively for views intended to accept surnames. When the “Letter-by-letter in fields” box is selected, the value of the `letterInFieldsOption` slot in the system’s user configuration data is set to `true`. For more information, see the description of this slot in “System-Wide Settings” (page 8-2) in *Newton Programmer’s Reference*.

The “Letter-by-letter in notes” checkbox enables letter-by-letter recognition for views based on the `clEditView` class that use the cursive recognizer. When the “Letter-by-letter in notes” box is selected, the `lettersCursiveOption` slot in the system’s user configuration data is set to `true`. The built-in Notes application and notes associated with items in the Names and Dates applications demonstrate this behavior. For more information, see the `lettersCursiveOption` description in “System-Wide Settings” (page 8-2) in *Newton Programmer’s Reference*.

RecToggle Views

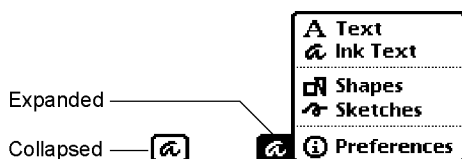
The `protoRecToggle` view is a button that allows the user to control the recognition behavior of one or more views easily. This button is usually provided as a child of your application’s status bar. When the user taps this button, it

CHAPTER 9

Recognition

displays a picker from which the user can choose recognition behaviors that you specify. When this picker is collapsed, the appearance of the button indicates the current recognition settings for the view or views that it controls. Figure 9-8 shows the appearance of typical `protoRecToggle` view when it is collapsed and when it is expanded to display the pick list of recognizers it can enable.

Figure 9-8 Use of `protoRecToggle` view in the Notes application



The default picker provides all of the items shown in Figure 9-8 in the order illustrated. You can specify that this picker display a subset of these items in the order you specify.

The topmost item in the picker indicates the recognizer that the `recToggle` view enables by default; unless you specify otherwise, the `recToggle` view enables the text recognizer by default, as shown in the figure.

You can also provide code that restores the user's most recent `recToggle` setting or initializes the `recToggle` to a predetermined setting each time your application opens.

The picker's Preferences item opens the Handwriting Recognition user preferences slip by default.

For more information on `protoRecToggle` views, see Chapter 10, "Recognition: Advanced Topics," as well as the description of this prototype in *Newton Programmer's Reference*.

Flag-Naming Conventions

This section describes conventions used to name recognition-related view flags, as well as the significance of the use of the words `Field` and `Allowed` in flag names.

The Entry Flags area of the Newton Toolkit (NTK) view editor actually sets view flags. The distinction that Newton Toolkit makes between "view flags" and "entry flags" is an artifact of the way certain views create child views dynamically at run time.

For example, when the user taps a `protoLabelInputLine` view, it creates and opens a `clParagraphView` child that is the input line view in which text

CHAPTER 9

Recognition

recognition takes place. The Entry Flags area of the NTK screen specifies the view flags for this dynamically created child view separately from the view flags for the container view in which it appears. When the system creates the child view, it copies the Entry Flags bits into the child view's `viewFlags` slot.

For simplicity's sake, this chapter refers to all recognition-oriented flags as "view flags." This chapter and its corresponding section of the *Newton Programmer's Reference* document all such flags as view flags.

Although the NTK view editor describes `vAnythingAllowed` as a "flag" it is actually a mask that sets all bits in a `clEditView` view's input mask. This chapter refers to this construct as the "`vAnythingAllowed` mask." See (page 9-8) for a graphical depiction of the relationships between bits in the input mask and recognition-related view flags.

The use of `Field` in the names of some flags and `Allowed` in others is meant to reflect these flags' intended use, rather than a functional difference.

The "field" flags are intended for setting up input views that accept a single kind of input, such as dates. For example, setting the `vDateField` flag specifies that the view accepts numeric input in a format commonly used for dates in the current locale. Setting this flag enables the set of dictionaries appropriate for recognizing such input.

On the other hand, the more inclusive "allowed" flags are intended for use with views that must recognize several kinds of input; for example, setting the `vNumbersAllowed` flag specifies that the view accepts a wide range of numeric input, such as currency values, times, and dates. Setting the `vNumbersAllowed` flag alone, then, enables a more inclusive set of dictionaries than obtained by setting the `vDateField` flag alone.

Despite differences in naming conventions (and despite the fact that the Field Type popup menu in the NTK view editor considers these flags mutually exclusive), the "field" and "allowed" flags can be mixed in any combination. Keep in mind, though, that the more choices the recognizer has, the more opportunity it has to make the wrong choice.

Recognition Compatibility

In addition to the cursive recognizer available in previous systems, version 2.0 of system software adds a recognizer optimized for printed characters. This recognizer, represented by the Printed radio button in the Handwriting Recognition preferences slip, is the default text recognizer used when you or the user do not specify otherwise.

Selecting the Cursive radio button in the Handwriting Recognition preferences slip equates to selecting the Mixed Cursive and Printed radio button available in previous versions of this slip: the cursive recognizer is enabled, all printed and

CHAPTER 9

Recognition

cursive letter styles in the system's handwriting model are enabled, and the system disables unused letter styles over time as the user corrects misrecognized words.

The default settings of the cursive recognizer in version 2.0 enable this recognizer's letter-by-letter recognition option. Previous versions of the system disabled this option by default, causing the cursive recognizer to return only words appearing in the set of dictionaries available to the view performing recognition.

The `protoLetterByLetter` prototype, which appears at the lower-left corner of the screen in the Notepad application on the MessagePad 100 and MessagePad 110, is obsolete. It has been replaced by the `protoRecToggle` prototype. For more information, see "RecToggle Views" (page 9-18).

Prior to version 2.0 of Newton system software, correction information was not accessible from NewtonScript. Version 2.0 of Newton system software makes this information available as frame data. For more information, see "Correction and Learning" (page 9-13).

Combining the `vLettersAllowed` flag with flags used to specify recognition of numeric values (such as `vPhoneField`, `vNumbersAllowed`, `vDateField`, `vTimeField`, and `vAddressField`) produced undesirable results in system software prior to version 2.0. System software version 2.0 supports these kinds of view flag combinations.

Deferred recognition—the ability to convert strokes to text at some time other than when the strokes are first entered on the screen—was introduced in Newton system software version 1.3 with no application programming interface. Version 2.0 of Newton system software provides a NewtonScript interface to this feature.

Using the Recognition System

This section describes how to use view flags to enable recognition in views. This chapter discusses only those view flags that interact with the recognition system. For a summary of these view flags, see "Constants" (page 9-31). For information on other kinds of view flags, see Chapter 3, "Views." For complete descriptions of all view flags, see *Newton Programmer's Reference*.

For information on the use of `recToggle` views, `recConfig` frames and advanced features of the recognition system, see Chapter 10, "Recognition: Advanced Topics."

Types of Views

The kind of view that you use to recognize input affects the amount of work you'll have to do to support recognition. Views based on the `clEditView` class handle most recognition events automatically once you've specified their intended

CHAPTER 9

Recognition

recognition behavior by setting view flags or providing a `recConfig` frame. Specifically, `clEditView` views create `clParagraphView` or `clPolygonView` child views automatically as required to display output from the recognition system. To use other kinds of views for recognition, you may need to provide `viewXxxScript` methods that create these child views and respond in other ways to recognition system events.

Configuring the Recognition System

You can take the following approaches to configuring the recognition system:

- Set view flags only. This approach works well for most applications and is described in this chapter.
- Set view flags and allow the user to configure recognition from a `protoRecToggle` view that you provide. The easiest way to do this is by setting the `vAnythingAllowed` mask, which is described in this chapter. This approach supports the use of ink text in `clEditView` views. Use of the `protoRecToggle` view is described in Chapter 10, “Recognition: Advanced Topics.”
- Set view flags and supply a recognition configuration frame based on `ROM_rcInkOrText`. This approach supports ink text in `clEditView` views. You should provide a `protoRecToggle` view as well, to allow the user to switch easily between text and ink text.
- Supply a recognition configuration frame of some other kind. This approach offers you the most control and flexibility, but also requires the most work to implement. The difficulty of enabling ink text according to the value of a `protoRecToggle` view depends on the particular implementation of your `recConfig` frame. Recognition configuration frames are described in Chapter 10, “Recognition: Advanced Topics.”
- Use the `RecogSettingsChanged` message sent by the `protoRecToggle` view to enable recognition behaviors dynamically. This technique is described in Chapter 10, “Recognition: Advanced Topics.”

Except where noted otherwise, all of the flags described in this chapter are set in the view’s `viewFlags` slot. When setting the values of `viewFlags` slots, remember that in order to produce useful behavior you may need to set other bits in addition to the recognition-oriented ones that this chapter describes. To preserve settings that your view’s `viewFlags` slot inherits from its view class, you should logically OR changes to bits in this slot.

For information on non-recognition view flags provided by the system, see Chapter 3, “Views.”

CHAPTER 9

Recognition

Obtaining Optimum Recognition Performance

To obtain the most accurate results from the recognition system, you must define as precisely as possible the type of input that the view is to recognize. Aside from potentially introducing errors, enabling superfluous recognizers may slow the recognition system's performance.

The view flags that enable text recognition also enable dictionaries suited to recognizing particular kinds of input, such as dates, phone numbers, and so on. Some view flags activate multiple dictionaries, and the sets of dictionaries activated by various flags may overlap. The system shows no preference towards any single dictionary in a set except for a slight weighting of results in favor of words found in the user dictionary, which most view flags enable.

The specific dictionaries that a particular flag enables varies according to the user's locale and the ROM version of the Newton device. You usually need not be concerned with this implementation detail, nor should you rely on the presence of a particular dictionary when setting view flags.

When you need to control precisely which dictionaries a view uses for recognition, you can set its `vCustomDictionaries` flag and use a `dictionaries` slot to specify explicitly which dictionaries are to be used. For information about custom dictionaries, see "Using Your RAM-Based Custom Dictionary" (page 10-28), in Chapter 10, "Recognition: Advanced Topics." For information about locale and the recognition system, see "How Locale Affects Recognition" (page 20-2), in Chapter 20, "Localizing Newton Applications."

For best performance, you need to specify the minimum combination of recognizers and dictionaries required to process the kind of input you expect the view to receive. This equates to enabling the minimum set of view flags that allow the view to recognize appropriate input correctly. By restricting the possible interpretations returned by the recognition system to only those that are appropriate for a particular view, you increase the system's chances of interpreting the input correctly. For example, when configuring a view for the entry of numeric data, you would not specify that the recognition system return alphabetic characters to that view.

The printed and cursive text recognizers appear nearly identical to NewtonScript applications. The main difference between them is that while the cursive recognizer can be made to use the value of the `viewFlags` slot as a strict definition of what it can recognize, the printed recognizer uses this value as only a hint—that is, it can always return values not specified by the input view's view flags. When configuring views for text recognition, you should set view flags that describe the input you anticipate the view to receive and then verify that you obtain acceptable results from both text recognizers.

Because the printed recognizer lets you write anything in the input view, it may be difficult to determine whether your `viewFlags` settings are appropriate when this recognizer is enabled; the cursive recognizer usually provides better feedback in

CHAPTER 9

Recognition

this regard. If necessary, you can provide a `ViewWordScript` or `ViewChangedScript` method that validates the recognizer's output; this method can be especially useful when working with the printed recognizer.

Accepting Pen Input

When setting up any view, you must specify whether it accepts pen input at all. If you set the `vNothingAllowed` flag (or turn off all recognition-oriented flags), the view does not accept pen input. If you want the view to accept pen input, you must set the `vClickable` flag in its `viewFlags` slot. Setting this flag only causes the view to accept pen taps and send `ViewClickScript` messages; it does not enable ink handling or send messages to any of the unit-handling methods that provide recognition behavior.

Setting the `vClickable` flag specifies that the view system is to send the `ViewClickScript` message to the view once for each pen tap that occurs within the view. Note that this is the case only when `vClickable` is the only flag set for the view—other flags, such as the `vCustomDictionaries` flag, set the `vClickable` bit in the view's input mask also.

When this flag is set, the system sends additional messages to the view to signal taps, strokes, gestures, and words. All pen input is signaled by the `ViewClickScript` message, which indicates that the pen contacted the screen or was lifted from it within the boundaries of the view. If you supply a `ViewClickScript` method, it should return `true` to indicate that the message was handled, or `nil` to pass the message on to another view. If this message is not handled by the view and additional recognition flags are set, other messages may be sent to the view, depending on what was written. These other messages include `ViewStrokeScript`, `ViewGestureScript`, and `ViewWordScript`—in that order, if all are sent.

Each of the corresponding input-related view methods accept as an argument a unit object passed to it by the system. The unit contains information about the pen input. You cannot examine the unit directly from `NewtonScript`, but you can pass it to other system-supplied functions that extract information from it such as the beginning and ending points of the stroke, an array of stroke points, the stroke bounds, and so on.

Taps and Overlapping Views

When views overlap, taps can “fall through” from the top view to the one beneath, causing unexpected results. For example, when the user taps in an area of the top view that doesn't handle taps, and the view beneath provides a button in the vicinity of the tap, the button may be activated unintentionally.

CHAPTER 9

Recognition

You can solve this problem by setting the top view's `vClickable` flag without providing a `ViewClickScript` method. (The top view need not handle the taps, only prevent them from being passed on to the other view.)

Recognizing Shapes

The `vShapesAllowed` flag enables the recognition of geometric shapes such as circles, straight lines, polygons, and so on. Do not set this flag for views that handle text input only. This flag is intended for use only in views based on the `clEditView` class. The `clEditView` class provides the built-in Notepad application's note stationery with much of its recognition behavior.

The shapes displayed on the screen are `clPolygon` views returned as the children of the `clEditView` that accepted the input strokes. There is no developer interface to shape recognition; to manipulate shapes returned by the recognition system, you must extract the polygon views from the edit view yourself. In some cases, you may find the `ViewAddChildScript` method useful for this purpose. The `ViewAddChildScript` message is sent when a child view is added to a view.

When multiple shapes are returned to an edit view, its `ViewAddChildScript` method is called once for each shape.

When multiple ink text words are returned to an edit view, the `ViewAddChildScript` method is invoked when the `clParagraphView` that holds the ink text is added as the child of the edit view, but this method is not invoked as ink text words are added to the paragraph view.

In views not based on the `clEditView` class, the arrival of each ink word is signalled by a `ViewInkWordScript` message.

Recognizing Standard Gestures

Setting the `vGesturesAllowed` flag supplies system-defined behavior for the gestures tap, double tap, highlight, scrub, line, caret, and caret-drag. Most input views set the `vGesturesAllowed` flag, as they need to respond to standard gestures such as scrubbing to delete text or ink. At present, you cannot define new gestures to the system.

When the `vGesturesAllowed` flag is set, the gesture recognizer invokes the view's `ViewGestureScript` method before handling the gesture. Normally, you don't need to supply a `ViewGestureScript` method for `clEditView` or `clParagraphView` views. These views handle all system-defined gestures automatically.

Your `ViewGestureScript` method is invoked only for gestures that the view system does not handle automatically. For information on intercepting standard gestures before the view system handles them, see "Customized Processing of

CHAPTER 9

Recognition

Double Taps” beginning on page 10-41. See also “ViewGestureScript” (page 8-71) in *Newton Programmer’s Reference*.

Combining View Flags

Generally, you must combine multiple view flags to produce useful recognition behavior. For example, most views that accept user input set the `vClickable` flag to enable pen input and the `vGesturesAllowed` flag to enable recognition of standard gestures such as scrubbing and inserting spaces.

Except where noted otherwise, the NewtonScript “plus” operator (+) is used to combine view flags, as in the following code fragment.

```
myViewTemplate :=
{
  // recognize taps, gestures, and shapes
  viewFlags: vClickable+vGesturesAllowed+vShapesAllowed,
  ...}
```

Note

Most combinations of view flags include the `vClickable` flag. If you do not set the `vClickable` flag, the view does not accept pen input at all. ♦

Sometimes a particular combination of view flags produces results that seem incorrect. For example, you might be surprised to discover that a view setting only the flags `vClickable+vLettersAllowed` can occasionally recognize numeric values. (The `vLettersAllowed` flag enables the recognition of single text characters by the cursive recognizer.) This behavior is caused by the presence of the symbols dictionary in the set of dictionaries available to the view. The symbols dictionary includes alphabetic characters, numerals and some punctuation marks. Most view flags enable this dictionary to support the correction of single letters or numerals by overwriting. As a side effect, it becomes possible to recognize extraneous characters or numerals in fields that ostensibly should not support such input. This behavior is rarely a problem, however, because the recognition system is designed to show a strong preference for “appropriate” interpretations of input as defined by the view flags set for the view.

Although you might expect that the presence of the symbols dictionary would allow a view setting only the flags `vClickable+vNumbersAllowed` to return alphabetic characters, this behavior is quite difficult to produce. Views that set the `vNumbersAllowed` flag show a much stronger preference for single-digit numbers than single alphabetic characters. However, letters that do not look similar to numeric values—for example, the letter *W*—may produce this particular form of misrecognition.

CHAPTER 9

Recognition

When troubleshooting recognition errors, remember that view flags may enable multiple dictionaries and that the sets of dictionaries enabled by various flags may overlap.

As a general rule, the fastest and most accurate recognition occurs when the fewest recognizers and dictionaries necessary to successfully analyze the input are enabled. Enabling unnecessary recognizers and dictionaries may decrease the speed and accuracy with which recognition is performed.

Recognizing Text

The `vCharsAllowed` and `vLettersAllowed` flags enable text recognition in views that accept pen input. Either flag enables the text recognizer specified by the Handwriting Recognition preferences slip.

Each of these flags specifies different recognition options and dictionary sets. The unique behaviors associated with each flag are demonstrated most clearly by the cursive recognizer. The cursive recognizer can be made to return only words present in the set of dictionaries available to the view performing recognition. In contrast, the printed recognizer can always return words or letter combinations that are not present in dictionaries.

The `vCharsAllowed` flag enables a default set of dictionaries that provide vocabulary used in common speech, names of days, names of months, proper names, and words in the user dictionary. When the `vCharsAllowed` flag is set and the `vLettersAllowed` flag is not, the cursive recognizer returns only words that appear in the set of dictionaries available to the view performing recognition.

Note that the complete set of dictionaries available to the view may include those enabled by other flags. For example, the NTK view editor provides a Field Type popup menu that allows you to specify whether the view is to accept phone, date, time, address or name data. The choices in this menu set the `vPhoneField`, `vDateField`, `vTimeField`, `vAddressField` and `vNameField` flags, respectively. Each of these flags enables one or more dictionaries suited to recognizing the specified input data. Custom dictionaries may also be made available to the view performing recognition by setting the `vCustomDictionaries` flag and providing a valid `dictionaries` slot in the view that performs recognition.

The `vLettersAllowed` flag enables the cursive recognizer's letter-by-letter recognition option. When the `vLettersAllowed` flag is set, the cursive recognizer may return words not appearing in dictionaries as well as nonword letter combinations. Note that this configuration increases the cursive recognizer's chances of misrecognizing words that appear in the set of dictionaries available to it.

Although both text recognizers provide a letter-based recognition feature, the two recognition engines are completely distinct. Consequently, the results produced by

CHAPTER 9

Recognition

the cursive recognizer's letter-by-letter option may be different from those returned by the printed recognizer for the same input data.

Although the printed recognizer can always return non-dictionary words, it does make extensive use of the dictionaries available to the view for recognition. Users may improve the printed recognizer's accuracy for problematic non-dictionary words by adding them to the user dictionary. You can supply custom dictionaries to improve the recognition of specialized vocabulary. It is recommended that applications do not add words to the user dictionary.

Recognizing Punctuation

The `vPunctuationAllowed` flag permits the cursive recognizer to return common punctuation marks such as the period (.); comma (,); question mark (?); single quotation marks (' and '); double quotation marks (" and "); and so on. The printed recognizer can always return these characters, regardless of whether this flag is set.

Views restricted to the entry of phone numbers, dates, or times need not set the `vPunctuationAllowed` flag because the `vPhoneField`, `vDateField`, and `vTimeField` flags already allow the entry of appropriate punctuation.

The cursive recognizer can also apply some simple rules when deciphering ambiguous input; for example, it can make use of the fact that most punctuation marks follow rather than precede words.

Suppressing Spaces Between Words

Setting the `vSingleUnit` flag causes the recognition system to ignore spatial information when grouping input strokes as words; instead, the system relies on temporal cues to determine when the user has finished writing a word. When this flag is set, the recognizer ignores short delays, such as those that occur between writing the individual characters in a word. Longer delays cue the recognizer to group the most recently completed set of strokes as a word. The amount of time considered to be a longer delay is a function of the speed of the processor and the recognition system, as well as the value of the `timeoutCursiveOption` user preference.

The `vSingleUnit` flag is useful for views in which the presence of gratuitous spaces may confuse the recognizer; for example, phone number entry fields usually suppress the recognition of spaces. If you want to suppress all spaces in the displayed text, you can use the `vNoSpaces` flag in conjunction with the `vSingleUnit` flag.

Rather than suppressing the input of spatial cues, the `vNoSpaces` flag suppresses the insertion of spaces between groups of strokes or recognized text in views based on the `clParagraphView` class. This post-processing flag does not restrict the interpretation of the input strokes or affect word segmentation, as the `vSingleUnit` flag does.

CHAPTER 9

Recognition

The `vNoSpaces` flag must appear in an evaluate slot named `textFlags` that you create in the view. The `vSingleUnit` flag appears in the view's `viewFlags` slot, as usual.

Forcing Capitalization

The `vCapsRequired` flag directs the system to capitalize the first letter of each word returned by the recognizer before displaying the text in the view.

Setting the `vCapsRequired` flag does not affect the recognizer's behavior—it affects post-processing performed on the recognizer's output before it is returned to the view.

Justifying to Width of Parent View

Setting the `vWidthIsParentWidth` flag for a view based on the `clParagraphView` class causes the view to extend its right boundary to match that of its parent automatically.

The `vWidthIsParentWidth` flag must appear in an evaluate slot named `textFlags` that you create in the view.

Like other flags set in the `textFlags` slot, the `vWidthIsParentWidth` flag does not affect the recognizer's behavior—it affects post-processing performed on the recognizer's output before it is returned to the view.

Restricting Input to Single Lines or Single Words

Including the `oneLineOnly` flag in your view's `viewJustify` slot causes the view to accept only a single line of text input, with no word wrapping provided.

You can restrict input to a single word by including the `oneWordOnly` flag in the view's `viewJustify` slot. If this flag is set, the view replaces the currently displayed word with the new one when the user writes in the view. You can also restrict input to single characters by using this flag in conjunction with a custom dictionary of single letters.

For more information on these flags, see their descriptions in Chapter 3, “Views.” For information on the use of custom dictionaries, see “Using Custom Dictionaries” beginning on page 10-24.

Validating Clipboard and Keyboard Input

It is possible for the user to enter invalid values in fields by dragging text from the Clipboard or by using a keyboard to type in the field. For example, setting the `vPhoneField` flag normally restricts input to numeric values in phone number formats; however, the user can still enter invalid values in such a field by dragging

CHAPTER 9

Recognition

or typing them. To prevent invalid input by these means, you can implement a `ViewChangedScript` method that validates its view's input.

Using the `vAnythingAllowed` Mask

The `vAnythingAllowed` mask can be used only with views based on the `clEditView` class. When used by itself, this mask sets all of the bits in the view's input mask, potentially enabling all of the system-supplied recognizers and dictionaries. However, the actual recognition behavior of views that use this mask varies according to current user preference settings.

For a view that sets the `vAnythingAllowed` mask, the recognition system replaces the set of view flags you've specified with a set of flags derived from the current settings of user preferences that affect recognition. The actual set of recognizers enabled for the view is controlled by

- user preferences specified in the system's user configuration data.
- the application's `protoRecToggle` view, if it has one.
- the view's `recConfig` frame, if it has one.

Slots in the system's user configuration data specify recognition behaviors that all views inherit. However, an optional `protoRecToggle` view can specify different behaviors for individual views by overriding values inherited from user configuration data. Similarly, each view can provide a `recConfig` frame that overrides settings specified by the `protoRecToggle` view or the system's user configuration data.

Thus, in practice, the `vAnythingAllowed` mask usually is not what its name implies: if any bit in this mask is turned off (by another flag, or by a `recToggle` view, for example), the input mask is no longer `vAnythingAllowed`.

The built-in Notepad application provides a good example of the behavior of views that use the `vAnythingAllowed` mask, including the use of a `protoRecToggle` view to change recognition settings.

CHAPTER 9

Recognition

Summary

Constants

Text Recognition View Flags

Constant	Value	Description
vCharsAllowed	1 << 12 or 0x01000	Enables default text recognizer and default dictionary set.
vLettersAllowed	1 << 14 or 0x04000	Enables letter-by-letter text recognition.
vAddressField	1 << 21 or 0x0200000	Enables recognizers and dictionaries suitable for the input of address data in the current locale.
vNumbersAllowed	1 << 13 or 0x02000	Enables the recognition of numeric characters, monetary values (for example, \$12.25), decimal points, and signs (+ or -).
vNameField	1 << 22 or 0x0400000	Enables text recognition optimized for name data; usually combined w/ vCapsRequired.
vCustomDictionaries	1 << 24 or 0x01000000	Enables text recognition using dictionaries specified by the view's dictionaries slot.
vPunctuationAllowed	1 << 15 or 0x08000	Enables recognition of punctuation marks by the cursive recognizer. (Printed recognizer always recognizes punctuation marks.)
vCapsRequired	1 << 23 or 0x0800000	Forces capitalization of the first character of each recognized word.

CHAPTER 9

Recognition

Non-Text Recognition View Flags

Constant	Value	Description
vNothingAllowed	0x00000000 or 0x0000	The view accepts no handwritten or keyboard input.
vAnythingAllowed	65535 << 9 or 0x01FFFE00	Recognize any input. Use only for views based on the <code>clEditView</code> class.
vClickable	1 << 9 or 0x0200	Accept taps and send <code>ViewClickScript</code> message to the view once for each tap that occurs within the view.
vStrokesAllowed	1 << 10 or 0x0400	Accept stroke input and send the <code>ViewStrokeScript</code> message at the end of each stroke.
vGesturesAllowed	1 << 11 or 0x0800	Recognize gesture strokes such as scrub, highlight, tap, double tap, caret, caret-drag, and line. Send the <code>ViewGestureScript</code> message when the view recognizes a gesture that it does not handle automatically.
vShapesAllowed	1 << 16 or 0x010000	Enables shape recognition. Use only for views based on the <code>clEditView</code> class.
vSingleUnit	1 << 8 or 0x0100	Disable the use of spatial cues (distance between strokes). Meaningful for text recognizers only.
vNoSpaces	1 << 1 or 0x0002	Directs a view based on the <code>clParagraphView</code> class to not insert spaces between existing text and new text.
vWidthIsParentWidth	1 << 0 or 0x0001	Extend right boundary of <code>clParagraphView</code> view to match that of its parent.

CHAPTER 9

Recognition

View Flags Enabling Lexical Dictionaries

Constant	Value	Description
vNumbersAllowed	1 << 13 or 0x02000	Enables recognition of numbers, monetary values (for example, \$12.25), decimal points, and mathematical signs (+ and -).
vPhoneField	1 << 18 or 0x040000	Enables recognition of phone numbers. Note that the set of lexical dictionaries enabled by this flag varies with the text recognizer currently in use.
vDateField	1 << 19 or 0x080000	Enables recognition of date formats (such as March 3-95), names of months, and names of days.
vTimeField	1 << 20 or 0x0100000	Enables recognition of times.

Data Structures

Recognition-Related User Configuration Slots

Use the `GetUserConfig` and `SetUserConfig` global functions to access these slots.

Slot name	Notes
letterSetSelection	Text recognizer in use.
learningEnabledOption	true enables cursive learning.
letterSpaceCursiveOption	Space between stroke groups.
timeoutCursiveOption	Time between individual strokes.
speedCursiveOption	Time spent analyzing input.
letterInFieldsOption	true enables cursive recognizer's letter-by-letter option in <code>protoLabelInputLine</code> views.
lettersCursiveOption	true enables cursive recognizer's letter-by-letter option in built-in Names and Dates applications' <code>protoLabelInputLine</code> views.
doAutoAdd	true adds new words to user dictionary and auto-add dictionary automatically.

continued

CHAPTER 9

Recognition

Slot name	Notes
doTextRecognition	true enables text recognition unconditionally.
doShapeRecognition	true enables shape recognition unconditionally.
doInkWordRecognition	true causes text recognizer to return ink text rather than sketch ink.

C H A P T E R 1 0

Recognition: Advanced Topics

This chapter describes advanced uses of the Newton recognition system. If you are developing an application that supports ink text, implements specialized recognition system behavior, or provides unusual input views, you'll need to understand one or more topics presented here. This chapter describes

- the use of `recConfig` frames. An individual view can use a `recConfig` frame to specify its own recognition behavior, support ink text, specify baseline information, support deferred recognition, and define input areas for single letters.
- the use of text-corrector views and text-correction information.
- the programmatic manipulation of system dictionaries and custom dictionaries.

Before reading this chapter, you should understand the contents of Chapter 9, "Recognition," which provides an overview of the recognition system and describes how to implement its most common behaviors. Depending on your application development goals, you may also find it helpful to be familiar with soups, as described in Chapter 11, "Data Storage and Retrieval."

About Advanced Topics in Recognition

This section provides conceptual information regarding

- how views configure recognizers and dictionaries based on the interaction of view flags, `recConfig` frames, `recToggle` views, and recognition-related user preferences.
- the use of `protoCharEdit` views.
- deferred recognition.

How the System Uses Recognition Settings

A number of settings that control the behavior of the various recognizers are specified by the system's user configuration data. All views that perform recognition inherit behavior from these values, which is why it's rarely appropriate for individual

CHAPTER 10

Recognition: Advanced Topics

applications to change these system-wide settings. Instead, individual views can customize their own recognition behavior by using a `recConfig` frame or `recToggle` view to override these inherited values locally.

In practice, most views' recognition behavior is defined by a combination of inherited and overridden values. For example, because most users tend not to change the speed at which they write, it's common for views to use inherited values for the `timeoutCursiveOption` slot, which specifies the relative delay required to consider a group of input strokes complete. At the same time, individual views may customize certain recognition settings by overriding values that would otherwise be inherited from the system's user configuration data. For example, a view can use a `recConfig` frame to disable the automatic addition of new words to the user dictionary.

A view based on the `protoRecToggle` system prototype provides another way to override inherited recognition settings. This view provides a picker that allows the user to change recognition settings easily. Each view controlled by this picker must provide a `_recogSettings` slot that the picker sets according to the user's current choice of recognition settings. The value in the `_recogSettings` slot overrides values inherited from the system's user configuration data.

Your application supplies only one `_recogSettings` slot for each `recToggle` view it provides. Because views use parent inheritance to find a `_recogSettings` slot, a single `recToggle` view and a single `_recogSettings` slot can control the recognition behavior of one view or multiple views, depending on the `_recogSettings` slot's position in the view hierarchy. For more information, see "Creating the `_recogSettings` Slot" beginning on page 10-20.

You can also provide an optional `RecogSettingsChanged` method in the `_parent` chain of any view controlled by the `recToggle` view. If a `RecogSettingsChanged` method is provided, the `recToggle` view sends this message to `self` when the user chooses an item in the `recToggle` picker. Your `RecogSettingsChanged` method can perform any application-specific task that is appropriate; typically, this method reconfigures recognition settings in response to the change in the `recToggle` view's state.

Finally, any view can provide an optional `recConfig` frame that specifies the view's recognition behavior at the local level.

Although `recConfig` frames have thus far been presented as simply an alternate interface to the recognition system, they are actually used internally by the system to represent the recognition behavior of each view. When the user writes, draws, or gestures in a view, the system builds a `recConfig` frame that specifies the precise settings of all the recognizers needed for the view. If you supply a `recConfig` frame for the view, the `recConfig` frame that the system builds is based on the `recConfig` frame you have supplied and any recognition-related user preferences that may apply.

CHAPTER 10

Recognition: Advanced Topics

On the other hand, if the view does not supply a `recConfig` frame, the recognition system builds one based on the set of view flags enabled for that view, the contents of its `dictionaries` slot (if present) and any recognition-related user preferences that may apply. Thus, every view that performs recognition is eventually associated with a `recConfig` frame that the system uses to perform setup tasks when the view is opened.

Note that the `recConfig` frame actually used to configure recognition is the one that the system builds, not the one that you supply. The `recConfig` frame that you supply is referenced by the `_proto` slot of the `recConfig` frame that the system builds.

The `recConfig` frame built by the system is passed to a *recognition area*, which is an object used internally by the system to describe the recognition characteristics of one or more views. Because similar views can share an area, the use of recognition areas minimizes the reconfiguration of the recognition system required to respond to changes in views on the screen.

A small number of recognition areas are kept in a cache. You can change the recognition behavior of a view dynamically by specifying new recognition settings and invalidating the area cache. The next time the view accepts input, the system builds a new recognition area reflecting its currently-specified recognition behavior and the dictionaries it is to use for recognition.

In addition to providing an efficient and flexible means of configuring the recognition system programmatically, `recConfig` frames provide support for future expansion of the recognition system. The `recConfig` frame allows applications to specify recognition configurations in a uniform way that is not dependent on the use of any particular recognizer engine. Although the Newton platform currently supports only its built-in recognizers, future versions of the system may permit the use of third-party recognizer engines.

The system provides several standard `recConfig` frames that can be placed in your view's `recConfig` slot or used as a starting point for building your own `recConfig` frames. For descriptions of system-supplied `recConfig` frames, see "System-Supplied `recConfig` Frames" (page 8-18) in *Newton Programmer's Reference*.

In summary, the recognition behavior that a view exhibits is ultimately determined by a combination of the following values:

- values inherited from the system's user configuration data.
- values in the view's `viewFlags` and `entryFlags` slots.
- values in the view's `dictionaries` slot when the `vCustomDictionaries` flag is set.
- values specified by an optional `recToggle` view, which may override values inherited from user configuration data or supply additional values.

CHAPTER 10

Recognition: Advanced Topics

- values specified by an optional `recConfig` frame, which may override values inherited from user configuration data, override values specified by a `recToggle` view, or supply additional values.

ProtoCharEdit Views

The `protoCharEdit` system prototype provides a comb-style entry view (or *comb view*) that allows the user to edit individual characters in words easily.

Figure 10-1 Example of `protoCharEdit` view

Individual character positions (or *cells*) in the comb view are delimited by vertical dotted lines. Each cell that can be edited has a dotted line beneath it to indicate that it can be changed. The user can edit a character by writing a new character over one currently occupying a cell; the recognized value of the character is displayed in the cell. When the user taps a cell, it displays a picker containing the best interpretations of the input strokes. The user can correct the character in that position by choosing an item from the picker.

The user can delete an individual character by tapping it and then selecting “Delete” from the picker that is displayed. Alternatively, the user can delete one or more characters by writing the scrub gesture over one or more cells.

The user can insert a space by tapping on the cell at the position that the new space is to occupy and choosing Insert from the picker that is displayed.

Alternatively, the user can enter the caret gesture in a cell to perform the same operation. When an insertion takes place in a cell already occupied by a character, the comb view shifts that character and those comprising the rest of the word to the right.

Tapping a blank cell before or after a word in the comb view displays a list of punctuation characters that may be appropriate for that position.

The recognition behavior of a `protoCharEdit` view is controlled by values you supply in an optional `template` slot. If this slot’s value is `nil`, the comb view is said to be **unformatted** because input is not restricted in any way. The recognition behavior of an unformatted comb view is similar to that of the text-corrector view provided by the built-in Notepad application: all characters are allowed, insertion and deletion are supported fully, and spaces are added at the ends of words to allow them to be extended.

CHAPTER 10

Recognition: Advanced Topics

A **formatted comb view** utilizes a template you define which specifies characteristics of the view's behavior or appearance. A comb view's template may specify an initial value for the string that the view displays, the editing characteristics for each position in the comb view, and filters that restrict the values recognized in each of these positions. The template may also define methods for initializing and post-processing the string displayed by the comb view. These methods may be useful when the string displayed in the comb needs to be different from the input string or when an externally-displayed string must differ from its internal representation.

When the user taps a character in a formatted comb view, it displays the list of characters specified by its template, if that list contains ten or fewer items. (Note that this value may change in future platforms.) Otherwise, it displays the list of top-ranking alternate interpretations returned by the text recognizer.

Ambiguous Characters in protoCharEdit Views

Because there are several characters that are ambiguous in appearance—for example, the value zero (0) and the letter *O*, or the value one (1) and the letter *L*—the built-in system fonts provide enhanced versions of these characters that improve their readability. However, continuous use of these characters can be distracting to the user. Thus, these fonts contain character codes that map to alternate versions of the ambiguous characters, and the system provides functions for mapping between the codes for the normal and enhanced characters. For more information, see the descriptions of the `MapAmbiguousCharacters` and `UnMapAmbiguousCharacters` functions under “protoCharEdit Functions and Methods” (page 8-47) in *Newton Programmer's Reference*.

Deferred Recognition

Deferred recognition is the ability to convert strokes to text at some time other than when the strokes are first entered on the screen. Views that are to perform deferred recognition must be capable of capturing ink text or ink. For example, a view that bases its `recConfig` frame on the system-supplied `ROM_InkOrText` frame and uses a `protoRecToggle` view to configure the recognition system need not do anything more to provide the deferred recognition feature.

This section describes the user interface to deferred recognition and then provides a programmer's overview of this feature.

User Interface to Deferred Recognition

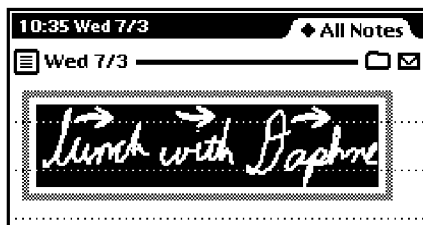
A view that performs deferred recognition uses the same settings as it would for real-time text recognition: a combination of settings specified by user preferences and settings specified by the view flags or `recConfig` frame associated with the view in which recognition takes place.

CHAPTER 10

Recognition: Advanced Topics

The user can enter unrecognized ink by enabling ink text or sketch ink. In this mode, strokes appear as ink. To convert the ink to text, the user double-taps the ink word; the user can cause multiple words to be recognized by selecting them beforehand and then double-tapping the selection. The recognition system responds by inverting the ink word or selection, as shown in Figure 10-2, and returning the recognized text, which replaces the selection.

Figure 10-2 User interface to deferred recognition, with inverted ink



Programmer's Overview of Deferred Recognition

Deferred recognition is available in views based on the `clEditView` class or `clParagraphView` views that support ink text. This feature works with any amount of input, from a single letter to a full page of text.

To initiate deferred recognition, the user double-taps the child views that display the ink to be recognized. The recognized text is added to an edit view as if the user had just written it. That is, a new `clParagraphView` child is added, or the recognized text is appended to a nearby `clParagraphView`. After the recognized text has been added, the original view containing the sketch ink or the ink text is removed from its edit view parent.

Deferred recognition also invokes the `ViewAddChildScript` and `ViewDropChildScript` methods of the recognized text and unrecognized ink views. Words added to nearby paragraphs invoke `ViewChangedScript` methods for those paragraphs, updating the `text` slot in those views; for some paragraph views, the `viewBounds` slot is updated as well.

You can pass `recConfig` frames to the global functions `Recognize`, `RecognizePara`, and `RecognizePoly` to implement your own form of deferred recognition. For more information, see “Deferred Recognition Functions” (page 8-89) in *Newton Programmer's Reference*.

CHAPTER 10

Recognition: Advanced Topics

Compatibility Information

The `ReadDomainOptions` function is obsolete. It has been replaced by the `ReadCursiveOptions` function.

The `AddToUserDictionary` function is obsolete. It has been replaced by the `AddWord` method of the review dictionary.

Two new dictionary constants, `kMoneyOnlyDictionary` and `kNumbersOnlyDictionary`, provide access to new lexical dictionaries used for recognizing monetary and numeric values, respectively.

Most lexical dictionaries are no longer locale-specific—aside from a few exceptions, each lexical dictionary is used for all locales. For detailed information, see “System-Supplied Dictionaries” (page 8-16) in *Newton Programmer’s Reference*.

All of the dictionary information provided by previous versions of system software is still present in version 2.0; however, certain dictionary constants now provide a superset of the information they previously referenced, as follows:

- The `kLastNamesDictionary` is obsolete. This information is now included in the `kSharedPropersDictionary` dictionary.
- The `kLocalCompaniesDictionary` constant is obsolete. This information is now included in the `kSharedPropersDictionary` dictionary.
- The `kLocalStatesAbbrevsDictionary` constant is obsolete. This information is now included in the `kSharedPropersDictionary` dictionary.
- The `kDateLexDictionary` constant is obsolete. It has been replaced by the `kLocalDateDictionary` constant.
- The `kTimeLexDictionary` constant is obsolete. It has been replaced by the `kLocalTimeDictionary` constant.
- The `kMoneyLexDictionary` constant is obsolete. This information is now included in the `kLocalNumberDictionary` dictionary.
- The `kNumberLexDictionary` constant is obsolete. This information is now included in the `kLocalNumberDictionary` dictionary.

Using Advanced Topics in Recognition

This section describes how to provide advanced recognition behaviors. It presumes understanding of conceptual material provided in this and other chapters. Topics discussed here include

- using `recConfig` frames to specify recognition behavior
- changing the recognition behavior of views dynamically

Using Advanced Topics in Recognition

10-7

CHAPTER 10

Recognition: Advanced Topics

- using `protoRecToggle` views to specify recognition behavior
- defining single-letter input areas within a view
- accessing text correction information
- using custom dictionaries for recognition
- manipulating the review dictionary (includes the user dictionary, expand dictionary, and auto-add dictionary)
- using `protoCharEdit` views for correcting text
- using stroke bundles

Using `recConfig` Frames

This section describes how to use a `recConfig` frame to specify a view's recognition behavior. Note that the use of view flags is generally the best (and simplest) way to configure views to recognize common input such as words and shapes. You need not use a `recConfig` frame unless you require some recognition behavior that cannot be provided using the view's `viewFlags` and `dictionaries` slots. For example, the use of a `recConfig` frame is required for views that restrict recognition of individual characters to a specified set, or implement customized forms of deferred recognition.

This section describes the use of `recConfig` frames for

- enabling recognizers
- supporting ink text
- fine-tuning recognition options
- specifying the dictionaries used for recognition

A `recConfig` frame can be used to specify any set of recognizers and dictionaries, including combinations not supported by the view flag model; however, views controlled by `recConfig` frames are subject to the same limitations as all views that perform recognition:

- The text recognizer (printed or cursive) made available to all views is determined by the value of the `letterSetSelection` slot in the system's user configuration data. Individual views cannot override this system-wide setting.
- The system's ability to save learning data is enabled by the value of the `learningEnabledOption` slot in the system's user configuration data. Individual views cannot override this system-wide setting.

Do not include `letterSetSelection` or `learningEnabledOption` slots in your `recConfig` frame.

CHAPTER 10

Recognition: Advanced Topics

Creating a recConfig Frame

For any view that is to use a `recConfig` frame, you must supply a `recConfig` slot, usually by defining it in your view's template. The frame in your view's `recConfig` slot must be modifiable; that is, it must be RAM-based. When your view template supplies a `recConfig` frame, the view system builds a RAM-based `recConfig` frame along with the view—you need not do anything more to cause the view to use the `recConfig` frame.

To create your own `recConfig` frame at run time, you need to call the `PrepRecConfig` function to create a RAM-based `recConfig` frame that the system can use. Although you could obtain similar results by cloning a `recConfig` frame that your view template defines, using the `PrepRecConfig` function is more efficient:

- The `PrepRecConfig` function creates a smaller frame than that obtained by cloning your view template's `recConfig` frame.
- The frame that the `PrepRecConfig` function returns can be used as it is by the recognition system. Any other frame that you place in the view's `recConfig` slot is used by the system to create the `recConfig` frame actually used by the view, with the result being the creation of two frames in RAM rather than just one.
- Consistent use of this function to create `recConfig` frames saves RAM by permitting similar `recConfig` frames to share the same frame map.

A function similar to the `PrepRecConfig` function, the `BuildRecConfig` function, is provided for debugging use. Do not use the `BuildRecConfig` function to create your RAM-based `recConfig` frame. The argument to the `BuildRecConfig` function is the view itself, rather than its `recConfig` frame. This function builds an appropriate `recConfig` frame for the specified view, regardless of whether the view defines one. The system does not use the `recConfig` frame that this function returns, however—as stated previously, this frame is for debugging use only.

IMPORTANT

The contents of the `inputMask` slot in the view's `recConfig` frame must match the input mask (the recognition-related bits) provided by the view's `viewFlags` slot. For more information on this slot and others that the `recConfig` frame may contain, see “`protoRecConfig`” (page 8-36) in *Newton Programmer's Reference*. ▲

You can base your `recConfig` frame on one of the system-supplied `recConfig` frames by simply placing the appropriate constant in your view template's `recConfig` slot. Alternatively, you can place in this slot a frame that uses its `_proto` slot to reference one of the system-supplied `recConfig` frames. A third way to define a `recConfig` frame is to supply all necessary values yourself. The

CHAPTER 10

Recognition: Advanced Topics

exact complement of slots and values required is determined by the recognition features your `recConfig` frame is intended to supply; for more information, including complete descriptions of the system-supplied `recConfig` frames, see “System-Supplied `recConfig` Frames” (page 8-18) in *Newton Programmer’s Reference*.

Once you’ve created a RAM-based `recConfig` frame, you must cause the recognition system to use it. This process is described in “Changing Recognition Behavior Dynamically” beginning on page 10-17. For a code example showing how to create a `recConfig` frame based on one of the system-supplied prototypes, see “Creating Single-Letter Input Views” beginning on page 10-15.

Using `RecConfig` Frames to Enable Recognizers

To enable or disable recognizers unconditionally, supply appropriate values for the `doTextRecognition`, `doShapeRecognition`, or `doInkWordRecognition` slots your view’s `recConfig` frame provides. For descriptions of these slots, see “`protoRecConfig`” (page 8-36) in *Newton Programmer’s Reference*.

For some operations, you may wish to restrict the recognizers that the user can enable in a view while still respecting the rest of the preferences indicated in the system’s user configuration data. The optional slots `allowTextRecognition` and `allowShapeRecognition` in the view’s `recConfig` frame are intended for use with views having an input mask that is `vAnythingAllowed`. For complete descriptions of these slots, see “`protoRecConfig`” (page 8-36) in *Newton Programmer’s Reference*. Note that you can also allow the user to set the values of these slots from a `protoRecToggle` view instead of setting them yourself in the `recConfig` frame.

Views that use the `allowSomethingRecognition` slots allow the user to turn on only the recognizers that you specify while respecting all other user preferences. Any subset of `allowSomethingRecognition` slots can be specified to allow the user to enable any appropriate combination of recognizers from the `protoRecToggle` view or user preferences.

For example, setting the value of the `allowTextRecognition` slot to `true` allows the user to enable the text recognizer in the view controlled by the `recConfig` frame while the `doTextRecognition` slot in the system’s user configuration data holds a non-`nil` value.

Returning Text, Ink Text or Sketch Ink

This section discusses the use of `recToggle` views with system-supplied view classes and `recConfig` frames to provide views that can display text, ink text, or sketch ink.

CHAPTER 10

Recognition: Advanced Topics

Sketch ink, like shapes, is displayed only in views based on the `clEditView` class. As a rule of thumb, consider sketch ink and ink text to be mutually exclusive when configuring recognition in views; for best results, configure your input view to recognize only one of these two data types.

Views based on the `clEditView` class handle sketch ink and ink text automatically. For other views, the system invokes the view's `ViewInkWordScript` or `ViewRawInkScript` method when ink arrives. For more details, see the descriptions of these methods in *Newton Programmer's Reference*.

The system-supplied `ROM_rcInkOrText` constant provides a ready-to-use `recConfig` frame that allows views based on the `clParagraphView` class to contain ink text in addition to normal text. To use this `recConfig` frame to create a view that supports ink text, you'll need to take the following steps:

- Create a view template that inherits from the `clParagraphView` class.
- In your view template, create a `recConfig` slot that holds the `ROM_rcInkOrText` constant. For more information, see “Creating a `recConfig` Frame” beginning on page 10-9.
- Provide a `protoRecToggle` view that allows the user to choose text or ink text settings; if your application provides a status bar, you need to provide the `recToggle` view as one of its children. For more information, see “Creating the `recToggle` View” beginning on page 10-19.
- Provide a `_recogSettings` slot at an appropriate position in the `recToggle` view's `_parent` chain. For more information see “Creating the `_recogSettings` Slot” beginning on page 10-20.

Normally, the input view tries to recognize input using all currently enabled recognizers. If no recognizers are enabled or if recognition fails for some reason—for example, due to messy input or some sort of error—then the view system converts the input strokes into ink. The `doInkWordRecognition` slot in the input view's `recConfig` frame specifies the kind of ink that the system creates from the input strokes.

When the `doInkWordRecognition` slot holds a non-`nil` value, the system returns ink text; when this slot holds the `nil` value, the system returns sketch ink. This slot is described fully in “`protoRecConfig`” (page 8-36) in *Newton Programmer's Reference*. Table 10-1 on page 10-12 summarizes the kinds of data returned by the recognition system when recognition fails in an edit view or paragraph view that is controlled by a `recToggle` view.

Note that when the input view is set to recognize shapes, the smoothed and cleaned up ink that is returned may be ink text but is more likely to be a curve shape. Aside from the failure of shape recognition, the only time raw ink is returned to the view is when its associated `recToggle` is set to “Sketches”.

CHAPTER 10

Recognition: Advanced Topics

Table 10-1 Recognition failure in paragraph or edit view controlled by `recToggle`

Recognizer enabled by <code>recToggle</code> view	Returns on failure
Text	Ink text
Ink text	Ink text (does not fail)
Shapes	Sketch ink, smoothed
Sketch ink	Nothing (occurs rarely)

As an alternative to using a `recConfig` frame to provide support for ink text, you can set your `clParagraphView` view's `vAnythingAllowed` mask. Although this is truly the easiest way to support ink text, it is less-preferred because it provides you the least control over the view's recognition behavior. A variation on this approach that may provide better performance is to enable an appropriate set of view flags rather than setting the `vAnythingAllowed` mask. The best way to support ink text, however, is through the use of a `recConfig` frame that provides appropriate values.

Regardless of the approach you take to provide ink text support, you should test your view's recognition behavior under both text recognizers, and under any other configurations your `recToggle` view provides.

To support both ink text and sketch ink in a single view, or to take other application-specific action in response to changes in the state of a `recToggle` view, your view can provide a `RecogSettingsChanged` method that reconfigures the its recognition behavior dynamically. For more information, see “Changing Recognition Behavior Dynamically” beginning on page 10-17.

For more information on `protoRecToggle` views, see “Using `protoRecToggle` Views” beginning on page 10-19. For detailed descriptions of `recConfig` frames, see “`protoRecConfig`” (page 8-36) and “System-Supplied `recConfig` Frames” (page 8-18) in *Newton Programmer's Reference*.

Fine-Tuning Text Recognition

To fine-tune either text recognizer's interpretation of input strokes, you can add the optional `speedCursiveOption`, `timeoutCursiveOption`, and `letterSpaceCursiveOption` slots to the `recConfig` frame. These mechanisms for controlling recognizer behavior may affect various recognizers differently. For more information, see “`protoRecConfig`” (page 8-36) in *Newton Programmer's Reference*. For a discussion of the dictionaries slot, see “Using Your RAM-Based Custom Dictionary” beginning on page 10-28.

CHAPTER 10

Recognition: Advanced Topics

Manipulating Dictionaries

You can control the view's use of dictionaries by including in your `recConfig` frame the `dictionaries`, `rcSingleLetters`, or `inhibitSymbolsDictionary` slots as appropriate. These slots are described in "protoRecConfig" (page 8-36) in *Newton Programmer's Reference*.

Single-Character Input Views

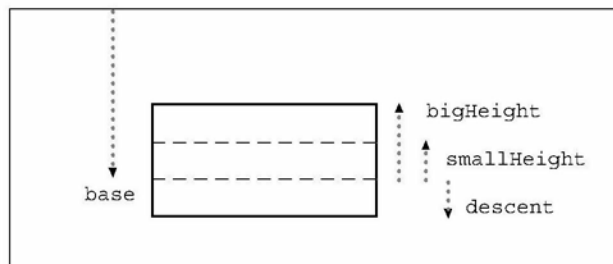
When recognizing single characters, the recognizer sometimes has difficulty determining individual characters' baseline or size; for example, it can be difficult to distinguish between an upper case *P* and a lower case *p* when relying strictly on user input. If you know where the user will be writing with respect to a well-defined baseline, you can provide an `rcBaseInfo` or `rcGridInfo` frame to specify to the recognition system precisely where characters are written.

The `rcBaseInfo` Frame

The `rcBaseInfo` frame is especially valuable in improving the recognition of single characters, for which it is sometimes difficult to derive baseline or letter-size values from user input.

Figure 10-3 depicts the editing box that an `rcBaseInfo` frame defines.

Figure 10-3 Single-character editing box specified by `rcBaseInfo` frame



The NewtonScript code used to create the baseline information for the editing box shown in Figure 10-3 looks like the following example.

```
rcBaseInfo := {
  base: 140, // global y-coordinate of baseline
  smallHeight: 15, // height of a lower case x
  bigHeight: 30, // height of an upper case X
  descent: 15, // size of descender below baseline
};
```

CHAPTER 10

Recognition: Advanced Topics

To obtain the best performance and to conserve available memory, create your `rcBaseInfo` frame by cloning the frame provided by the `ROM_canonicalBaseInfo` constant. Store your frame in a slot named `rcBaseInfo` in your input view's `recConfig` frame.

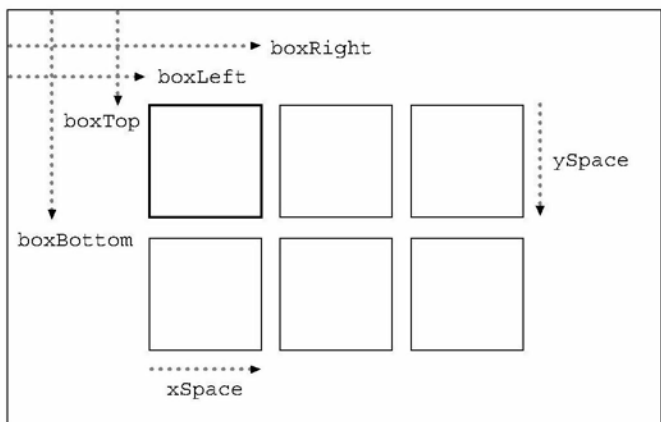
For a detailed description of the `rcBaseInfo` frame, see “Data Structures Used in `recConfig` Frames” (page 8-24) in *Newton Programmer's Reference*.

The `rcGridInfo` Frame

The `rcGridInfo` frame allows you to define the position of one or more single-letter input areas within a single input view. Its purpose is to facilitate the creation of views having multiple single-letter input areas, such as might be used by a crossword puzzle application. Providing a separate view for each single letter input area would be extremely inefficient; the use of an `rcGridInfo` frame allows you to draw one view that provides the illusion of many input views, by defining to the recognizer the size of an individual input area and the spacing between input areas.

Figure 10-4 depicts an example of the grid that an `rcGridInfo` frame defines. The boxes shown in this figure are not views themselves, just lines on the screen that indicate the location of the input areas to the user. The recognition behavior is provided by the view that draws these boxes; the `rcGridInfo` frame helps the recognizer determine the precise location of user input, and, consequently, where to display its output. By providing the proper slots, you can use an `rcGridInfo` frame to define a row, column, or matrix (as shown in the figure) of single-letter input areas within a view.

Figure 10-4 Two-dimensional array of input boxes specified by `rcGridInfo` frame



CHAPTER 10

Recognition: Advanced Topics

If you provide a grid in which the user is to write characters or words, you need to use an `rcGridInfo` frame to define the grid to the text recognizer. For example, the `protoCharEdit` system prototype uses an `rcGridInfo` frame internally to define the input areas (cells) in the comb view it provides.

The recognizer uses the information in an `rcGridInfo` frame to make character-segmentation decisions. You can use the `rcGridInfo` frame in conjunction with an `rcBaseInfo` frame to provide more accurate recognition within boxes in a single view. Recognition in the most recently used grid box begins as soon as the user writes in a new box in the grid.

The NewtonScript code used to create the grid shown in Figure 10-4 looks like the following example.

```
rcGridInfo := {
    boxLeft: 100, // x coordinate of left of top-left box
    boxRight: 145, // x coordinate of right of top-left box
    xSpace: 55, // x distance from boxLeft to boxLeft
    boxTop: 50, // y coordinate of top of top-left box
    boxBottom: 95, // y coordinate of bottom of top-left box
    ySpace: 55 // y distance from boxTop to boxTop
};
```

To obtain the best performance and to conserve available memory, create your `rcGridInfo` frame by cloning the frame provided by the `ROM_canonicalCharGrid` constant. Store your frame in a slot named `rcGridInfo` in your view's `recConfig` frame.

For a detailed description of the `rcGridInfo` frame, see “Data Structures Used in `recConfig` Frames” (page 8-24) in *Newton Programmer's Reference*

Creating Single-Letter Input Views

The following code fragment creates a single-letter input view's `recConfig` frame. This frame, which includes `rcBaseInfo` and `rcGridInfo` frames, is based on the `ROM_rcSingleCharacterConfig` frame supplied by the system.

```
// specify box (or horizontal array of boxes)
// into which character(s) are written.

myView := {
    recConfig: ROM_rcsinglecharacterconfig,
    ...}

// height of a lowercase letter
constant kSmallHeight := 11;
```

CHAPTER 10

Recognition: Advanced Topics

```

// indent from left of view to first letter
constant kBoxIndent := 4;
// width of a single box in the grid
constant kCellWidth := 24;

// create editable recConfig frame and set initial values
myView.ViewSetupDoneScript := func()
begin
    // prebuild RAM copy that we can change
    recConfig := PrepRecConfig(recConfig);

    // set these same flags in myView.viewFlags
    recConfig.inputMask :=
        vClickable+vGesturesAllowed+vCustomDictionaries;

    // get global bounds of enclosing view
    local box := :GlobalBox();
    // calc left edge of boxes in grid
    local leftX := box.left + kBoxIndent;

    // specify baseline and expected letter height
    recConfig.rcBaseInfo :=
    {
        // baseline for writing
        base: box.top + viewLineSpacing,
        // height of a small letter
        smallHeight: kSmallHeight,
    };

    // specify horizontal info for an array of boxes
    recConfig.rcGridInfo :=
    {
        // left edge of first box
        boxLeft: leftX,
        // right edge of first box
        boxRight: leftX + kCellWidth,
        // width to left edge of next box
        xSpace: kCellWidth,
    };

    // use new settings
    PurgeAreaCache();
end;

```

CHAPTER 10

Recognition: Advanced Topics

The `PurgeAreaCache` function causes the recognition system to adopt the settings that the `recConfig` frame specifies. This function is explained in more detail in the next section, “Changing Recognition Behavior Dynamically.”

Normally, you need not call the `PurgeAreaCache` function when specifying a `recConfig` frame as part of a view’s template. However, you must call this function to change a `recConfig` frame at run time. For example, the previous code fragment calculates values determining the size and location of the grid view according to the size of the enclosing parent view; thus, the parent view must already exist before the grid view’s `recConfig` frame can be constructed. Therefore, the grid view’s `recConfig` frame is constructed from within the `ViewSetupDoneScript` method of the parent view that encloses the grid view. At the time the `viewSetupDoneScript` method is executed, the system has already used the `recConfig` frame supplied by the enclosing view’s template. In order to cause the system to use the new `recConfig` frame—the one that defines the grid view—the `ViewSetupDoneScript` method must call the `PurgeAreaCache` function.

Changing Recognition Behavior Dynamically

To change a view’s recognition behavior dynamically, you must indicate the view’s new configuration (by setting view flags, changing the view’s dictionaries slot, or defining a `recConfig` frame) and make the recognition system use the new settings. The system supplies three functions that you can use to make the system adopt new recognition settings; each is appropriate for a particular situation.

The function you use to adopt new settings depends on whether you are changing the recognition behavior of all views or just changing the behavior of individual views. Changes to user preferences for recognition affect the recognition behavior of all views. On the other hand, changing the value of a single view’s `viewFlags` or `recConfig` slot affects that view only.

Note

It is recommended that you do not change any user settings without confirmation from the user. ♦

To change the recognition behavior of a single view dynamically, use the global function `SetValue` to change the value of the view’s `viewFlags` slot or `recConfig` slot. In addition to setting the new value, the `SetValue` function invalidates the area cache, which is a buffer that stores a small number of recognition areas. Your changes to recognition behavior are incorporated when the recognition area for your view is rebuilt.

CHAPTER 10

Recognition: Advanced Topics

▲ WARNING

The `SetValue` function may not be appropriate for setting the `entryFlags` slot in views that do not have a `viewFlags` slot. In these kinds of views, set the value of the `entryFlags` slot directly and then call the `PurgeAreaCache` function to invalidate the area cache. If you have changed values in the system's user configuration data, call the `ReadCursiveOptions` function instead of the `PurgeAreaCache` function. ▲

You can also use the `PurgeAreaCache` function to invalidate the area cache. This function provides an efficient way to force the reconstruction of recognition areas after you've changed the values of slots in multiple views. Note, however, that this function does not resynchronize the recognition system with changes in the system's user configuration data. Do not call `PurgeAreaCache` to effect changes in user preferences for recognition.

User preferences that affect recognition behavior are saved as slot values in the system's user configuration data. Some of these values, such as that of the `timeoutCursiveOption` slot, affect all views; others affect only views that set the `vAnythingAllowed` mask. For detailed information about the slot you need to set, see its description in "System-Wide Settings" (page 8-2) in *Newton Programmer's Reference*.

When setting user preferences for recognition, do not modify the system's user configuration data directly. Instead, use the `GetUserConfig` and `SetUserConfig` global functions to manipulate user configuration values.

After calling the `SetUserConfig` function to set one or more new values, you must call the `ReadCursiveOptions` function to cause the recognition system to use the new values. Do not call the `PurgeAreaCache` function after changing values in the system's user configuration data—this function does not even test for changes to user preferences. Because the `ReadCursiveOptions` function invalidates the area cache, you need not call the `PurgeAreaCache` function after calling the `ReadCursiveOptions` function.

IMPORTANT

The view's `viewFlags` slot must contain the same recognition flags as the `inputMask` slot in its `recConfig` frame. Certain view system operations depend on the `viewFlags` slot being set up properly. ▲

CHAPTER 10

Recognition: Advanced Topics

Using protoRecToggle Views

A `protoRecToggle` view changes the recognition behavior of views by overriding values inherited from the system's user configuration data. Note that values in the view's `recConfig` frame override settings specified by the `protoRecToggle` view.

The `protoRecToggle` view is usually used with `clEditView` views that set the `vAnythingAllowed` mask or `clParagraphView` views that support ink text.

Take the following steps to use a `protoRecToggle` view.

- Create the `recToggle` view in NTK. If your application has a status bar, you need to provide the `recToggle` view as a child of the status bar.
- Configure input views appropriately to support the choices your `recToggle` view provides. To do so, you need to provide an appropriate `recConfig` frame or set the `vAnythingAllowed` mask for each view that is to be controlled by the `recToggle` view.
- Provide a `_recogSettings` slot at a place in the `_parent` chain that allows each view controlled by the `recToggle` view to inherit this slot.

You can take the following optional steps to customize your `recToggle` view's appearance or behavior:

- Provide a `_recogPopup` slot specifying the items to be included in the `protoRecToggle` picker.
- Implement a `RecogSettingsChanged` method in the `_parent` chain of any view controlled by the `recToggle` view.

The next several sections describes these steps in detail.

Creating the recToggle View

To create a `recToggle` view, you'll first need to sketch it out in the NTK layout editor. When you do so, you'll notice that regardless of where you draw it, the view will appear in the upper-left corner of the layout. This is because the `recToggle` view is intended to be displayed as a child of the status bar in applications that have one.

When a `recToggle` view is a child of your application's status bar, the view system positions the `recToggle` view on the status bar automatically, ignoring the value of the `recToggle` view template's `viewBounds` slot in the process. When the `recToggle` view is not a child of the status bar, you must create a `viewBounds` slot for it and set appropriate values for this slot.

CHAPTER 10

Recognition: Advanced Topics

Configuring Recognizers and Dictionaries for recToggle Views

Regardless of whether you use a `recConfig` frame or view flags to specify your view's recognition behavior, the view must be capable of enabling recognizers and dictionaries appropriate for each choice in the `recToggle` picker. If your view does not support all of the recognition settings provided by the default `recToggle` view, you need to provide a `_recogPopup` slot that restricts the choices appearing in the picker that the `recToggle` view displays. For more information, see "Providing the `_recogPopup` Slot" beginning on page 10-22.

If you are using a `recConfig` frame to specify your view's recognition behavior, you can place the `ROM_rcPrefsConfig` constant in your `recConfig` frame's `_proto` slot to provide a general-purpose `recConfig` frame that allows recognition of all forms of pen input. Note that you must also enable recognition behavior and dictionaries as appropriate in order to produce useful behavior.

Creating the `_recogSettings` Slot

Applications that use a `recToggle` view must provide a `_recogSettings` slot in a view that is a parent to both the `recToggle` view and the input view it controls. Your view template should specify an initial value of `nil` for this slot. Each time the user chooses an item from the `recToggle` picker, it saves a value representing its current setting in this slot. You can preserve the user's recognition settings by saving the contents of this slot when your application closes and restoring this slot's value when your application reopens.

When a single `recToggle` view controls recognition for all of your application's views, the `_recogSettings` slot can reside in the application's base view, as shown in Figure 10-5.

This approach can be used to synchronize the recognition behavior of multiple views; for example, the built-in Notes application uses a single `recToggle` view to control the recognition behavior of all currently visible notes. All of the views controlled by a single `recToggle` view must provide the same set of recognizers and dictionaries.

When each of several `recToggle` views must control individual input views, you must provide a `_recogSettings` slot for each `recToggle` view at an appropriate place in the `_parent` chain of each view that performs recognition, as shown in Figure 10-6.

CHAPTER 10

Recognition: Advanced Topics

Figure 10-5 One recToggle controls all views

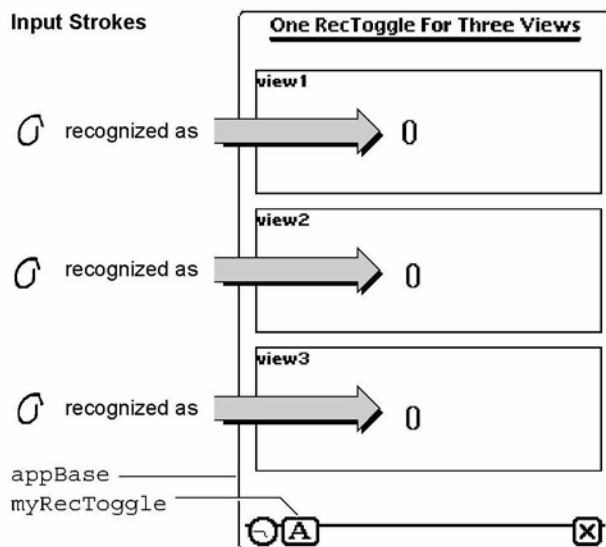
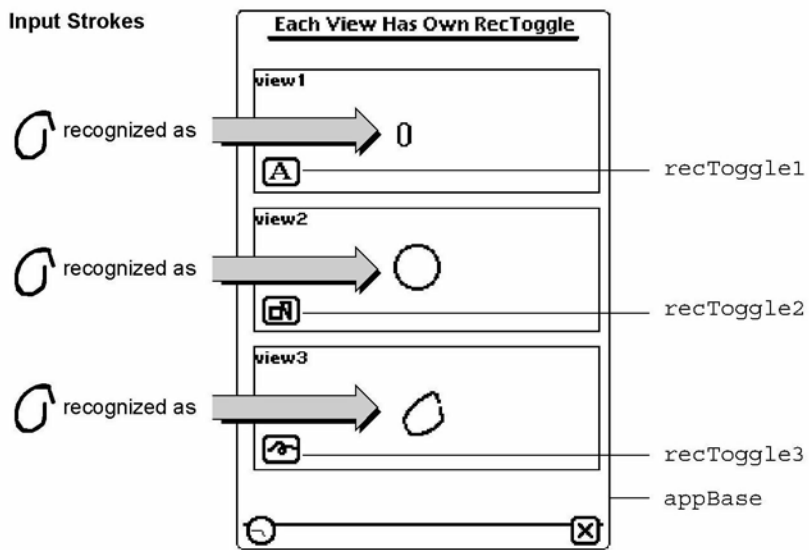


Figure 10-6 Each recToggle view controls a single input view



CHAPTER 10

Recognition: Advanced Topics

When the view receives input, it uses parent inheritance to find configuration information. If a `_recogSettings` slot exists in the view's `_parent` chain, the view uses the value of this slot, along with values supplied by an optional `recConfig` frame and values inherited from the system's user configuration data.

The `recToggle` view's `ViewSetupFormScript` method uses the value of the `_recogSettings` slot to set the state of the `recToggle` view. To restore the recognition settings that were in effect the last time your application was used, you can save the value of the `_recogSettings` slot when the application closes and restore the value of this slot when the application reopens. If you prefer that the `recToggle` view always open to a default setting, rather than a saved one, you can place the value `nil` in the `_recogSettings` slot when your application opens.

Providing the `_recogPopup` Slot

You can customize the appearance and behavior of your `recToggle` view by providing a `_recogPopup` slot in its view template. This slot contains an array of symbols corresponding to items included in the picker that the `recToggle` view displays. The first item in the array appears at the top of the picker and specifies the default recognizer enabled by the `recToggle` view. The picker includes subsequent items in the order in which they appear in the array.

Table 10-2 summarizes the symbols that may appear in the `_recogPopup` slot, along with the corresponding item each produces in the `recToggle` picker.

Table 10-2 Symbols appearing in the `_recogPopup` slot

Symbol	Represents	Picker item
'recogText	Text recognizer	Text
'recogInkText	Ink text	Ink Text
'recogShapes	Shape recognizer	Shapes
'recogSketches	Raw ink	Sketches
'recToggleSettings	Handwriting Recognition preferences slip	Preferences
'pickSeparator	No selection	Dashed line

To specify that the `recToggle` view enable a default recognizer other than the one specified by the first symbol in the `_recogPopup` array, your `recToggle` view's template can provide a `defaultItem` slot. This slot holds an integer value specifying the array element to be used as the default.

CHAPTER 10

Recognition: Advanced Topics

Avoid including inappropriate items in the `recToggle` popup, such as an ink text item for a view that does not support ink text. It is your responsibility to ensure that the `_recogPopup` array includes only symbols representing valid choices for the view that the `recToggle` configures.

Accessing Correction Information

As words are recognized, the system saves correction information that includes

- the stroke bundle. (See “Unrecognized Strokes” on page 9-7.)
- alternate interpretations returned by the recognizer. (See “Classifying Strokes” on page 9-3.)
- learning data. (See “Correction and Learning” on page 9-13.)

You can call the `GetCorrectInfo` global function at any time to obtain correction information for recently-recognized words. This function returns a `correctInfo` frame based on the `protoCorrectInfo` system prototype.

The `info` slot in the `correctInfo` frame holds an array of `wordInfo` frames based on the `protoWordInfo` system prototype. Each `wordInfo` frame represents a single written word.

The `max` slot in the `correctInfo` frame specifies the maximum number of words for which it holds correction information. When adding a new element to the `info` array will cause this array to exceed the size specified by the `max` slot, the system removes the first element of the `info` array, uses its learning data if necessary, and adds the new `wordInfo` frame to the `info` array.

The `correctInfo` frame provides a number of methods that you can use to manipulate its contents; for more information, see “CorrectInfo Functions and Methods” (page 8-54) in *Newton Programmer’s Reference*.

Each `wordInfo` frame specifies the following information:

- the view that contains the word.
- the position of the word within the `clParagraphView` view that displays it.
- the list of alternate interpretations of the input strokes.
- an identifier specifying the recognizer that interpreted the input.
- a stroke bundle (optional).
- learning data (optional).

As an alternative to obtaining `wordInfo` frames from the `correctInfo` frame, you can extract these frames from the word unit passed to an optional `ViewWordScript` method that your view provides. For a description of this method, see “Application-Defined Recognition Methods” (page 8-66) in *Newton Programmer’s Reference*.