CHAPTER 4

List Manager

Contents

List Manager

This chapter describes how your application can use the List Manager to create scrollable lists that allow the user to select one or more of a group of items. The List Manager lets you create one-column lists or multicolumn lists. By default, it creates lists that contain only unstyled text, but with extra effort, you can use the List Manager to create lists that display items graphically.

Read the information in this chapter if you need to allow users to select one or more items from a group of items. If you only need to allow the user to select one item from a small group of items, a pop-up menu may be more appropriate than a list. If, however, you would like the user to be able to select one of many items or to be able to select multiple items, the List Manager provides a convenient and intuitive interface.

If the contents of a group of items might change, use a list rather than a pop-up menu. Users generally expect the contents of pop-up menus to remain the same, whereas a list provides instant visual feedback when its contents change, thus preventing user confusion. For example, you might use the List Manager to create a list of appointments and allow the user to add or remove appointments to or from the list.

Although the List Manager can handle small, simple lists effectively, it is not suitable for displaying large amounts of data (such as that used by a spreadsheet application). The List Manager cannot maintain lists that occupy more than 32 KB of memory, and performance degrades sharply well before the 32 KB limit. Also, the List Manager expects all cells to be equal in size. Thus, if you are writing a spreadsheet application, you should use the Control Manager and your own internal data structures. However, you should still read the sections of this chapter that concern selection of list items so that your application can have a user interface consistent with the List Manager's.

To use this chapter, you should be familiar with the concepts of the Control Manager, the Event Manager, and the Window Manager, and, if you plan to create a list in a modal or modeless dialog box, with the Dialog Manager. For more information on these topics, see *Inside Macintosh: Macintosh Toolbox Essentials*.

This chapter begins by describing lists and the user interface for them. The chapter then discusses how you can

- create lists
- respond to events affecting lists
- get information about a list
- get or change the contents of list items
- search through a list for a particular item
- support keyboard navigation of lists
- manage multiple lists within the same window or dialog box
- write your own list definition procedure to handle nonstandard lists, such as lists of pictures

Introduction to Lists

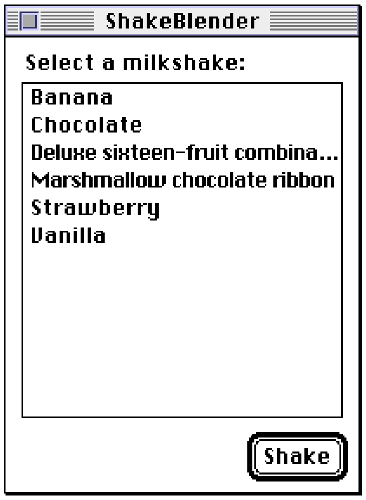
You can use the List Manager to store and update elements of data within a list and to display the list in a rectangle within a window. The List Manager provides routines that allow you to create, manipulate, and display lists. It can also respond appropriately to a mouse click within a list by, for example, scrolling a list when the user clicks in a scroll bar. Thus, using the List Manager is easier than using the Control Manager and QuickDraw to create a scrolling list of items.

Appearance of Lists

A **list** is a series of items displayed within a rectangle. Each item in a list is contained within an invisible rectangular **cell.** All cells in a list created by the List Manager are the same size, but cells may contain different types of data. Your application may allow the user to select one or more items in a list by clicking them. When a user selects an item, the List Manager highlights the cell containing the item.

Figure 4-1 illustrates a window that includes a list of six items.

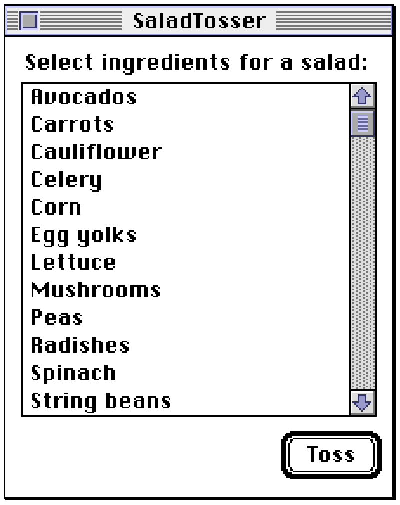
Figure 4-1 A one-column, text-only list without a scroll bar



The font used for a text-only list is determined by the font of the current graphics port. Usually, you should create lists in the system font. Regardless of the font your application uses, if a string is too long to fit in a list using the current font, the List Manager uses condensed type in an effort to fit it. If the string is still too long, the List Manager truncates the string displayed and appends an ellipsis to it. Both of these techniques are illustrated in Figure 4-1. Both the strings “Deluxe sixteen-fruit combination” and “Marshmallow chocolate ribbon” are condensed; the first of these is also truncated.

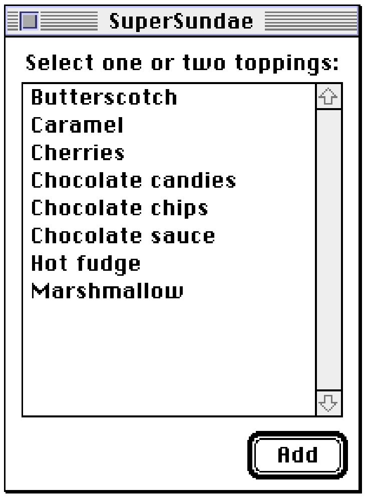
Lists may contain a vertical scroll bar, a horizontal scroll bar, or both. By using scroll bars, you can include more items in a list than can fit in the list’s rectangle, and the user can scroll to view multiple items. If there is any chance that a list may contain more cells than can fit within the list’s rectangle, you should include a scroll bar in the list. Figure 4-2 illustrates a list that includes a vertical scroll bar.

Figure 4-2 A one-column, text-only list with a vertical scroll bar



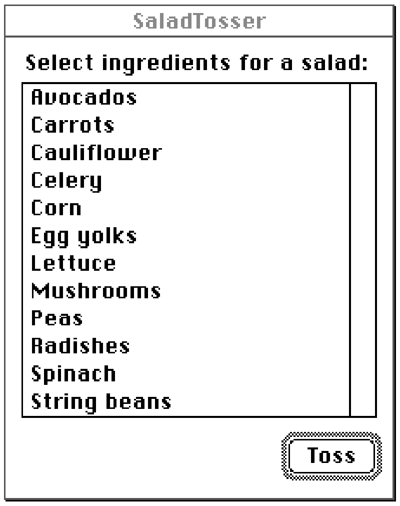
If a list includes a scroll bar but there are a small enough number of items in the list that all the list's items are visible, the List Manager automatically disables the scroll bar. For example, Figure 4-3 shows such a list.

Figure 4-3 A list whose scroll bar has been disabled



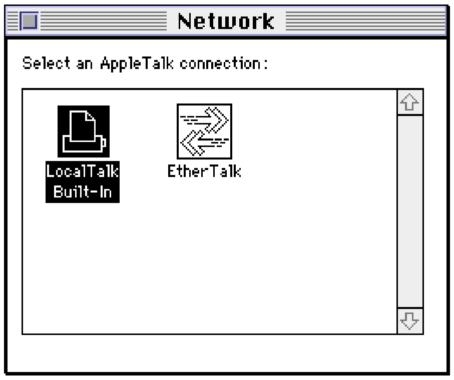
When a window containing one or more lists becomes deactivated, your application should call the List Manager to deactivate the lists as well. Figure 4-4 shows a deactivated list.

Figure 4-4 A deactivated list



Your application can create one-column lists of the type illustrated in Figure 4-2 through Figure 4-4 using the List Manager. Your application can also create lists that contain two or more columns. For example, the Network control panel allows the user to select a network connection from a three-column list. In Figure 4-5, there are only two possible network connections, so there are no items in the third column of the list.

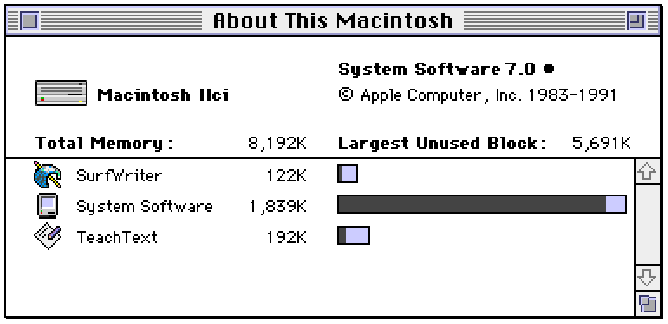
Figure 4-5 A list containing multiple columns and graphical elements



Note that the list in Figure 4-5 contains graphical elements rather than just text. To create a list with graphical elements, you must write a custom list definition procedure, because the default list definition procedure supports only the display of text. A **list definition procedure** is a code resource of type `'LDEF'` that defines the characteristics of a list. In addition to using a list definition procedure to support graphical items in lists, you can write one to customize the display of text in a list. For example, to use styled text in a list, you would need to create a list definition procedure.

You can also use a list definition procedure to create lists that contain cells which display more than one type of information. For example, the Finder's "About This Macintosh" modeless dialog box contains a list of applications that are currently in use. Each cell in the list includes a small icon of the application, the name of the application, the amount of memory in the application's partition, and a graphical indication of how much of that memory has been used, as illustrated in Figure 4-6.

Figure 4-6 A list of items whose cells display more than one type of information



About This Macintosh

Macintosh IIci

System Software 7.0 •
© Apple Computer, Inc. 1983-1991

Total Memory:	8,192K	Largest Unused Block:	5,691K
SurfWriter	122K		
System Software	1,839K		
TeachText	192K		

Note that the list in Figure 4-6 is not a multicolumn list. It is a one-column list, but each cell of the list displays several types of information.

Your application specifies whether the List Manager should leave room for a size box, although your application is responsible for drawing any grow icon; the List Manager does not draw the grow icon automatically. Usually, size boxes are useful only for lists that are on the bottom of the windows that contain them, like the list in Figure 4-6. In this case, resizing the window changes the size of the list. Your application should ensure that the user cannot shrink the size of the window so much that the list is no longer visible.

In addition to requesting a vertical scroll bar, your application may request that the List Manager use a horizontal scroll bar for your list. A second scroll bar is useful mainly if your application allows the user to resize a window containing a list both horizontally and vertically so that only a portion of the list is visible. A second scroll bar is also useful to allow the user to scroll through a table of cells. Usually, however, if you are implementing a spreadsheet-like application, you should not be using the List Manager. Most multicolumn lists created by the List Manager, such as the one illustrated in Figure 4-5, should not include two scroll bars.

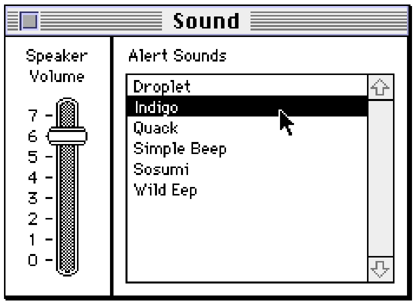
Selection of List Items

Sometimes, an application might create a list simply for the user to view. For example, a desktop-publishing application might create a list of fonts used in a document. The user should be able to scroll the list to examine all of the fonts, but the application can ensure (by ignoring mouse clicks on list cells) that clicking cells of the list has no effect. More often, however, applications create lists so that users can select items from them by clicking the items' cells.

Your application can allow the user to select items in a list by calling the `LClick` function whenever a mouse-down event occurs. The `LClick` function handles all user interaction, including highlighting of items, until the user releases the mouse button. The `LClick` function also examines the state of the modifier keys (specifically the Shift and Command keys) and changes the selection appropriately.

Figure 4-7 illustrates the Sound control panel, which allows users to select a system alert sound from a list of available alert sounds.

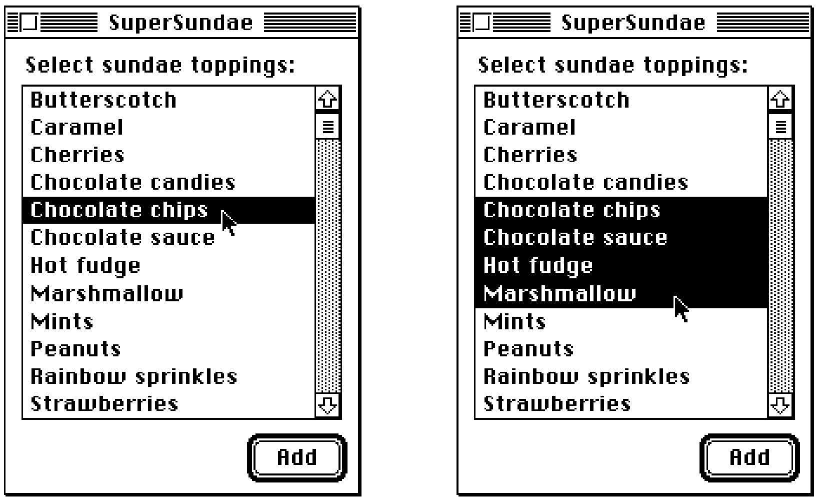
Figure 4-7 A list with an item selected



When the user selects a cell (such as the "Indigo" system alert sound) by clicking the item's cell, the List Manager highlights the cell. In the list shown in Figure 4-7, the user can also select a cell by clicking another cell and dragging the cursor to the desired cell (such as the cell containing "Indigo") before releasing the mouse button. This type of list allows the user to select only one item, because there can be only one system alert sound. While you can create a list that has this behavior, the List Manager by default allows the user to select a range of cells or even several discontiguous ranges of cells by using the Shift and Command keys.

The user can use the Shift key to select a range of cells. By pressing the Shift key when clicking a cell, the user can select all items in a given range. For example, in Figure 4-8 the user extends a selection of just one item to cover several items by pressing the Shift key and clicking another item. The List Manager then highlights all cells ranging from the already selected cell to the newly selected cell, thus making the entire range of cells selected. In a one-column list, like that in Figure 4-8, the List Manager highlights a rectangular range of cells in response to a Shift-click.

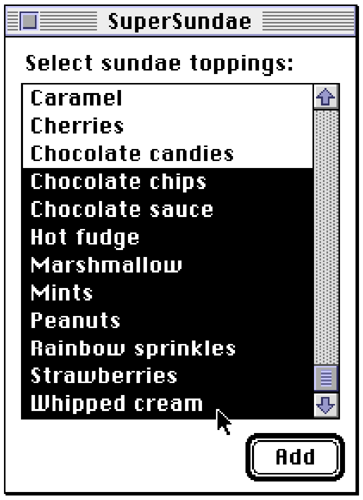
Figure 4-8 Selection of a range of items in a list



User selects an item. User then Shift-clicks another item.

After pressing the mouse button while also pressing the Shift key (but before releasing the mouse button), the user can extend or shrink the range of cells selected by dragging the cursor. The user can even drag the cursor below the list to select a range that includes items not initially visible. For example, Figure 4-9 illustrates the effect of dragging after the initial selection of the range of cells illustrated in Figure 4-8.

Figure 4-9 Effect of dragging after Shift-clicking

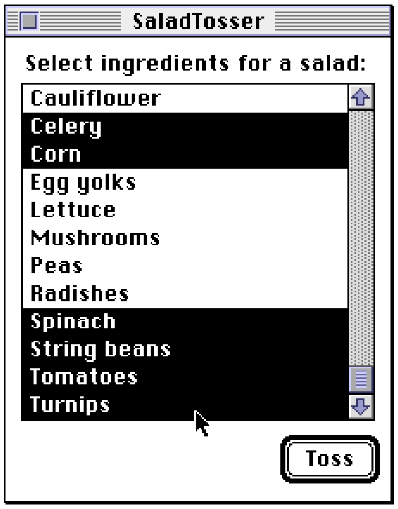


Virtually every application that supports Shift-clicking to extend list selections should also support the selection of discontiguous ranges of list cells. The default behavior of the List Manager is to allow a user to add a cell to the current selection by pressing the Command key when clicking a cell. If a user Command-clicks a cell that is already selected, the List Manager removes the cell from the selection.

To add or remove a range of cells from the current selection, a user can press the mouse button while also pressing the Command key and then drag the cursor over other cells. The List Manager determines whether to add or remove selections in a range of cells by checking the status of the first cell clicked in. If that cell is initially selected, then Command-dragging deselects all cells in the range over which the cursor passes. If that cell is initially not selected, then Command-dragging selects all cells in the range over which the cursor passes. Once the user changes a cell's selection status by Command-dragging over a cell, the selection status of the cell stays the same for the duration of the drag even if the user moves the cursor back over the cell. In this way, the use of the Command key differs from that of the Shift key.

Figure 4-10 illustrates use of the Command key. This example shows a list created by an application that allows a user to choose what vegetables to include in a salad to be tossed by a device attached to the computer.

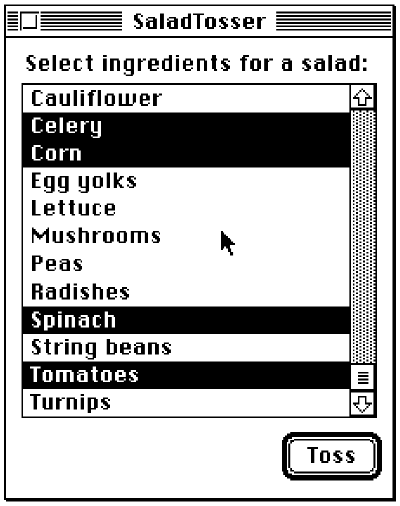
Figure 4-10 Selection of discontiguous items in a list



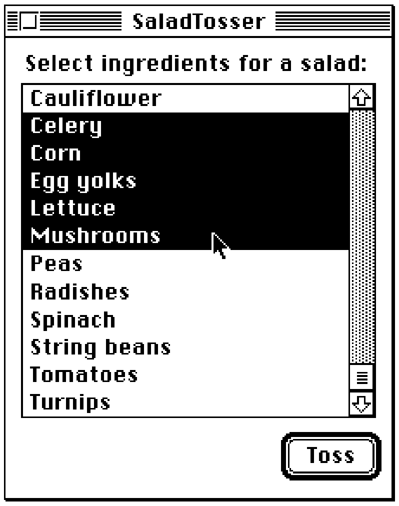
Initially, the user has selected “Celery” and “Corn.” By pressing the Command key and the mouse button while the cursor is over the item “Spinach,” then dragging the cursor downward to “Turnips” (which automatically scrolls into view), the user can select additional items. Without the feature of Command-clicking to select discontiguous ingredients, a user of this list would be able to select only alphabetical ranges of ingredients for the salad.

If a user Shift-clicks a cell after having created discontiguous selection ranges, the discontiguity is lost. The List Manager selects all cells in the range of the first selected cell and the newly selected cell, unless the newly selected cell precedes the first selected cell, in which case the List Manager selects all cells in the range of the newly selected cell and the last selected cell. Figure 4-11 illustrates how the selection changes when a user Shift-clicks a cell that follows one range of selected cells but precedes another. In this example, after selecting “Celery,” “Corn,” “Spinach,” and “Tomatoes,” the user Shift-clicks the item labeled “Mushrooms.”

Figure 4-11 Effect of Shift-clicking in a list that contains discontiguous items



User selects discontiguous items using Command-click.



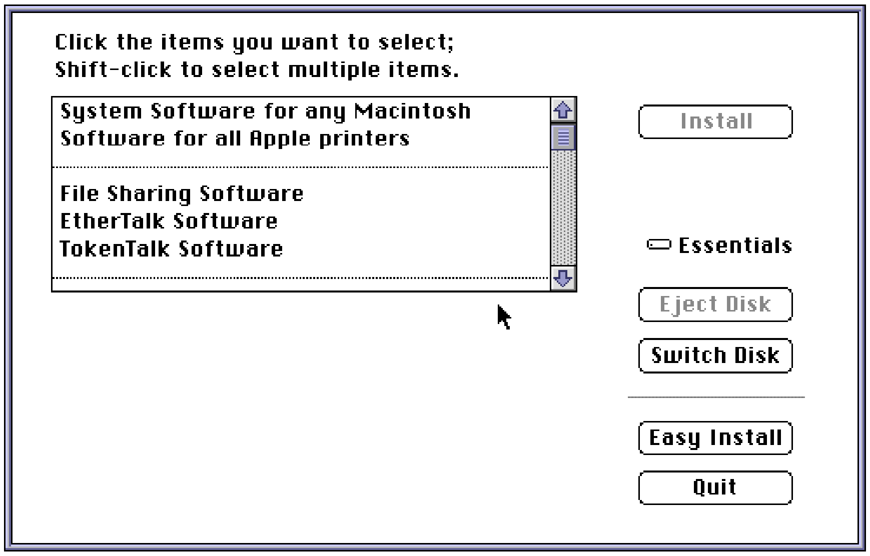
User then Shift-clicks the next item.

If a user presses both the Command and Shift keys when clicking a cell, then the pressing of the Shift key is ignored and the List Manager behaves as if only the Command key is pressed.

Your application can customize the algorithm the List Manager uses to manage the selection of list items. (You can do this by setting one or more flags in the `selFlags` field of the list record.) For example, your application can permit the user to select only one element of a list at a time, in which case the Shift and Command keys are ignored.

Some applications may wish to make the Shift key work in lists just like the Command key. This is especially useful for applications geared toward novice users, who might not think of using the Command key to select several discontiguous items in a list. If your application uses a nonstandard behavior, then it should make this clear to the user. For example, the Installer application includes a list that treats Shift-clicks like Command-clicks, and it indicates to the user that Shift-clicking selects multiple items. This is illustrated in Figure 4-12.

Figure 4-12 Notifying the user of nonstandard list behavior



Click the items you want to select;
Shift-click to select multiple items.

System Software for any Macintosh
Software for all Apple printers

File Sharing Software
EtherTalk Software
TokenTalk Software

The List Manager provides a number of other ways that your application can customize the selection of items within a list. In particular, your application can

- allow only one item to be selected at a time. (By default, the List Manager allows multiple items to be selected.)
- allow the user to select a range of items by clicking the first item and dragging to the last item without necessarily pressing the Shift or Command key. Ordinarily, dragging in this manner results in only the last item's being selected.
- disable discontiguous selections, while still allowing the user to select a range of items.
- cause all previously selected cells to be deselected when the user Shift-clicks.
- allow the user to deselect a range of cells by Shift-dragging. Ordinarily, Shift-dragging causes cells to become selected even if the first cell clicked is already selected.

- disable the feature that allows the user to shrink a selection by Shift-clicking to select a range of cells and then dragging the cursor to a position within that range. When this feature is disabled, all cells in the cursor's path during a Shift-drag become selected even if the user drags the cursor back over the cell.
- turn off the highlighting of selected cells that contain no data.

"Customizing Cell Highlighting" beginning on page 4-38 discusses the techniques that your application can use to customize the selection of lists.

Keyboard Navigation of Lists

Although it is easy to use the mouse to select list items, some users prefer to use the keyboard. Keyboard navigation and selection of list items is a particularly useful feature for long lists. Your application should support keyboard navigation of lists in two ways. First, your application should support the use of the arrow keys to move or extend a selection. Second, if your application uses text-only lists (or lists whose items can be identified by text strings), your application should allow the user to select an item simply by typing the text associated with it.

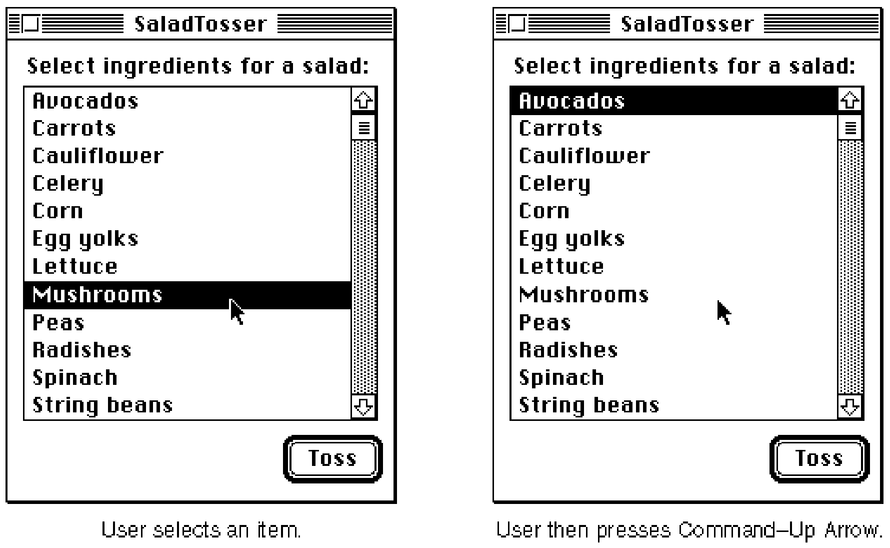
The List Manager does not provide any routines to automatically handle keyboard navigation of lists, but your application can provide code to manage keyboard navigation of lists. "Supporting Keyboard Navigation of Lists" beginning on page 4-45 shows code that handles keyboard navigation.

Movement of a Selection With Arrow Keys

When a user presses an arrow key and is not pressing the Shift or Command key, the user is attempting to move the selection one cell. For example, your application should respond to the pressing of the Up Arrow key by selecting the cell that is one cell above the first selected cell and deselecting any other selected cells. If the first selected cell is already in the first row, then your application should respond simply by deselecting all cells other than that first selected cell. Your application should respond to the pressing of the Left Arrow key by moving the selection one cell to the left. Your application should respond to the pressing of the Down Arrow key or the Right Arrow key by selecting the cell that is one cell below or to the right of the last selected cell and deselecting any other selected cells. If the last selected cell is already in the last row, then your application should respond simply by deselecting all cells other than that last selected cell.

When a user presses an arrow key while pressing the Command key, your application should move the first (or last) selected cell as far as it can move in the appropriate direction. For example, Command–Left Arrow indicates that the first selected cell should be moved as far left as possible (and all other cells should be deselected). Figure 4-13 illustrates how an application responds to the pressing of the Command–Up Arrow keys.

Figure 4-13 Response to pressing the Command–Up Arrow keys



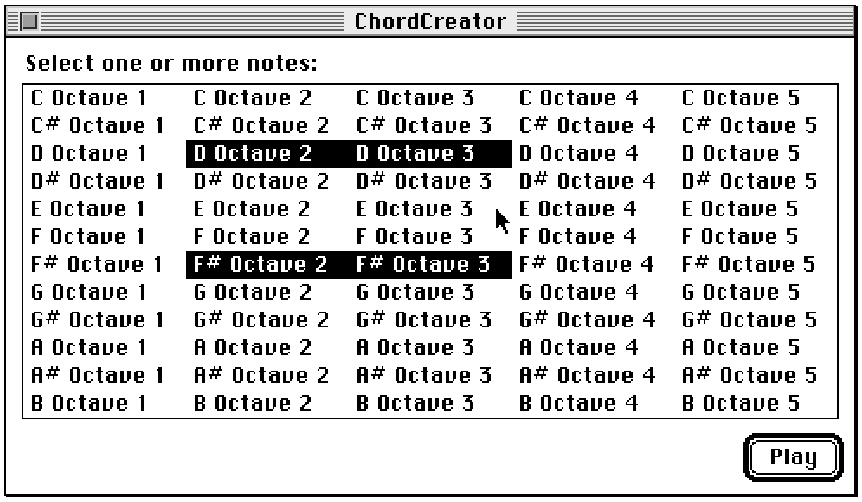
Extension of a Selection With Arrow Keys

A user may press the Shift key when pressing an arrow key to extend the current selection. There are two different algorithms that your application can use to respond to a Shift–arrow key combination.

The first potential response is the *extend algorithm*, in which your application simply finds the first (or last) selected cell, and then selects another cell in the direction of the arrow key. For example, if the user presses Shift–Right Arrow, your application should find the last selected cell and highlight the cell one column to the right of it, unless that cell is already highlighted. If the user presses Command–Shift–Up Arrow, your application should select the cell in the first row that was in the same column as the first selected cell and select all cells in between.

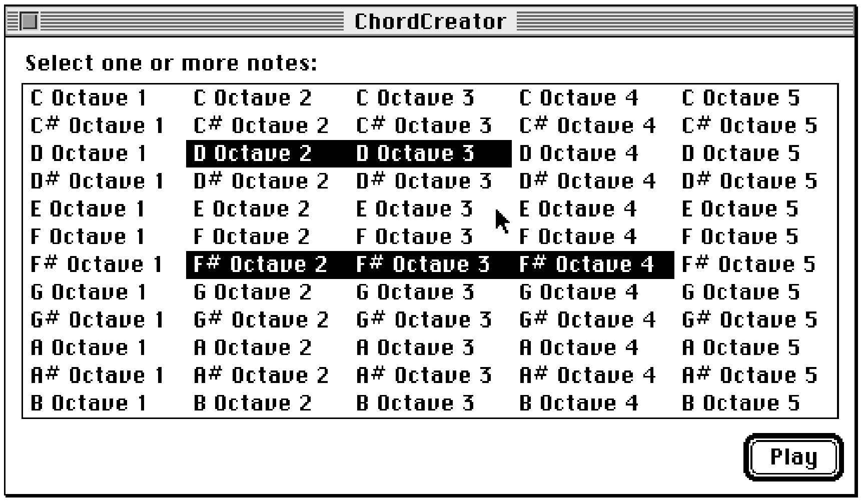
Figure 4-14 shows the effect of the extend algorithm when the user selects items using the Shift key and arrow keys. In this example, after selecting two discontiguous ranges, the user then presses Shift–Right Arrow, extending the last selected cell by one cell to the right. The user then presses Shift–Left Arrow, extending the selection one cell to the left of the first selected cell.

Figure 4-14 Response to user making a discontiguous selection, then pressing Shift–Right Arrow followed by Shift–Left Arrow using the extend algorithm



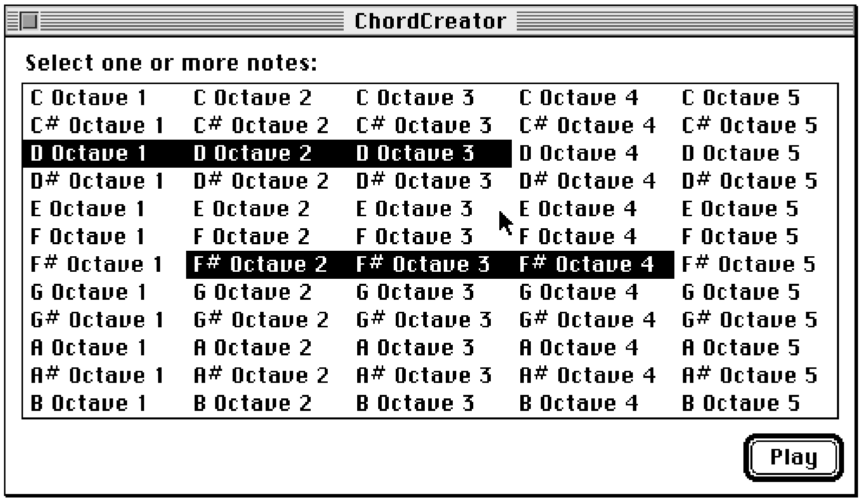
Select one or more notes:

C Octave 1	C Octave 2	C Octave 3	C Octave 4	C Octave 5
C# Octave 1	C# Octave 2	C# Octave 3	C# Octave 4	C# Octave 5
D Octave 1	[✓ D Octave 2]	[✓ D Octave 3]	D Octave 4	D Octave 5
D# Octave 1	D# Octave 2	D# Octave 3	D# Octave 4	D# Octave 5
E Octave 1	E Octave 2	E Octave 3	E Octave 4	E Octave 5
F Octave 1	F Octave 2	F Octave 3	F Octave 4	F Octave 5
F# Octave 1	[✓ F# Octave 2]	[✓ F# Octave 3]	F# Octave 4	F# Octave 5
G Octave 1	G Octave 2	G Octave 3	G Octave 4	G Octave 5
G# Octave 1	G# Octave 2	G# Octave 3	G# Octave 4	G# Octave 5
A Octave 1	A Octave 2	A Octave 3	A Octave 4	A Octave 5
A# Octave 1	A# Octave 2	A# Octave 3	A# Octave 4	A# Octave 5
B Octave 1	B Octave 2	B Octave 3	B Octave 4	B Octave 5



Select one or more notes:

C Octave 1	C Octave 2	C Octave 3	C Octave 4	C Octave 5
C# Octave 1	C# Octave 2	C# Octave 3	C# Octave 4	C# Octave 5
D Octave 1	[✓ D Octave 2]	[✓ D Octave 3]	D Octave 4	D Octave 5
D# Octave 1	D# Octave 2	D# Octave 3	D# Octave 4	D# Octave 5
E Octave 1	E Octave 2	E Octave 3	E Octave 4	E Octave 5
F Octave 1	F Octave 2	F Octave 3	F Octave 4	F Octave 5
F# Octave 1	[✓ F# Octave 2]	[✓ F# Octave 3]	[✓ F# Octave 4]	F# Octave 5
G Octave 1	G Octave 2	G Octave 3	G Octave 4	G Octave 5
G# Octave 1	G# Octave 2	G# Octave 3	G# Octave 4	G# Octave 5
A Octave 1	A Octave 2	A Octave 3	A Octave 4	A Octave 5
A# Octave 1	A# Octave 2	A# Octave 3	A# Octave 4	A# Octave 5
B Octave 1	B Octave 2	B Octave 3	B Octave 4	B Octave 5



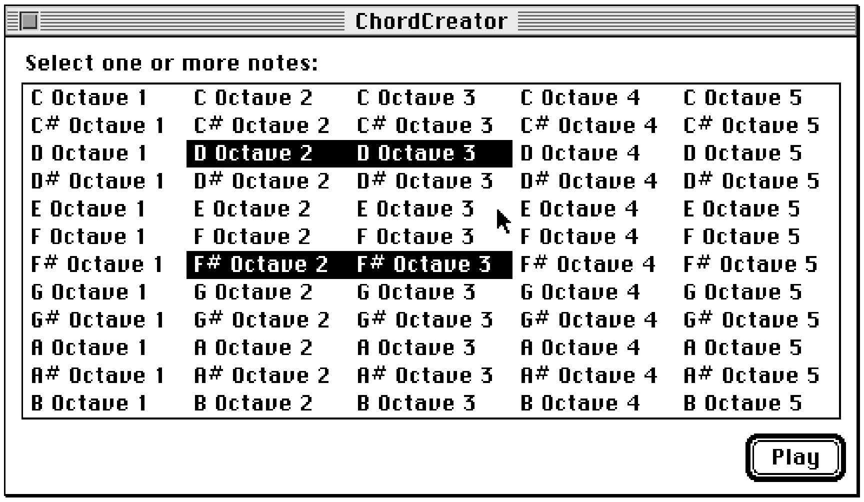
Select one or more notes:

C Octave 1	C Octave 2	C Octave 3	C Octave 4	C Octave 5
C# Octave 1	C# Octave 2	C# Octave 3	C# Octave 4	C# Octave 5
[✓ D Octave 1]	[✓ D Octave 2]	[✓ D Octave 3]	D Octave 4	D Octave 5
D# Octave 1	D# Octave 2	D# Octave 3	D# Octave 4	D# Octave 5
E Octave 1	E Octave 2	E Octave 3	E Octave 4	E Octave 5
F Octave 1	F Octave 2	F Octave 3	F Octave 4	F Octave 5
F# Octave 1	[✓ F# Octave 2]	[✓ F# Octave 3]	[✓ F# Octave 4]	F# Octave 5
G Octave 1	G Octave 2	G Octave 3	G Octave 4	G Octave 5
G# Octave 1	G# Octave 2	G# Octave 3	G# Octave 4	G# Octave 5
A Octave 1	A Octave 2	A Octave 3	A Octave 4	A Octave 5
A# Octave 1	A# Octave 2	A# Octave 3	A# Octave 4	A# Octave 5
B Octave 1	B Octave 2	B Octave 3	B Octave 4	B Octave 5

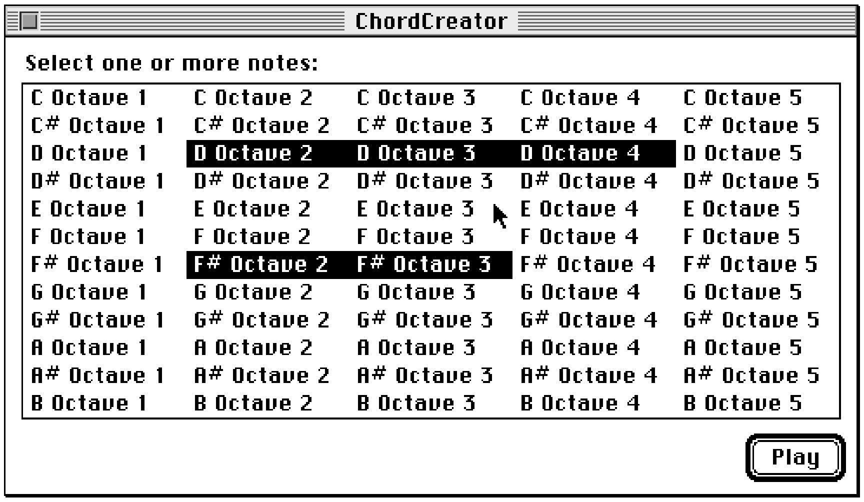
While the extend algorithm is intuitive and works well for simple lists, a more powerful algorithm for managing extensions of selections with the arrow keys is the *anchor algorithm*. This algorithm is far more difficult to implement than the extend algorithm, but allows the user more power than the extend algorithm to extend a list in whatever way is desired, and it works more intuitively both for lists that are likely to contain many discontiguous items and for multicolumn lists.

The anchor algorithm works by moving the user's selection relative to an anchor cell. The application should determine which cell to make the anchor cell by examining the last cell in the rectangular range of cells last selected by the user. If the user has pressed either the Right Arrow or Down Arrow key, the anchor cell should be the first cell in this range; otherwise, it should be the last cell. The application then finds the cell that is on the other end of the rectangular range of cells last selected by the user. It then attempts to move this cell in the direction specified by the arrow key, and it highlights all cells in the rectangle whose corners are the anchor cell and the moving cell. Figure 4-15 illustrates this process.

Figure 4-15 Response to Shift–Right Arrow using the anchor algorithm



C Octave 1	C Octave 2	C Octave 3	C Octave 4	C Octave 5
C# Octave 1	C# Octave 2	C# Octave 3	C# Octave 4	C# Octave 5
D Octave 1	**D Octave 2**	**D Octave 3**	D Octave 4	D Octave 5
D# Octave 1	D# Octave 2	D# Octave 3	D# Octave 4	D# Octave 5
E Octave 1	E Octave 2	E Octave 3	E Octave 4	E Octave 5
F Octave 1	F Octave 2	F Octave 3	F Octave 4	F Octave 5
F# Octave 1	**F# Octave 2**	**F# Octave 3**	F# Octave 4	F# Octave 5
G Octave 1	G Octave 2	G Octave 3	G Octave 4	G Octave 5
G# Octave 1	G# Octave 2	G# Octave 3	G# Octave 4	G# Octave 5
A Octave 1	A Octave 2	A Octave 3	A Octave 4	A Octave 5
A# Octave 1	A# Octave 2	A# Octave 3	A# Octave 4	A# Octave 5
B Octave 1	B Octave 2	B Octave 3	B Octave 4	B Octave 5



C Octave 1	C Octave 2	C Octave 3	C Octave 4	C Octave 5
C# Octave 1	C# Octave 2	C# Octave 3	C# Octave 4	C# Octave 5
D Octave 1	**D Octave 2**	**D Octave 3**	**D Octave 4**	D Octave 5
D# Octave 1	D# Octave 2	D# Octave 3	D# Octave 4	D# Octave 5
E Octave 1	E Octave 2	E Octave 3	E Octave 4	E Octave 5
F Octave 1	F Octave 2	F Octave 3	F Octave 4	F Octave 5
F# Octave 1	**F# Octave 2**	**F# Octave 3**	F# Octave 4	F# Octave 5
G Octave 1	G Octave 2	G Octave 3	G Octave 4	G Octave 5
G# Octave 1	G# Octave 2	G# Octave 3	G# Octave 4	G# Octave 5
A Octave 1	A Octave 2	A Octave 3	A Octave 4	A Octave 5
A# Octave 1	A# Octave 2	A# Octave 3	A# Octave 4	A# Octave 5
B Octave 1	B Octave 2	B Octave 3	B Octave 4	B Octave 5

The top window of Figure 4-15 shows two rectangular ranges of selected cells. Suppose the application determines that the range of cells last selected by the user is the range containing “D Octave 2” and “D Octave 3.” Because the user pressed Shift–Right Arrow, the application designates the first cell in this range to be the anchor cell. It then extends each row of the rectangular range one cell to the right, as shown in the bottom window of Figure 4-15.

The application must remember the anchor cell in case the user clicks another Shift–arrow key combination before making any other changes to the list. If this occurs, the application should keep the same anchor cell. Thus in Figure 4-15, if, after pressing Shift–Right Arrow, the user presses Shift–Left Arrow, then the application keeps the same anchor cell (ordinarily, if the Shift–Left Arrow keys are pressed, the last cell in the range becomes the anchor cell). The rectangular range of cells previously extended one cell to the right thus reverts to its original state. Therefore, if your application supports the anchor algorithm, the user can use Shift–arrow key combinations to extend a rectangular range of cells in any direction around an anchor cell that is determined by the first arrow key pressed.

Type Selection in a Text-Only List

In a text-only list, when the user types the name of an item in a list, your application should respond by scrolling to that item and selecting it. This behavior (allowing a user to type the name of an item in a list to select it) is known as **type selection.** Rather than requiring the user to type the entire name of a list item, however, your application should continually attempt to determine the best match in the list for the user's typing.

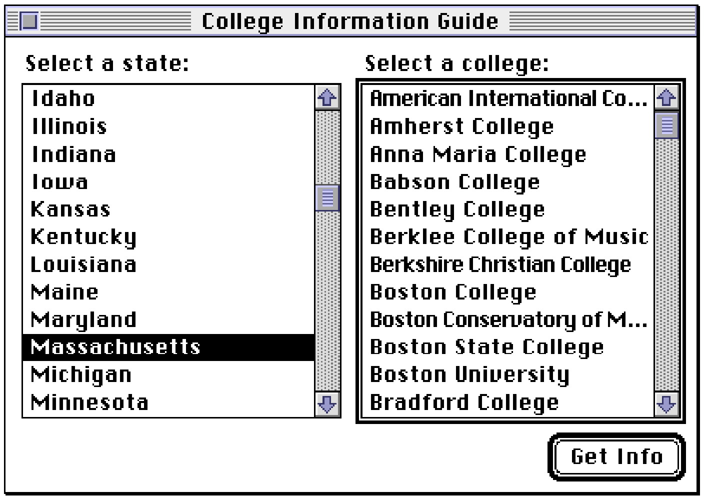
In particular, every time the user types a character, your application should add it to a string that keeps track of the characters the user has typed in searching the list. Your application should attempt to find an exact match for this string, or if no exact match exists, your application should select the first item that alphabetically follows the text indicated by the string.

Sometimes the user may start to type the name of one list item and then type the name of another. Your application should support this by automatically resetting the internal string used to keep track of the user's typing after a given amount of time has elapsed without the user's pressing a key. To compute the amount of time after which your application should reset the string, you can use a formula (provided later in this chapter) that depends on the value the user sets for the autokey threshhold in the Keyboard control panel. For users who specify a long delay until keys repeat, your application should use a long time span before it resets the internal string it uses to keep track of the user's typing.

Multiple Lists in a Window

In a window with multiple lists that support keyboard navigation, you need to show which list is the target of keyboard input. To help the user in such a window, your application should draw a 2-pixel-wide outline around the current list, that is, the list that would be affected by typing. The box should surround the entire list, including any scroll bars, and there should be 1 pixel of white space between the outline and the list's border. Figure 4-16 illustrates a window containing more than one list.

Figure 4-16 An outlined list in a window with more than one list



In Figure 4-16, the second list is outlined. Thus, the user knows that using the keyboard affects this list only. Your application should allow the user to press the Tab key to move the outline to the next list in a window. In a window with more than two lists, your application should allow the user to press Shift-Tab to move the outline to the previous list in a window.

Ordinarily, your application should not outline a list that is the only list in its window. However, if there is an editable text item in a dialog box containing a list, or if keyboard input could have some other effect, then your application should outline a list when the user can navigate it with the keyboard. The user should be able to use the Tab key to switch between a list and an editable text item; however, there is no need to outline the editable text item, since the insertion point indicates to the user that using the keyboard results in any text being inserted there.

When a window containing multiple lists is deactivated, your application should remove the outline from the current list and not replace it until the window is activated.

About the List Manager

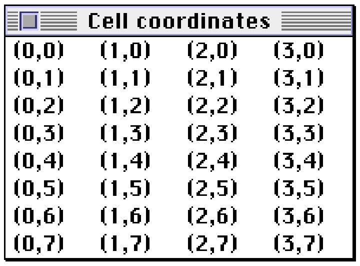
The List Manager uses a list record to keep track of information about a list. In most cases your application can get or set the information in a list record using List Manager routines. When necessary, your application can examine fields of the list record directly.

Each cell in a list can be described by a data structure of type `Cell`:

```
TYPE Cell         = Point;
```

The `Cell` data type has the same structure as the `Point` data type; however, the fields (horizontal and vertical coordinates) of a cell record have different meaning. The horizontal coordinate of a cell specifies its column number, and the vertical coordinate of a cell specifies the cell's row number. Note, however, that the first cell in a list is defined to be cell (0,0). So a cell with coordinates (3,4) is in the fourth column and fifth row. Thus you can visually identify a cell's coordinates using the formula (*column*–1, *row*–1). Figure 4-17 illustrates a list in which each cell item's text is set to the coordinates of the cell.

Figure 4-17 Coordinates of cells



```
(0,0)   (1,0)   (2,0)   (3,0)
(0,1)   (1,1)   (2,1)   (3,1)
(0,2)   (1,2)   (2,2)   (3,2)
(0,3)   (1,3)   (2,3)   (3,3)
(0,4)   (1,4)   (2,4)   (3,4)
(0,5)   (1,5)   (2,5)   (3,5)
(0,6)   (1,6)   (2,6)   (3,6)
(0,7)   (1,7)   (2,7)   (3,7)
```

A list record is defined by the `ListRect` data type.

```
TYPE ListRec   =
RECORD
   rView:      Rect;             {list's display rectangle}
   port:       GrafPtr;          {list's graphics port}
   indent:     Point;            {indent distance for drawing}
   cellSize:   Point;            {size in pixels of a cell}
   visible:    Rect;             {boundary of visible cells}
   vScroll:    ControlHandle;    {vertical scroll bar}
   hScroll:    ControlHandle;    {horizontal scroll bar}
   selFlags:   SignedByte;       {selection flags}
   lActive:    Boolean;          {TRUE if list is active}
   lReserved:  SignedByte;       {reserved}
   listFlags:  SignedByte;       {automatic scrolling flags}
   clikTime:   LongInt;          {TickCount at time of last click}
   clikLoc:    Point;            {position of last click}
   mouseLoc:   Point;            {current mouse location}
   lClikLoop:  Ptr;              {routine called by LClick}
   lastClick:  Cell;             {last cell clicked}
   refCon:     LongInt;          {for application use}
   listDefProc:                  {list definition procedure}
               Handle;
   userHandle: Handle;           {for application use}
   dataBounds: Rect;             {boundary of cells allocated}
   cells:      DataHandle;       {cell data}
   maxIndex:   Integer;          {used internally}
   cellArray:                    {offsets to data}
               ARRAY[1..1] OF Integer;
END;
   ListPtr     = ^ListRec;       {pointer to a list record}
   ListHandle  = ^ListPtr;       {handle to a list record}
```

The only fields of a list record that you need to be familiar with are the `rView`, `port`, `cellSize`, `visible`, and `dataBounds` fields.

The `rView` field specifies the rectangle in which the list's visible rectangle is located, in local coordinates of the graphics port specified by the `port` field. Note that the list's visible rectangle does not include the area needed for the list's scroll bars. The width of a vertical scroll bar (which equals the height of a horizontal scroll bar) is 15 pixels.

The `cellSize` field specifies the size in pixels of each cell in the list. Usually, you let the List Manager automatically calculate the dimensions of a cell. It determines the default vertical size of a cell by adding the ascent, descent, and leading of the port's font. (This is 16 pixels for 12-point Chicago, for example.) For best results, you should make the height of your application's list equal to a multiple of this height. The List Manager determines the default horizontal size of a cell by dividing the width of the list's visible rectangle by the number of columns in the list.

The `visible` field specifies which cells in a list are visible within the area specified by the `rView` field. The List Manager sets the `left` and `top` fields of `visible` to the coordinates of the first visible cell; however, the List Manager sets the `right` and `bottom` fields so that each is 1 greater than the horizontal and vertical coordinates of the last visible cell. For example, if a list contains 4 columns and 10 rows but only the first 2 columns and the first 5 rows are visible (that is, the last visible cell has coordinates (1,4)), the List Manager sets the `visible` field to (0,0,2,5).

The List Manager sets the `visible` field using this method so that you can test whether a cell is visible within a list by calling QuickDraw's `PtInRect` function with a given cell and the contents of this field. Also, this allows your application to compute the number of visible rows, for example, by subtracting the `top` field of `visible` from `bottom`.

The dataBounds field (located near the end of the list record) specifies the total cell dimensions of the list, including cells that are not visible. It works much like the visible field; that is, its right and bottom fields are each 1 greater than the horizontal and vertical coordinates of the last cell in the list. For example, if a list contains 4 columns and 10 rows (that is, the last cell in the list has coordinates (3,9)), the List Manager sets the dataBounds field to (0,0,4,10).

Your application seldom needs to access the remaining fields of the list record, although they are described here for your quick reference.

The indent field indicates the location, relative to the top-left corner of a cell, at which drawing should begin. For example, the default list definition procedure sets the vertical coordinate of this field to near the bottom of the cell, so that characters drawn with QuickDraw's DrawText procedure appear in the cell.

The vScroll and hScroll fields are handles to the vertical and horizontal scroll bars associated with a list. You can determine which scroll bars a list contains by checking whether these fields are NIL.

The lActive field is TRUE if a list is active or FALSE if it is inactive. You should not change the value in this field directly, but should use the LActivate procedure to activate or inactivate a list.

The selFlags field specifies the algorithm that the List Manager uses to select cells in response to a click in a list. This field is described in more detail in "Customizing Cell Highlighting" on page 4-38.

The listFlags field indicates whether automatic vertical and horizontal scrolling is enabled. If automatic scrolling is enabled, then a list scrolls when the user clicks a cell but then drags the cursor out of the rectangle specified by the rView field. For example, if a user drags the cursor below this field, the list scrolls downward. By default, the List Manager enables vertical automatic scrolling if your list has a vertical scroll bar; it enables horizontal scrolling if your list has a horizontal scroll bar. Your application can disable or enable automatic scrolling by using the following bit values:

```
CONST lDoVAutoScroll = 2;  {allows vertical autoscrolling}
      lDoHAutoScroll = 1;  {allows horizontal autoscrolling}
```

The clikTime and clikLoc fields indicate the time at which the user last clicked the mouse and the local coordinates of the click. The lastClick field (located later in the list record) indicates the cell coordinates of the last click. You can access the value in the lastClick field via the LLastClick function. If your application depends on the accuracy of the values in these fields, and if your application treats keyboard selection of list items identically to mouse selection of list items, then it should update the values of these fields after highlighting a cell in response to a keyboard event. (In particular, this is necessary if your application implements the anchor algorithm for extending cell selections with the arrow keys.)

The `mouseLoc` field indicates the current location of the cursor in local coordinates (v, h). Ordinarily, you should use the Event Manager's `GetMouse` procedure to obtain this information, but this field may be more convenient to access from within a click-loop procedure (explained next).

The `lClikLoop` field usually contains `NIL`, but your application may place a pointer to a custom click-loop procedure in this field. A click-loop procedure manages the selection of list items and the scrolling of a list in response to a mouse click in the visible rectangle of a list. It is unlikely that your application will need to define its own click-loop procedures, because the List Manager's `LClick` function provides a default click-loop procedure that uses a robust algorithm to respond to mouse clicks. Your application needs to use a custom click-loop procedure only if it needs to perform some special processing while the user drags the cursor after clicking in a list. For more information on click-loop procedures, see "Click-Loop Procedures" on page 4-100.

The `refCon` and `userHandle` fields are for your application's use. You might, for example, use the `refCon` field to store the value of the A5 register, or to keep track of whether a list should be outlined. Typically, an application uses the `userHandle` field to store a handle to some additional storage associated with a list, but you can use the field in any way that is convenient for your application.

The `listDefProc` field contains a handle to the code used by the list definition procedure.

The `cells` field contains a handle to data that stores the list contents. The handle is declared like this:

```
TYPE  DataArray      = PACKED ARRAY[0..32000] OF Char;
      DataPtr        = ^DataArray;
      DataHandle     = ^DataPtr;
```

Because of the way the `cells` field is defined, no list can contain more than 32,000 bytes of data. The List Manager slows down considerably when a list approaches this size, and the List Manager may fail if you attempt to store more data than this in a list.

The List Manager uses the `cellArray` field to store offsets to data in the relocatable block specified by the `cells` field.

Your application will never need to access the `lReserved` and `maxIndex` fields.

▲ **WARNING**
Your application should not change the `cells` field directly or access the information in the `cellArray` field directly. The List Manager provides routines that you can use to manipulate the information in a list. ▲

Using the List Manager

This section explains how you can take advantage of the List Manager's features and how you can customize lists that your application creates by providing support for features not built into the List Manager. In particular, this section explains how you can

- use the `LNew` function and the `LDispose` procedure to create a list within a rectangle in a window and then dispose of that list
- add rows and columns to a list by using the `LAddColumn` and `LAddRow` functions along with the `LSetCell` procedure; delete them by using the `LDelColumn` and `LDelRow` procedures; and temporarily disable drawing of a list while adding multiple columns or rows by using the `LSetDrawingMode` procedure
- call the `LClick` function to let the List Manager automatically respond to mouse clicks in a list by scrolling the list and changing the selection as appropriate; and call the `LUpdate` and `LActivate` procedures to respond to update and activate events
- use the `LGetSelect` function and the `LSetSelect` procedure to get information about which cells are selected or to change the selection; and use the `LAutoScroll` and `LScroll` procedures to scroll to a particular cell
- customize the algorithm that the List Manager uses to highlight cells in response to a mouse click by modifying the `selFlags` field of the list record
- manipulate list items by using the `LAddToCell`, `LClrCell`, `LGetCellDataLocation`, and `LGetCell` procedures
- search through a list for a particular item by using the `LSearch` function and writing a custom match function
- respond to arrow-key and other key-down events to change or extend the selection
- manage multiple lists within the same window or dialog box by drawing an outline around the list that would be affected by keyboard input using the `refCon` field of the list record to link the lists
- write your own list definition procedure

Creating a List

To create a list, you can use the `LNew` function. Listing 4-1 shows a typical use of the `LNew` function to create a vertically scrolling list in a rectangular space in a window.

Listing 4-1 Creating a list with a vertical scroll bar

```
FUNCTION MyCreateVerticallyScrollingList
                  (myWindow: WindowPtr; myRect: Rect;
                   columnsInList: Integer;
                   myLDEF: Integer): ListHandle;
CONST
   kDoDraw             = TRUE;  {always draw list after changes}
   kNoGrow             = FALSE; {don't leave room for size box}
   kIncludeScrollBar   = TRUE;  {leave room for scroll bar}
   kScrollBarWidth     = 15;    {width of vertical scroll bar}
VAR
   myDataBounds:        Rect;    {initial dimensions of the list}
   myCellSize:          Point;   {size of each cell in list}
BEGIN
   {specify dimensions of the list}
   {start with a list that contains no rows}
   SetRect(myDataBounds, 0, 0, columnsInList, 0);

   {let the List Manager calculate the size of a cell}
   SetPt(myCellSize, 0, 0);

   {adjust the rectangle to leave room for the scroll bar}
   myRect.right := myRect.right - kScrollBarWidth;

   {create the list}
   MyCreateVerticallyScrollingList :=
      LNew(myRect, myDataBounds, myCellSize, myLDEF, myWindow,
           kDoDraw, kNoGrow, NOT kIncludeScrollBar,
           kIncludeScrollBar);
END;
```

The `LNew` function called in the last line of Listing 4-1 takes a number of parameters that let you specify the characteristics of the list you wish to create.

The first parameter to `LNew` sets the rectangle for the list's visible rectangle, specified in local coordinates of the window specified in the fifth parameter to `LNew`. Because this rectangular area does not include room for scroll bars, the `MyCreateVerticallyScrollingList` function adjusts the right of this rectangle to leave enough room.

The second parameter to `LNew` specifies the data bounds of the list. By setting the `topLeft` field of this rectangle to (0,0), you can use the `botRight` field to specify the number of columns and rows you want in the list. The `MyCreateVerticallyScrollingList` function initially creates a list of no rows. While your application is free to preallocate rows when creating a list, it is often easier to only preallocate columns and then add rows after creating the list, as described in the next section.

The third parameter is the size of a cell. By setting this parameter to (0,0), you let the List Manager compute the size automatically. The algorithm the List Manager uses to compute this size is given in the discussion of the `cellSize` field of the list record in "About the List Manager" beginning on page 4-22.

To specify that you wish to use the default list definition procedure, pass 0 as the fourth parameter to `LNew`. To use a custom list definition procedure, pass the resource ID of the list definition procedure. Note that the code for the appropriate list definition procedure is loaded into your application's heap; the code for the default list definition procedure is about 150 bytes in size.

In the sixth parameter to `LNew`, your application can specify whether the List Manager should initially enable the automatic drawing mode. When this mode is enabled, the List Manager always redraws the list after changes. Usually, your application should set this parameter to `TRUE`. This does not preclude your application from temporarily disabling the automatic drawing mode.

The last three parameters to `LNew` specify whether the List Manager should leave room for a size box, whether it should include a horizontal scroll bar, and whether it should include a vertical scroll bar. Note that while the List Manager draws scroll bars automatically, it does not draw the grow icon in the size box. Usually, your application can draw the grow icon by calling the Window Manager's `DrawGrowIcon` procedure.

The `LNew` function creates a list according your specifications and returns a handle to the list's list record. Your application uses the returned handle to refer to the list when using other List Manager routines.

Lists are often used in dialog boxes. Because the Control Manager does not define a control for lists, you must define a list in a dialog item list as a user item. Listing 4-2 shows an application-defined procedure that creates a one-column, text-only list in a dialog box.

Listing 4-2 Installing a list in a dialog box

```
FUNCTION MyCreateTextListInDialog (myDialog: DialogPtr;
                                   myItemNumber: Integer)
                                   : ListHandle;
CONST
   kTextLDEF = 0;                {resource ID of default LDEF}
VAR
   myUserItemRect:   Rect;       {enclosure of user item}
   myUserItemType:   Integer;    {for GetDialogItem}
   myUserItemHdl:    Handle;     {for GetDialogItem}
BEGIN
   GetDialogItem(myDialog, myItemNumber, myUserItemType,
                  myUserItemHdl, myUserItemRect);
   MyCreateTextListInDialog :=
      MyCreateVerticallyScrollingList(myDialog, myUserItemRect,
                                          1, kTextLDEF);
END;
```

The `MyCreateTextListInDialog` function defined in Listing 4-2 calls the `MyCreateVerticallyScrollingList` function defined in Listing 4-1, after finding the rectangle in which to install the new list by using the Dialog Manager's `GetDialogItem` procedure. For more information on the Dialog Manager, see *Inside Macintosh: Macintosh Toolbox Essentials*.

The List Manager does not automatically draw a 1-pixel border around a list. Listing 4-3 shows an application-defined procedure that draws a border around a list.

Listing 4-3 Drawing a border around a list

```
PROCEDURE MyDrawListBorder (myList: ListHandle);
VAR
   myBorder:      Rect;          {box for list}
   myPenState:    PenState;      {current status of pen}
BEGIN
   myBorder := myList^^.rView;   {get view rectangle}
   GetPenState(myPenState);      {store pen state}
   PenSize(1, 1);                {set pen to 1 pixel}
   InsetRect(myBorder, -1, -1);  {adjust rectangle for framing}
   FrameRect(myBorder);          {draw border}
   SetPenState(myPenState);      {restore old pen state}
END;
```

The `MyDrawListBorder` procedure defined in Listing 4-3 uses standard QuickDraw routines to save the state of the pen, set the pen size to 1 pixel, draw the border, and restore the pen state.

When you are finished using a list, you should dispose of it using the `LDispose` procedure, passing a handle to the list as the only parameter. The `LDispose` procedure disposes of the list record, as well as the data associated with the list; however, it does not dispose of any application-specific data that you might have stored in a relocatable block specified by the `userHandle` field of the list record. Thus, if you use this field to store a handle to a relocatable block, you should dispose of the relocatable block before calling `LDispose`.

Adding Rows and Columns to a List

Your application can choose to preallocate the cells it needs when it creates a list. For example, an application might preallocate the columns it needs, and then add rows to the list one by one. Other applications might create a list and add both rows and columns to it later. Regardless of the technique your application uses to create its cells, it can set the data in a cell by using the `LSetCell` procedure.

You specify the data, the length of the data, the location of the cell whose data you wish to set, and a handle to the list containing the cell, as parameters to the `LSetCell` procedure. Listing 4-4 demonstrates an application-defined procedure that adds rows to a one-column list based on the contents of a string list resource. The `MyAddItemsFromStringList` procedure adds each row to the list using the `LAddRow` function, then sets the data of the cell in the first (and only) column of the newly added row using the `LSetCell` procedure.

Listing 4-4 Adding items from a string list to a one-column, text-only list

```
PROCEDURE MyAddItemsFromStringList (myList: ListHandle;
                                    stringListID: Integer);
VAR
   index:      Integer;             {index within string list}
   rowNum:     Integer;             {row number to add string to}
   myString:   Str255;              {string to add}
   aCell:      Cell;                {cell to store string in}
BEGIN
                                    {compute new row number}
   rowNum := myList^^.dataBounds.bottom;
   index := 1;                      {start with first string}
   REPEAT
      GetIndString(myString, stringListID, index);
      IF myString <> '' THEN
      BEGIN    {add new row for string}
         {specify #rows to add, row number of first new row}
         rowNum := LAddRow(1, rowNum, myList);
         {prepare to set cell data--specify }
         { the cell's column number, row number}
         SetPt(aCell, 0, rowNum);
         {set cell data to string}
         LSetCell(@myString[1], Length(myString), aCell,
                  myList);
      END;
      rowNum := rowNum + 1;
      index := index + 1;
   UNTIL myString = '';
END;
```

The MyAddItemsFromStringList procedure defined in Listing 4-4 adds strings from a string list resource to the end of a list. It keeps track of the index of the string in the string list with the index variable, and it tracks the number of the new row to add in the rowNum variable.

The MyAddItemsFromStringList procedure adds a new row by calling the LAddRow function. The first parameter to LAddRow specifies the number of rows to add, and the second parameter specifies the row number of the first new row. LAddRow returns the row number of the first row added, which differs from the second parameter only if that parameter specifies a row number that is out of range.

After creating a new row, MyAddItemsFromStringList sets the cell in the first column of the added row to the text contained within the string. Note that the procedure does not copy the length byte of the string.

To add columns to a list, your application can use the LAddColumn function, which works just like LAddRow.

To delete a row or column from a list, your application can call the LDelRow procedure or the LDelColumn procedure. The first parameter of each of these procedures is the number of rows (or columns) to delete, and the second parameter is the row or column number of the first to be deleted. For example, this code deletes the first row of a list:

```
LDelRow(1, 0, myList);{#rows to delete, starting row number}
```

When making many changes to a list, your application should temporarily disable the automatic drawing mode (unless the list is in a window that is not yet visible). To do so, call the LSetDrawingMode procedure to turn off the automatic drawing mode, make the changes to the list, turn the automatic drawing mode back on, and redraw the list (by invalidating a rectangle containing the list and its scroll bars and later calling the LUpdate procedure when your application receives an update event). You might do these steps as follows:

```
LSetDrawingMode (FALSE, myList);
{...(make changes to the list)...}
LSetDrawingMode (TRUE, myList);
InvalRect(myList^^.rView);
IF (myList^^.vScroll <> NIL) THEN
   InvalRect(myList^^.vScroll^^.contrlRect);
IF (myList^^.hScroll <> NIL) THEN
   InvalRect(myList^^.hScroll^^.contrlRect);
```

Responding to Events Affecting a List

Your application must respond to several different types of events involving a list by calling appropriate List Manager routines. If a mouse-down event occurs in a list, your application should call the LClick function. If your application receives an update event, and some part of the list is within the update region, then it should call the LUpdate procedure. If a window containing a list is activated or deactivated, your application should activate or deactivate the list by calling the LActivate procedure. Finally, if a key-down event occurs, your application may need to call its own internal procedures to scroll the list or select items as necessary. This section explains how to handle mouse-down, update, and activate events; for information on handling key-down events, see “Supporting Keyboard Navigation of Lists” on page 4-45.

The LClick function automatically responds to a mouse-down event by handling user interaction until the user releases the mouse button. The List Manager performs any scrolling as necessary and changes the selection as appropriate. After handling the event, the LClick function returns TRUE if the click was a double click. Listing 4-5 shows an application-defined procedure that uses the LClick function to handle mouse-down events in a list.

Listing 4-5 Responding to a mouse-down event in a list

```
PROCEDURE MyHandleMouseDownInList (theEvent: EventRecord;
                                   theList: ListHandle);
BEGIN
   SetPort(theList^^.port);
   GlobalToLocal(theEvent.where);
   IF LClick(theEvent.where, theEvent.modifiers, theList) THEN
      MyDoubleClick(theList);
END;
```

In response to a double click, your application should simulate the selection of the default button if there is one. If your dialog box does not contain a default button, then your application can respond to a double click with some other appropriate behavior.

Listing 4-6 illustrates an application-defined procedure that responds to an update event affecting a list.

Listing 4-6 Responding to an update event in a list

```
PROCEDURE MyUpdateList (theList: ListHandle);
BEGIN
   SetPort(theList^^.port);      {set up the drawing environment}
                                 {update list and scroll bars}
   LUpdate(theList^^.port^.visRgn, theList);
   MyDrawListBorder(theList);    {draw border around list}
END;
```

Your list update procedure might also do some other drawing appropriate to a particular list. For example, if your application supports multiple lists in a window, then your list-updating procedure should redraw an outline around the current list in response to an update event. For more information on outlining the current list, see "Outlining the Current List" on page 4-53.

Note that the call to the `LUpdate` procedure must be bracketed by calls to the Window Manager's `BeginUpdate` and `EndUpdate` procedures. See the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for more information.

In response to an activate event, your application should call `LActivate` for each list in the window. For example, this code deactivates a list:

```
LActivate (FALSE, myList);
```

To activate a list, pass `TRUE` as the first parameter to `LActivate`.

Working With List Selections

The List Manager provides routines that make it easy to determine what the selection is or to change the selection, whether your list allows just one item to be selected at a time or allows many items to be selected. The List Manager also provides a routine that allows you to automatically scroll the first selected cell to the upper-left corner of the list's visible rectangle. In addition, you can write your own routine to scroll a list just enough so that a particular cell is visible.

Your application can use the `LGetSelect` function to determine whether a given cell is selected or to find the next selected cell. Your application can use the `LSetSelect` procedure to select or deselect a given cell.

Listing 4-7 shows an application-defined procedure that finds the first cell in a selection.

Listing 4-7 Finding the first selected cell in a list

```
FUNCTION MyGetFirstSelectedCell (theList: ListHandle;
                                 VAR theCell: Cell): Boolean;
BEGIN
   SetPt(theCell, 0, 0);
   MyGetFirstSelectedCell := LGetSelect(TRUE, theCell, theList);
END;
```

The first parameter (TRUE) passed to the LGetSelect function indicates that LGetSelect should search the list (beginning with the cell specified in the second parameter) for the first selected cell. If you pass TRUE as the first parameter, LGetSelect sets the cell specified in the second parameter to the coordinates of the first selected cell that it finds, or it returns FALSE if no cells including or after the cell specified by the second parameter are selected. If you pass FALSE as the first parameter to LGetSelect, then the function returns TRUE only if the cell specified in the second parameter is selected. The MyGetFirstSelectedCell function defined in Listing 4-7 thus returns TRUE only if at least one cell is selected, in which case the second parameter to the function is set to the coordinates of that cell.

Finding the last selected cell in a list is slightly more complex. Listing 4-8 illustrates how this might be done.

Listing 4-8 Finding the last selected cell in a list

```
PROCEDURE MyGetLastSelectedCell (theList: ListHandle;
                                 VAR theCell: Cell);
VAR
   aCell:           Cell;
   moreCellsInList:  Boolean;
BEGIN
   IF MyGetFirstSelectedCell(theList, aCell) THEN
   REPEAT
      theCell := aCell;
      moreCellsInList := LNextCell(TRUE, TRUE, aCell, theList);
   UNTIL NOT LGetSelect(TRUE, aCell, theList);
END;
```

The MyGetLastSelectedCell procedure goes from one selected cell to the next until there are no more selected cells. It calls the LNextCell function to move from one cell to the next cell in the list. If it did not do this, then the procedure would loop infinitely, since LGetSelect would repeatedly return TRUE for the first selected cell. The first two parameters to LNextCell indicate whether the function should return the next cell in the current row, the next cell in the current column, or, if both are set to TRUE, the next cell regardless of location.

Your application can use the `LSetSelect` procedure to set or deselect a cell by passing `TRUE` or `FALSE`, respectively, as the first parameter to the routine. Listing 4-9 illustrates a useful procedure that uses `LSetSelect` and `LGetSelect` to select a single cell in a list while deselecting all other cells.

Listing 4-9 Selecting a cell and deselecting other cells

```
PROCEDURE MySelectOneCell (theList: ListHandle; theCell: Cell);
VAR
   nextSelectedCell:    Cell;
   moreCellsInList:     Boolean;
BEGIN
   IF MyGetFirstSelectedCell(theList, nextSelectedCell) THEN
   WHILE LGetSelect(TRUE, nextSelectedCell, theList) DO
   BEGIN                  {move to next selected cell...}
      IF (nextSelectedCell.h <> theCell.h) OR
            (nextSelectedCell.v <> theCell.v) THEN
                          {...and remove cell from selection}
         LSetSelect(FALSE, nextSelectedCell, theList)
      ELSE
         moreCellsInList :=
                          {move to next cell}
            LNextCell(TRUE, TRUE, nextSelectedCell, theList);
   END;
   LSetSelect(TRUE, theCell, theList);
END;
```

The `MySelectOneCell` procedure defined in Listing 4-9 deselects each selected cell, except that if it encounters the cell that is ultimately to be selected, then it does not deselect that cell. This prevents an annoying flickering that would otherwise occur if you were to call `MySelectOneCell` to select a cell already selected.

The List Manager provides the `LAutoScroll` procedure to enable your application to scroll the first selected cell to the upper-left corner of the list's visible rectangle—for example:

```
LAutoScroll(myList);
```

Sometimes, you might want your application to scroll a list just enough so that a certain cell (such as a cell the user has just selected using the keyboard) is visible. For example, this is how the Standard File Package responds if the user presses the Down Arrow key when the currently selected item is on the bottom of the list's visible rectangle. You can mimic this effect by calling the `LScroll` procedure, which requires that your application indicate how many columns and rows to scroll. Negative numbers indicate scrolling up or to the left. Positive numbers indicate scrolling down or to the right. Listing 4-10 illustrates the use of the `LScroll` procedure.

Listing 4-10 Scrolling so that a particular cell is visible

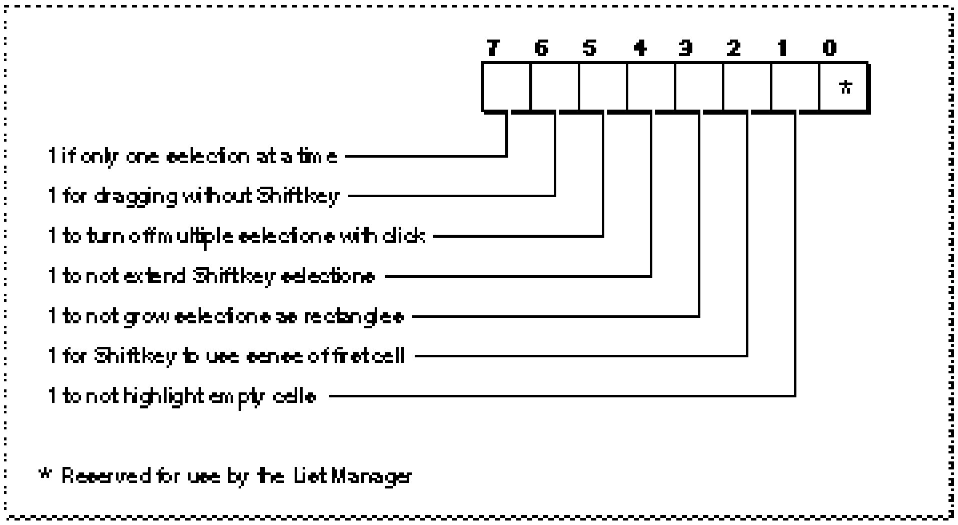
```
PROCEDURE MyMakeCellVisible (theList: ListHandle; theCell: Cell);
VAR
   visibleRect:      Rect;    {rectangle enclosing visible cells}
   dCols, dRows:     Integer; {number of rows to scroll}
BEGIN
   visibleRect := theList^^.visible;
   IF NOT PtInRect(theCell, visibleRect) THEN
   BEGIN                      {cell is not already visible}
      WITH theCell, visibleRect DO
      BEGIN
         IF h > right - 1 THEN
            dCols := h - right + 1     {move to left}
         ELSE IF h < left THEN
            dCols := h - left;         {move to right}
         IF v > bottom - 1 THEN
            dRows := v - bottom + 1    {move up}
         ELSE IF v < top THEN
            dRows := v - top;          {move down}
      END;
      LScroll(dCols, dRows, theList);
   END;
END;
```

The `MyMakeCellVisible` procedure defined in Listing 4-10 simply computes the number of cells between the last visible row and column and the selected cell. Note that the last visible column for a list is equal to `theList^^.visible.right – 1`, and the last visible row is `theList^^.visible.bottom – 1`.

Customizing Cell Highlighting

You can change the `selFlags` field of the list record to modify the algorithm the List Manager uses to select cells in response to a mouse click. “Selection of List Items” beginning on page 4-9 explains the different customizations you can make. Figure 4-18 illustrates the bits in the `selFlags` field.

Figure 4-18 Selection flags


7 6 5 4 3 2 1 0
*
1 if only one selection at a time
1 for dragging without Shiftkey
1 to turn off multiple selections with click
1 to not extend Shiftkey selections
1 to not grow selections as rectangles
1 for Shiftkey to use sense of first cell
1 to not highlight empty cells
* Reserved for use by the List Manager

The List Manager defines constants for each flag:

```
CONST
   lOnlyOne    = -128;  {allow only 1 item to be selected at once}
   lExtendDrag = 64;    {enable selection of multiple items }
                        { by dragging without Shift}
   lNoDisjoint = 32;    {prevent discontiguous selections }
                        { using Command key}
   lNoExtend   = 16;    {deselect all items before }
                        { responding to Shift-click}
   lNoRect     = 8;     {select all items in cursor's path }
                        { during Shift-drag}
   lUseSense   = 4;     {allow user to use Shift key to }
                        { deselect one or more items}
   lNoNilHilite= 2;     {disable highlighting of empty cells}
```

When you create a list, the List Manager clears all bits in the `selFlags` fields. To change any of these defaults, set the appropriate bits in the `selFlags` field. For example, this code sets the `selFlags` field so that only one selection is allowed in a list:

```
myList^^.selFlags := lOnlyOne;
```

Many of the constants are often used additively. For example, your application might allow the user to select a new range of cells simply by dragging over them, as shown in the following code:

```
myList^^.selFlags := lExtendDrag + lNoDisjoint + lNoExtend
                     + lNoRect + lUseSense;
```

The `lExtendDrag` constant allows users to select a range of items simply by dragging the cursor. Ordinarily, if the user clicks one cell and drags the cursor to another, only the last cell remains set.

The `lNoDisjoint` constant ensures that only one range of cells can be selected.

The `lNoExtend` constant disables the List Manager feature that responds to a Shift-click by selecting all cells in the range of the newly clicked cell and the first (or last) selected cell. Instead, the List Manager simply deselects all cells in the range if this bit is set.

To allow the user to select a number of cells simply by moving the cursor over them, you can set the bit corresponding to the `lNoRect` constant. This prevents the deselection of cells should the user drag the cursor first in one direction and then the other.

You can set the bit corresponding to the `lUseSense` constant so that if a user Shift-clicks a selected cell, the cell is deselected. Ordinarily, Shift-clicking a selected cell has no effect.

You might also wish to make the Shift key work just like the Command key in your application. You can accomplish that with the following code:

```
myList^^.selFlags := lNoRect + lNoExtend + lUseSense;
```

The `lNoNilHilite` constant is somewhat different from the others, in that it affects the display of a list, not the way that the List Manager selects items in response to a click. If the bit corresponding to this constant is set, then the List Manager does not select or highlight cells that do not contain any data.

Manipulating List Cells

In addition to the `LSetCell` procedure, the List Manager provides four procedures, `LAddToCell`, `LClrCell`, `LGetCellDataLocation`, and `LGetCell`, that allow you to manipulate cell item data. You can use the `LAddToCell` procedure to append data to list items and the `LClrCell` procedure to remove all data from a list item. The `LGetCellDataLocation` procedure indicates the location of the beginning of a cell's data within the `cells` field of the list record as well as the length of that data, and the `LGetCell` procedure copies a cell's data to a buffer that your application specifies.

Listing 4-11 illustrates the use of `LClrCell` to clear the data from all cells in a list.

Listing 4-11 Clearing all cell data

```
PROCEDURE MyClearAllCellData (myList: ListHandle);
VAR
   aCell: Cell;
BEGIN
   SetPt(aCell, 0, 0);
   REPEAT
      LClrCell(aCell, myList);
   UNTIL NOT LNextCell(TRUE, TRUE, aCell, myList);
END;
```

Because `LClrCell` simply does nothing if passed a cell not in the list, the `MyClearAllCellData` procedure defined in Listing 4-11 will not crash when attempting to clear the first cell even if there are no cells in the list.

Listing 4-12 uses the `LGetCell` procedure to return the data of a specific cell.

Listing 4-12 Getting a copy of the data of a cell

```
PROCEDURE MyGetCellData (dataPtr: Ptr; VAR dataLen: Integer;
                         aCell: Cell; myList: ListHandle);
BEGIN
   LGetCell(dataPtr, dataLen, aCell, myList);
END;
```

The `LGetCell` procedure copies cell data to memory beginning at the location specified by `dataPtr`. It copies only the number of bytes specified by the value passed in the `dataLen` parameter; it returns in that parameter the number of bytes actually copied.

Because the `LGetCell` procedure duplicates existing bytes of memory, if your application needs to access a cell's data but does not need to manipulate the data, then it should use the `LGetCellDataLocation` procedure to access cell data directly. Listing 4-13 uses the `LGetCellDataLocation` procedure to get a cell's data.

Listing 4-13 Directly accessing a cell's data

```
PROCEDURE MyGetDirectAccessToCellData
                  (VAR offset: Integer; VAR len: Integer;
                   aCell: Cell; myList: ListHandle);
BEGIN
   LGetCellDataLocation(offset, len, aCell, myList);
END;
```

The `LGetCellDataLocation` procedure simply returns in the `offset` and `len` parameters the offset and length of the appropriate cell's data within the `cells` field of the list record.

Listing 4-14 shows an application-defined procedure that uses `LGetCellDataLocation` in conjunction with the `LSetCell` procedure and the `LAddRow` function to add a new string to a one-column, alphabetical text-only list. To compare two strings, the procedure uses the Text Utilities `CompareText` function, which requires that data be specified by a pointer and length, thus making `LGetCellDataLocation` perfect for this purpose. For more information on the `CompareText` function, see *Inside Macintosh: Text*.

Listing 4-14 Adding an item to a one-column, alphabetical text list

```
PROCEDURE MyAddItemAlphabetically (myList: ListHandle; myString: Str255);
VAR
   found:      Boolean;                     {flag variable}
   myRows:     Integer;                     {number of rows in list}
   currentRow: Integer;                     {row being examined}
   cellDataOffset, cellDataLength: Integer;  {data being compared to string}
   aCell:      Cell;                        {cell coordinates}
BEGIN
   found := FALSE;                          {initialize flag variable}
   WITH myList^^.dataBounds DO
      myRows := bottom - top;               {compute number of rows}
   currentRow := -1;                        {start before first row}
   WHILE NOT found DO
   BEGIN                                    {try to insert before next row}
      currentRow := currentRow + 1;         {move to next row}
      IF currentRow = myRows THEN           {past the end of the list?}
         found := TRUE                      {insert string at this row}
      ELSE
      BEGIN
         SetPt(aCell, 0, currentRow);       {prepare to check cell data}
                                            {find location of data}
         LGetCellDataLocation(cellDataOffset, cellDataLength, aCell, myList);
         HLockHi(Handle(myList^^.cells));   {lock list data in memory}
         IF CompareText(@myString[1],       {skip length byte of string}
                        Ptr(ORD4(myList^^.cells^) + cellDataOffset),
                        Length(myString), cellDataLength, gitl2Hdl) = -1 THEN
            found := TRUE;                  {new string should precede }
                                            { this row's string}
         HUnlock(Handle(myList^^.cells));   {unlock list data}
      END;
   END;
                                            {add new row for string}
   currentRow := LAddRow(1, currentRow, myList);
   SetPt(aCell, 0, currentRow);             {prepare to set cell data}
                                            {set data}
   LSetCell(@myString[1], Length(myString), aCell, myList);
END;
```

The `MyAddItemAlphabetically` procedure defined in Listing 4-14 simply compares a string to the text in each row of a list, until the string follows the row text alphabetically or until there are no more rows, that is, the row number (which is 0-based) is equal to the number of rows, in which case the string is appended to the end of the list.

Searching a List for a Particular Item

Sometimes, your application might need to search through a list for a particular item. For example, your application might need to search a list of pictures to see which cell contains a certain picture, or your application might wish to search for an item that matches a certain string. You can use the `LSearch` function and specify your own match function to make this possible.

The `LSearch` function returns `TRUE` if it is able to find the specified data in a cell greater than or equal to the specified cell. If it does find the data, it also returns the coordinates of the cell that contains the data.

In addition to specifying the cell to search, your application also specifies a pointer to a match function, the data to search for, and the length of the data, as parameters to the `LSearch` function.

If your application specifies `NIL` for the match function, the `LSearch` function searches the list for the first cell whose data matches the specified data. In particular, the `LSearch` function calls the Text Utilities `IUMagIDString` function to compare each cell's data with the specified data until `IUMagIDString` returns 0. Because `IUMagIDString` compares strings for equality without regard for secondary ordering, using this default match function is useful only for text-only lists. For more information on `IUMagIDString`, see *Inside Macintosh: Text*.

Your application can use a different match function from `IUMagIDString` as long as it is defined just like `IUMagIDString`. For example, your application could use the `IUMagString` function so that secondary ordering is taken into consideration. To do so, your application might use the following code:

```
found := LSearch(myData, myLength, @IUMagString, myCell, myList);
```

You can also write your own match function. Listing 4-15 shows an example match function.

Listing 4-15 A match function

```
FUNCTION MySearchPartialMatch
                  (cellDataPtr, searchDataPtr: Ptr;
                   cellDataLen, searchDataLen: Integer): Integer;
BEGIN
   IF (cellDataLen > 0) AND (cellDataLen >= searchDataLen) THEN
      MySearchPartialMatch :=
                        IUMagIDString(cellDataPtr, searchDataPtr,
                                     searchDataLen, searchDataLen)
   ELSE
      MySearchPartialMatch := 1;
END;
```

Your match function should return 0 if it finds a match and 1 otherwise. The match function defined in Listing 4-15 works just like the default match function but allows the cell data to be longer than the data being searched for. For example, a search for the text “rose” would match a cell containing the text “Rosebud”.

Listing 4-16 defines a more complex but potentially more useful match function for text-only lists.

Listing 4-16 Searching a list for a cell containing certain text or the next cell alphabetically

```
FUNCTION MyMatchNextAlphabetically
               (cellDataPtr, searchDataPtr: Ptr;
                cellDataLen, searchDataLen: Integer): Integer;
BEGIN
   MyMatchNextAlphabetically := 1;     {set default return value}
   IF (cellDataLen > 0) THEN
   BEGIN
      IF IUMagIDString(cellDataPtr, searchDataPtr,
                        searchDataLen, searchDataLen) = 0 THEN
         MyMatchNextAlphabetically := 0{strings are equal}
      ELSE IF IUMagString(cellDataPtr, searchDataPtr,
                           cellDataLen, searchDataLen) = 1 THEN
      MyMatchNextAlphabetically := 0;  {search data is after }
                                       { cell data}
   END;
END;
```

Using the `LSearch` function with the `MyMatchNextAlphabetically` function defined in Listing 4-16 results in the finding of the cell that is alphabetically greater than or equal to the search text. For example, if you use the `LSearch` function with this match function to search a list of the 50 states (not including the District of Columbia) for the text “Washington, D.C.”, then the `LSearch` function returns the coordinates of the cell containing the text “West Virginia”.

Note

The `MyMatchNextAlphabetically` function defined in Listing 4-16 works only for lists that are alphabetically arranged. ◆

Supporting Keyboard Navigation of Lists

This section discusses how your application can support keyboard navigation of lists. In particular, this section first shows how your application can respond to the user's typing to select an item in a text-only list. Second, this section shows how your application can respond to the user's pressing of the arrow keys.

Supporting Type Selection of List Items

To support type selection of list items, your application must keep a record of the characters the user has typed, the time when the user last typed a character, and which list the last typed character affected. For example, the SurfWriter application defines the following four variables to keep track of this information:

```
VAR
   gListNavigateString:       {current string being searched}
               Str255;
   gTSThresh:     Integer;    {ticks before type selection resets}
   gLastKeyTime:  LongInt;    {time in ticks of last click time}
   gLastListHit:  ListHandle; {last list type selection affected}
```

The `gListNavigateString` variable stores the current status of the type selection. For example, if the user types `'h'` and then `'e'` and then `'l'` and then `'l'` and then `'o'`, this string should be `'hello'`.

The `gTSThresh` variable stores the number of ticks after which type selection resets. For example, if the user has typed `'hello'` but then waits more than this amount of time before typing `'g'`, the SurfWriter application sets `gListNavigateString` to `'g'`, not to `'hellog'`. The value of `gTSThresh` is dependent on the value the user sets for "Delay Until Repeat" in the Keyboard control panel. SurfWriter also resets the type selection if the user begins typing in a different list from the list last typed in. Thus, if the difference between the current tick count and the `gLastKeyTime` variable is greater than `gTSThresh`, or if `gLastListHit` is not equal to the current list, then the SurfWriter application must reset the type selection.

Listing 4-17 shows how the SurfWriter application initializes or resets its type-selection variables.

Listing 4-17 Resetting variables related to type selection

```
PROCEDURE MyResetTypeSelection;
CONST
   KeyThresh = $18E;    {location of low-memory word}
   kMaxKeyThresh = 120; {120 ticks = 2 seconds}
TYPE
   IntPtr = ^Integer;   {for accessing low memory}
BEGIN
   gListNavigateString := '';          {reset navigation string}
   gLastListHit := NIL;                {remember active list}
   gLastKeyTime := 0;                  {no keys yet hit}
   gTSThresh := 2 * IntPtr(KeyThresh)^;{update type-selection }
                                       { threshold}
   IF gTSThresh > kMaxKeyThresh THEN
      gTSThresh := kMaxKeyThresh;      {set threshold to maximum}
END;
```

The `MyResetTypeSelection` procedure defined in Listing 4-17 initializes three of the variables to default values and sets the `gTSThresh` variable to twice the value of the system global variable `KeyThresh`, up to a maximum of 120 ticks. By using the same formula as `MyResetTypeSelection` for computing the type-selection threshold, you make sure your application is consistent with other applications as well as with the Finder. The SurfWriter application calls the `MyResetTypeSelection` procedure when it starts up and when it wishes to reset the type selection because the type-selection threshold has expired. It also calls the procedure whenever it receives a resume event, because the user might have used the Keyboard control panel, in which case SurfWriter needs to update the value of the type-selection threshold.

Having initialized variables related to type selection, the SurfWriter application needs to respond to appropriate key-down events. Listing 4-18 illustrates an application-defined procedure that does this.

Listing 4-18 Selecting an item in response to a key-down event

```
PROCEDURE MyKeySearchInList (theList: ListHandle; theEvent: EventRecord);
VAR
   newChar: Char;                      {character to add to search string}
   theCell: Cell;                      {cell containing found string}
BEGIN
   newChar := CHR(BAnd(theEvent.message, charCodeMask));
   IF (gLastListHit <> theList) OR
            (theEvent.when - gLastKeyTime >= gTSThresh) OR
            (Length(gListNavigateString) = 255) THEN
      MyResetTypeSelection;
   gLastListHit := theList;            {remember list keyed in}
   gLastKeyTime := theEvent.when;      {record time of key-down event}
                                       {set length of string}
   gListNavigateString[0] := Char(Length(gListNavigateString) + 1);
                                       {add character to string}
   gListNavigateString[Length(gListNavigateString)] := newChar;

   SetPt(theCell, 0, 0);
   IF LSearch(@gListNavigateString[1], Length(gListNavigateString),
               @MyMatchNextAlphabetically, theCell, theList) THEN
   BEGIN
                                       {deselect all cells but new cell}
      MySelectOneCell(theList, theCell);
                                       {make sure new selection is visible}
      MyMakeCellVisible(theList, theCell);
   END;
END;
```

The MyKeySearchInList procedure defined in Listing 4-18 first updates variables related to type selection. Then it searches through the list for a cell containing the current search string or for the next cell alphabetically. It searches using the LSearch function in conjunction with a custom match function defined in Listing 4-15 on page 4-43. The procedure also uses the MySelectOneCell procedure defined in Listing 4-9 on page 4-36 and the MyMakeCellVisible procedure defined in Listing 4-10 on page 4-37.

Note

If your compiler enforces range checking, you may need to disable it before using the code in Listing 4-18, because the code accesses the length byte of a string directly. See your development system's documentation for more information on range checking. ◆

Supporting Arrow-Key Navigation of Lists

This section discusses how your application can support the use of arrow keys to move the current selection or to extend the current selection using a simple extension algorithm. For information on implementing a more complex anchor algorithm for extending the selection, read this section and then the next section, beginning on page 4-52.

The following constants define the ASCII character codes for the various arrow keys. These ASCII values for these keys are the same for U.S. and international keyboards.

```
CONST
   kLeftArrow     = Char(28);             {move left}
   kRightArrow    = Char(29);             {move right}
   kUpArrow       = Char(30);             {move up}
   kDownArrow     = Char(31);             {move down}
```

To support both the moving of a selection (the user's pressing an arrow key without pressing the Shift key) and the extending of a selection (the user's pressing of an arrow key while pressing the Shift key), your application needs to define a routine that computes a new selection location given an old one. For example, if the user presses Command–Left Arrow, the routine should find the cell as far to the left of the first currently selected cell as possible. Listing 4-19 illustrates an application-defined procedure that does this.

Listing 4-19 Determining the location of a new cell in response to an arrow-key event

```
PROCEDURE MyFindNewCellLoc
            (theList: ListHandle; oldCellLoc: Cell;
             VAR newCellLoc: Cell; keyHit: Char;
             moveToExtreme: Boolean);
VAR
   listRows, listColumns: Integer;     {list dimensions}
BEGIN
   WITH theList^^.dataBounds DO
   BEGIN
      listRows := bottom - top;        {number of rows in list}
      listColumns := right - left;     {number of columns in list}
   END;
   newCellLoc := oldCellLoc;
   IF moveToExtreme THEN
      CASE keyHit OF
         kUpArrow:
            newCellLoc.v := 0;               {move to row 0}
         kDownArrow:
            newCellLoc.v := listRows - 1;    {move to last row}
         kLeftArrow:
            newCellLoc.h := 0;               {move to column 0}
         kRightArrow:
            newCellLoc.h := listColumns - 1; {move to last column}
      END
   ELSE
      CASE keyHit OF
         kUpArrow:
            IF oldCellLoc.v <> 0 THEN
               newCellLoc.v := oldCellLoc.v - 1;   {row up}
         kDownArrow:
            IF oldCellLoc.v <> listRows - 1 THEN
               newCellLoc.v := oldCellLoc.v + 1;   {row down}
         kLeftArrow:
            IF oldCellLoc.h <> 0 THEN
               newCellLoc.h := oldCellLoc.h - 1;   {column left}
         kRightArrow:
            IF oldCellLoc.h <> listColumns - 1 THEN
               newCellLoc.h := oldCellLoc.h + 1;   {column right}
      END;
END;
```

The `MyFindNewCellLoc` procedure defined in Listing 4-19 computes the coordinates of the cell referenced by the `newCellLoc` parameter based on the coordinates of the `oldCellLoc` parameter and the direction of the arrow key pressed. The `oldCellLoc` parameter contains the coordinates of the first or last cell in a selection, depending on which arrow key was pressed. The behavior of `MyFindNewCellLoc` also depends on the value passed in the `moveToExtreme` parameter. For example, if the user pressed the Command key while pressing an arrow key, the SurfWriter application passes `TRUE`; otherwise, it passes `FALSE`. If `moveToExtreme` is `TRUE`, then `MyFindNewCellLoc` returns in `newCellLoc` a cell that is as far as possible from the cell specified in `oldCellLoc`. Otherwise, it returns a cell that is within one cell of `oldCellLoc`. If a cell cannot be moved in the direction specified by the arrow key, `newCellLoc` is equivalent on exit to `oldCellLoc`.

Having defined the `MyFindNewCellLoc` procedure, it is easy to move or extend a selection in response to an arrow-key event. Listing 4-20 illustrates an application-defined procedure that moves the selection in response to the user's pressing an arrow key without pressing the Shift key.

Listing 4-20 Moving the selection in response to an arrow-key event

```
PROCEDURE MyArrowKeyMoveSelection (theList: ListHandle;
                                   keyHit: Char;
                                   moveToExtreme: Boolean);
VAR
   currentSelection:       Cell;
   newSelection:           Cell;
BEGIN
   IF MyGetFirstSelectedCell(theList, currentSelection) THEN
   BEGIN
      IF (keyHit = kRightArrow) OR (keyHit = kDownArrow) THEN
         {find last selected cell}
         MyGetLastSelectedCell(theList, currentSelection);
      {move relative to appropriate cell}
      MyFindNewCellLoc(theList, currentSelection,
                       newSelection, keyHit, moveToExtreme);
      {make this cell the selection}
      MySelectOneCell(theList, newSelection);
      {make sure new selection is visible}
      MyMakeCellVisible(theList, newSelection);
   END;
END;
```

The MyArrowKeyMoveSelection procedure defined in Listing 4-20 calls the MyFindNewCellLoc procedure defined in Listing 4-19 to find the coordinates of a cell to select. It computes the coordinates of that new cell relative to the first selected cell if the user pressed a Left Arrow or Up Arrow key; otherwise, it computes the coordinates of the new cell relative to the last selected cell. After computing the coordinates of the new cell, MyArrowKeyMoveSelection selects it by calling routines defined in Listing 4-9 and Listing 4-10.

Listing 4-21 illustrates an application-defined procedure that extends the selection in response to the user's pressing an arrow key while pressing the Shift key.

Listing 4-21 Extending the selection in response to an arrow-key event

```
PROCEDURE MyArrowKeyExtendSelection (theList: ListHandle;
                                     keyHit: Char;
                                     moveToExtreme: Boolean);
VAR
   currentSelection: Cell;
   newSelection:     Cell;
BEGIN
   IF MyGetFirstSelectedCell(theList, currentSelection) THEN
   BEGIN
      IF (keyHit = kRightArrow) OR (keyHit = kDownArrow) THEN
                              {find last selected cell}
         MyGetLastSelectedCell(theList, currentSelection);
                              {move relative to appropriate cell}
      MyFindNewCellLoc(theList, currentSelection,
                       newSelection, keyHit, moveToExtreme);
                              {add a new cell to the selection}
      IF NOT LGetSelect(FALSE, newSelection, theList) THEN
         LSetSelect(TRUE, newSelection, theList);
                              {make sure new selection is visible}
      MyMakeCellVisible(theList, newSelection);
   END;
END;
```

The MyArrowKeyExtendSelection procedure defined in Listing 4-21 works just like the MyArrowKeyMoveSelection procedure defined in Listing 4-20, but it does not deselect all other cells besides the newly selected cell.

Listing 4-22 shows an application-defined procedure that takes advantage of the code listings provided in this section. The SurfWriter application calls the procedure in Listing 4-22 every time it receives an arrow-key event that affects a list.

Listing 4-22 Processing an arrow-key event

```
PROCEDURE MyArrowKeyInList (theList: ListHandle; theEvent: EventRecord;
                           allowExtendedSelections: Boolean);
BEGIN
   IF (NOT allowExtendedSelections) OR
         (BAnd(theEvent.modifiers, shiftKey) = 0) THEN
      MyArrowKeyMoveSelection(theList,
                              CHR(BAnd(theEvent.message, charCodeMask)),
                              BAnd(theEvent.modifiers, cmdKey) <> 0)
   ELSE
      MyArrowKeyExtendSelection(theList,
                                CHR(BAnd(theEvent.message, charCodeMask)),
                                BAnd(theEvent.modifiers, cmdKey) <> 0);
END;
```

The `MyArrowKeyInList` procedure defined in Listing 4-22 takes three parameters, the third of which is a Boolean variable that indicates whether the application supports the use of Shift–arrow key combinations to extend the current selection. If the application does support this and the user held down the Shift key, the `MyArrowKeyInList` procedure calls the procedure in Listing 4-21 to extend the selection. Otherwise, it calls the procedure in Listing 4-20 to move the selection. Either way, it checks the status of the Command key to determine whether the appropriate procedure should move as far in the direction of the arrow key as possible before selecting a new cell.

Supporting the Anchor Algorithm for Extending Lists With Arrow Keys

This section summarizes how your application can support the anchor method for extending lists with arrow keys. Implementing this method takes a lot of work, but the extra work may pay off if you expect many users of your application's lists to make range selections or if your application uses multicolumn lists. For a comparison between the anchor algorithm and the extension algorithm illustrated in the previous section, see "Extension of a Selection With Arrow Keys" on page 4-16.

To support the anchor algorithm, your application must keep track of several types of information between Shift–arrow key events. Most importantly, your application must store information about which cell in a list is the anchor cell and which cell is the moving cell. In response to a Shift–arrow key event, your application should change the location of the moving cell. It should then highlight all cells in the rectangle whose corners are the anchor cell and the moving cell. This permits the user to use several consecutive Shift–arrow key combinations to move a rectangular range of cells around the anchor cell.

Your application must thus save the location of the anchor cell the first time the user uses a Shift–arrow key combination to affect a certain rectangular range of cells. For example, if the user presses Shift–Right Arrow and the user has not before used a Shift–arrow key combination, then your application should store as the anchor cell the upper-left cell in the rectangular range of cells to be affected. The moving cell is then one cell to the right of what was the lower-right corner of this range.

Your application can determine what rectangular range of cells a Shift–arrow key combination is meant to affect by using the `LLastClick` function, which returns the coordinates of the last cell that was clicked. (If your application relies on this function, it must always update the `lastClick` field of the list record in response to keyboard selection of any list item, since keyboard selection of a list item is functionally equivalent to clicking.) Your application must check the selection status of adjacent cells to find as big a rectangular range of selected cells surrounding this cell as possible.

Your application can check whether a Shift–arrow key event is affecting a new range of cells simply by checking the `clikTime` field of the list record. (Your application must thus also update this field in response to keyboard selection of any list item.) If the last click time changes between Shift–arrow key events, your application knows that the user has clicked the list or used the keyboard to change the selection. In this case, your application must compute a new anchor cell and moving cell based on the `LLastClick` function and the direction of the arrow key pressed. Otherwise, your application can keep the same anchor cell, move the moving cell in the direction specified by the arrow key, and highlight cells in the rectangular range of the anchor cell and the moving cell.

In summary, if your application is to support the anchor algorithm for extending a list selection, it must keep track of an anchor cell, a moving cell, and the time of the last click in a list. (Your application might store a handle to a relocatable block containing this information in the `userHandle` field of the list record.) Whenever a Shift–arrow key event is meant to affect a new range of cells, your application updates all three of these variables. Otherwise, it only changes the coordinates of the moving cell from one Shift–arrow key event to the next.

Outlining the Current List

If a window in your application contains two lists, or contains one list and an editable text item, then your application should place a 2-pixel outline around a list whenever the list is the current list and active, that is, whenever typing would affect the list. Your application should outline the current list so that the user knows that typing affects the list.

Listing 4-23 shows an application-defined procedure that checks whether a list is the current list. If it is both current and active, it draws a 2-pixel outline around the list. Otherwise, it draws in the background color of the dialog box to remove the outline.

Listing 4-23 Drawing an outline around a list

```
PROCEDURE MyDrawOutline (myList: ListHandle);
CONST
   kScrollBarWidth = 15;         {width of scroll bar}
VAR
   myOutlineRect:    Rect;       {rectangle for outline border}
   myPenState:       PenState;   {current status of pen}
BEGIN
   {get list's visible rectangle}
   myOutlineRect := myList^^.rView;
   {compensate for scroll bars}
   IF myList^^.vScroll <> NIL THEN
      myOutlineRect.right := myOutlineRect.right
                           + kScrollBarWidth;
   IF myList^^.hScroll <> NIL THEN
      myOutlineRect.bottom := myOutlineRect.bottom
                           + kScrollBarWidth;
   {draw 2-pixel outline 3 pixels from border}
   SetPort(myList^^.port);       {set port to list's port}
                                 {move out 4 pixels}
   InsetRect(myOutlineRect, -4, -4);
   GetPenState(myPenState);      {store pen state}
   IF (myList = gCurrentList) AND myList^^.lActive THEN
      PenPat(black)              {draw border}
   ELSE
      PenPat(white);             {remove border}
   PenSize(2, 2);                {use 2-pixel pen}
   FrameRect(myOutlineRect);     {draw outline}
   SetPenState(myPenState);      {restore old pen state}
END;
```

The `MyDrawOutline` procedure defined in Listing 4-23 determines the rectangle to draw in by adjusting the list's visible rectangle to compensate for scroll bars and by then moving each side of the rectangle 4 pixels. (One pixel is already taken by the list border, an additional pixel is needed for space between the border and the outline, and the pen size for the outline is 2 pixels.) The list determines whether to draw or remove a list by comparing the list passed in with an an application-defined global variable, gCurrentList. If the variable indicates that a list is the current list, and the MyDrawOutline procedure determines that the list is active, then it draws the outline; otherwise, it removes it.

Your application can use the refCon field of the list record to create a linked ring list of all of the lists in a window to make it easier to support outlining. That is, the refCon field of the first list in a window contains a handle to the second list in a window; the refCon field of the second list in a window contains a handle to the third, and so on, until the refCon field of the last list in a window contains a handle to the first.

The advantage of implementing such a ring list is that it makes it easy to change which list is the current list. In response to a Tab-key event, your application need only find the next list in a window by looking at the current list's refCon field and setting the gCurrentList variable to the list referenced by that field. Without using such a strategy, your application would need to examine the gCurrentList variable, determine which of a window's lists the variable corresponded to, determine which list in the window is the next list, and then set the gCurrentList variable to this next list.

Listing 4-24 shows an application-defined procedure that adds a list to a ring being maintained for a particular window.

Listing 4-24 Adding a list to the ring

```
PROCEDURE MyTrackList (myList: ListHandle);
VAR
   aList:      ListHandle;
BEGIN
   aList := gCurrentList;
   IF aList = NIL THEN
      gCurrentList := myList     {first ListHandle to be tracked}
   ELSE
   BEGIN
      {look for last ListHandle in ring}
      WHILE (ListHandle(aList^^.refCon) <> gCurrentList) DO
         {move to next ListHandle in ring}
         aList^^.refCon := ListHandle(aList^^.refCon)^^.refCon;
      {insert myList into ring}
      ListHandle(aList^^.refCon) := myList;
   END;
   {add link from myList to current list}
   ListHandle(myList^^.refCon) := gCurrentList;
END;
```

The SurfWriter application calls the MyTrackList procedure defined in Listing 4-24 once for each list in a window when it first opens that window. The first list added to the ring is automatically set to be the current list. SurfWriter initializes the gCurrentList variable to NIL before creating a ring for each window that uses multiple lists. In addition, SurfWriter stores the value of the gCurrentList variable whenever a window containing multiple lists is deactivated and then resets it when the window is activated again. That way, the gCurrentList variable always stores a handle to the current list of the active window.

Once all the lists in a window are linked in a ring, it is easy to write a routine that ensures that only the current list is outlined. Listing 4-25 illustrates such a routine.

Listing 4-25 Updating the outline of all lists in a window

```
PROCEDURE MyUpdateListOutlines;
VAR
   listToUpdate:     ListHandle;
BEGIN
   listToUpdate := gCurrentList;
   IF listToUpdate <> NIL THEN
   REPEAT
      {move to next list in ring}
      listToUpdate := ListHandle(listToUpdate^^.refCon);
      MyDrawOutline(listToUpdate);
   UNTIL listToUpdate = gCurrentList;
END;
```

The MyUpdateListOutlines procedure defined in Listing 4-25 simply calls the MyDrawOutline procedure for each list in the active window's ring of lists. The SurfWriter application calls this procedure each time your application changes which list is current.

Listing 4-26 shows an application-defined procedure that responds to the user's pressing the Tab key when the Shift key is not also pressed.

Listing 4-26 Moving the outline to the next list in a window

```
PROCEDURE MyOutlineNextList;
BEGIN
   gCurrentList := ListHandle(gCurrentList^^.refCon);
   MyUpdateListOutlines;
END;
```

If the user presses Shift-Tab, your application should respond by changing the current list to the previous list. Listing 4-27 shows an application-defined procedure that does this.

Listing 4-27 Moving the outline to the previous list in a window

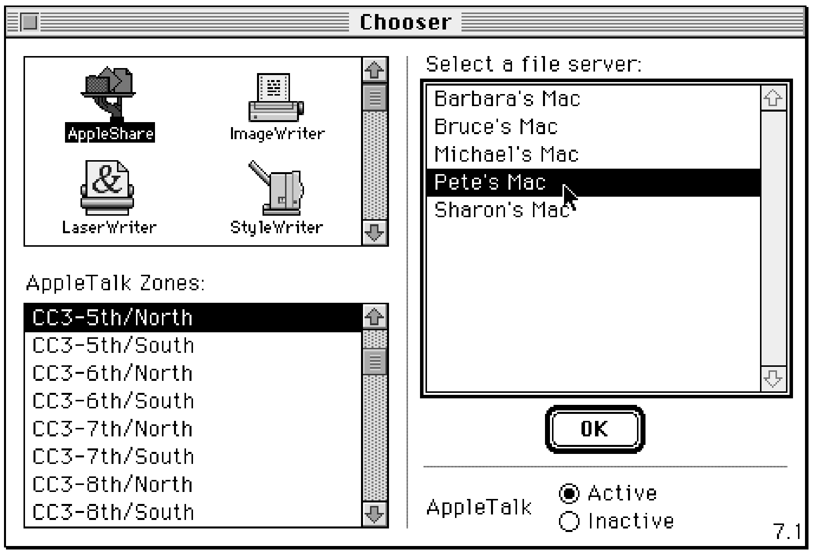
```
PROCEDURE MyOutlinePreviousList;
VAR
   previousList:     ListHandle;
BEGIN
   {compute the coordinates of the list before the current list}
   previousList := gCurrentList;
   WHILE (ListHandle(previousList^^.refCon) <> gCurrentList) DO
      previousList := ListHandle(previousList^^.refCon);
   {now switch the outline to this list}
   gCurrentList := previousList;
   MyUpdateListOutlines;
END;
```

The `MyOutlineNextList` and `MyOutlinePreviousList` procedures defined in Listing 4-26 and Listing 4-27 work the same if a window contains exactly two lists.

Writing Your Own List Definition Procedure

The default list definition procedure supports only the display of unstyled text. If your application needs to display items graphically, you can create your own list definition procedure. For example, the Chooser desk accessory uses its own list definition procedure to display icons and names corresponding to Chooser extensions. Figure 4-19 illustrates the Chooser's use of a custom list definition procedure.

Figure 4-19 The Chooser's use of a custom list definition procedure



Select a file server:

Barbara's Mac
Bruce's Mac
Michael's Mac
Pete's Mac
Sharon's Mac

CC3-5th/North
CC3-5th/South
CC3-6th/North
CC3-6th/South
CC3-7th/North
CC3-7th/South
CC3-8th/North
CC3-8th/South

AppleTalk [✓ Active] Inactive

This section explains how you can write a list definition procedure. After writing a list definition procedure, you must compile it as a resource of type `'LDEF'` and store it in the resource fork of any application that uses the list definition procedure.

This section provides code for a list definition procedure that supports the display of QuickDraw pictures. It works by requiring the application that uses it to store as cell data variables of type `PicHandle`. That way, each cell stores only 4 bytes of data, and the List Manager's 32 KB limit is not at risk of being approached for small lists. This list definition procedure provides enough versatility to display virtually any type of image.

You can write your own list definition procedure to store some type of data other than unstyled text. You can give your list definition procedure any name you choose, but it must be defined like this:

```
PROCEDURE MyLDEF (message: Integer; selected: Boolean;
                  VAR cellRect: Rect; theCell: Cell;
                  dataOffset: Integer; dataLen: Integer;
                  theList: ListHandle);
```

The List Manager can send four types of messages to your list definition procedure, as indicated by a value passed in the `message` parameter. The following constants define the different types of messages:

```
CONST
   lInitMsg          = 0;     {do any special list initialization}
   lDrawMsg          = 1;     {draw the cell}
   lHiliteMsg        = 2;     {invert cell's highlight state}
   lCloseMsg         = 3;     {take any special disposal action}
```

Of the second through seventh parameters to a list definition procedure, only the `theList` parameter, which contains a handle to a list record, can be accessed by your list definition procedure in response to all four messages.

The `selected`, `cellRect`, `theCell`, `dataOffset`, and `dataLen` parameters pass information to your list definition procedure only when the value in the `message` parameter contains the `lDrawMsg` or the `lHiliteMsg` constant. These parameters provide information about the cell to be affected by the message. The `selected` parameter indicates whether the cell should be highlighted. The `cellRect` and `theCell` parameters indicate the cell's rectangle and coordinates. Finally, the `dataOffset` and `dataLen` parameters specify the offset and length of the cell's data within the relocatable block referenced by the `cells` field of the list record.

Listing 4-28 shows a list definition procedure that processes messages sent to it by the List Manager.

Listing 4-28 Processing messages to a list definition procedure

```
PROCEDURE MyLDEF (message: Integer; selected: Boolean;
                  VAR cellRect: Rect; theCell: Cell;
                  dataOffset: Integer; dataLen: Integer;
                  theList: ListHandle);
BEGIN
   CASE message OF
      lInitMsg:
         MyLDEFInit(theList);
      lDrawMsg:
         MyLDEFDraw(selected, cellRect, theCell, dataOffset,
                    dataLen, theList);
      lHiliteMsg:
         MyLDEFHighlight(selected, cellRect, theCell,
                         dataOffset, dataLen, theList);
      lCloseMsg:
         MyLDEFClose(theList);
   END;
END;
```

The MyLDEF procedure defined in Listing 4-28 calls procedures defined later in this section to handle the various messages specified by the message parameter. It passes all relevant parameters to these message-handling procedures. Thus, it passes only the theList parameter to the procedures that handle the initialization and close messages.

Responding to the Initialization Message

The List Manager automatically allocates memory for a list and fills out the fields of a list record before calling your list definition procedure with a lInitMsg message. Your application might respond to the initialization message by changing fields of the list record, such as the cellSize and indent fields. (These fields are by default set according to a formula discussed in “About the List Manager” beginning on page 4-22.)

Many list definition procedures do not need to perform any action in response to the initialization message. For example, the list definition procedure that allows the Standard File Package to display small icons next to the names of files uses the standard cell size and thus does not need to perform any special initialization.

Since pictures can come in a variety of sizes, the pictures list definition procedure introduced in Listing 4-28 does not need to perform any special initialization either; it depends on the application that uses the list definition procedure to define the correct cell size. Thus, Listing 4-29 shows how the pictures list definition procedure responds to the initialization method.

Listing 4-29 Using the default initialization method

```
PROCEDURE MyLDEFInit (theList: ListHandle);
BEGIN
END;
```

Note
Your list definition procedure does not actually need to call a procedure that responds to the initialization message if it does not need to perform any special action. ◆

Responding to the Draw Message

Your list definition procedure must respond to the lDrawMsg message by examining the specified cell’s data and drawing the cell as appropriate. At the same time, your list definition procedure must ensure that it does not alter the characteristics of the drawing environment.

Listing 4-30 shows how the pictures list definition procedure responds to the draw message.

Listing 4-30 Responding to the `lDrawMsg` message

```
PROCEDURE MyLDEFDraw (selected: Boolean; cellRect: Rect;
                      theCell: Cell; dataOffset: Integer;
                      dataLen: Integer; theList: ListHandle);
VAR
   savedPort:     GrafPtr;          {old graphics port}
   savedClip:     RgnHandle;        {old clip region}
   savedPenState: PenState;         {old pen state}
   myPicture:     PicHandle;        {handle to a picture}
BEGIN
   {set up the drawing environment}
   GetPort(savedPort);              {remember the port}
   SetPort(theList^^.port);         {set port to list's port}
   savedClip := NewRgn;             {create new region}
   GetClip(savedClip);              {set region to clip region}
   ClipRect(cellRect);              {set clip region to cell }
                                    { rectangle}
   GetPenState(savedPenState);      {remember pen state}
   PenNormal;                       {use normal pen type}
   {draw the cell if it contains data}
   EraseRect(cellRect);             {erase before drawing}
   IF dataLen = SizeOf(PicHandle) THEN
   BEGIN
                                    {get handle to picture}
      LGetCell(@myPicture, dataLen, theCell, theList);
                                    {draw the picture}
      DrawPicture(myPicture, cellRect);
   END;
   {select the cell if necessary}
   IF selected THEN                 {highlight cell}
      MyLDEFHighlight(selected, cellRect, theCell, dataOffset,
                      dataLen, theList);
   {restore graphics environment}
   SetPort(savedPort);              {restore saved port}
   SetClip(savedClip);              {restore clip region}
   DisposeRgn(savedClip);           {free region memory}
   SetPenState(savedPenState);      {restore pen state}
 END;
```

The `MyLDEFDraw` procedure defined in Listing 4-30 begins by saving characteristics of the current graphics environment, such as the graphics port, the clipping region, and the pen state. It also sets the pen to a normal state, and sets the clipping region to the cell's rectangle. The `MyLDEFDraw` procedure then draws in the cell rectangle by erasing the rectangle, getting the handle stored as the cell's data, and drawing the picture referenced by that handle. Then, if the cell should be selected, it simply calls the `MyLDEFHighlight` procedure defined in the next section. Before returning, `MyLDEFDraw` restores the graphics environment to its previous state and disposes of the memory it used to remember the clipping region.

Note
For more information on the QuickDraw routines and data structures used in Listing 4-30, see *Inside Macintosh: Imaging With QuickDraw*. ◆

Responding to the Highlighting Message

Virtually every list definition procedure should respond to the `lHiliteMsg` message in the same way, by inverting the bits in the cell's rectangle. Your list definition procedure would need to respond in a different way if selected list items should not simply be highlighted. For example, in a list of patterns, simply highlighting selected cells could confuse the user because highlighted patterns look just like other patterns.

Listing 4-31 shows how your list definition procedure can respond to the `lHiliteMsg` message in a way that is compatible with all Macintosh models, including models that do not support Color QuickDraw.

Listing 4-31 Responding to the `lHiliteMsg` message

```
PROCEDURE MyLDEFHighlight (selected: Boolean; cellRect: Rect;
                           theCell: Cell; dataOffset: Integer;
                           dataLen: Integer; theList: ListHandle);
BEGIN
                           {use color highlighting if possible}
   BitClr(Ptr(HiliteMode), pHiliteBit);
   InvertRect(cellRect);   {highlight cell rectangle}
END;
```

For more information on highlighting, see *Inside Macintosh: Imaging With QuickDraw.*

Responding to the Close Message

The List Manager sends your list definition procedure the `lCloseMsg` message immediately before disposing of the data occupied by a list. Your list definition procedure needs to respond to the close message only if it needs to perform some special processing before a list is disposed, such as releasing memory associated with a list that would not be released by the `LDispose` procedure.

The pictures list definition procedure responds to the close message by freeing memory occupied by the list's pictures, whose handles are stored in the list. While the `LDispose` procedure will dispose of the picture handles themselves, it cannot dispose of the relocatable blocks referenced by the picture handles.

Listing 4-32 shows how the pictures list definition procedure responds to the `lCloseMsg` message.

Listing 4-32 Responding to the `lCloseMsg` message

```
PROCEDURE MyLDEFClose (theList: ListHandle);
   VAR
      aCell:         Cell;          {cell in the list}
      myPicHandle:   PicHandle;     {handle stored as cell data}
      myDataLength:  Integer;       {length in bytes of cell data}
BEGIN
   SetPt(aCell, 0, 0);
   IF PtInRect(aCell, theList^^.dataBounds) THEN
   REPEAT
      {free memory only if cell's data is 4 bytes long}
      myDataLength := SizeOf(PicHandle);
      LGetCell(@myPicHandle, myDataLength, aCell, theList);
      IF myDataLength = SizeOf(PicHandle) THEN
         KillPicture(myPicHandle);
   UNTIL NOT LNextCell(TRUE, TRUE, aCell, theList);
END;
```

Using the Pictures List Definition Procedure

The pictures list definition procedure introduced in Listing 4-28 can display a list containing pictures. For example, the SurfWriter application uses it to display a list of icons. SurfWriter first creates a list using the `MyCreateVerticallyScrollingList` function shown in Listing 4-1 on page 4-27. After creating the list, rather than using the default cell size as calculated by the List Manager, the SurfWriter application sets the size of the cells using the `LCellSize` procedure, as shown in Listing 4-33.

Listing 4-33 Setting the cell size of a list

```
PROCEDURE MySetCellSizeForIconList(myCellSize: Point;
                                   myList: ListHandle);
BEGIN
   LCellSize(myCellSize, myList);
END;
```

To later add an icon to a list of icons, the SurfWriter application uses the procedure shown in Listing 4-34.

Listing 4-34 Adding an icon to a list of icons

```
PROCEDURE MyAddIconToList(myCellRect: Rect; myPlotRect: Rect;
                          myCell: Cell; theList: ListHandle;
                          VAR myPicHandle: PicHandle;
                          resID: Integer);
CONST
   kIconWidth     = 32; {width of an icon}
   kIconHeight    = 32; {height of an icon}
   kExtraSpace    = 2;  {extra space on top and to left of icon}
VAR
   myIcon:  Handle;
BEGIN
   {picture occupies entire cell rectangle}
   SetRect(myCellRect, 0, 0, kIconWidth + kExtraSpace,
           kIconHeight + kExtraSpace);
   {plot icon over portion of rectangle}
   SetRect(myPlotRect, kExtraSpace, kExtraSpace, kIconWidth +
           kExtraSpace, kIconHeight + kExtraSpace);
   {load icon from resource file}
   myIcon := GetIcon(resID);
   {create the picture}
   myPicHandle := OpenPicture(myCellRect);
   PlotIcon(myPlotRect, myIcon);
   ClosePicture;
   {store handle to picture as cell data}
   LSetCell(@myPicHandle, SizeOf(PicHandle), myCell, theList);
   {release icon resource}
   ReleaseResource(myIcon);
END;
```

Note that the `MyAddIconToList` procedure uses the QuickDraw routines `OpenPicture` and `ClosePicture` to bracket the set of drawing commands that it uses to define the picture data for a particular cell. It then stores the handle to the picture as the cell's data, so that the pictures list definition procedure can draw the picture within the cell.

List Manager Reference

This section describes the data structures and routines that are specific to the List Manager. The "Data Structures" section shows the data structures for the cell, the data handle, and the list record. The "List Manager Routines" section beginning on page 4-70 describes the routines that your application can use to create, manipulate, get information about, and dispose of lists. The "Application-Defined Routines" section beginning on page 4-96 describes list definition procedures, match functions, and click-loop procedures.

Data Structures

This section describes the data structures that the List Manager uses to store information about a list.

Your application can use the cell record to specify the coordinates of a cell. For example, your application must specify cell coordinates to the `LAddToCell` procedure to add data to a cell.

The List Manager uses a data handle internally to store information about the contents of a list's cells. The List Manager provides routines that allow you to access information contained in this data handle.

Finally, the List Manager uses a list record to store a variety of information about a list. To obtain some types of information about a list, your application might need to access the fields of the list record directly.

The Cell Record

A cell record specifies the coordinates of a cell in a list. The `Cell` data type defines a cell record.

```
TYPE  Cell = Point;
```

Field descriptions

`v`	The row number of the cell.
`h`	The column number of the cell.

Note that column and row numbers are 0-based. Also note that this chapter designates cells using the notation (*column*–1, *row*–1), so that a cell with coordinates (2,5) is in the third column and sixth row of a list. You specify a cell with coordinates (2,5) by setting the cell's `h` field to 2 and its `v` field to 5.

The Data Handle

The List Manager uses a data handle to store information about a list. The `DataHandle` data type defines a data handle.

```
TYPE  DataArray         = PACKED ARRAY[0..32000] OF Char;
      DataPtr           = ^DataArray;
      DataHandle        = ^DataPtr;
```

Your application should not change the information in a data handle directly. Your application can, however, read data stored in a list's data handle directly by calling the `GetCellDataLocation` procedure to find the offset of a cell's data into the data handle and the length of the cell's data.

The List Record

The List Manager uses a list record to store many types of information about a list. Usually you access a list record through a handle to the list record defined by the data type `ListHandle`. The `ListRec` data type defines a list record.

```
TYPE ListRec   =
RECORD
   rView:      Rect;             {list's display rectangle}
   port:       GrafPtr;          {list's graphics port}
   indent:     Point;            {indent distance for drawing}
   cellSize:   Point;            {size in pixels of a cell}
   visible:    Rect;             {boundary of visible cells}
   vScroll:    ControlHandle;    {vertical scroll bar}
   hScroll:    ControlHandle;    {horizontal scroll bar}
   selFlags:   SignedByte;       {selection flags}
   lActive:    Boolean;          {TRUE if list is active}
   lReserved:  SignedByte;       {reserved}
   listFlags:  SignedByte;       {automatic scrolling flags}
   clikTime:   LongInt;          {TickCount at time of last click}
   clikLoc:    Point;            {position of last click}
   mouseLoc:   Point;            {current mouse location}
   lClikLoop:  Ptr;              {routine called by LClick}
   lastClick:  Cell;             {last cell clicked}
   refCon:     LongInt;          {for application use}
   listDefProc:                  {list definition procedure}
               Handle;
   userHandle: Handle;          {for application use}
   dataBounds: Rect;            {boundary of cells allocated}
   cells:      DataHandle;      {cell data}
   maxIndex:   Integer;         {used internally}
   cellArray:                   {offsets to data}
               ARRAY[1..1] OF Integer;
END;
   ListPtr     = ^ListRec;      {pointer to a list record}
   ListHandle  = ^ListPtr;      {handle to a list record}
```

Field descriptions

`rView`	Specifies the rectangle in which the list's visible rectangle is located, in local coordinates of the graphics port specified by the `port` field. Note that the list's visible rectangle does not include the area needed for the list's scroll bars. The width of a vertical scroll bar (which equals the height of a horizontal scroll bar) is 15 pixels.
`port`	Specifies the graphics port of the window containing the list.
`indent`	Defines the location, relative to the upper-left corner of a cell, at which drawing should begin. List definition procedures should set this field to a value appropriate to the type of data that a cell in a list is to contain.
`cellSize`	Contains the size in pixels of each cell in the list. When your application creates a list, it can either specify the cell size or let the List Manager calculate the cell size. You should not change the `cellSize` field directly; if you need to change the cell size after creating a list, use the `LCellSize` procedure.
`visible`	Specifies the cells in a list that are visible within the area specified by the `rView` field. The List Manager sets the `left` and `top` fields of `visible` to the coordinates of the first visible cell; however, the List Manager sets the `right` and `bottom` fields so that each is 1 greater than the horizontal and vertical coordinates of the last visible cell. For example, if a list contains 4 columns and 10 rows but only the first 2 columns and the first 5 rows are visible (that is, the last visible cell has coordinates (1,4)), the List Manager sets the `visible` field to (0,0,2,5).
`vScroll`	Contains a control handle for a list's vertical scroll bar, or `NIL` if a list does not have a vertical scroll bar.
`hScroll`	Contains a control handle for a list's horizontal scroll bar, or `NIL` if a list does not have a vertical scroll bar.

`selFlags`	Indicates the selection flags for a list. When your application creates a list, the List Manager clears the `selFlags` field to 0. This defines the List Manager's default selection algorithm. To change the default behavior for a particular list, set the desired bits in the list's `selFlags` field.

You can use these constants to refer to bits in this field:

```
CONST
   {allow only one item to be selected at once}
   lOnlyOne       = -128;
   {enable multiple item selection without Shift}
   lExtendDrag    = 64;
   {prevent discontiguous selections}
   lNoDisjoint    = 32;
   {reset list before responding to Shift-click}
   lNoExtend      = 16;
   {Shift-drag selects items passed by cursor}
   lNoRect        = 8;
   {allow use of Shift key to deselect items}
   lUseSense      = 4;
   {disable highlighting of empty cells}
   lNoNilNilite   = 2;
```

`lActive`	Indicates whether the list is active (`TRUE` if active, `FALSE` if inactive).
`lReserved`	Reserved.
`listFlags`	Indicates whether the List Manager should automatically scroll the list if the user clicks the list and then drags the cursor outside the list display rectangle.

The following constants define bits in this field that determine whether horizontal autoscrolling and vertical autoscrolling are enabled:

```
CONST
   {allow automatic vertical scrolling}
   lDoVAutoscroll    = 2;
   {allow automatic horizontal scrolling}
   lDoHAutoscroll    = 1;
```

By default, the List Manager enables horizontal autoscrolling for a list if the list includes a horizontal scroll bar, and enables vertical autoscrolling for a list if the list includes a vertical scroll bar.

`clikTime`	Specifies the time in ticks of the last click in the list. If your application depends on the value contained in this field, then your application should update the field should the application select a list item in response to keyboard input.
`clikLoc`	Specifies the location in local coordinates of the last click in the list.
`mouseLoc`	Indicates the current location of the cursor in local coordinates. This value is continuously updated by the `LClick` function after the user clicks a list.
`lClikLoop`	Contains a pointer to a click-loop procedure repeatedly called by the `LClick` function, or `NIL` if the default click-loop procedure is to be used. For information on click-loop procedures, see "Click-Loop Procedures" beginning on page 4-100.
`lastClick`	Specifies the coordinates of the last cell in the list that was clicked. This may not be the same as the last cell selected if the user selects a range of cells by Shift-dragging or Command-dragging. If your application depends on the value contained in this field, then your application should update the field whenever your application selects a list item in response to keyboard input.
`refCon`	Contains 4 bytes for use by your application.
`listDefProc`	Contains a handle to the code for the list definition procedure that defines how the list is drawn.
`userHandle`	Contains 4 bytes that your application can use as needed. For example, your application might use this field to store a handle to additional storage associated with the list. However, the `LDispose` procedure does not automatically release this storage when disposing of the list.
`dataBounds`	Specifies the range of cells in a list. When your application creates a list, it specifies the initial bounds of the list. As your application adds rows and columns, the List Manager updates this field. The List Manager sets the `left` and `top` fields of `dataBounds` to the coordinates of the first cell in the list; the List Manager sets the `right` and `bottom` fields so that each is 1 greater than the horizontal and vertical coordinates of the last cell. For example, if a list contains 4 columns and 10 rows (that is, the last cell in the list has coordinates (3,9)), the List Manager sets the `dataBounds` field to (0,0,4,10).
`cells`	Contains a handle to a relocatable block used to store cell data. Your application should not change the contents of this relocatable block directly.
`maxIndex`	Used internally.
`cellArray`	Contains offsets to data that indicate the location of different cells' data within the data handle specified by the `cells` parameter. Your application should not access this field directly.

List Manager Routines

This section describes the routines you can use to

- create and dispose of lists
- add and delete rows and columns to and from lists
- find or change cells' selection status
- read or change cell data
- respond to list events
- affect the display of a list
- get information about cells
- change the size of a list or of a cell contained in a list

▲ **WARNING**

The List Manager's routines are contained in a resource of resource type `'PACK'`. Calling any of the routines described in this section could result in the loading of the package resource and the allocation of memory. Thus, your application should not call any of the routines described in this section at interrupt time. For more information on packages, see *Inside Macintosh: Operating System Utilities.* ▲

Creating and Disposing of Lists

You can create a list by calling the `LNew` function. When you are through with the list, you can dispose of it by calling the `LDispose` procedure.

LNew

You can use the `LNew` function to create a new list in a window.

```
FUNCTION LNew (rView, dataBounds: Rect; cSize: Point;
               theProc: Integer; theWindow: WindowPtr;
               drawit, hasGrow, scrollHoriz, scrollVert: Boolean)
               : ListHandle;
```

`rView`	The rectangle in which to display the list, in local coordinates of the window specified by the `theWindow` parameter. This rectangle does not include the area to be taken up by the list's scroll bars.
`dataBounds`	The initial data bounds for the list. By setting the `left` and `top` fields of this rectangle to (0,0) and the `right` and `bottom` fields to (`kMyInitialColumns,kMyInitialRows`), your application can create a list that has `kMyInitialColumns` columns and `kMyInitialRows` rows.

`cSize`	The size of each cell in the list. If your application specifies (0,0) and is using the default list definition procedure, the List Manager sets the `v` coordinate of this parameter to the sum of the ascent, descent, and leading of the current font, and it sets the `h` coordinate using the following formula:

```
cSize.h := (rView.right - rView.left) DIV
                 (dataBounds.right – dataBounds.left)
```

`theProc`	The resource ID of the list definition procedure to use for the list. To use the default list definition procedure, which supports the display of unstyled text, specify a resource ID of 0.
`theWindow`	A pointer to the window in which to install the list.
`drawIt`	A Boolean value that indicates whether the List Manager should initially enable the automatic drawing mode. When the automatic drawing mode is enabled, the List Manager automatically redraws the list whenever a change is made to it. You can later change this setting using the `LSetDrawingMode` procedure. Your application should leave the automatic drawing mode disabled only for short periods of time when making changes to a list (by, for example, adding rows and columns).
`hasGrow`	A Boolean value that indicates whether the List Manager should leave room for a size box. The List Manager does not actually draw the grow icon. Usually, your application can draw it with the Window Manager's `DrawGrowIcon` procedure.
`scrollHoriz`	A Boolean value that indicates whether the list should contain a horizontal scroll bar. Specify `TRUE` if your list requires a horizontal scroll bar; specify `FALSE` otherwise.
`scrollVert`	Indicates whether the list should contain a vertical scroll bar. Specify `TRUE` if your list requires a vertical scroll bar; specify `FALSE` otherwise.

DESCRIPTION

The `LNew` function attempts to create a list defined by the function's parameters and returns a handle to the newly created list. If the `LNew` function cannot allocate the list, it returns `NIL`. This might happen if there is not enough memory available or if `LNew` cannot load the resource specified by the `theProc` parameter. If the `LNew` function returns successfully, then all of the fields of the list record referenced by the returned handle are correctly set.

If the list contains a horizontal or vertical scroll bar and the window specified by the parameter `theWindow` is visible, `LNew` draws the scroll bar for the new list in the window just outside the list's visible rectangle specified by the `rView` parameter. The `LNew` function does not, however, draw a 1-pixel border around the list's visible rectangle.

SPECIAL CONSIDERATIONS

You should not call the `LNew` function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LNew` function are

Trap macro	Selector
`_Pack0`	$0044

SEE ALSO

See Listing 4-1 on page 4-27 for an example of how to use the `LNew` function.

LDispose

You can use the `LDispose` procedure to dispose of the memory associated with a list that you no longer need.

```
PROCEDURE LDispose (lHandle: ListHandle);
```

`lHandle`	The list to be disposed of.

DESCRIPTION

The `LDispose` procedure releases all memory allocated by the List Manager in creating a list. First, `LDispose` issues a close request to the list definition procedure and calls the Control Manager procedure `DisposeControl` for the list's scroll bars (if any). `LDispose` then uses the Memory Manager to free the memory referenced by the cells field, then disposes of the list record itself.

Because `LDispose` disposes of data associated with cells in your list, there is no need to clear the data from list cells or to delete individual rows and columns before calling `LDispose`.

The `LDispose` procedure does not dispose of any memory associated with a list that the List Manager has not allocated. In particular, `LDispose` does not dispose of any memory referenced by the `userHandle` field of the list record. Your application is responsible for deallocating any memory it has allocated through the `userHandle` field before calling `LDispose`.

SPECIAL CONSIDERATIONS

You should not call the `LDispose` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LDispose` procedure are

Trap macro	Selector
`_Pack0`	$0028

Adding and Deleting Columns and Rows To and From a List

You can use the `LAddColumn` and `LAddRow` functions to add one or more columns or rows to a list, and you can use the `LDelColumn` and `LDelRow` procedures to delete one or more columns or rows from a list.

LAddColumn

You can use the `LAddColumn` function to add one or more columns to a list.

```
FUNCTION LAddColumn (count: Integer; colNum: Integer;
                     lHandle: ListHandle): Integer;
```

`count`	The number of columns to add.
`colNum`	The column number of the first column to add.
`lHandle`	The list to which to add the columns.

DESCRIPTION

The `LAddColumn` function inserts into the given list the number of columns specified by the `count` parameter, starting at the column specified by the `colNum` parameter. The `LAddColumn` function returns as its function result the column number of the first column added, which is equal to the value specified by the `colNum` parameter if that value is a valid column number.

If the column number specified by `colNum` is not already in the list, then new last columns are added. The value returned by the `LAddColumn` function thus has significance only in this case.

▲ **WARNING**
If there is insufficient memory in the heap to add the new columns, the `LAddColumn` function may fail to add the new columns although it returns a positive function result. Be sure there is enough memory in the heap to allocate the new columns before calling `LAddColumn`. ▲

Columns whose column numbers are initially greater than `colNum` have their column numbers increased by `count`.

If the automatic drawing mode is enabled and the columns added by `LAddColumn` are visible, then the list (including its scroll bars) is updated. New cells created by a call to `LAddColumn` are initially empty.

You may add columns to a list that does not yet have rows. The `dataBounds` field of the list record reflects that the list has columns, but you can only access cells when both rows and columns have been added.

SPECIAL CONSIDERATIONS

You should not call the `LAddColumn` function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LAddColumn` function are

Trap macro	Selector
`_Pack0`	$0004

LAddRow

You can use the `LAddRow` function to add one or more rows to a list.

```
FUNCTION LAddRow (count: Integer; rowNum: Integer;
                  lHandle: ListHandle): Integer;
```

`count`	The number of rows to add.
`rowNum`	The row number of the first row to add.
`lHandle`	The list to add the rows to.

DESCRIPTION

The `LAddRow` function inserts into the given list the number of rows specified by the `count` parameter, starting at the row specified by the `rowNum` parameter. The `LAddRow` function returns as its function result the row number of the first row added, which is equal to the value specified by the `rowNum` parameter if that value is a valid row number.

If the row number specified by `rowNum` is not already in the list, then new last rows are added. The value returned by the `LAddRow` function thus has significance only in this case.

▲ **WARNING**

If there is insufficient memory in the heap to add the new rows, the `LAddRow` function may fail to add the new rows although it returns a positive function result. Be sure there is enough memory in the heap to allocate the new rows before calling `LAddRow`. ▲

Rows whose row numbers are initially greater than `rowNum` have their row numbers increased by `count`.

If the automatic drawing mode is enabled and the rows added by `LAddRow` are visible, then the list (including its scroll bars) is updated. New cells created by a call to `LAddRow` are initially empty.

You may add rows to a list that does not yet have columns. The `dataBounds` field of the list record reflects that the list has rows, but you can only access cells when both rows and columns have been added.

SPECIAL CONSIDERATIONS

You should not call the `LAddRow` function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LAddRow` function are

Trap macro	Selector
`_Pack0`	$0008

SEE ALSO

For an example that adds rows to a list, see Listing 4-4 on page 4-31.

LDelColumn

You can use the `LDelColumn` procedure to delete one or more columns from a list.

```
PROCEDURE LDelColumn (count: Integer; colNum: Integer;
                      lHandle: ListHandle);
```

`count`	The number of columns to delete, or 0 to delete all columns.
`colNum`	The column number of the first column to delete.
`lHandle`	The list from which to delete the columns.

DESCRIPTION

The `LDelColumn` procedure deletes the number of columns specified by the `count` parameter, starting at the column specified by the `colNum` parameter.

If the column specified by `colNum` is invalid, then nothing is done.

Your application can quickly delete all columns from a list (and thus delete all cell data) simply by setting the `count` parameter to 0. The number of rows is left unchanged. Your application can achieve the same effect by setting the `colNum` parameter to `lHandle^^.dataBounds.left` and setting the `count` parameter to a value greater than `lHandle^^.dataBounds.right – lHandle^^.dataBounds.left`.

Columns whose column numbers are initially greater than `colNum` have their column numbers decreased by `count`.

If the automatic drawing mode is enabled and one or more of the columns deleted by `LDelColumn` are visible, then the list (including its scroll bars) is updated.

SPECIAL CONSIDERATIONS

You should not call the `LDelColumn` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LDelColumn` procedure are

Trap macro	Selector
`_Pack0`	$0020

LDelRow

You can use the `LDelRow` procedure to delete one or more rows from a list.

```
PROCEDURE LDelRow (count: Integer; rowNum: Integer;
                   lHandle: ListHandle);
```

`count`	The number of rows to delete, or 0 to delete all rows.
`rowNum`	The row number of the first row to delete.
`lHandle`	The list from which to delete the rows.

DESCRIPTION

The `LDelRow` procedure deletes the number of rows specified by the `count` parameter, starting at the row specified by the `rowNum` parameter.

If the row specified by `rowNum` is invalid, then nothing is done.

Your application can quickly delete all rows from a list (and thus delete all cell data) simply by setting the `count` parameter to 0. The number of columns is left unchanged. Your application can achieve the same effect by setting the `rowNum` parameter to `lHandle^^.dataBounds.top` and setting the `count` parameter to a value greater than `lHandle^^.dataBounds.bottom - lHandle^^.dataBounds.top`.

Rows whose row numbers are initially greater than `rowNum` have their row numbers decreased by `count`.

If the automatic drawing mode is enabled and one or more of the rows deleted by `LDelRow` are visible, then the list (including its scroll bars) is updated.

SPECIAL CONSIDERATIONS

You should not call the `LDelRow` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LDelRow` procedure are

Trap macro	Selector
`_Pack0`	$0024

Determining or Changing the Selection

Your application can use the `LGetSelect` function to determine whether a certain cell is selected or to find the next selected cell. To select or deselect a specific cell, your application can use the `LSetSelect` procedure.

LGetSelect

You can use the `LGetSelect` function to get information about which cells are selected.

```
FUNCTION LGetSelect (next: Boolean; VAR theCell: Cell;
                     lHandle: ListHandle): Boolean;
```

`next`	A Boolean value that indicates whether `LGetSelect` should check only the cell specified by the parameter `theCell` (`next = FALSE`), or whether it should try to find the next selected cell (`next = TRUE`).
`theCell`	On input, specifies the first cell whose selection status should be checked. If `next` is `TRUE`, then this parameter on output indicates the next selected cell greater than or equal to the cell specified on input. Otherwise, this parameter remains unchanged.
`lHandle`	The list in which the selection is being checked.

DESCRIPTION

The behavior of the `LGetSelect` function depends on the value specified in the `next` parameter.

If `next` is `TRUE`, then `LGetSelect` searches the list for the first selected cell beginning at the cell specified by `theCell`. (In particular, `LGetSelect` first checks cells in row `theCell.v`, and then cells in the next row, and so on.) If it finds a selected cell, `LGetSelect` returns `TRUE` and sets the parameter `theCell` to the coordinates of the selected cell. If it does not find a selected cell, `LGetSelect` returns `FALSE`.

If `next` is `FALSE`, then `LGetSelect` checks only the cell specified by the parameter `theCell`. If the cell is selected, `LGetSelect` returns `TRUE`. Otherwise, it returns `FALSE`.

SPECIAL CONSIDERATIONS

You should not call the `LGetSelect` function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LGetSelect` function are

Trap macro	Selector
`_Pack0`	$003C

SEE ALSO

For examples that determine which items are selected in a list, see "Working With List Selections" beginning on page 4-34.

LSetSelect

You can use the `LSetSelect` procedure to select or deselect a cell.

```
PROCEDURE LSetSelect (setIt: Boolean; theCell: Cell;
                      lHandle: ListHandle);
```

`setIt`	A Boolean value that indicates whether the `LSetSelect` procedure should select or deselect the specified cell. Specify `TRUE` to select the cell; specify `FALSE` to deselect the cell.
`theCell`	The cell to be selected or deselected.
`lHandle`	The list containing the cell to be selected or deselected.

DESCRIPTION

If `setIt` is `TRUE`, then the `LSetSelect` procedure selects the cell specified by the `theCell` parameter in the list specified by `lHandle`. If the cell is already selected, `LSetSelect` does nothing.

If `setIt` is `FALSE`, then `LSetSelect` deselects the cell specified by `theCell`. If the cell is already deselected, `LSetSelect` does nothing.

If a cell's selection status is changed and the cell is visible, `LSetSelect` redraws the cell.

SPECIAL CONSIDERATIONS

You should not call the `LSetSelect` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LSetSelect` procedure are

Trap macro	Selector
`_Pack0`	$005C

SEE ALSO

For examples that change the items selected in a list, see "Working With List Selections" beginning on page 4-34.

Accessing and Manipulating Cell Data

To change the data contained in a cell, your application ordinarily uses the `LSetCell` procedure. Alternatively, it can use the `LAddToCell` procedure to append data to a cell, or the `LClrCell` procedure to clear all data from a cell. To find the data in a cell, your application can use the `LGetCellDataLocation` procedure to find the location of a cell's data in memory. Or, your application can use the `LGetCell` procedure to copy the data elsewhere in memory.

LSetCell

You can use the `LSetCell` procedure to change the data contained in a cell.

```
PROCEDURE LSetCell (dataPtr: Ptr; dataLen: Integer; theCell: Cell;
                    lHandle: ListHandle);
```

`dataPtr`	A pointer to the new data for a cell.
`dataLen`	The length in bytes of the data pointed to by the `dataPtr` parameter.
`theCell`	The coordinates of the cell to hold the new data.
`lHandle`	The list containing the cell given in the `theCell` parameter.

DESCRIPTION

The `LSetCell` procedure sets the data of the cell specified by the parameter `theCell` to `dataLen` bytes of data beginning at the location specified by `dataPtr`. Any previous cell data in `theCell` is replaced.

If the cell coordinates specified by the `theCell` parameter are invalid, then `LSetCell` does nothing.

▲ **WARNING**
If there is insufficient memory in the heap, the `LSetCell` procedure may fail to set the cell's data. ▲

If the data of a visible cell is changed and the automatic drawing mode is enabled, `LSetCell` updates the list.

SPECIAL CONSIDERATIONS

You should not call the `LSetCell` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LSetCell` procedure are

Trap macro	Selector
`_Pack0`	$0058

SEE ALSO

For an example that sets the data of cells in a list, see Listing 4-4 on page 4-31.

LAddToCell

You can use the `LAddToCell` procedure to append data to the data already contained in a cell.

```
PROCEDURE LAddToCell (dataPtr: Ptr; dataLen: Integer;
                      theCell: Cell; lHandle: ListHandle);
```

`dataPtr`	A pointer to the data to be appended.
`dataLen`	The length in bytes of the data pointed to by the `dataPtr` parameter.
`theCell`	The coordinates of the cell to which the data should be appended.
`lHandle`	The list containing the cell given in the `theCell` parameter.

DESCRIPTION

The `LAddToCell` procedure appends `dataLen` bytes of data beginning at the location specified by `dataPtr` to data already contained in the cell specified by the parameter `theCell`.

If the cell coordinates specified by the parameter `theCell` are invalid, then the `LAddToCell` procedure does nothing.

If the data of a visible cell is changed and the automatic drawing mode is enabled, `LAddToCell` updates the list.

SPECIAL CONSIDERATIONS

You should not call the `LAddToCell` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LAddToCell` procedure are

Trap macro	Selector
`_Pack0`	$000C

LClrCell

You can use the `LClrCell` procedure to clear the data contained in a cell.

```
PROCEDURE LClrCell (theCell: Cell; lHandle: ListHandle);
```

`theCell`	The coordinates of the cell to be cleared.
`lHandle`	The list containing the cell given in the `theCell` parameter.

DESCRIPTION

The `LClrCell` procedure clears the data contained in the cell specified by the `theCell` parameter.

If the cell coordinates specified by the `theCell` parameter are invalid, then the `LClrCell` procedure does nothing.

If the data of a visible cell is cleared and the automatic drawing mode is enabled, `LClrCell` updates the list.

SPECIAL CONSIDERATIONS

You should not call the `LClrCell` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LClrCell` procedure are

Trap macro	Selector
`_Pack0`	$001C

LGetCellDataLocation

You can find the memory location of cell data by using the `LGetCellDataLocation` procedure. The `LGetCellDataLocation` procedure is also available as the `LFind` procedure.

```
PROCEDURE LGetCellDataLocation (VAR offset, len: Integer;
                               theCell: Cell;
                               lHandle: ListHandle);
```

`offset`	The `LGetCellDataLocation` procedure returns in this parameter the offset of the cell's data, specified from the beginning of the data handle referenced by the `cells` field of the list record.
`len`	The `LGetCellDataLocation` procedure returns in this parameter the length of the cell's data in bytes.
`theCell`	The cell whose data's location is sought.
`lHandle`	The list containing the cell specified by the parameter `theCell`.

DESCRIPTION

Your application can use the `LGetCellDataLocation` procedure to read cell data. The `cells` field of the list record contains a handle to a relocatable block used to store all cell data. When `LGetCellDataLocation` returns, the `offset` parameter contains the offset of the specified cell's data in this relocatable block, and the `len` parameter specifies the length in bytes of the cell's data. In other words, the first byte of cell data is located at `Ptr(ORD4(lHandle^^.cells^) + offset)`, and the last byte of cell data is located at `Ptr(ORD4(lHandle^^.cells^) + offset + len)`.

If the cell coordinates specified by the parameter `theCell` are invalid, then `LGetCellDataLocation` sets the `offset` and `len` parameters to –1.

▲ **WARNING**
Your application should not modify the contents of the `cells` field directly. To change a cell's data, use the `LSetCell` procedure or the `LAddToCell` procedure. ▲

SPECIAL CONSIDERATIONS

You should not call the `LGetCellDataLocation` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LGetCellDataLocation` procedure are

Trap macro	Selector
`_Pack0`	$0034

SEE ALSO

For an example that uses the `LGetCellDataLocation` procedure to get the data of a cell, see Listing 4-13 on page 4-41.

LGetCell

You can use the `LGetCell` procedure to copy a cell's data.

```
PROCEDURE LGetCell (dataPtr: Ptr; VAR dataLen: Integer;
                    theCell: Cell; lHandle: ListHandle);
```

`dataPtr`	A pointer to the location to which to copy the cell's data.
`dataLen`	On entry, specifies the maximum number of bytes to copy. On exit, indicates the number of bytes actually copied.
`theCell`	The cell whose data is to be copied.
`lHandle`	The list containing the cell specified by the parameter `theCell`.

DESCRIPTION

The `LGetCell` procedure copies up to `dataLen` bytes of the data of the cell specified by `theCell` to the memory location pointed to by `dataPtr`. If the cell data is longer than `dataLen`, only `dataLen` bytes are copied and the `dataLen` parameter is unchanged. If the cell data is shorter than `dataLen`, then `LGetCell` sets `dataLen` to the length in bytes of the cell's data.

SPECIAL CONSIDERATIONS

You should not call the `LGetCell` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LGetCell` procedure are

Trap macro	Selector
`_Pack0`	$0038

Responding to Events Affecting Lists

Your application can respond to mouse-down events in a list, activate events for a window containing a list, and update events for a window containing a list simply by calling the `LClick` function, the `LActivate` procedure, and the `LUpdate` procedure, respectively. The List Manager does not include a routine that automatically responds to keyboard events; for information on responding to those, see "Supporting Keyboard Navigation of Lists" beginning on page 4-45.

LClick

To process a mouse-down event in a list, use the `LClick` function.

```
FUNCTION LClick (pt: Point; modifiers: Integer;
                 lHandle: ListHandle): Boolean;
```

`pt`	The location in local coordinates of the mouse-down event. Your application can simply call `GlobalToLocal(myEvent.where)` and then pass `myEvent.where` in this parameter.
`modifiers`	An integer value corresponding to the `modifiers` field of the event record.
`lHandle`	The list in which the mouse-down event occurred.

DESCRIPTION

The `LClick` function responds to the mouse-down event whose location and modifiers are specified by the `pt` and `modifiers` parameters. The `LClick` function handles all user interaction until the user releases the mouse button. The `LClick` function returns `TRUE` if the click was a double-click, or `FALSE` otherwise.

If the `pt` parameter specifies a portion of the list's visible rectangle, then cells are selected with an algorithm that depends on the list's selection flags and on the `modifiers` parameter. If the user drags the cursor above or below the list's visible rectangle and vertical autoscrolling is enabled, then the List Manager vertically autoscrolls the list. If the user drags the cursor to the right or the left of the list's visible rectangle and horizontal autoscrolling is enabled, then the List Manager horizontally autoscrolls the list.

If the `pt` parameter specifies a point within the list's scroll bar, then the List Manager calls the scroll bar's control definition procedure to track the cursor and it scrolls the list appropriately.

SPECIAL CONSIDERATIONS

You should not call the `LClick` function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LClick` function are

Trap macro	Selector
`_Pack0`	$0018

SEE ALSO

For information on enabling and disabling autoscrolling, see "About the List Manager" beginning on page 4-22. For information on responding to mouse-down events, see "Responding to Events Affecting a List" on page 4-32.

LActivate

When your application receives an activate event for a window containing a list, it should activate or deactivate the list as appropriate. You can use the `LActivate` procedure to perform highlighting of the cells and to show or hide any scroll bars.

```
PROCEDURE LActivate (act: Boolean; lHandle: ListHandle);
```

`act`	A Boolean value that indicates whether the list should be activated. Specify `TRUE` to activate the list. Specify `FALSE` to deactivate the list.
`lHandle`	The list to be activated or deactivated.

DESCRIPTION

The `LActivate` procedure activates the list specified by the `lHandle` parameter if `act` is `TRUE` and deactivates it otherwise.

If a list is being deactivated, `LActivate` removes highlighting from selected cells and hides the scroll bars. If a list is being activated, `LActivate` highlights selected cells and shows the scroll bars.

The `LActivate` procedure has no effect on a list's size box, if one exists.

SPECIAL CONSIDERATIONS

You should not call the `LActivate` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LActivate` procedure are

Trap macro	Selector
`_Pack0`	$0000

SEE ALSO

For information on responding to activate events in lists, see “Responding to Events Affecting a List” beginning on page 4-32. For general information on events, see the chapter “Event Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

LUpdate

To respond to an update event, use the `LUpdate` procedure.

```
PROCEDURE LUpdate (theRgn: RgnHandle; lHandle: ListHandle);
```

`theRgn`	The visible region of the list’s port after a call to the Window Manager’s `BeginUpdate` procedure.
`lHandle`	The list to be updated.

DESCRIPTION

The `LUpdate` procedure redraws all visible cells in the list specified by the `lHandle` parameter that intersect the region specified by the parameter `theRgn`. It also redraws the scroll bars if they intersect the region.

You should bracket calls to `LUpdate` by calls to the Window Manager procedures `BeginUpdate` and `EndUpdate`.

SPECIAL CONSIDERATIONS

You should not call the `LUpdate` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LUpdate` procedure are

Trap macro	Selector
`_Pack0`	$0064

SEE ALSO

For information on responding to update events in lists, see “Responding to Events Affecting a List” beginning on page 4-32. For general information on events, see the chapter “Event Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

Modifying a List's Appearance

Your application can use the LSetDrawingMode procedure to enable or disable automatic drawing of lists. If your application uses LSetDrawingMode to temporarily disable list drawing, then it must call the LDraw procedure to draw a cell when its appearance changes, or when new rows or columns are added to the list. To automatically scroll a list so that the first selected cell is the first cell visible, your application can use the LAutoScroll procedure. To scroll a list a certain number of cells horizontally and vertically, your application can use the LScroll procedure.

LSetDrawingMode

You can use the LSetDrawingMode procedure to change the automatic drawing mode specified when creating a list. The LSetDrawingMode procedure is also available as the LDoDraw procedure.

```
PROCEDURE LSetDrawingMode (drawIt: Boolean; lHandle: ListHandle);
```

drawIt	A Boolean value that indicates whether the List Manager should enable the automatic drawing mode. Specify TRUE to enable the automatic drawing mode. Specify FALSE to disable the automatic drawing mode.
lHandle	The list whose drawing mode is being changed.

DESCRIPTION

The LSetDrawingMode procedure sets the List Manager's drawing mode for the list specified by the lHandle parameter to the state specified by the drawIt parameter.

While the automatic drawing mode is turned off, all cell drawing and highlighting are disabled, and the scroll bar does not function properly. Thus, your application should disable the automatic drawing mode only for short periods of time. After enabling it, your application should ensure that the list is redrawn.

SPECIAL CONSIDERATIONS

You should not call the LSetDrawingMode procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the LSetDrawingMode procedure are

Trap macro	Selector
_Pack0	$002C

SEE ALSO

For an example that disables and then reenables the automatic drawing mode, see "Adding Rows and Columns to a List" beginning on page 4-30.

LDraw

You can use the `LDraw` procedure to draw a cell in a list. Ordinarily, you should only need to use `LDraw` when the automatic drawing mode has been disabled.

```
PROCEDURE LDraw (theCell: Cell; lHandle: ListHandle);
```

`theCell`	The cell to draw.
`lHandle`	The list containing the cell identified by the parameter `theCell`.

DESCRIPTION

The `LDraw` procedure draws the cell specified by the parameter `theCell`. The List Manager makes the list's graphics port the current port, sets the clipping region to the cell's rectangle, and calls the list definition procedure to draw the cell. It restores the clipping region and port before exiting.

SPECIAL CONSIDERATIONS

You should not call the `LDraw` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LDraw` procedure are

Trap macro	Selector
`_Pack0`	$0030

LAutoScroll

You can use the `LAutoScroll` procedure to make the first selected cell visible.

```
PROCEDURE LAutoScroll (lHandle: ListHandle);
```

`lHandle`	The list to be scrolled.

DESCRIPTION

The `LAutoScroll` procedure scrolls the list specified by the `lHandle` parameter so that the first selected cell is in the upper-left corner of the list's visible rectangle.

SPECIAL CONSIDERATIONS

You should not call the `LAutoScroll` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LAutoScroll` procedure are

Trap macro	Selector
`_Pack0`	$0010

LScroll

You can use the `LScroll` procedure to scroll a list a specified number of rows and columns.

```
PROCEDURE LScroll (dCols: Integer; dRows: Integer;
                      lHandle: ListHandle);
```

`dCols`	The number of columns to scroll. Specify a positive number to scroll down (that is, each cell moves up), and a negative number to scroll up.
`dRows`	The number of rows to scroll. Specify a positive number to scroll right (that is, each cell moves left), and a negative number to scroll left.
`lHandle`	The list to be scrolled.

DESCRIPTION

The `LScroll` procedure scrolls the list specified by the `lHandle` procedure the number of columns and rows specified by `dCols` and `dRows`. The List Manager will not scroll beyond the data bounds of the list.

If the automatic drawing mode is enabled, `LScroll` does all necessary updating of the list.

SPECIAL CONSIDERATIONS

You should not call the `LScroll` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LScroll` procedure are

Trap macro	Selector
`_Pack0`	$0050

Searching a List for a Particular Item

You can use the `LSearch` function to search a list for a particular item.

LSearch

You can use the `LSearch` function to find a cell whose data matches data that you specify.

```
FUNCTION LSearch (dataPtr: Ptr; dataLen: Integer; searchProc: Ptr;
                  VAR theCell: Cell; lHandle: ListHandle)
                  : Boolean;
```

`dataPtr`	A pointer to the data being searched for.
`dataLen`	The length in bytes of the data being searched for.
`searchProc`	A pointer to a function to be used to compare the data being searched for with cell data. If `NIL`, the Text Utilities Package function `IUMagIDString` is used.
`theCell`	The first cell to be searched. If `LSearch` finds a match, it returns in this parameter the coordinates of the first cell whose data matches the data being searched for.
`lHandle`	The list to be searched.

DESCRIPTION

Your application can use the `LSearch` function to search the list specified by the `lHandle` parameter beginning at the cell specified by the parameter `theCell` for a match. If `LSearch` finds a match, it returns `TRUE` and sets the parameter `theCell` to the coordinates of the first cell whose data matches the data specified by the `dataPtr` and `dataLen` parameters. Otherwise, `LSearch` returns `FALSE`.

The `LSearch` function determines whether a cell's data matches the search data by calling the `IUMagIDString` function, or the function specified by the `searchProc` parameter. If that function returns 0, `LSearch` has found a match; otherwise, `LSearch` checks the next cell in the list.

SPECIAL CONSIDERATIONS

You should not call the `LSearch` function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LSearch` function are

Trap macro	Selector
`_Pack0`	$0054

SEE ALSO

For examples of use of the `LSearch` function, see “Searching a List for a Particular Item” beginning on page 4-43. For information on the syntax of a custom match function, see “Match Functions” beginning on page 4-99. For information on the `IUMagIDString` function, see *Inside Macintosh: Text*.

Changing the Size of Cells and Lists

Usually, once your application creates a list, it should not need to change the cell size or the size of the list itself. However, there may be instances in which changing one or both is desirable. For example, if your list is on the lower-right side of a window that is resizable, then your application must resize the list if the window containing it is resized. Your application can do that with the `LSize` procedure. To resize the cells in a list, your application can use the `LCellSize` procedure.

LSize

You can change the size of a list by using the `LSize` procedure. Usually, you need to do this only after calling the Window Manager procedure `SizeWindow`.

```
PROCEDURE LSize (listWidth: Integer; listHeight: Integer;
                 lHandle: ListHandle);
```

`listWidth`	The new width (in pixels) of the list’s visible rectangle.
`listHeight`	The new height (in pixels) of the list’s visible rectangle.
`lHandle`	The list whose size is being changed.

DESCRIPTION

The `LSize` procedure adjusts the lower-right side of the list specified by the `lHandle` parameter so that the list's visible rectangle is the width and height specified by the `listWidth` and `listHeight` parameters.

Because the list's visible rectangle does not include room for the scroll bars, your application should make `listWidth` 15 pixels less than the desired width of the list if it contains a vertical scroll bar, and it should make `listHeight` 15 pixels less than the desired height of the list if it contains a horizontal scroll bar.

The contents of the list and the scroll bars are adjusted and redrawn as necessary. However, `LSize` does not draw a border around the list's rectangle. Also, it does not erase any portions of the old list that may still be visible. However, this approach should not be a problem if your application only calls `LSize` after the user resizes a window containing a list in its lower-right corner.

SPECIAL CONSIDERATIONS

You should not call the `LSize` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LSize` procedure are

Trap macro	Selector
`_Pack0`	$0060

SEE ALSO

For information on the Window Manager's `SizeWindow` procedure, see the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

LCellSize

You can change the size of cells by using the `LCellSize` procedure. All cells in a list must be the same size, however.

```
PROCEDURE LCellSize (cSize: Point; lHandle: ListHandle);
```

`cSize`	The new size of each cell in the list.
`lHandle`	The list whose cells' size is being changed.

DESCRIPTION

The `LCellSize` procedure sets the `cellSize` field of the list record referenced by the `lHandle` parameter to the value of the `cSize` parameter. That is, the list's new cells will be of width `cSize.h` and of height `cSize.v`.

The `LCellSize` procedure updates the list's visible rectangle to contain cells of the specified size. However, `LCellSize` does not redraw any cells.

SPECIAL CONSIDERATIONS

You should not call the `LCellSize` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LCellSize` procedure are

Trap macro	Selector
`_Pack0`	$0014

Getting Information About Cells

The List Manager provides three routines that allow your application to obtain information related to cells. Your application can use the `LNextCell` function to find the next cell in a list; this is useful, for example, when performing some operation on all cells in a list. To find the local QuickDraw coordinates of a cell's rectangle, your application can use the `LRect` procedure. Finally, to determine the cell coordinates of the last cell clicked, your application can use the `LLastClick` function.

LNextCell

You can use the `LNextCell` function to find the next cell in a given row, in a given column, or in an entire list.

```
FUNCTION LNextCell (hNext: Boolean; vNext: Boolean;
                    VAR theCell: Cell; lHandle: ListHandle)
                    : Boolean;
```

`hNext`	A Boolean value that indicates whether `LNextCell` should check columns other than the current column.
`vNext`	A Boolean value that indicates whether `LNextCell` should check rows other than the current row.
`theCell`	The coordinates of the current cell.
`lHandle`	The list in which to find the next cell.

DESCRIPTION

The behavior of the `LNextCell` function hinges on the values of the `hNext` and `vNext` parameters.

If `hNext` is TRUE and `vNext` is FALSE, then `LNextCell` tries to find a cell whose coordinates are greater than those of the cell specified in `theCell` parameter but that is in the same row as `theCell`. If successful, `LNextCell` sets the value of the `theCell` parameter to the first such cell and returns TRUE. If the cell initially specified by `theCell` is the last cell in its row, however, `LNextCell` returns FALSE.

If `hNext` is FALSE and `vNext` is TRUE, then `LNextCell` tries to find a cell whose coordinates are greater than those of the cell specified in `theCell` parameter but that is in the same column as `theCell`. If successful, `LNextCell` sets the value of the `theCell` parameter to the first such cell and returns TRUE. If, however, the cell initially specified by `theCell` is the last cell in its column, `LNextCell` returns FALSE.

If both `hNext` and `vNext` are TRUE, then `LNextCell` tries to find a cell whose coordinates are greater than those of the cell specified in the parameter `theCell`. If successful, `LNextCell` sets the value of the `theCell` parameter to the first such cell and returns TRUE. If, however, the cell initially specified by `theCell` is the last cell in the list, `LNextCell` returns FALSE.

Finally, if both `hNext` and `vNext` are FALSE, `LNextCell` simply returns FALSE.

SPECIAL CONSIDERATIONS

You should not call the `LNextCell` function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LNextCell` function are

Trap macro	Selector
`_Pack0`	$0048

SEE ALSO

Listing 4-7 on page 4-34 and Listing 4-8 on page 4-35 show how to find the first and last selected cell in a list.

LRect

You can use the `LRect` procedure to find a rectangle that encloses a cell. Because the List Manager automatically draws cells, few applications need to call this procedure directly.

```
PROCEDURE LRect (VAR cellRect: Rect; theCell: Cell;
                 lHandle: ListHandle);
```

`cellRect`	The `LRect` procedure returns in this parameter the rectangle enclosing the cell, specified in local coordinates of the list's graphics port. This rectangle is not necessarily within the list's rectangle.
`theCell`	The cell for which an enclosing rectangle is sought.
`lHandle`	The list containing the cell specified by the parameter `theCell`.

DESCRIPTION

The `LRect` procedure calculates the coordinates of the rectangle enclosing the cell specified by the `theCell` parameter. The procedure does not check whether the cell is actually contained within the list's visible rectangle.

If the `theCell` parameter specifies cell coordinates not contained within the list, the `LRect` procedure sets the `cellRect` parameter to (0,0,0,0).

SPECIAL CONSIDERATIONS

You should not call the `LRect` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LRect` procedure are

Trap macro	Selector
`_Pack0`	$004C

LLastClick

You can use the `LLastClick` function to determine the coordinates of the last cell clicked in a particular list.

```
FUNCTION LLastClick (lHandle: ListHandle): Cell;
```

`lHandle`	The list to be checked for the last cell clicked.

DESCRIPTION

The `LLastClick` function returns the cell coordinates of the last cell clicked. If the user has not clicked a cell since the creation of the list, then both the `h` and `v` fields of the cell returned contain negative numbers.

Note that the last cell clicked is not necessarily the last cell selected. The user could Shift-click in one cell and then drag the cursor to select other cells.

SPECIAL CONSIDERATIONS

You should not call the `LLastClick` function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LLastClick` function are

Trap macro	Selector
`_Pack0`	$0040

Application-Defined Routines

The List Manager provides several ways that your application can customize its routines. First, your application can define a list definition procedure to create a list that displays cells graphically. Second, your application can create a custom match function to search for a particular item in a list. Finally, you can override the default click-loop procedure by providing a custom click-loop procedure.

List Definition Procedures

Your application can write a list definition procedure to customize list display. For example, you can write a list definition procedure to support the display of color icons. A custom list definition procedure must be compiled as a code resource of type `'LDEF'` and added to the resource file of the application that needs to use it.

MyLDEF

A list definition procedure has the following syntax:

```
PROCEDURE MyLDEF (message: Integer; selected: Boolean;
                  VAR cellRect: Rect; theCell: Cell;
                  dataOffset: Integer; dataLen: Integer;
                  theList: ListHandle);
```

`message` A value that identifies the operation to be performed. These constants specify the four types of messages:

```
CONST
   lInitMsg    = 0;  {do any special initialization}
   lDrawMsg    = 1;  {draw the cell}
   lHiliteMsg  = 2;  {invert cell's highlight state}
   lCloseMsg   = 3;  {do any special disposal action}
```

`selected` A Boolean value that indicates whether the cell specified by the `theCell` parameter should be highlighted. This parameter is defined only for the `lDrawMessage` and `lHiliteMsg` messages.

`cellRect` The rectangle (in local coordinates of the list's graphics port) that encloses the cell specified by the `theCell` parameter. Although this parameter is defined as a VAR parameter, your list definition procedure must not change the coordinates of the rectangle. This parameter is defined only for the `lDrawMessage` and `lHiliteMsg` messages.

`theCell` The coordinates of the cell to be drawn or highlighted. This parameter is defined only for the `lDrawMessage` and `lHiliteMsg` messages.

`dataOffset`
The location of the cell data associated with the cell specified by the `theCell` parameter. The location is specified as an offset from the beginning of the relocatable block referenced by the `cells` field of the list record. This parameter is defined only for the `lDrawMessage` and `lHiliteMsg` messages.

`dataLen` The length in bytes of the cell data associated with the cell specified by the `theCell` parameter. This parameter is defined only for the `lDrawMessage` and `lHiliteMsg` messages.

`theList` The list for which a message is being sent. Your application can access the list's list record, or it can call List Manager routines to manipulate the list.

DESCRIPTION

The List Manager calls your list definition procedure whenever an application using the procedure creates a new list with the `LNew` function, needs a cell to be drawn, needs a cell's highlighting state to be reversed, or has called the `LDispose` procedure to dispose of a list.

In response to the `lInitMsg` message, your list definition procedure should perform any special initialization needed for a list. For example, the procedure might set fields of the list record, such as the `cellSize` and `indent` fields, to appropriate values. Your list definition procedure does not necessarily need to do anything in response to the initialization message. If it does nothing, then memory is still allocated for the list, and fields of the list record are set to the same values as they would be set to if the default list definition procedure were being used. (For more information on those values, see "About the List Manager" beginning on page 4-22.)

Your list definition procedure should draw the cell specified by the `theCell` parameter after receiving an `lDrawMsg` message. The procedure must ensure that it does not draw anywhere but within the rectangle specified by the `cellRect` parameter. If the `selected` parameter is `TRUE`, then your list definition procedure should draw the cell in its highlighted state; otherwise, it should draw the cell without highlighting. When drawing, your list definition procedure should take care not to permanently change any characteristics of the drawing environment.

Your list definition procedure should respond to the `lHiliteMsg` message by reversing the selection status of the cell contained within the rectangle specified by the `cellRect` parameter. If a cell is highlighted, your list definition procedure should remove the highlighting; if a cell is not highlighted, your list definition procedure should highlight it.

The List Manager sends your list definition procedure an `lCloseMsg` message before it disposes of a list and its data. Your list definition procedure need only respond to this message if additional memory has been allocated for the list. For example, your list definition procedure might allocate a relocatable block in response to the `lInitMsg` message. In this case, your list definition procedure would need to dispose of this relocatable block in response to the `lCloseMsg` message. Or, if your list definition procedure defines cells simply to contain pointers or handles to data stored elsewhere in memory, it would need to dispose of that memory in response to the `lCloseMsg` message.

SPECIAL CONSIDERATIONS

You must compile a list definition procedure as a resource of type `'LDEF'` before it can be used by an application.

Because a list definition procedure is stored in a code resource, it cannot have its own global variables that it accesses through the A5 register. (Some development systems, however, may allow code resources to access global variables through some other register, such as A4. See your development system's documentation for more information.) If your list definition procedure needs access to global data, it might store a handle to such data in the `refCon` or `userHandle` fields of the list record; however, applications would not then be able to use these fields for their own purposes.

ASSEMBLY-LANGUAGE INFORMATION

The entry point of a list definition procedure must be at the beginning.

SEE ALSO

For an example of a list definition procedure, see "Writing Your Own List Definition Procedure" beginning on page 4-58.

Match Functions

You can pass a pointer to a custom match function as the third parameter to the `LSearch` function. Alternatively, your application can specify `NIL` to use the Text Utilities function `IUMagIDString`, the default match function.

MyMatchFunction

A match function must have the following syntax:

```
FUNCTION MyMatchFunction (cellDataPtr, searchDataPtr: Ptr;
                          cellDataLen, searchDataLen: Integer)
                          : Integer;
```

`cellDataPtr`
: A pointer to the data contained in a cell.

`searchDataPtr`
: A pointer to the data being searched for.

`cellDataLen`
: The number of bytes of data contained in the cell specified by the `cellDataPtr` parameter.

`searchDataLen`
: The number of bytes of data contained in the cell specified by the `searchDataPtr` parameter.

DESCRIPTION

A custom match function must compare the data defined by the `cellDataPtr` and `cellDataLen` parameters with the data defined by the `searchDataPtr` and `searchDataLen` parameters. If the cell data matches the search data, your match function should return 0. Otherwise, your match function should return 1.

Your match function can use any technique you choose to compare the data. For example, your match function might consider the search data to be equivalent to the cell data if both are the same length. Or, your match function might only report a match if the search data can be found somewhere within the cell data.

The default match function, `IUMagIDString`, returns 0 if the search data exactly matches the cell data, but `IUMagIDString` considers the strings `'Rose'` and `'rosé'` to be equivalent. If your application simply needs a match function that works like `IUMagIDString` but considers `'Rose'` to be different from `'rosé'`, you do not need to write a custom match function. Instead, your application can simply pass `@IUMagString` as the third parameter to the `LSearch` function.

SPECIAL CONSIDERATIONS

A custom match function does not execute at interrupt time. Instead, it is called directly by the `LSearch` function. Thus, a match function can allocate memory, and it does not need to adjust the value contained in the A5 register.

SEE ALSO

For information on the `IUMagIDString` function and the `IUMagString` function, see *Inside Macintosh: Operating System Utilities*.

For examples of match functions, see “Searching a List for a Particular Item” beginning on page 4-43.

Click-Loop Procedures

The List Manager supports the use of custom click-loop procedures to allow you to override the standard click-loop procedure that is used to select cells and automatically scroll a list. To define a custom click-loop procedure, specify a pointer to your procedure in the `lClikLoop` field of the list record. Because the `selFlags` field of the list record (described in “Customizing Cell Highlighting” beginning on page 4-38) already provides a means of customizing the algorithm the List Manager uses to highlight list cells, in most cases you should not need to define a custom click-loop procedure.

MyClickLoop

A click-loop procedure must have the following syntax:

```
PROCEDURE MyClickLoop;
```

DESCRIPTION

If your application defines a custom click-loop procedure, then the `LClick` function repeatedly calls the procedure until the user releases the mouse button. A click-loop procedure may perform any processing desired when it is executed.

Because no parameters are passed to the click-loop procedure, your click-loop procedure probably needs to access a global variable that contains a handle to the list record, which contains information about the location of the cursor and other information potentially of interest to a click-loop procedure. You might also create a global variable that stores the state of the modifier keys immediately before a call to the `LClick` function. You would need to set these global variables immediately before calling `LClick`.

A click-loop procedure should ordinarily set the Z flag to 1 just before returning. If a click-loop procedure sets the Z flag to 0, then the `LClick` function immediately returns.

SPECIAL CONSIDERATIONS

A click-loop procedure does not execute at interrupt time. Instead, it is called directly by the `LClick` function. Thus, a click-loop procedure can allocate memory, and it does not need to adjust the value contained in the A5 register.

ASSEMBLY-LANGUAGE INFORMATION

Your click-loop procedure should ordinarily set register D0 to 1. To stop the `LClick` function from calling your procedure for the current mouse-down event, set register D0 to 0.

For your convenience, register D5 contains the current mouse location.

Summary of the List Manager

Pascal Summary

Constants

```
CONST
   {masks for listFlags field of list record}
   lDoVAutoScroll = 2;        {allow vertical autoscrolling}
   lDoHAutoScroll = 1;        {allow horizontal autoscrolling}

   {masks for selFlags field of list record}
   lOnlyOne       = -128;     {allow only one item to be selected at once}
   lExtendDrag    = 64;       {enable multiple item selection without Shift}
   lNoDisjoint    = 32;       {prevent discontiguous selections}
   lNoExtend      = 16;       {reset list before responding to Shift-click}
   lNoRect        = 8;        {Shift-drag selects items passed by cursor}
   lUseSense      = 4;        {allow use of Shift key to deselect items}
   lNoNilHilite   = 2;        {disable highlighting of empty cells}

   {messages to list definition procedure}
   lInitMsg       = 0;        {do any special list initialization}
   lDrawMsg       = 1;        {draw the cell}
   lHiliteMsg     = 2;        {invert cell's highlight state}
   lCloseMsg      = 3;        {take any special disposal action}
```

Data Types

```
TYPE
   Cell = Point;                      {cell.v contains row number}
                                      {cell.h contains column number}

   DataArray   = PACKED ARRAY[0..32000] OF Char;
   DataPtr     = ^DataArray;
   DataHandle  = ^DataPtr;
```

```
ListRec =
RECORD
   rView:      Rect;              {list's display rectangle}
   port:       GrafPtr;           {list's graphics port}
   indent:     Point;             {indent distance for drawing}
   cellSize:   Point;             {size in pixels of a cell}
   visible:    Rect;              {boundary of visible cells}
   vScroll:    ControlHandle;     {vertical scroll bar}
   hScroll:    ControlHandle;     {horizontal scroll bar}
   selFlags:   SignedByte;        {selection flags}
   lActive:    Boolean;           {TRUE if list is active}
   lReserved:  SignedByte;        {reserved}
   listFlags:  SignedByte;        {automatic scrolling flags}
   clikTime:   LongInt;           {TickCount at time of last click}
   clikLoc:    Point;             {position of last click}
   mouseLoc:   Point;             {current mouse location}
   lClikLoop:  Ptr;               {routine called by LClick}
   lastClick:  Cell;              {last cell clicked}
   refCon:     LongInt;           {for application use}
   listDefProc:                   {list definition procedure}
               Handle;
   userHandle: Handle;            {for application use}
   dataBounds: Rect;              {boundary of cells allocated}
   cells:      DataHandle;        {cell data}
   maxIndex:   Integer;           {used internally}
   cellArray:                     {offsets to data}
               ARRAY[1..1] OF Integer;
END;

ListPtr     = ^ListRec;           {pointer to a list record}
ListHandle  = ^ListPtr;           {handle to a list record}
```

List Manager Routines

Creating and Disposing of Lists

```
FUNCTION LNew              (rView: Rect; dataBounds: Rect; cSize: Point;
                            theProc: Integer; theWindow: WindowPtr;
                            drawIt, hasGrow, scrollHoriz,
                            scrollVert: Boolean): ListHandle;
PROCEDURE LDispose         (lHandle: ListHandle);
```

Adding and Deleting Columns and Rows To and From a List

```
FUNCTION LAddColumn          (count: Integer; colNum: Integer;
                              lHandle: ListHandle): Integer;
FUNCTION LAddRow             (count: Integer; rowNum: Integer;
                              lHandle: ListHandle): Integer;
PROCEDURE LDelColumn         (count: Integer; colNum: Integer;
                              lHandle: ListHandle);
PROCEDURE LDelRow            (count: Integer; rowNum: Integer;
                              lHandle: ListHandle);
```

Determining or Changing the Selection

```
FUNCTION LGetSelect          (next: Boolean; VAR theCell: Cell;
                              lHandle: ListHandle): Boolean;
PROCEDURE LSetSelect         (setIt: Boolean; theCell: Cell;
                              lHandle: ListHandle);
```

Accessing and Manipulating Cell Data

```
PROCEDURE LSetCell           (dataPtr: Ptr; dataLen: Integer; theCell: Cell;
                              lHandle: ListHandle);
PROCEDURE LAddToCell         (dataPtr: Ptr; dataLen: Integer; theCell: Cell;
                              lHandle: ListHandle);
PROCEDURE LClrCell           (theCell: Cell; lHandle: ListHandle);
{the LGetCellDataLocation procedure is also available as the LFind procedure}
PROCEDURE LGetCellDataLocation
                             (VAR offset, len: Integer; theCell: Cell;
                              lHandle: ListHandle);
PROCEDURE LGetCell           (dataPtr: Ptr; VAR dataLen: Integer;
                              theCell: Cell; lHandle: ListHandle);
```

Responding to Events Affecting Lists

```
FUNCTION LClick              (pt: Point; modifiers: Integer;
                              lHandle: ListHandle): Boolean;
PROCEDURE LActivate          (act: Boolean; lHandle: ListHandle);
PROCEDURE LUpdate            (theRgn: RgnHandle; lHandle: ListHandle);
```

Modifying a List's Appearance

```
{the LSetDrawingMode procedure is also available as the LDoDraw procedure}
PROCEDURE LSetDrawingMode   (drawIt: Boolean; lHandle: ListHandle);
PROCEDURE LDraw             (theCell: Cell; lHandle: ListHandle);
PROCEDURE LAutoScroll       (lHandle: ListHandle);
PROCEDURE LScroll           (dCols: Integer; dRows: Integer;
                             lHandle: ListHandle);
```

Searching a List for a Particular Item

```
FUNCTION LSearch            (dataPtr: Ptr; dataLen: Integer;
                             searchProc: Ptr; VAR theCell: Cell;
                             lHandle: ListHandle): Boolean;
```

Changing the Size of Cells and Lists

```
PROCEDURE LSize             (listWidth: Integer; listHeight: Integer;
                             lHandle: ListHandle);
PROCEDURE LCellSize         (cSize: Point; lHandle: ListHandle);
```

Getting Information About Cells

```
FUNCTION LNextCell          (hNext: Boolean; vNext: Boolean;
                             VAR theCell: Cell;
                             lHandle: ListHandle): Boolean;
PROCEDURE LRect             (VAR cellRect: Rect; theCell: Cell;
                             lHandle: ListHandle);
FUNCTION LLastClick         (lHandle: ListHandle): Cell;
```

Application-Defined Routines

```
PROCEDURE MyLDEF            (message: Integer; selected: Boolean;
                             VAR cellRect: Rect; theCell: Cell;
                             dataOffset: Integer; dataLen: Integer;
                             theList: ListHandle);
FUNCTION MyMatchFunction    (cellDataPtr, searchDataPtr: Ptr;
                             cellDataLen, searchDataLen: Integer): Integer;
PROCEDURE MyClickLoop;
```

C Summary

Constants

```
/*masks for listFlags field of list record*/
enum {
   lDoVAutoScroll = 2,     /*allow vertical autoscrolling*/
   lDoHAutoScroll = 1,     /*allow horizontal autoscrolling*/

/*masks for selFlags field of list record*/
   lOnlyOne = -128,        /*allow only one item to be selected at once*/
   lExtendDrag = 64,       /*enable multiple item selection without Shift*/
   lNoDisjoint = 32,       /*prevent discontiguous selections*/
   lNoExtend = 16,         /*reset list before responding to Shift-click*/
   lNoRect = 8,            /*Shift-drag selects items passed by cursor*/
   lUseSense = 4,          /*allow use of Shift key to deselect items*/
   lNoNilHilite = 2,       /*disable highlighting of empty cells*/
/*messages to list definition procedure*/
   lInitMsg = 0,           /*do any special list initialization*/
   lDrawMsg = 1,           /*draw the cell*/
   lHiliteMsg = 2,         /*invert cell's highlight state*/
   lCloseMsg = 3           /*take any special disposal action*/
};
```

Data Types

```
typdef Point Cell;         /*cell.v contains row number*/
                           /*cell.h contains column number*/

typedef char DataArray[32001], *DataPtr, **DataHandle;
```

```
struct ListRec {
   Rect           rView;            /*list's display rectangle*/
   GrafPtr        ptr;              /*list's graphics port*/
   Point          indent;           /*indent distance for drawing*/
   Point          cellSize;         /*size in pixels of a cell*/
   Rect           visible;          /*boundary of visible cells*/
   ControlHandle  vScroll;          /*vertical scroll bar*/
   ControlHandle  hScroll;          /*horizontal scroll bar*/
   char           selFlags;         /*selection flags*/
   Boolean        lActive;          /*TRUE if list is active*/
   char           lReserved;        /*reserved*/
   char           listFlags;        /*automatic scrolling flags*/
   long           clikTime;         /*TickCount at time of last click*/
   Point          clikLoc;          /*position of last click*/
   Point          mouseLoc;         /*current mouse location*/
   ProcPtr        lClikLoop;        /*routine called by LClick*/
   Cell           lastClick;        /*last cell clicked*/
   long           refCon;           /*for application use*/
   Handle         listDefProc;      /*list definition procedure*/
   Handle         userHandle;       /*for application use*/
   Rect           dataBounds;       /*boundary of cells allocated*/
   DataHandle     cells;            /*cell data*/
   short          maxIndex;         /*used internally*/
   short          cellArray[1];     /*offsets to data*/
};

typedef struct ListRect ListRect;
typedef ListRect *ListPtr, **ListHandle;
```

List Manager Routines

Creating and Disposing of Lists

```
pascal ListHandle LNew      (const Rect *rView, Rect *dataBounds,
                             Point *cSize, short theProc,
                             WindowPtr theWindow, Boolean drawIt,
                             Boolean hasGrow, Boolean scrollHoriz,
                             Boolean scrollVert);
pascal void LDispose        (ListHandle lHandle);
```

Adding and Deleting Columns and Rows To and From a List

```
pascal short LAddColumn     (short count, short colNum, ListHandle lHandle);
pascal short LAddRow        (short count, short rowNum, ListHandle lHandle);
pascal void LDelColumn      (short count, short colNum, ListHandle lHandle);
pascal void LDelRow         (short count, short rowNum, ListHandle lHandle);
```

Determining or Changing the Selection

```
pascal Boolean LGetSelect   (Boolean next, Cell *theCell,
                             ListHandle lHandle);
pascal void LSetSelect      (Boolean setIt, Cell theCell,
                             ListHandle lHandle);
```

Accessing and Manipulating Cell Data

```
pascal void LSetCell        (const void *dataPtr, short dataLen,
                             Cell theCell, ListHandle lHandle);
pascal void LAddToCell      (const void *dataPtr, short dataLen,
                             Cell theCell, ListHandle lHandle);
pascal void LClrCell        (Cell theCell, ListHandle lHandle);
/*the LGetCellDataLocation procedure is also available as */
/* the LFind procedure*/
pascal void LGetCellDataLocation
                            (short *offset, short *len, Cell theCell,
                             ListHandle lHandle);
pascal void LGetCell        (void *dataPtr, short *dataLen, Cell theCell,
                             ListHandle lHandle);
```

Responding to Events Affecting Lists

```
pascal Boolean LClick       (Point pt, short modifiers, ListHandle lHandle);
pascal void LActivate       (Boolean act, ListHandle lHandle);
pascal void LUpdate         (RgnHandle theRgn, ListHandle lHandle);
```

Modifying a List's Appearance

```
/*the LSetDrawingMode procedure is also available as the LDoDraw procedure*/
pascal void LSetDrawingMode
                            (Boolean drawIt, ListHandle lHandle);
pascal void LDraw           (Cell theCell, ListHandle lHandle);
pascal void LAutoScroll     (ListHandle lHandle);
pascal void LScroll         (short dCols, short dRows, ListHandle lHandle);
```

Searching for a List Containing a Particular Item

```
pascal Boolean LSearch      (const void *dataPtr, short dataLen,
                             SearchProcPtr searchProc, Cell *theCell,
                             ListHandle lHandle);
```

Changing the Size of Cells and Lists

```
pascal void LSize           (short listWidth, short listHeight,
                             ListHandle lHandle);
pascal void LCellSize       (Point cSize, ListHandle lHandle);
```

Getting Information About Cells

```
pascal Boolean LNextCell    (Boolean hNext, Boolean vNext, Cell *theCell,
                             ListHandle lHandle);
pascal void LRect           (Rect *cellRect, Cell theCell,
                             ListHandle lHandle);
pascal Cell LLastClick      (ListHandle lHandle);
```

Application-Defined Routines

```
pascal void MyLDEF          (short message, Boolean selected,
                             Rect *cellRect, Cell theCell,
                             short dataOffset, short dataLen,
                             ListHandle theList);
pascal short MyMatchFunction
                            (Ptr cellDataPtr, Ptr searchDataPtr,
                              short cellDataLen, short searchDataLen);
pascal void MyClickLoop     (void);
```

Assembly-Language Summary

Data Structures

ListRect Data Structure

0	`rView`	8 bytes	list's display rectangle
8	`port`	long	list's graphics port
12	`indent`	4 bytes	indent distance for drawing
16	`cellSize`	4 bytes	size in pixels of a cell
20	`visible`	8 bytes	boundary of visible cells
28	`vScroll`	long	vertical scroll bar
32	`hScroll`	long	horizontal scroll bar
36	`selFlags`	byte	selection flags
37	`lActive`	byte	nonzero if list is active
38	`lReserved`	byte	reserved
39	`listFlags`	byte	automatic scrolling flags
40	`clikTime`	long	ticks at time of last click
44	`clikLoc`	4 bytes	position of last click
48	`mouseLoc`	4 bytes	current mouse location
52	`lClikLoop`	long	pointer to routine called by `LClick`
56	`lastClick`	4 bytes	last cell clicked
60	`refCon`	long	for application use
64	`listDefProc`	long	handle to code for list definition procedure
68	`userHandle`	long	for application use
72	`dataBounds`	8 bytes	boundary of cells allocated
80	`cells`	long	handle to cell data
84	`maxIndex`	word	used internally
86	`cellArray`	variable	offsets to data

Trap Macros

Trap Macros Requiring Routine Selectors

`_Pack0`

Selector	Routine
$0000	`LActivate`
$0004	`LAddColumn`
$0008	`LAddRow`
$000C	`LAddToCell`
$0010	`LAutoScroll`
$0014	`LCellSize`
$0018	`LClick`
$001C	`LClrCell`
$0020	`LDelColumn`
$0024	`LDelRow`
$0028	`LDispose`
$002C	`LSetDrawingMode`
$0030	`LDraw`
$0034	`LGetCellDataLocation`
$0038	`LGetCell`
$003C	`LGetSelect`
$0040	`LLastClick`
$0044	`LNew`
$0048	`LNextCell`
$004C	`LRect`
$0050	`LScroll`
$0054	`LSearch`
$0058	`LSetCell`
$005C	`LSetSelect`
$0060	`LSize`
$0064	`LUpdate`