
Data Browser Programming Guide

[Carbon](#) > [User Experience](#)



2007-08-07



Apple Inc.
© 2004, 2007 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

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

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

Apple, the Apple logo, Carbon, Mac, Mac OS, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Finder is a trademark of Apple Inc.

Simultaneously published in the United States and Canada.

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

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

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

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

Contents

Introduction **Introduction to Data Browser Programming Guide 7**

- Who Should Read This Document? 7
- Organization of This Document 7
- See Also 8

Chapter 1 **Data Browser Concepts 9**

- The Data Browser User Interface 9
- How the Data Browser Works 12
 - How Views are Implemented 12
 - Referring to Data 13
 - Callbacks and the Data Browser 14
 - How a Data Browser Gets Populated With Data 17
 - What Can Be Displayed 19

Chapter 2 **Data Browser Tasks 21**

- Tips for Using the Data Browser API 21
- Displaying Data in a Simple List View 22
 - Design and Configure a List View Data Browser 22
 - Declare the Necessary Global Constants 26
 - Write an Item-Data Callback 27
 - Write Code That Initializes the List View 28
- Displaying Hierarchical Data in a List View 30
 - Write an Item-Data Callback 31
 - Write an Item-Notification Callback for Hierarchical Data 32
 - Write Code That Initializes the Data Browser 33
- Displaying Data in a Column View 33
 - Design and Configure a Column View Data Browser 34
 - Write an Item-Data Callback 36
 - Write an Item-Notification Callback 38
 - Write Code That Initializes the Data Browser 38
- Switching Between List View and Column View 39
 - Set Up a Menu for Switching Views 40
 - Declare the Global Constants Needed for Switching 40
 - Write an Item-Data Callback to Handle Both Views 41
 - Write Code That Initializes a Switchable Data Browser 43
 - Write Code to Switch Between Views 45
 - Configure the Data Browser 45
 - Populate the Initial List View With Data 46
- Supporting Text Editing 47

Applying an Operation to Each Item 48
Sorting Items 48
Supporting Drag and Drop 48
Providing a Contextual Menu 49
Providing Help Tags 49
Customizing Drawing, Tracking, and Dragging 50

Document Revision History 51

Figures, Tables, and Listings

Chapter 1

Data Browser Concepts 9

| | | |
|------------|---|----|
| Figure 1-1 | Data displayed in a list view data browser | 9 |
| Figure 1-2 | Hierarchical data displayed in a list view data browser | 10 |
| Figure 1-3 | A list with movable columns | 10 |
| Figure 1-4 | A sortable column | 11 |
| Figure 1-5 | Data displayed using column view | 11 |
| Figure 1-6 | A list view data browser that displays a variety of data | 12 |
| Figure 1-7 | Column and list views are subclasses of table view | 13 |
| Figure 1-8 | An item is data at a row, column intersection | 14 |
| Table 1-1 | Properties you can respond to in an item-data callback | 15 |
| Table 1-2 | Messages sent by the data browser to the item-notification callback | 16 |

Chapter 2

Data Browser Tasks 21

| | | |
|--------------|--|----|
| Figure 2-1 | A simple list view | 22 |
| Figure 2-2 | Setting up the Control pane for a list view | 23 |
| Figure 2-3 | Setting up the Attributes pane for a list view | 24 |
| Figure 2-4 | Setting up a header and number of columns for a list view | 24 |
| Figure 2-5 | Setting up the Columns pane for a list view | 25 |
| Figure 2-6 | Hierarchical data displayed in a list view | 30 |
| Figure 2-7 | A column view data browser | 34 |
| Figure 2-8 | Setting up the Control pane for a column view | 35 |
| Figure 2-9 | Setting up the Attributes pane for a column view | 36 |
| Figure 2-10 | Assigning a command to a menu item | 40 |
| Figure 2-11 | A contextual menu in the Finder | 49 |
| Figure 2-12 | A help tag in a data browser | 50 |
| Listing 2-1 | An item-data callback that populates a simple list view | 27 |
| Listing 2-2 | Initializing a simple list view data browser | 29 |
| Listing 2-3 | An item-data callback for a hierarchical list | 31 |
| Listing 2-4 | An item-notification callback for a hierarchical list | 32 |
| Listing 2-5 | Initializing a hierarchical list view data browser | 33 |
| Listing 2-6 | An item-data callback for a column view data browser | 36 |
| Listing 2-7 | Initializing a column view data browser | 38 |
| Listing 2-8 | An item-data callback for a data browser that can switch between views | 41 |
| Listing 2-9 | Initializing a data browser that can switch between views | 43 |
| Listing 2-10 | Configuring the data browser for each view | 45 |
| Listing 2-11 | Reading data and populating the data browser | 47 |

Introduction to Data Browser Programming Guide

Note: This document was formerly titled *Displaying Data in a Data Browser*.

A data browser provides a user interface for displaying and selecting items from a list of data or from hierarchically organized lists of data such as directory paths. The list and column views in the Finder are representative of the kind of behavior your application can provide using the data browser API.

The data browser API supports the following:

- The ability to display an unlimited number of cells
- Built-in drag-and-drop handling
- Display of a variety of data—text, icons, checkboxes, pop-up menus, progress bars, relevance indicators, and sliders
- Contextual menus and help tags
- Built-in support for editing text displayed in the data browser
- Display of hierarchically organized data in a list
- Keyboard navigation and accessibility

Who Should Read This Document?

This document is targeted at Mac OS X Carbon developers who want to display data that can be browsed in a way similar to data browsing in the Mac OS X Finder. The document assumes you are familiar with Mac OS X programming and with Carbon in particular.

Some of the features available in a data browser assume you are familiar with the technologies underlying the features. For example, if you want to provide help tags for data browser items, knowledge of the Carbon Help Manager is useful. Cross-references are provided for situations in which information from another technology is helpful.

Organization of This Document

This document contains the following chapters:

[“Data Browser Concepts”](#) (page 9) shows typical data browser user interfaces, introduces data browser terminology, describes how the data browser works, provides in-depth information on what can be displayed, and defines the terms needed to refer to the displayed data.

[“Data Browser Tasks”](#) (page 21) describes how to use the data browser API to display data in list view and column view, to switch between views, to support text editing in a cell. It also provides information on the callbacks you need to supply to support drag and drop, provide help tags and contextual menus, and to customize drawing, tracking, and dragging behavior.

See Also

Data Browser Reference. This document is a complete reference for the functions, callbacks, data types, and constants provided by the data browser API.

Data Browser Concepts

This section provides definitions for key concepts needed to understand and use the data browser API. It contains information on the following topics:

- [“The Data Browser User Interface”](#) (page 9). Provides a description of list and column views and shows a variety of implementations that call out the use of such things as disclosure triangles, placards, icons with text, relevance ranking, and date-time information.
- [“How the Data Browser Works”](#) (page 12). Discusses how data is identified in each view, how your application specifies the identifying values, the tasks performed by callbacks you supply, how a view gets populated with data, the view hierarchy, and the types of data you can display.

The Data Browser User Interface

The data browser can present data in two different kinds of displays—list view and column view. List view displays data in a list with one or more columns. A simple list with four columns—Title, Time, Genre, and My Rating—is shown in Figure 1-1.

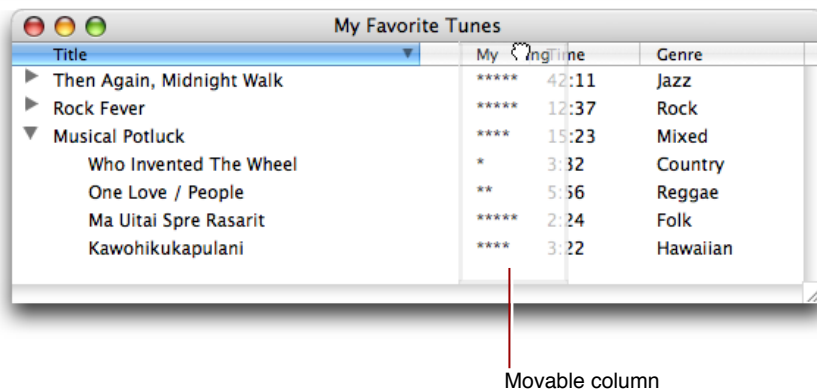
Figure 1-1 Data displayed in a list view data browser

| Title | Time | Genre | My Rating |
|-----------------------------|------|----------|-----------|
| A Box Full Of Sharp Objects | 5:57 | Rock | ***** |
| Crazy | 4:38 | Rock | **** |
| Down With Sickness | 2:56 | Rock | *** |
| Kawahikukapulani | 3:22 | Hawaiian | **** |
| Ma Uitai Spre Rasarit | 2:24 | Folk | ***** |
| One Love / People | 5:56 | Reggae | ** |

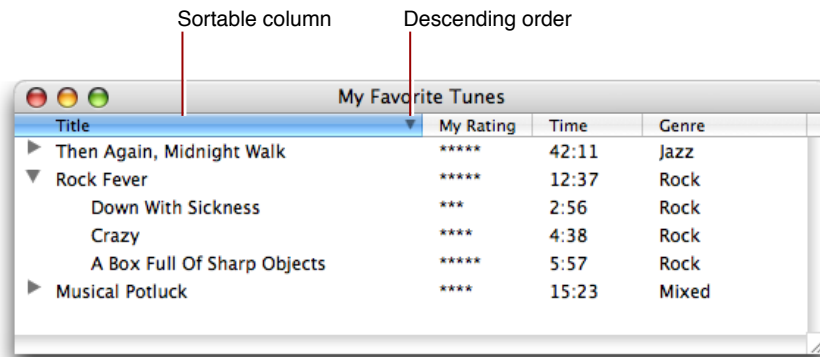
A list can display hierarchically arranged data, as shown in Figure 1-2. A disclosure triangle next to an item indicates that the item is a container. The user can open and close a container by clicking its disclosure triangle.

Figure 1-2 Hierarchical data displayed in a list view data browser

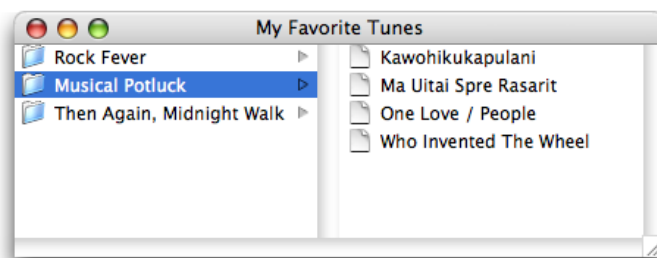
A list can be set up to provide the user with the option to move the columns, as shown in Figure 1-3. This example shows the user dragging the My Rating column so it is positioned between the Title column and the Time column.

Figure 1-3 A list with movable columns

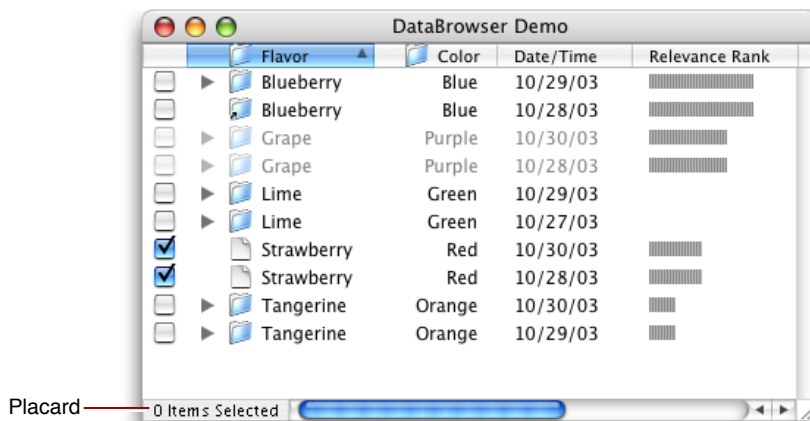
List view can also support sorting, as shown in Figure 1-4. The triangle to the right of a column title indicates whether the current sorting order is ascending or descending. In the figure, the sorting order is descending. The user can change the sorting order by clicking the column title.

Figure 1-4 A sortable column

Column view provides in-place browsing using fixed navigation columns. Figure 1-5 shows the same data that is displayed in Figure 1-4, but in column view. Column view is typically used to navigate to items in a data hierarchy, while list view is used to find out information about a specific data item. The Finder is an example of an application that can display data in either list view or column view.

Figure 1-5 Data displayed using column view

A data browser can display various types of data. The list view data browser in Figure 1-6 shows checkboxes, icons, text, dates, and relevance indicators, but a data browser can also display pop-up menus and progress indicators.

Figure 1-6 A list view data browser that displays a variety of data

Other options available in a data browser include:

- Displaying items as active or inactive
- Providing a placard, similar to that shown in the lower left of the window in Figure 1-6
- Using items that are aliases

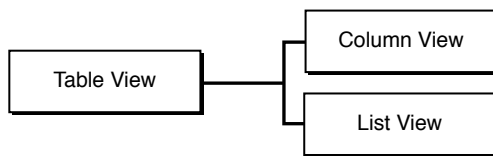
How the Data Browser Works

This section describes how the data browser works and provides information you need to know before you begin to write any code. You'll find out:

- How list and column views are implemented and the impact that has on the functions available in the API
- How the data browser refers to data, including the assumptions it makes about the data you want to display
- The callbacks you can provide and what the most important callbacks—item-data and item-notification—handle
- How a data browser gets populated with data
- What types of data can be displayed by the data browser

How Views are Implemented

The data browser implementation is object oriented. Its views—list and column—are derived from the table view class, as shown in Figure 1-7. Although there isn't a user interface for table view, the data browser API defines a number of table-view functions, most of which can be used with list view and some of which can be used with column view.

Figure 1-7 Column and list views are subclasses of table view

The API contains sets of functions that format and set attributes for the data browser and other sets of functions that get, set, or operate on the displayed data. Although the formatting can be set up programmatically, it's easier to set up the initial look and behavior of the data browser graphically using Interface Builder.

Referring to Data

An **item** in a data browser refers to the data displayed at a particular row and column intersection. Two values are associated with each item—an item ID and a property ID.

Keep in mind that the data browser displays data; it does not store data. It is the responsibility of your application to store data and to supply data to the data browser when requested. Your application stores all data and sets up item IDs to refer to your stored data.

An **item ID** is a unique 32-bit value that your application uses to refer to data. When you ask the data browser to display one or more items in a data browser, you provide an item ID for each data item. You can store the actual data in memory, on disk, or across a network. The data browser passes item IDs to the appropriate callback to get information about properties of the data items and to signal your application to provide the data associated with an item ID. Item IDs must be greater than 0. (Zero is reserved for use as the data browser constant `kDataBrowserNoItem`.) Item IDs can be pointer values, data file offsets, and 32-bit TCP/IP host addresses.

Item IDs are position independent—they specify data, not its specific position, in the data browser. The constant `kDataBrowserNoItem` is an API-defined item ID used by the data browser to mean none of the item IDs currently stored in the data browser.

Item IDs have slightly different interpretations in column view and in list view. In list view, an item ID refers to a row of data. In column view, an item ID refers to one entry in a column. In list view, to refer to an entry in a cell (a column, row intersection), you must supply the item ID as well as a property ID.

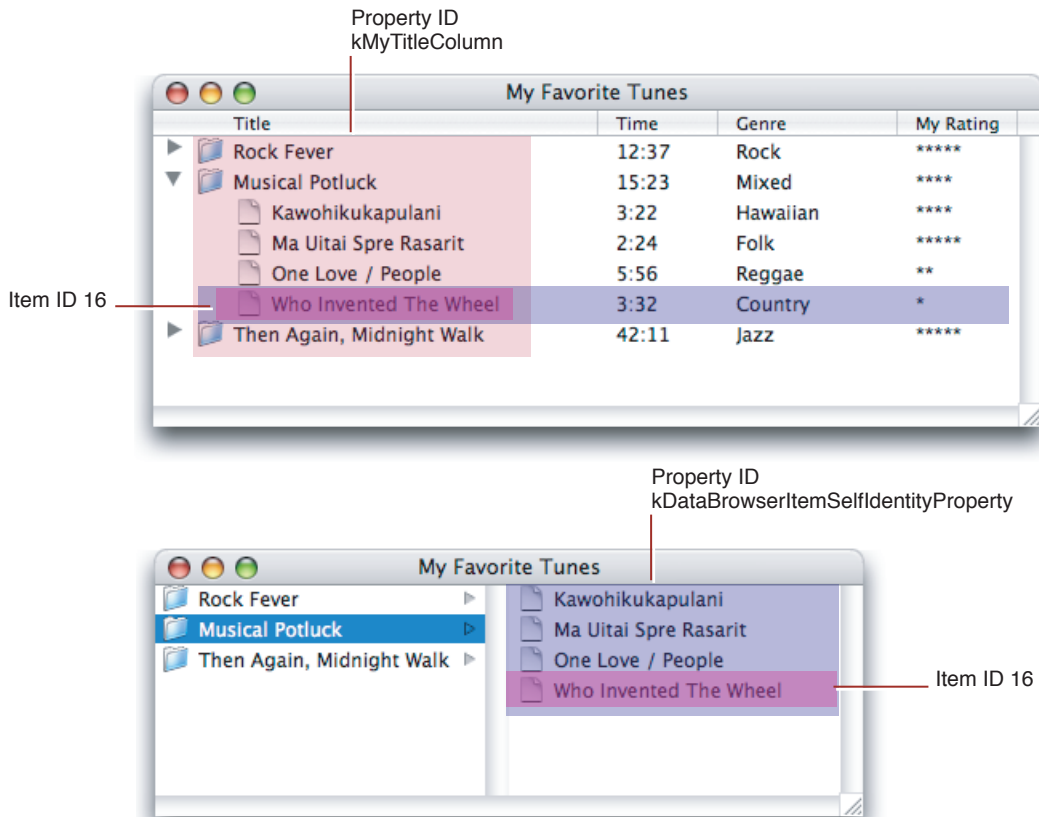
A **property ID** is a nonzero, 32-bit unsigned integer value that uniquely identifies a list view column. Property IDs do not need to be ordered or sequential, but they cannot be values 0 through 1023 because those values are reserved by Apple. A property ID is typically defined as a four-character sequence. For example, a column that displays dates could be assigned the property ID `DATE`.

Columns in column view don't use application-defined property IDs. Instead, they have the predefined property `kDataBrowserItemSelfIdentityProperty`.

[Figure 1-8](#) (page 14) shows item IDs and property IDs in list and column view. Take a look at the song “Who Invented The Wheel.” List view displays the song title and its three attributes—time (length of song), genre, and a listener-assigned rating. Every attribute in this row has the same item ID—the value 16. This value is assigned by the application. The song title is an item that is uniquely identified by its item ID (16) and its property ID (`kMyTitleColumn`). The genre data for the song “Who Invented The Wheel” is uniquely identified by its item ID (16) and its property ID (`kMyGenreColumn`).

Data items in column view are uniquely identified by an item ID. In the case of columns, the property ID is always `kDataBrowserItemSelfIdentityProperty`, so it's really the item ID that is the distinguishing value for an item. Compare the song title “Who Invented The Wheel” in list and column views. In each case the song title has the same item ID. But for list view the property ID is `kMyTitleColumn` whereas for column view it is the generic `kDataBrowserItemSelfIdentityProperty`.

Figure 1-8 An item is data at a row, column intersection



Callbacks and the Data Browser

The data browser relies on callbacks provided by your application to take care of any tasks that relate to the displayed data, including populating the data browser with data. At a minimum, you must supply an **item-data callback** that responds to requests for data; this is the main path of communication between the data browser and your application. Without a callback to respond data requests, the data browser is not able to display and update your data. In all cases except for the simplest list-view data browser, you also need to supply an **item notification callback** to respond to messages sent to your application from the data browser. Messages are generated by the data browser in response to a changed condition in the browser. You'll see how to write an item-data (`DataBrowserItemDataProcPtr`) and item-notification (`DataBrowserItemNotificationWithItemProcPtr`) callbacks in “[Data Browser Tasks](#)” (page 21).

You can optionally set up callbacks to:

- Compare items for the purpose of sorting them
- Support drag and drop

- Provide a contextual menu in the data browser
- Provide help tags
- Control how custom items are drawn, tracked, and dragged. See [“What Can Be Displayed”](#) (page 19) for more information on custom items.

You can find more information about using the optional callbacks in [“Data Browser Tasks”](#) (page 21) and in [Data Browser Reference](#). But right now let’s focus on the two most important callbacks you need to use—the item-data and item-notification callbacks.

The Item-Data Callback

The purpose of an item-data callback is to respond to data requests. The data browser sends an item ID and a property to the item-data callback. In response, your callback provides the data associated with the property. Properties are either the API-defined properties shown in [Table 1-1](#) (page 15) or a four-character sequence that you assign to represent a column in list view.

Your callback responds only to the properties that it makes sense to respond to for your application. [Table 1-1](#) (page 15) provides some guidance. For example, if the browser you create doesn’t display hierarchical data, then you don’t need to respond to the properties that support hierarchical data.

Table 1-1 Properties you can respond to in an item-data callback

| Properties | Respond to this . . . | Provide |
|--|--|------------------------|
| kDataBrowserItemIs-ActiveProperty | when items can be active or inactive | Boolean; default true |
| kDataBrowserItemIs-SelectableProperty | to support item selection | Boolean; default true |
| kDataBrowserItemIs-EditableProperty | to support editing items | Boolean; default false |
| kDataBrowserItemIs-ContainerProperty | to support display of hierarchical data | Boolean; default false |
| kDataBrowserItem-ParentContainerProperty | to support display of hierarchical data | item ID for parent |
| kDataBrowser-ContainerIsOpenableProperty | to report whether a container in a hierarchical list can be opened | Boolean; default true |
| kDataBrowser-ContainerIsClosableProperty | to report whether a container in a hierarchical list can be closed | Boolean; default true |
| kDataBrowser-ContainerIsSortableProperty | to report whether a container in a hierarchical list can be sorted | Boolean; default true |
| kDataBrowserItemSelf-IdentityProperty | to display data in column view | data to display |

| Properties | Respond to this . . . | Provide |
|--|--|--------------------------------|
| <code>kDataBrowserContainerAlias-IDProperty</code> | to report whether a container is an alias or symbolic link to some other container | <code>DataBrowserItemID</code> |

You respond to an application-defined property by providing the data displayed for that item. Recall that application-defined properties are relevant only to list view, because you use them to distinguish one column in list view from another. For example, in [Figure 1-8](#) (page 14), the song “Who Invented the Wheel” has four application-defined properties—each associated with a column, Title, Time, Genre, and MyRating. You define a four character sequence for each property—such as, 'NAME', 'TIME', 'GENR', and 'RATE'. When the data browser passes your item-data callback an item ID and one of these four properties, you supply the data to display in the column. For example, for the data shown in [Figure 1-8](#) (page 14), if your item-data callback receives an item ID equal to 16 and the `GENR` property, you'd supply `Country` as the data.

The Item-Notification Callback

The purpose of an item-notification callback is to respond to messages, as appropriate. The data browser sends an item ID and a message to your item-notification callback. The messages, shown in [Table 1-2](#) (page 16), inform your application of changes to the state of your browser. You take action when it makes sense to do so for your application. For example, when a container is opened, you add items to the display by calling the function `AddDataBrowserItems`.

Table 1-2 Messages sent by the data browser to the item-notification callback

| Notification | Sent when . . . |
|---|--|
| <code>kDataBrowserItemAdded</code> | An item is added to the data browser |
| <code>kDataBrowserItemRemoved</code> | An item is removed from the data browser |
| <code>kDataBrowserEditStarted</code> | A text editing session is started for an item |
| <code>kDataBrowserEditStopped</code> | A text editing session is stopped for an item |
| <code>kDataBrowserItemSelected</code> | An item is added to the selection set |
| <code>kDataBrowserItemDeselected</code> | An item is removed from the selection set |
| <code>kDataBrowserDoubleClicked</code> | The user double clicks an item |
| <code>kDataBrowserContainerOpened</code> | A container is opened |
| <code>kDataBrowserContainerClosing</code> | A container is in the process of closing; at the start of the close operation. |
| <code>kDataBrowserContainerClosed</code> | A container is closed, after the close operation. |
| <code>kDataBrowserContainerSorting</code> | A container is about to be sorted. |
| <code>kDataBrowserContainerSorted</code> | A container is sorted. |
| <code>kDataBrowserUserStateChanged</code> | The user reformats the view for the target. For example, the user changes a sorting order or a column width. |

| Notification | Sent when . . . |
|---|--|
| <code>kDataBrowserSelectionSetChanged</code> | The selection set is modified. |
| <code>kDataBrowserTargetChanged</code> | The target changes to the specified item. |
| <code>kDataBrowserUserToggledContainer</code> | The user toggles a container (opened or closed) by clicking a disclosure triangle. This does not get sent if the container is programmatically opened or closed through one of the routines in the Data Browser API. |

How a Data Browser Gets Populated With Data

To populate the data browser with data, your application can call these functions: `AddDataBrowserItems` and `SetDataBrowserTarget`. In response to each of these functions, the data browser invokes your item-data and item-notification callbacks, sending the callbacks a variety of data requests or messages. In this section, you'll take a behind-the-scenes look at the requests and messages the data browser sends for these three scenarios:

- Calling the function `AddDataBrowserItems` to populate a simple list view data browser
- Calling the function `AddDataBrowserItems` to populate a hierarchical list view data browser
- Calling the function `SetDataBrowserTarget` to populate a column view data browser

Adding Items to a Simple List View

You add items to a simple list of nonhierarchical data, similar to that shown in [Figure 1-1](#) (page 9) by calling the function `AddDataBrowserItems`. When you implement a simple list you need only to supply an item-data callback. You don't need an item-notification callback. You'll see exactly how to write an item-data callback in ["Displaying Data in a Simple List View"](#) (page 22).

After you call the function `AddDataBrowserItems`, the data browser invokes your item-data callback to find out whether the items to be added are active, and then to obtain the data to display for each item. As you add each item, the data browser:

1. Sends a request for the active property for the item. The data browser passes you an item ID and the API-defined property `kDataBrowserItemIsActiveProperty`. If data in the list can be active or inactive, you can respond to this request by setting the active state of the item using the function `SetDataBrowserItemDataBooleanValue`. If items are always active, you don't need to respond to the request.
2. Sends a request for data you want displayed in each column of the list. The data browser passes you an item ID and an application-defined property that specifies the column. You must respond to this request by calling the appropriate function for setting the data.

To populate a simple list with the six rows of data shown in [Figure 1-1](#) (page 9), the data browser would invoke your item-data callback to fetch data as needed. Each time the data browser needs data, it sends a request for the active property followed by a request for data.

Adding Items to a Hierarchical List View

To display hierarchical data, you must provide both an item-data callback and an item-notification callback. The data browser populates a list of hierarchical data similar to the way it populates a simple list, but there is a message and an additional data request sent to your callbacks. See [Figure 1-2](#) (page 10) for an example of such a list. When you call the function `AddDataBrowserItems`, the data browser:

1. Sends a message for each item ID provided to the list, to the item-notification callback informing you the item is added (`kDataBrowserItemAdded`).
2. Sends a request for the active property for the item. The data browser passes you an item ID and the API-defined property `kDataBrowserItemIsActiveProperty`. If data in the list can be active or inactive, you can respond to this request by setting the active state of the item using the function `SetDataBrowserItemDataBooleanValue`. If items are always active, you don't need to respond to the request.
3. Sends a request for data you want displayed for that item. The data browser passes you an item ID and an application-defined property that specifies the column. You must respond to this request by calling the appropriate function for setting the data.
4. The data browser sends a request to check whether the item is a container (`kDataBrowserItemIsContainerProperty`)
5. Repeats steps 2 through 4 for each application-defined property of each item ID.

Setting a Target for a Column View

You can populate a column view data browser, similar to that shown in [Figure 1-5](#) (page 11), by calling the function `SetDataBrowserTarget`. You can set the target to any item in the hierarchy to have the data browser open with that item selected. When you call the function `SetDataBrowserTarget`, the data browser:

1. Sends a request to your item-data callback for the container property for the target item (`kDataBrowserItemIsContainerProperty`).
2. If the target item is a container, sends a container-opened message (`kDataBrowserItemContainerOpened`) to your item-notification callback. You respond by calling the function `AddDataBrowserItems`, supplying the items that are in the container. This function call sets off the requests and messages described previously in [“Adding Items to a Simple List View”](#) (page 17) for the function `AddDataBrowserItems`.

If the target item is not a container, sends a request to your item-data callback to get the parent of the item (`kDataBrowserItemParentContainerProperty`).

The next set of steps describe what happens for a target item after you supply the item ID of the target's parent:

1. Sends a container-opened message (`kDataBrowserContainerOpened`) to your item-notification callback. Your callback responds by supplying the items in the container by calling the function `AddDataBrowserItems`.

2. After all items in the target's parent container are added, the data browser sends a request for the `is-sortable-property` (`kDataBrowserItemIsSortableProperty`) to your `item-data` callback. If you don't respond to this request, the data browser assumes the items in the container are sortable. If the container is sortable, the data browser invokes your `item-compare` callback if you supply one (see [“Sorting Items”](#) (page 48)). Otherwise, it uses its built-in sorting routine.
3. If it is sortable, sends a `container-sorting` message (`kDataBrowserItemContainerSorting`) followed by a `container-sorted` message (`kDataBrowserItemContainerSorted`) to your `item-notification` callback.
4. Sends a `target-changed` message (`kDataBrowserTargetChanged`) to your `item-notification` callback.

What Can Be Displayed

The data browser API defines several display types that your application uses to designate the kind of data shown in a column. For these predefined types, the data browser automatically handles a number of tasks, such as drawing the item. If you prefer to draw the items and provide custom behavior, you can use a custom type.

Each display type is listed below. Only icon and text can be used for a column view data browser. All display types can be used for a list view data browser.

- **Text.** Columns with the display type `kDataBrowserTextType` display text in list view. For an example of a column that displays text, see [Figure 1-3](#) (page 10).
- **Icon and text.** Columns with the display type `kDataBrowserIconAndTextType` display an icon with text next to it in list or column view. Your application provides the icon to draw in a cell and the text to draw next to the icon. For an example of a column that displays icon and text, see [Figure 1-5](#) (page 11).
- **Icon.** Columns with the display type `kDataBrowserIconType` display icons in list view. Your application provides the icon to draw in a cell. You can also provide other drawing information such as color and an icon transformation type.
- **Date and time.** Columns with the display type `kDataBrowserDateTimeType` can display values, in list view, returned by the date and time utilities routines. Time can be displayed as a relative time value (Today, Yesterday, and so forth) or absolute. For an example of a column that displays time, see [Figure 1-6](#) (page 12).
- **Checkboxes.** Columns created with the display type `kDataBrowserCheckboxType` display checkboxes in list view. Checkboxes can be shown in three states—on, off, or mixed. To allow users to change the checkbox setting, your application sets the flag `kDataBrowserPropertyIsEditable`. For an example of a column that displays checkboxes, see [Figure 1-6](#) (page 12).
- **Progress indicators.** Columns with the display type `kDataBrowserProgressBarType` can show progress indicators in list view. The appearance of a progress indicator is defined by minimum, maximum, and current values.
- **Relevance indicators.** Columns with the display type `kDataBrowserRelevanceRankType` display relevance indicators in list view. The appearance of a relevance indicator is defined by minimum, maximum, and current values. For an example of a column that displays relevance indicators, see [Figure 1-6](#) (page 12).
- **Pop-up menus.** Columns with the display type `kDataBrowserPopupMenuType` display pop-up menus in list view. Your application provides the menu and sets the value of the menu item to display. To allow a user to choose menu items, your application sets the flag `kDataBrowserPropertyIsEditable`.

- **Custom.** Columns with the display type `kDataBrowserCustomType` display data, in list view, for which your application provides the routines needed to display the data and support custom behavior for tasks such as dragging or tracking.

Data Browser Tasks

This chapter provides instructions and sample code on how to implement a data browser in an application. It provides general tips for using the data browser API and shows how to create list and column view data browsers. You'll also see how to switch between the two views.

The chapter includes these sections:

- [“Tips for Using the Data Browser API”](#) (page 21)
- [“Displaying Data in a Simple List View”](#) (page 22)
- [“Displaying Hierarchical Data in a List View”](#) (page 30)
- [“Displaying Data in a Column View”](#) (page 33)
- [“Switching Between List View and Column View”](#) (page 39)
- [“Supporting Text Editing”](#) (page 47)
- [“Applying an Operation to Each Item”](#) (page 48)
- [“Sorting Items”](#) (page 48)
- [“Supporting Drag and Drop”](#) (page 48)
- [“Providing a Contextual Menu”](#) (page 49)
- [“Providing Help Tags”](#) (page 49)
- [“Customizing Drawing, Tracking, and Dragging”](#) (page 50)

Tips for Using the Data Browser API

For best results, follow these tips when using a data browser in your application:

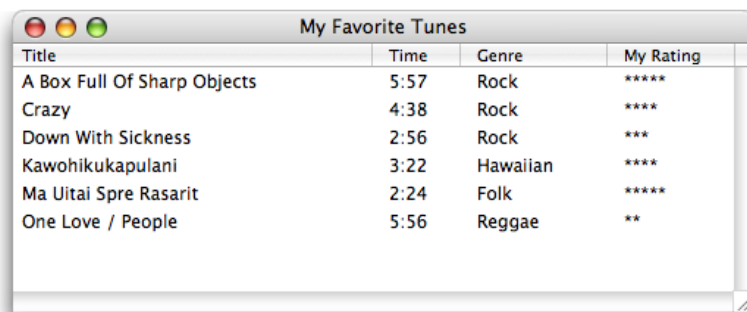
- Adhere to the guidelines discussed in *Apple Human Interface Guidelines*. Using the data browser API doesn't ensure you'll have a stunning interface. Only you can do that by following the human interface guidelines.
- Text editing works only in list view; don't plan to make text editing available in column view.
- Avoid using pop-up menus and sliders. The implementation is such that these items can look crowded in a data browser. In addition, the rectangle returned by the function `GetDataBrowserItemPartBounds` for these items doesn't provide the values you need to make the interface align properly.

Displaying Data in a Simple List View

This section shows how to create the list view data browser shown in Figure 2-1 and to display data in the list. This data browser has the following characteristics:

- It is a list view.
- It uses minimal highlighting, does not show the focus ring, has variable column width, can show scroll bars, and has drag select enabled.
- It uses a header row and has four columns.
- Each column holds text information and can be moved, sorted, and selected.

Figure 2-1 A simple list view



| Title | Time | Genre | My Rating |
|-----------------------------|------|----------|-----------|
| A Box Full Of Sharp Objects | 5:57 | Rock | ***** |
| Crazy | 4:38 | Rock | **** |
| Down With Sickness | 2:56 | Rock | *** |
| Kawohikukapulani | 3:22 | Hawaiian | **** |
| Ma Uitai Spre Rasarit | 2:24 | Folk | ***** |
| One Love / People | 5:56 | Reggae | ** |

To create this list, you'll perform these tasks, each of which is described in the sections that follow:

1. Design and configure a list view data browser. In Mac OS X, the easiest way to accomplish this task is to use Interface Builder.
2. Declare the necessary global constants in your Carbon application.
3. Write an item-data callback that populates the list with data.
4. Write code that initializes the data browser control.

Design and Configure a List View Data Browser

Follow these steps to design and configure the list view data browser shown in Figure 2-1 (page 22):

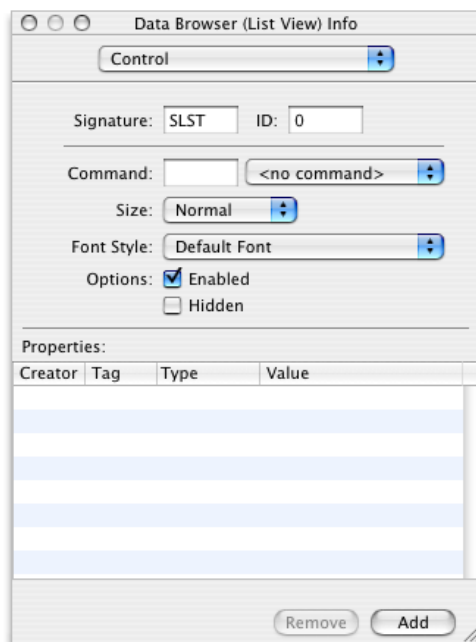
1. Create a new Carbon application in Xcode.
2. Double-click the `main.nib` file provided in the new project in the NIB Files folder.

You access Interface Builder from within Xcode to ensure that the nib file you create is properly bundled with the Xcode project.

3. Click the empty window (titled Window), and choose Tools > Show Info.

4. In the Size pane of the Info window, set the width and height to 500 by 180.
5. Choose Tools > Palettes > Show Palettes.
6. Click the Carbon Browsers & Tab palette button.
7. Drag the Data Browser (List View) item to the window.
8. Drag the data browser control to resize it to fit in the window.
9. With the data browser control selected, make the Info window active.
10. In the Control pane, enter a signature and make sure the Enabled option is selected, as shown in Figure 2-2. You can enter any signature you'd like, as long as there is at least one uppercase letter. Apple reserves signatures that are all lowercase. For this example, enter SLST.

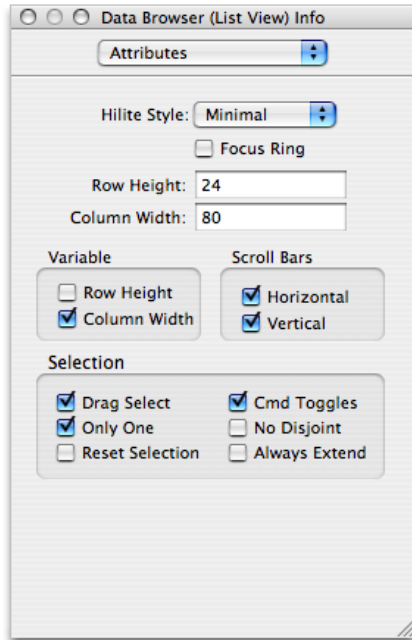
Figure 2-2 Setting up the Control pane for a list view



11. In the Attributes pane, make the following settings, as shown in Figure 2-3:
 - Set Hilite Style to Minimal.
 - Make sure Focus Ring is not selected.
 - Under Variable, select the Column Width option.
 - Under Scroll Bars, select Horizontal and Vertical.
 - Under Selection, make sure Drag Select is selected. If you want to restrict the user to selecting only one item at a time, select Only One. You can leave the other items at the default setting.

- Don't make any changes to the default row height and column width.

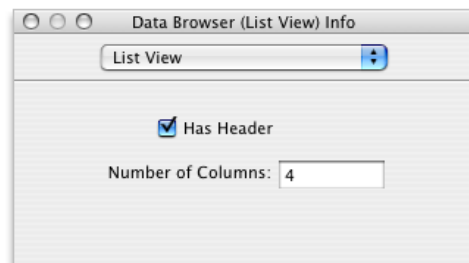
Figure 2-3 Setting up the Attributes pane for a list view



12. In the List View pane, make these settings, as shown in Figure 2-4:

- Make sure Has Header is selected. A header is a title row.
- Set the number of columns to 4.

Figure 2-4 Setting up a header and number of columns for a list view

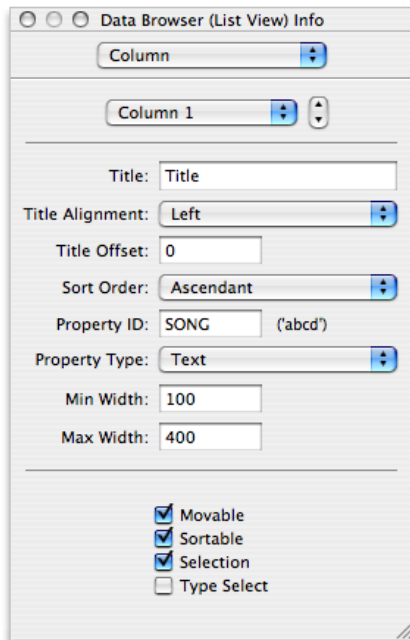


13. In the Columns pane, shown in Figure 2-5, set the following for column 1:

- Title: Title
- Title Alignment: Left
- Title Offset: 0
- Sort Order: Ascendant

- Property ID: SONG
- Property Type: Text
- Min Width: 100
- Max Width: 400
- Make sure the column is movable, sortable, and can be selected.

Figure 2-5 Setting up the Columns pane for a list view



14. In the Columns pane for column 2:

- Title: Time
- Title Alignment: Left
- Title Offset: 0
- Sort Order: Ascendant
- Property ID: TIME
- Property Type: Text
- Min Width: 20
- Max Width: 100
- Make sure the column is movable, sortable, and can be selected.

15. In the Columns pane for column 3:

- Title: Genre

- Title Alignment: Left
- Title Offset: 0
- Sort Order: Ascendant
- Property ID: GENR
- Property Type: Text
- Min Width: 50
- Max Width: 150
- Make sure the column is movable, sortable, and can be selected.

16. In the Columns pane for column 4:

- Title: MyRating
- Title Alignment: Left
- Title Offset: 0
- Sort Order: Ascendant
- Property ID: RATE
- Property Type: Text
- Min Width: 75
- Max Width: 100
- Make sure the column is movable, sortable, and can be selected.

17. Save the file and quit Interface Builder.

When you create a list view data browser for your own application, you'd set the options to support the number of columns and functionality appropriate for your application.

Declare the Necessary Global Constants

In the previous section, you assigned each of the columns in list view a unique property ID. You have to make sure the code in your Carbon application uses the same property IDs that you assigned in Interface Builder because you refer to the columns from within your application using the property ID. Rather than use the four-character property IDs directly, it's useful to declare an enumeration of constants set to the property IDs. You'll use these constants later in an item-data callback to determine which column requires updating.

In Xcode, open the `main.c` file and add the following global enumeration to the code:

```
enum {
    kMyTitleColumn = 'SONG',
    kMyTimeColumn = 'TIME',
    kMyGenreColumn = 'GENR',
    kMyRatingColumn = 'RATE'
}
```

Write an Item-Data Callback

The item-data callback communicates data between the data browser and your application; it adds and modifies data in the browser. Your application must provide this callback because without it, no data is displayed. The data browser invokes the callback when it needs to display a value for an item. Typically, the callback is invoked under the following conditions:

- The user interacts with the data browser by clicking a disclosure triangle to open a container, scrolling, editing text, or performing some other action that changes the display.
- Your application calls a function that requires the data browser to add or modify data. For example, the functions `AddDataBrowserItems`, `SetDataBrowserTarget`, and `SetDataBrowserColumnViewPath` cause the data browser to invoke your item-data callback.

An item-data callback consists of a switch statement that responds to application-defined properties that you define in your application as well as to the list of properties shown in [Table 1-1](#) (page 15). For each case in the switch statement, you call the appropriate function from the `SetDataBrowserItemData` and `GetDataBrowserItemData` set of functions available in the data browser API. (See *Data Browser Reference* for the complete list of “getting and setting” functions.)

The data browser shown in [Figure 2-1](#) (page 22) is a simple list view. The data is not hierarchical, there are no symbolic links, it uses list view, editing is not allowed, and the user cannot change the active or selection states. For this example, an item-data callback simply needs to look for the property ID for a given item ID, and provide the data that’s associated with that item ID, property ID pair by calling the function `SetDataBrowserItemDataText`.

There are four columns of data, so the switch statement in the item-data callback must respond to the property ID assigned to each column. Listing 2-1 shows the item-data callback needed to populate the list view data browser shown in [Figure 2-1](#) (page 22). An explanation for each numbered line of code appears following the listing.

Listing 2-1 An item-data callback that populates a simple list view

```
OSStatus MyDataBrowserItemDataCallback (ControlRef browser,
    DataBrowserItemID itemID,
    DataBrowserPropertyID property,
    DataBrowserItemDataRef itemData,
    Boolean changeValue)
{
    OSStatus status = noErr;

    if (!changeValue) switch (property) // 1
    {
        case kMyTitleColumn:
            status = SetDataBrowserItemDataText (itemData,
                myTunesDatabase[itemID].song); // 2
            break;
        case kMyTimeColumn:
            status = SetDataBrowserItemDataText(itemData,
                myTunesDatabase[itemID].time);
            break;
        case kMyGenreColumn:
            status = SetDataBrowserItemDataText(itemData,
                myTunesDatabase[itemID].genre);
            break;
    }
}
```

```

        case kMyRatingColumn:
            status = SetDataBrowserItemDataText(itemData,
                myTunesDatabase[itemID].rating);
            break;
        default:
            status = errDataBrowserPropertyNotSupported;
            break;
    }
    else status = errDataBrowserPropertyNotSupported; // 3

return status;
}

```

Here's what the code does:

1. Makes sure that this is a set request, and if so, that it examines the property. The `changeValue` parameter is `false` for a set request. The property is the same property ID you defined for each column in Interface Builder, and is the same property ID you declared global constants for in your code.
2. Calls the function `SetDataBrowserItemDataText` to provide the data to display for that item. In this example, all the data in the data browser, regardless of the property ID of the column, is text. Your application needs to use the item ID associated with the data to obtain the appropriate data. In this example, data from a database was previously read into the structure `myTunesDatabase`. When the data was read, the application assigned an item ID equal to the array location in the `myTunesDatabase` structure. For a more complex set of data, your application could provide a function that looks up the data location based on the item ID.
3. Sets `status` to a result code indicating that get requests are not supported for any property. Get requests need to be supported only if the data in a browser can be edited.

The item-data callback in this example is a simple one. The item-data callbacks in the sections “[Displaying Hierarchical Data in a List View](#)” (page 30), “[Displaying Data in a Column View](#)” (page 33), and “[Switching Between List View and Column View](#)” (page 39) show item-data callbacks that handle progressively more complex situations.

Write Code That Initializes the List View

There are a number of tasks that need to be done to initialize a data browser. You need to:

- Get the data browser control that you defined in Interface Builder so that you can then pass it to other functions.
- Initialize the data browser callbacks structure and fill it with callbacks appropriate for your application. In this case there is only one callback—an item-data callback.
- Load your data.
- Populate the data browser with data.

Listing 2-2 shows a function that initializes the list view data browser depicted in [Figure 2-1](#) (page 22). A detailed explanation for each numbered line of code appears following the listing.

Listing 2-2 Initializing a simple list view data browser

```

OSStatus MyInitializeDataBrowserControl (WindowRef window)
{
    const      ControlID  dbControlID  = { 'SLST', 0 };                // 1
    OSStatus   status = noErr;
    UInt32     i;
    SInt32     numRows;
    ControlRef dbControl;
    DataBrowserCallbacks dbCallbacks;

    GetControlByID (window, &dbControlID, &dbControl);                // 2
    dbCallbacks.version = kDataBrowserLatestCallbacks;                // 3
    InitDataBrowserCallbacks (&dbCallbacks);                          // 4
    dbCallbacks.u.v1.itemDataCallback =
        NewDataBrowserItemDataUPP((DataBrowserItemDataProcPtr)
            MyDataBrowserItemDataCallback);                            // 5
    SetDataBrowserCallbacks(dbControl, &dbCallbacks);                // 6
    SetAutomaticControlDragTrackingEnabledForWindow (window, true);  // 7

    numRows = MyLoadData ();                                          // 8
    status = AddDataBrowserItems (dbControl, kDataBrowserNoItem, numRows,
        NULL, kDataBrowserItemNoProperty );                            // 9
    return status;
}

```

Here's what the code does:

1. Initializes a control ID variable with the values you set up in Interface Builder for the data browser.
2. Obtains the data browser using the control ID from the previous step.
3. Sets the version of the data browser callbacks structure to the latest version.
4. Initializes a data browser callbacks structure, effectively setting all fields to `NULL` in preparation for assigning the callback provided by your application.
5. Creates a universal procedure pointer (UPP) to an item-data callback and assigns the callback to the appropriate field in the data browser callbacks structure.
6. Installs the callback by calling the function `SetDataBrowserCallbacks`, passing the data browser callbacks structure as a parameter.
7. Enables automatic drag tracking for the window that contains the data browser. This lets a user drag a column in list view to a new position.
8. Calls your application's function to read data from a database in preparation for writing it to the data browser.
9. Populates the data browser by calling the function `AddDataBrowserItems`. This function causes the data browser to invoke your item-data callback, which in turn provides the data browser with the data associated with each row.

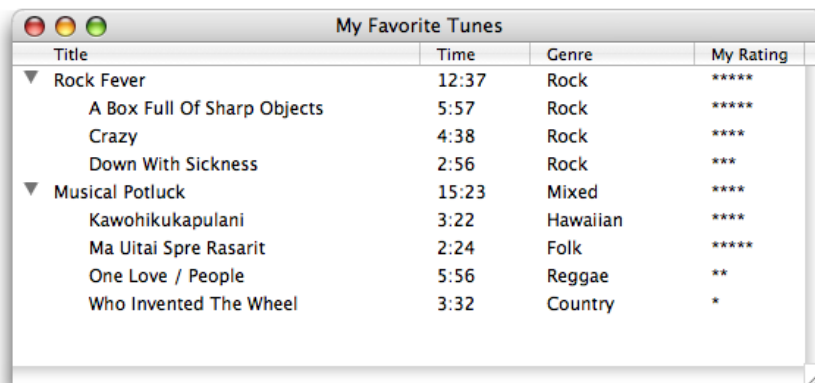
Note that the fourth parameter, the `items` parameter, is `NULL`. In all but the most simplest of lists, such as this one, this parameter should be a pointer to an array of item ID values for the items you want to add to the data browser. You supply item ID values based on your own identification scheme. Passing `NULL` causes the data browser to generate item ID values for you, starting at a value of 1. Calling the

function in this way clears whatever items are in the container. Because of this clearing behavior, passing `NULL` is not recommended unless your application uses a data browser to display a simple list that is populated only once with data. We pass `NULL` here just to point out the usage of the function. In other examples, we pass application-generated item IDs.

Displaying Hierarchical Data in a List View

This section shows how to create a data browser that uses list view to display hierarchical data. Figure 2-6 shows the list that's created in this section.

Figure 2-6 Hierarchical data displayed in a list view



| Title | Time | Genre | My Rating |
|-----------------------------|-------|----------|-----------|
| Rock Fever | 12:37 | Rock | ***** |
| A Box Full Of Sharp Objects | 5:57 | Rock | ***** |
| Crazy | 4:38 | Rock | **** |
| Down With Sickness | 2:56 | Rock | *** |
| Musical Potluck | 15:23 | Mixed | **** |
| Kawohikukapulani | 3:22 | Hawaiian | **** |
| Ma Uitai Spre Rasarit | 2:24 | Folk | ***** |
| One Love / People | 5:56 | Reggae | ** |
| Who Invented The Wheel | 3:32 | Country | * |

To create this list, you'll perform the following tasks:

1. Design and configure a data browser that uses list view. This example uses the same design and configuration options as that used for the simple list view discussed in the previous example. See [“Design and Configure a List View Data Browser”](#) (page 22) for instructions on performing this task.
2. Declare the necessary global constants. These are the same constants declared for the simple list view example discussed in the previous example. See [“Declare the Necessary Global Constants”](#) (page 26) for instructions on performing this task.
3. Write an item-data callback that writes data to the list and sets each item in the list to the appropriate value for the property `kDataBrowserItemIsContainerProperty`. See [“Write an Item-Data Callback”](#) (page 8).
4. Write an item-notification callback that responds to a container-opened notification. See [“Write an Item-Notification Callback for Hierarchical Data”](#) (page 32).
5. Write code that initializes the data browser control. See [“Write Code That Initializes the Data Browser”](#) (page 33).

Tasks 3 through 5 are discussed in the sections that follow.

Write an Item-Data Callback

The item-data callback needed for a list view data browser that displays hierarchical data is almost identical to the callback supplied to display nonhierarchical data. There is one difference. In addition to responding to the property IDs of each column, the callback must respond to the API-defined property `kDataBrowserItemIsContainerProperty`. Your callback checks to see if an item is a container, and if it is, you set the container property to `true`. Otherwise, set the property to `false`. The code in Listing 2-3 shows how to do this. A detailed explanation for each numbered line of code appears following the listing.

Listing 2-3 An item-data callback for a hierarchical list

```
OSStatus MyDataBrowserItemDataCallback (ControlRef browser,
    DataBrowserItemID itemID,
    DataBrowserPropertyID property,
    DataBrowserItemDataRef itemData,
    Boolean changeValue)
{
    OSStatus status = noErr;

    if (!changeValue) switch (property) // 1
    {
        case kMyTitleColumn:
            status = SetDataBrowserItemDataText (itemData,
                myTunesDatabase[itemID].song);
            break;
        case kMyTimeColumn:
            status = SetDataBrowserItemDataText(itemData,
                myTunesDatabase[itemID].time);
            break;
        case kMyGenreColumn:
            status = SetDataBrowserItemDataText(itemData,
                myTunesDatabase[itemID].genre);
            break;
        case kMyRatingColumn:
            status = SetDataBrowserItemDataText(itemData,
                myTunesDatabase[itemID].rating);
            break;
        case kDataBrowserItemIsContainerProperty: // 2
            status = SetDataBrowserItemDataBooleanValue (itemData, // 3
                MyCheckIfContainer (itemID));
            break;
    }
    else status = errDataBrowserPropertyNotSupported; // 4

    return status;
}
```

Here's what the code does:

1. Makes sure this is a set request, and if so, examines the property. The `changeValue` parameter is `false` for a set request. In this case, the property ID can be one of the unique four-character sequences you assigned to the columns in list view or it can be one of the predefined properties from the data browser API.
2. Checks for the container property.

3. Sets the container property for the item to `true` if the item is a container or `false` if the item is not a container. Your application determines whether the item is a container or not.
4. Sets `status` to a result code indicating that get requests are not supported for any property. Get requests need to be supported only if the data in a browser can be edited.

Write an Item-Notification Callback for Hierarchical Data

The item-notification callback informs your application of actions taken by the user, such as initiating an editing session or opening a container. Your application must provide this callback if it displays hierarchical data in a list view or uses column view. Without it, users cannot open containers in list view or navigate the data hierarchy in column view.

The data browser communicates changes of its internal state by sending messages to your item-notification callback. Your callback responds to each event notification by taking the appropriate action. In the case of hierarchical data, your application responds to a container-opened notification by populating the data browser with the items that are in the container. [Table 1-2](#) (page 16) shows the message that can be sent to an item-notification callback. What you choose to handle in your item-notification callback depends on the nature of your application.

To implement the hierarchical list shown in [Figure 2-6](#) (page 30), you need to respond only to the container opened (`kDataBrowserContainerOpened`) notification. The data browser handles closing automatically, so you don't need to handle a container closed notification. [Listing 2-4](#) shows the code necessary to handle an open container event for the list view data browser shown in [Figure 2-6](#) (page 30). A detailed explanation for each numbered line of code appears following the listing.

Listing 2-4 An item-notification callback for a hierarchical list

```
void MyDataBrowserItemNotificationCallback( ControlRef browser,
                                           DataBrowserItemID itemID,
                                           DataBrowserItemNotification message)
{
    switch (message)
    {
        case kDataBrowserContainerOpened: // 1
        {
            int i, myItemsPerContainer;

            myItemsPerContainer = myTunesDatabase[itemID].songsInAlbum; // 2

            DataBrowserItemID myItems [myItemsPerContainer];
            for ( i = 0; i < myItemsPerContainer; i++) // 3
                myItems[i] = MyGetChild (itemID, i);
            AddDataBrowserItems (browser, itemID,
                                myItemsPerContainer,
                                myItems, kTitleColumn); // 4

            break;
        }
    }
}
```

Here's what the code does:

1. Checks for a container-opened notification. For this simple example, that's all this item-notification callback needs to check for. The data browser API defines a number of other item notifications that can be used within this kind of callback.
2. Retrieves the number of items in this container. You pass this value to the function `AddDataBrowserItems`.
3. Builds an array of the item IDs for the items in the container. This example calls an application-defined function (`MyGetChild`) that looks up item ID values for items in a container. Your application would need to build an array of item IDs appropriately.
4. Calls the function `AddDataBrowserItems` to add the items to the container. You supply the item ID of the container, the number of items in the container, an array of the item IDs for the items in the container, and the property ID of the column whose sorting order matches the sorting order of the items array. In this case, passes the property ID of the title column. You can pass `kDataBrowserItemNoProperty` if the items array is not sorted or if you don't know the sorting order of your data. You'll get the best performance from this function if you provide a sorting order.

After calling `AddDataBrowserItems`, the data browser invokes your item-data callback to obtain the data to display. For more information on the requests sent to your item-data callback see [“Adding Items to a Simple List View”](#) (page 17).

Write Code That Initializes the Data Browser

The initialization function needed for a list view data browser that displays hierarchical data is almost the same as that needed for nonhierarchical data. There are two differences:

- You need to assign your item-notification callback UPP to the appropriate field in the data browser callbacks structure.
- You must add code to specify which column can display a disclosure triangle next to a container items.

The code you need to add is shown in Listing 2-5. Just add this code to the function shown in [“Write Code That Initializes the List View”](#) (page 28).

Listing 2-5 Initializing a hierarchical list view data browser

```
dbCallbacks.u.v1.itemNotificationCallback =
    NewDataBrowserItemNotificationUPP(
        (DataBrowserItemNotificationProcPtr)
        MyDataBrowserItemNotificationCallback);

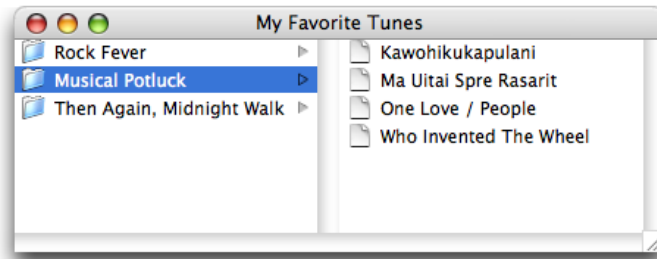
SetDataBrowserListViewDisclosureColumn (dbControl,
    kMyTitleColumn,
    true);
```

Displaying Data in a Column View

This section shows how to create a data browser that uses column view to display data. Figure 2-7 shows the data browser that's created in this section. There are two notable features about this data browser:

- Each column displays icon and text data. You'll need to provide the icons to display and the text strings.
- The columns are of equal width. A column view data browser can't have columns of unequal widths.

Figure 2-7 A column view data browser



To create this list, you'll perform the following tasks:

1. Design and configure a column view data browser.
2. Write an item-data callback that populates the columns.
3. Write an item-notification callback that handles container notifications.
4. Write code that initializes the data browser control.

Design and Configure a Column View Data Browser

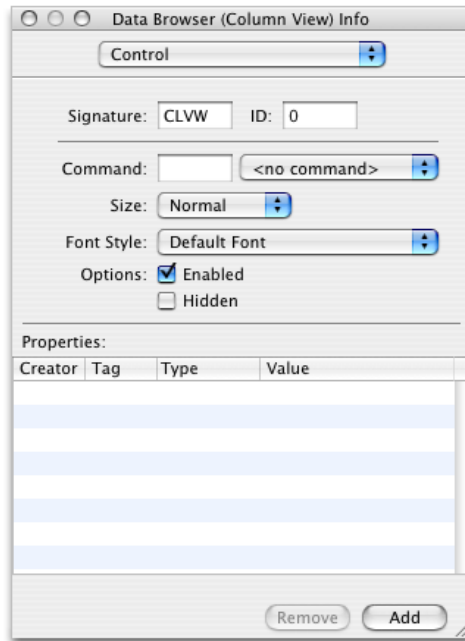
Follow these steps to design and configure the column view data browser shown in [Figure 2-7](#) (page 34):

1. Create a new Carbon application in Xcode.
2. Double-click the `main.nib` file provided in the new project.

You access Interface Builder from within Xcode to ensure that the nib file you create is properly bundled with the Xcode project.
3. Click the window, and choose Tools > Show Info.
4. Choose Size in the Info window and set the width and height to 250 by 300.
5. Choose Tools > Palettes > Show Palettes.
6. Click the Carbon Browsers & Tab palette button.
7. Drag the Data Browser (Column View) item to the window.
8. Drag the data browser control to resize it to fit in the window.
9. With the data browser control selected, make the Info window active.

10. In the Control pane, enter a signature, and make sure the Enabled option is selected, as shown in Figure 2-8. You can enter any signature you like, as long as there is at least one uppercase letter. Apple reserves signatures that are all lowercase. For this example, enter `CLVW`.

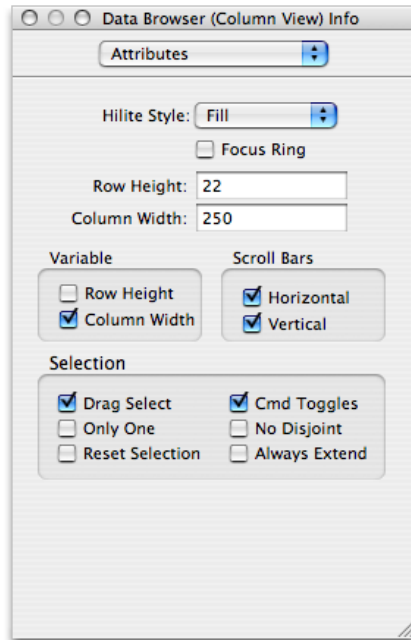
Figure 2-8 Setting up the Control pane for a column view



11. In the Attributes pane, make the following settings, as shown in Figure 2-9:
- Set Hilite Style to Fill.
 - Make sure Row Height is set to 22 (the default) and the Column Width is set to 250.
 - Make sure Focus Ring is not selected.
 - Select Column Width to make it variable.
 - Select Horizontal and Vertical under Scroll Bars.

- Under Selection, do not change the default selections.

Figure 2-9 Setting up the Attributes pane for a column view



12. Save the file and quit Interface Builder.

Write an Item-Data Callback

A column view data browser is populated with data the same way a list view data browser is—through an item-data callback provided by your application. Unlike list view columns, columns in column view do not have property IDs assigned by your application. Instead, every column in column view has the API-defined property `kDataBrowserItemSelfIdentityProperty`. This is one of the properties to which your item-data callback responds. The two other API-defined properties handled by an item-data callback for a column view are `kDataBrowserItemIsContainerProperty` and `kDataBrowserItemParentContainerProperty`. Column view is, by definition, a way to view hierarchically organized data. As such, your item-data callback provides item IDs for the items within a container and for the parent of an item if the item has a parent.

Figure 2-6 shows the item-data callback needed to populate the column view data browser shown in [Figure 2-7](#) (page 34). A detailed explanation for each numbered line of code appears following the listing.

Listing 2-6 An item-data callback for a column view data browser

```
OSStatus MyDataBrowserItemDataCallback (ControlRef browser,
    DataBrowserItemID itemID,
    DataBrowserPropertyID property,
    DataBrowserItemDataRef itemData,
    Boolean changeValue)
{
    OSStatus status = noErr;
```

```

switch (property)
{
    case kDataBrowserItemSelfIdentityProperty: // 1
    {
        status = SetDataBrowserItemDataIcon (itemData,
            MyCheckIfContainer (itemID) ?
            icon[kFolder] : icon[kDocument]); // 2
        status = SetDataBrowserItemDataText (itemData,
            myTunesDatabase[itemID].song); // 3
        break;
    }
    case kDataBrowserItemIsContainerProperty: // 4
    {
        status = SetDataBrowserItemDataBooleanValue (itemData,
            MyCheckIfContainer (itemID));
        break;
    }
    case kDataBrowserItemParentContainerProperty: // 5
    {
        DataBrowserItemID myParent;
        myParent = MyGetParent (itemID); // 6
        status = SetDataBrowserItemDataItemID (itemData, myParent); // 7
        break;
    }
}
return status;
}

```

Here's what the code does:

1. Checks for the self-identity property. This property is relevant only to column view.
2. Sets the icon for the item. There are two types of icons used in this example—a folder, used for a container, and a document, used for an item. The code checks to see if the item is a container, and then supplies the appropriate icon.
3. Sets the text data for the item.
4. Checks for the container property. It sets the property to `true` if the item is a container or `false` if the item is not a container. Your application determines whether the item is a container or not.
5. Checks for the parent container property.
6. Calls your routine to look up the item ID for the item's parent. If the item is at the top of the hierarchy, it has no parent, and the returned value should be 0.
7. Sets the item's parent ID by calling the function `SetDataBrowserItemDataItemID`. If the item has a parent, the data browser calls your item-data callback again with the parent ID. It keeps doing this until the top of the hierarchy is reached.

Write an Item-Notification Callback

The item-notification callback needed for this column-view data browser is identical to that supplied for the list view data browser that displays hierarchical data. The notification you must respond to is `kDataBrowserContainerOpened`. See [“Write an Item-Notification Callback for Hierarchical Data”](#) (page 32) for details.

Write Code That Initializes the Data Browser

Listing 2-7 shows code that initializes the column view data browser shown in [Figure 2-7](#) (page 34). A detailed explanation for each numbered line of code appears following the listing.

Listing 2-7 Initializing a column view data browser

```
ControlRef MyInitializeDataBrowserControl (WindowRef window)
{
    const      ControlID  dbControlID = {'CLVW', 0};                // 1
    OSStatus   status = noErr;
    ControlRef dbControl;
    DataBrowserCallbacks dbCallbacks;

    GetControlByID (window, &dbControlID, &dbControl);            // 2
    dbCallbacks.version = kDataBrowserLatestCallbacks;            // 3
    InitDataBrowserCallbacks (&dbCallbacks);                      // 4
    dbCallbacks.u.v1.itemDataCallback =
        NewDataBrowserItemDataUPP((DataBrowserItemDataProcPtr)
            MyDataBrowserItemDataCallback);                        // 5
    dbCallbacks.u.v1.itemNotificationCallback =
        NewDataBrowserItemNotificationUPP(
            (DataBrowserItemNotificationProcPtr)
            MyDataBrowserItemNotificationCallback);                // 6

    status = SetDataBrowserCallbacks(dbControl, &dbCallbacks);    // 7
    return dbControl;
}
```

Here's what the code does:

1. Initializes a control variable with the values you set up in Interface Builder for the data browser.
2. Obtains the data browser using the control ID from the previous steps.
3. Sets the version of the data browser callbacks structure to the latest version.
4. Initializes a data browser callback structure, effectively setting all fields to `NULL` in preparation for assigning the callbacks provided by your application.
5. Creates a universal procedure pointer (UPP) to your item-data callback and assigns the callback to the appropriate field in the data browser callbacks structure.
6. Creates a UPP to your item-notification callback and assigns the callback to the appropriate field in the data browser callbacks structure.

7. Installs the callbacks by calling the function `SetDataBrowserCallbacks`, passing the data browser callbacks structure as a parameter.

The only task left to perform is to populate the data browser with data when your application first launches. You can add a function call to the initialization function here, or to the main part of your application. For a column view data browser, you can call the function `SetDataBrowserColumnViewPath` to open the data browser so the specified target item is selected. Calling this function populates the entire data browser, not just the items in the specified path. As an alternative, you can populate the data browser by calling the function `SetDataBrowserTarget` to add data along a specific path, with the target item selected.

Switching Between List View and Column View

This section shows how to create a data browser that opens in list view, but the user can switch to column view. You'll find it easier to follow the instructions in this section if you have already read the sections [“Displaying Hierarchical Data in a List View”](#) (page 30) and [“Displaying Data in a Column View”](#) (page 33).

To create a data browser that can switch between list view and column view, perform the following tasks:

1. Design and configure a data browser that uses list view. The data browser opens in list view, so you'll use Interface Builder to configure the list view portion of the data browser. You will configure the column view programmatically.

This example uses the same design and configuration options as that used for the simple list view discussed in the section [“Design and Configure a List View Data Browser”](#) (page 22), except that you'll set up the Title column to display both icon and text data. This will make the list and column views look more consistent when you switch between them. Follow the instructions in the [“Design and Configure a List View Data Browser”](#) (page 22), but choose Icon and Text as the display type for Column 1.

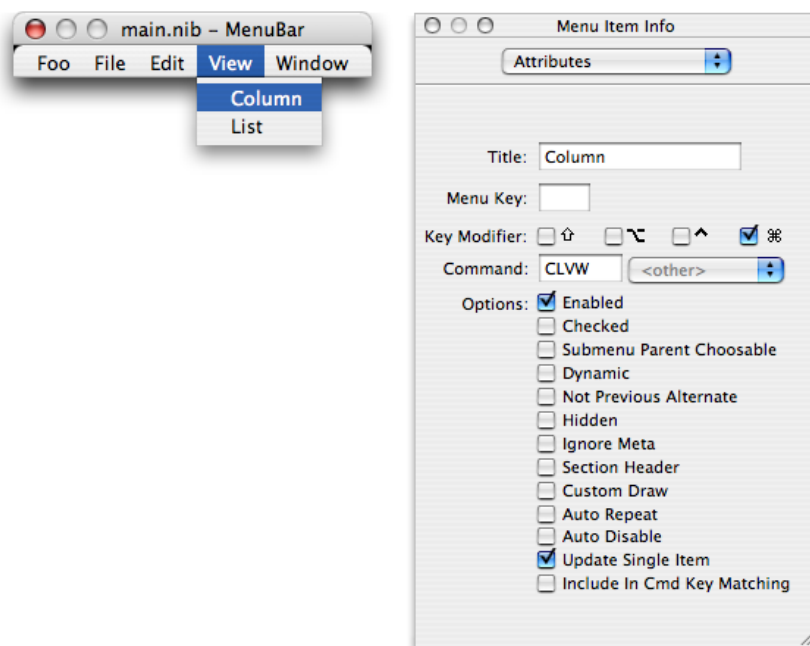
2. Set up a menu that allows the user to switch from one view to another. See the section [“Set Up a Menu for Switching Views”](#) (page 40).
3. Declare the necessary global constants. See [“Declare the Global Constants Needed for Switching”](#) (page 40).
4. Write an item-data callback that can write data to the data browser regardless of whether its in list or column view. See [“Write an Item-Data Callback to Handle Both Views”](#) (page 41).
5. Write an item-notification callback that handles container notifications. The item-notification callback needed for this data browser is identical to that supplied for the list view data browser that displays hierarchical data. The notification you must respond to is `kDataBrowserContainerOpened`. See [“Write an Item-Notification Callback for Hierarchical Data”](#) (page 32) for details.
6. Write code that initializes the data browser control. See [“Write Code That Initializes a Switchable Data Browser”](#) (page 43).
7. Write code that changes the view style of the browser. See [“Write Code to Switch Between Views”](#) (page 45).
8. Write code that configures the data browser for the current view. See [“Configure the Data Browser”](#) (page 45).

9. Write data to the data browser when the application first launches. See [“Populate the Initial List View With Data”](#) (page 46).

Set Up a Menu for Switching Views

You can set up any user interface element you’d like for switching views (menu, button, and so forth), but in this example you’ll create a menu in Interface Builder. The View menu shown in Figure 2-10 contains two menu items—Column and List. You assign each item a command that is issued when the user chooses the menu item from the View menu. In this case the Column menu item is assigned the command 'CLVW', as shown in the figure. You can assign 'LSVW' for the List menu item command. Your application needs to handle each of these commands appropriately, as shown in the section [“Write Code to Switch Between Views”](#) (page 45).

Figure 2-10 Assigning a command to a menu item



If you are unfamiliar with how to create a menu in Interface Builder, see the following resources:

- Managing Menus under Objects in Interface Builder Help
- *Learning Carbon*, available from O’Reilly publishers

Declare the Global Constants Needed for Switching

When you designed and configured the list view data browser previously (see [“Design and Configure a List View Data Browser”](#) (page 22)), you assigned each of the list view columns a unique property ID. You must make sure the code in your Carbon application uses the same property IDs you assigned in Interface Builder

because you refer to the columns from within your application using the property ID. Rather than use the four-character property IDs directly, it's useful to declare an enumeration of constants set to the property IDs. You'll use these constants later in an item-data callback to determine which column requires updating.

In addition to the property IDs, you also need to declare constants for the menu commands you set up in Interface Builder. In Xcode, open the `main.c` file and add the following global enumeration to the code:

```
enum
{
    kMyTitleColumn = 'TITL',
    kMyTimeColumn = 'TIME',
    kMyGenreColumn = 'GENR',
    kMyRatingColumn = 'RATE',
    kCommandListView = 'LISV',
    kCommandColumnView = 'COLV'
};
```

Write an Item-Data Callback to Handle Both Views

An item-data callback for a data browser that switches between views must respond to inquiries from both views. This means your switch statement responds to the property IDs assigned to each list view column, as well as the API-defined properties used for column view columns—`kDataBrowserSelfIdentityProperty`, `kDataBrowserItemIsContainerProperty`, and `kDataBrowserItemParentContainerProperty`.

Listing 2-8 shows an item-data callback for a data browser that switches between views. A detailed explanation for each numbered line of code appears following the listing.

Listing 2-8 An item-data callback for a data browser that can switch between views

```
OSStatus MyDataBrowserItemDataCallback (ControlRef browser,
    DataBrowserItemID itemID,
    DataBrowserPropertyID property,
    DataBrowserItemDataRef itemData,
    Boolean changeValue)
{
    UInt32 index = MyGetIndex (itemID);
    OSStatus status = noErr;

    switch (property)
    {
        case kMyTitleColumn: // 1
            status = SetDataBrowserItemDataIcon (itemData,
                MyCheckIfContainer (itemID) ?
                icon[kFolder] : icon[kDocument]);
            SetDataBrowserItemDataText (itemData,
                myTunesDatabase[index].song);
            break;
        case kMyTimeColumn: // 2
            SetDataBrowserItemDataText (itemData,
                myTunesDatabase[index].time);
            break;
        case kMyGenreColumn: // 3
            SetDataBrowserItemDataText (itemData,
                myTunesDatabase[index].genre);
            break;
        case kMyRatingColumn: // 4
```

```

        SetDataBrowserItemDataText(itemData,
                                   myTunesDatabase[index].rating);
        break;
    case kDataBrowserItemSelfIdentityProperty: // 5
    {
        status = SetDataBrowserItemDataIcon (itemData,
                                             MyCheckIfContainer (itemID) ?
                                             icon[kFolder] : icon[kDocument]);
        status = SetDataBrowserItemDataText (itemData,
                                             myTunesDatabase[index].song);
        break;
    }
    case kDataBrowserItemIsContainerProperty: // 6
    {
        status = SetDataBrowserItemDataBooleanValue (itemData,
                                                      MyCheckIfContainer (itemID));
        break;
    }
    case kDataBrowserItemParentContainerProperty: // 7
    {
        DataBrowserItemID myParent;
        myParent = MyGetParent (itemID); // 8
        status = SetDataBrowserItemDataItemID (itemData, myParent); // 9
        break;
    }
}

return status;
}

```

Here's what the code does:

1. Checks for the title property ID, then sets the icon and text data for the item. Recall that the display type for the title column is icon and text, so you have to set the appropriate icon as well as the text. This example uses a folder icon for a container and a document icon for an item that is not a container. The code checks to see if the item is a container, and then supplies the appropriate icon.
2. Checks for the time property ID, then sets the text data for the item. This is not the time of day, but the duration of a song, so the data is text.
3. Checks for the genre property ID, then sets the text data for the item.
4. Checks for the rating property ID, then sets the text data for the item.
5. Checks for the self-identity property (relevant only in column view). Sets the icon for the item—a folder, used for a container, or a document, used for an item. Then sets the text data for the item.
6. Checks for the container property. It sets the property to `true` if the item is a container or `false` if the item is not a container. Your application determines whether the item is a container or not.
7. Checks for the parent container property.
8. Calls your routine to look up the item ID for the item's parent. If the item is at the top of the hierarchy, it has no parent, and the returned value should be 0.

9. Sets the item's parent ID by calling the function `SetDataBrowserItemDataItemID`. If the item has a parent, the data browser calls your item-data callback again with the parent ID you provide. It keeps doing this until the top of the hierarchy is reached.

Write Code That Initializes a Switchable Data Browser

As with any data browser, you initialize the data browser by setting the appropriate callbacks. For a data browser that can switch views, you also need to set the initial view (in this case, list view) and to store configuration information about the list view columns. Whenever the user switches from column view back to list view, you need to retrieve the list view configuration information so you can restore the data browser to the list view state. You'll declare the following global structures to store list view configuration information for each column in the list. Note that `myNumColumns` should be replaced with the number of columns you need in list view.

```
struct ListViewColumns{
    DataBrowserTableViewColumnID column;
    DataBrowserListViewHeaderDesc description;
};
typedef struct ListViewColumns ListViewColumns;

ListViewColumns myListViewColumns[myNumColumns];
DataBrowserPropertyDesc myColumnPropertyDescription[myNumColumns];
```

You'll save list view configuration information to these global data structures in the initialization function shown in Listing 2-9, so that later you can restore the configuration. A detailed explanation for each numbered line of code appears following the listing.

Listing 2-9 Initializing a data browser that can switch between views

```
ControlRef MyInitializeDataBrowserControl (WindowRef window)
{
    const ControlID dbControlID = {'CLVW', 0};
    OSStatus status = noErr;
    ControlRef dbControl;
    DataBrowserCallbacks dbCallbacks;
    int i;

    GetControlByID (window, &dbControlID, &dbControl); // 1
    dbCallbacks.version = kDataBrowserLatestCallbacks; // 2
    InitDataBrowserCallbacks (&dbCallbacks);

    dbCallbacks.u.v1.itemDataCallback =
        NewDataBrowserItemDataUPP ((DataBrowserItemDataProcPtr)
            MyDataBrowserItemDataCallback); // 3
    dbCallbacks.u.v1.itemNotificationCallback =
        NewDataBrowserItemNotificationUPP (
            (DataBrowserItemNotificationProcPtr)
            MyDataBrowserItemNotificationCallback); // 4

    status = SetDataBrowserCallbacks(dbControl, &dbCallbacks);

    SetDataBrowserListViewDisclosureColumn (dbControl,
        kMyTitleColumn, true); // 5
```

```

SetDataBrowserSelectionFlags (dbControl, kDataBrowserSelectOnlyOne); // 6
SetAutomaticControlDragTrackingEnabledForWindow (window, true); // 7
SetDataBrowserViewStyle (dbControl, kDataBrowserListView); // 8
myListViewColumns[0].column = kTitleColumn; // 9
myListViewColumns[1].column = kGenreColumn;
myListViewColumns[2].column = kTimeColumn;
myListViewColumns[3].column = kRatingColumn;
for (i = 0; i < kNumListViewColumns; i++) // 10
{
    DataBrowserListViewHeaderDesc description;

    GetDataBrowserListViewHeaderDesc (dbControl,
        myListViewColumns[i].column, &description); // 11
    myListViewColumns[i].description = description; // 12
    myColumnPropertyDescription[i].propertyID =
        myListViewColumns[i].column; // 13
    if (i == 0) // 14
        myColumnPropertyDescription[i].propertyType =
            kDataBrowserIconAndTextType;
    else
        myColumnPropertyDescription[i].propertyType =
            kDataBrowserTextType;
}

return dbControl;
}

```

Here's what the code does:

1. Initializes a control ID variable to the values you set up in Interface Builder to identify the data browser.
2. Sets the version of the data browser callbacks structure to the current one.
3. Creates a universal procedure pointer (UPP) to an item-data callback and assigns the callback to the appropriate field in the data browser callback structure.
4. Creates a UPP to an item-notification callback and assigns the callback to the appropriate field in the data browser callback structure.
5. Sets the title column as the one to display a disclosure triangle. You need to do this because the data is organized hierarchically. In this instance, the title column either denotes an album or a song title. Album titles disclose song titles.
6. Sets the selection flags so that only one item can be selected at a time. Your application would set up the selection flags appropriate to its needs.
7. Enables drag tracking. This allows users to reorder the columns by dragging. The system automatically handles the drag operation for you.
8. Sets the view style to a list view style. The data browser first opens to a list view but the user can change the view to a column view after the application launches.
9. Starts to fill the list view column structure with configuration information. This line and the next three lines of code save the property IDs for each column.
10. Iterates through the columns to save the appropriate configuration information.

11. Obtains the current list view header description, which you previously set up in Interface Builder.
12. Assigns the description to the list view column structure.
13. Assigns the property ID to the column property description structure.
14. Assigns the appropriate display type to the column property description structure. Recall that you previously set up the title column to contain icon and text data and all other columns to contain text only.

Write Code to Switch Between Views

In one of the Carbon event handlers for your application you need to write code that checks for, and responds to, the menu commands you set up in Interface Builder. Recall that you declared constants for these commands in the section “[Declare the Global Constants Needed for Switching](#)” (page 40).

Respond to the menu commands `kCommandListView` and `kCommandColumnView` by calling the function `SetDataBrowserViewStyle` with the `style` parameter set to the appropriate value—either `kDataBrowserListView` or `kDataBrowserColumnView`. Then, call your function to configure the data browser for the new view.

Configure the Data Browser

Whenever the user switches the view, you must configure the data browser for the new view. Listing 2-10 shows how to configure each view. A detailed explanation for each numbered line of code appears following the listing.

Listing 2-10 Configuring the data browser for each view

```
void ConfigureDataBrowser (ControlRef dbControl)
{
    UInt32          i;
    SInt32          numAlbums;
    DataBrowserItemID target;
    DataBrowserViewStyle viewStyle;

    numAlbums = CFStringGetIntValue (CFCopyLocalizedString
                                     (CFSTR("NumAlbums"), NULL));
    DataBrowserItemID myItems[numAlbums];
    SetDataBrowserTarget (dbControl, 0); // 1
    GetDataBrowserViewStyle (dbControl, &viewStyle); // 2

    switch (viewStyle)
    {
        case kDataBrowserListView:
        {
            for (i = 0; i < kNumListViewColumns; i++) // 3
            {
                DataBrowserListViewColumnDesc myColDescription;
                DataBrowserListViewHeaderDesc description;
                description = myListViewColumns[i].description;
                myColDescription.headerBtnDesc =
```

```

        myListViewColumns[i].description;
        myColDescription.propertyDesc =
            myColumnPropertyDescription[i];
        status = AddDataBrowserListViewColumn (dbControl,
            &myColDescription, i);
    }
    SetDataBrowserListViewDisclosureColumn (dbControl,
        kTitleColumn, true); // 4
    SetDataBrowserSelectionFlags (dbControl,
        kDataBrowserSelectOnlyOne); // 5
    MyPopulateListView(); // 6
    break;
}
case kDataBrowserColumnView:
{
    int j = 0;
    MyPopulateColumnView(); // 7
}break;
}
}
}

```

Here's what the code does:

1. Sets the target for the data browser to the top level. You could also save the current target at the time the user switches the view, then set the target as the target for the new view.
2. Gets the view style. This is the view the user just switched to, and that you set when you processed the view command.
3. Retrieves the previously saved list view configuration and adds each column to the list view by calling the function `AddDataBrowserListViewColumn`.
4. Sets the title column as the one that has a disclosure triangle.
5. Sets a flag to restrict selection in list view to one item at a time. Your application would set whatever user selection flags are appropriate.
6. Calls your function to populate list view. Your function should build an array of the item IDs for the container items, and then call the function `AddDataBrowserItems` to add the container items to the list.
7. Calls your function to populate column view. Your function can call `SetDataBrowserTarget`, supplying the item ID for the item you want as the target. You must make sure your item-data callback responds to the request `kDataBrowserItemParentContainerProperty` to ensure the data browser is populated properly.

Populate the Initial List View With Data

When your application first launches, you need to read in your data and populate the data browser. The code shown in Listing 2-11 is from the main part of the application. It calls the function `AddDataBrowserItems` for each container to display in the initial list view. Your application needs to take similar action when the application launches.

Listing 2-11 Reading data and populating the data browser

```

DataBrowserItemID myItemID;

myNumRows = MyLoadData (dbControl);
for (i = 0; i < MyNumRows; i++)
{
    if (MyCheckIfContainer (i))
    {
        myItemID = MyGetItemID(i);
        AddDataBrowserItems (dbControl, kDataBrowserNoItem,
                             1, &myItemID, kTitleColumn);
    }
}

```

Supporting Text Editing

The data browser provides built-in text editing capability for columns with the display type `kDataBrowserTextType` or `kDataBrowserIconAndTextType`. The user clicks an item in a cell and edits the text displayed in that cell. The editing session is open as long as the user has the cell selected. Your application needs to capture any changes the user makes and save those changes to its internal data store.

To support text editing in a data browser, follow these steps:

1. When you create a column that you want users to be able to edit, set the property `kDataBrowserPropertyIsEditable` to `true` for that column. Otherwise, the user won't be able to modify the data in the column.
2. In the switch statement in your item-data callback, handle the `kDataBrowserItemIsEditableProperty` inquiry. In response to this inquiry, call the function `SetDataBrowserItemDataBooleanValue` with the `theData` parameter set to `true`. If your application has more than one column that contains editable text, call the function `GetDataBrowserItemDataProperty` to find out which column is being edited.
3. After you pass `true` in response to the `kDataBrowserItemIsEditableProperty` inquiry, the data browser issues two notifications to your item-notification callback: `kDataBrowserEditStarted` and `kDataBrowserEditStopped`. Respond to each of these appropriately. For example, after you've been notified that editing has started, you can lock any data that might need to be locked, or update the user interface as needed. After the editing has stopped, you can unlock any locked data or update the user interface.
4. When the editing session is finished, your item-data callback is invoked with the `property` parameter set to the ID of the column that was modified and the `setValue` parameter set to `true`. Your callback extracts the edited text by calling the function `GetDataBrowserItemDataText`, supplying the item ID of the item whose text was modified. You can then make sure the string is valid and decide to accept or reject the new value before saving the edited text to your application's private data store. If your application decides to reject the edited text, you can inform the user that you are rejecting the edit.

Applying an Operation to Each Item

The item-iterator callback (`DataBrowserItemProcPtr`) performs a task, defined by your application, on each item in a data browser that meets your criteria. You provide the callback and the criteria for applying it as parameters to the function `ForEachDataBrowserItem`. For example, you could write a callback to remove inactive items or to set values displayed in a data browser to default values.

Sorting Items

Items in a list view column can be sorted in ascending or descending order. Items in a container appear in the same sorting order as the items in the parent container. You supply a callback (`DataBrowserItemCompareProcPtr`) to handle sorting. Your callback can perform simple sorting or more complex sorting, such as the following:

- Sort containers in a hierarchical list independently of each other
- Use secondary and tertiary sorting

In either of the complex cases, your callback must keep track of previous sort operations and preserve sorting orders for secondary and tertiary items.

Supporting Drag and Drop

If your application wants to support drag-and-drop behavior in a data browser, it can supply callbacks to:

- Add an item to a drag object. The callback `DataBrowserAddDragItemProcPtr` is invoked when a drag operation is started and is used to support dragging objects from the data browser. Your callback adds the item to the drag reference passed to it by the data browser.
- Determine if a drag object can be accepted in a particular location (`DataBrowserItemAcceptDragProcPtr`)
- Receive a drag item by extracting the item from the drag reference passed to it by the data browser and then processing the item appropriately (`DataBrowserItemReceiveDragProcPtr`)
- Perform drag postprocessing tasks, such as deallocating resources (`DataBrowserPostProcessDragProcPtr`)

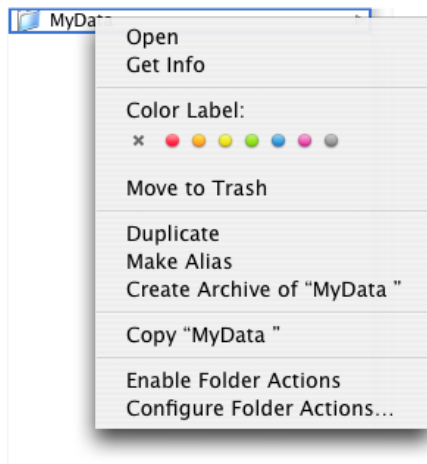
If you want only to support the ability for the user to change column positions in a list view, you don't need to provide drag callbacks.

Any time you want a data browser to support drag-and-drop, you must call the Control Manager function `SetAutomaticControlDragTrackingEnabledForWindow`. This is true even if you want only column dragging or only item dragging.

Providing a Contextual Menu

Providing a contextual menu in a data browser is optional. You need to supply two callbacks if you want to support a contextual menu in a data browser—`get-contextual-menu` (`DataBrowserGetContextualMenuProcPtr`) and `select-contextual-menu` (`DataBrowserSelectContextualMenuProcPtr`). The `get-contextual-menu` callback is called by the data browser when the user Control-clicks an item in the display. Figure 2-11 shows how such a menu looks; this one is from the Finder.

Figure 2-11 A contextual menu in the Finder



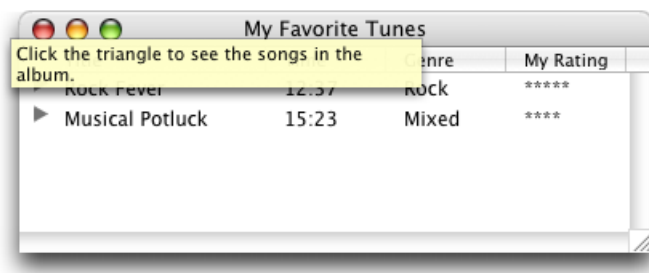
Your `get-contextual-menu` callback provides a menu and information about help that is (or is not) available. You can customize the content of the menu and determine what information to provide in it by calling the function `GetDataBrowserItems` with the `state` parameter set to `kDataBrowserItemSelected`. This function obtains the items that are selected, which you can then use to choose the appropriate information to supply.

Your `select-contextual-menu` callback is invoked by the data browser when the user selects an item from the contextual menu. Your callback can retrieve the selection and take the appropriate action.

Providing Help Tags

The `item-help-content` callback (`DataBrowserItemHelpContentProcPtr`) is invoked by the data browser when the user hovers the pointer over an item. Your callback fills in a help tag structure provided to you by the data browser and, when requested, disposes of previously supplied help content. As a result of the information you supply, a help tag is displayed similar to that shown in Figure 2-12. You are not required to provide this callback.

Figure 2-12 A help tag in a data browser



If you want to provide a single help tag for the entire data browser, you can simply type the help tag content into the Help pane of the Info window in Interface Builder. The system takes care of displaying and disposing of the content; a callback is unnecessary.

Customizing Drawing, Tracking, and Dragging

If you want to control how items are drawn, tracked, and dragged in a list view data browser, you need to set the display type of the columns to custom (`kDataBrowserCustomType`). (See [“What Can Be Displayed”](#) (page 19) for more information on display types.) The custom type signals the data browser that, at the very least, you want to handle drawing the items. If you plan to support editing, drag and drop, or anything that requires tracking and hit-testing, you must also handle these tasks because none of these tasks are handled by the data browser for columns that display custom types.

The data browser API has a suite of callbacks that operate only on custom types:

- The draw-item callback (`DataBrowserDrawItemProcPtr`), which is invoked by the data browser whenever your custom content needs to be drawn.
- The edit-item callback (`DataBrowserEditItemProcPtr`), which is invoked by the data browser when a user clicks a custom item.
- The hit-testing callback (`DataBrowserHitTestProcPtr`), which determines if the mouse is over content that can be selected or dragged.
- The tracking callback (`DataBrowserTrackingProcPtr`), which is invoked after your hit-testing callback returns `true` and when a mouse-click is inside the content area of the custom item.
- The item-drag callback (`DataBrowserItemDragRgnProcPtr`), which determines which part of an item to use to create a transparent image for a dragged item.
- The item-accept callback (`DataBrowserItemAcceptDragProcPtr`), which is invoked after your item-drag callback starts a drag operation, determines whether or not the associated item can accept the drag object.
- The item-receive callback (`DataBrowserItemReceiveDragProcPtr`), which is called after your item-accept-drag callback has determined that a location can accept a drag object. Your application takes whatever actions are necessary to add the dropped data to the data browser.

Document Revision History

This table describes the changes to *Data Browser Programming Guide*.

| Date | Notes |
|------------|---|
| 2007-08-07 | Corrected typographical error. |
| 2006-09-05 | Made minor technical improvements and changed the title from "Displaying Data in a Data Browser." |
| 2005-07-07 | Revised information on sorting. |
| 2004-08-31 | Made corrections to Table 1-1 (page 15) and Listing 2-4 (page 32). |
| 2004-03-05 | Added information to explain how the data browser works, including details on the item-data and item-notification callbacks, and how the data browser is populated with data. |
| | Moved information on the optional callbacks from the Concepts to the Tasks chapter. |
| | Added information on how to use the function <code>SetDataBrowserListViewDisclosureColumn</code> to set up a hierarchical list properly. |
| | Made minor code revisions to fix errors and remove unused variables. |
| 2003-12-11 | Preliminary version of this document. |

REVISION HISTORY

Document Revision History