22

LISTS AND CUSTOM LIST DEFINITION FUNCTIONS

Demonstration Program: Lists

Introduction to Lists

If you need the user to be able to select a single item from a small group of items, you typically provide a pop-up menu. Pop-up menus, however, do not allow the user to select multiple items from a group of items, are not especially suitable for the presentation of large numbers of items, and cannot present items in columns as well as rows. Furthermore, the items in a pop-up menu remain displayed only as long as the user holds the mouse button down.

By using **lists** to present a group of items to the user, you can overcome these limitations. Although lists, like pop-up menus, are generally used to solicit the user's choices, they can also be used to simply present information. Perhaps the most familiar example of such a list is that at the bottom of the window opened when you choose **About This Computer...** from the Mac OS 8/9 Apple menu.

In essence, then, the List Manager allows you to create either one-column or multi-column scrollable lists which may be used to simply present items of information or, as is most often the case, to enable the user to select one or more of a group of items.

By default, the List Manager creates lists which contain only monostyled text. However, with a little additional effort, you can create lists which display items graphically (as does the list on the left side of the window opened when you choose **Chooser** from the Mac OS 8/9 Apple menu), or which display more than one type of information in each item (as does the list in the Mac OS 8/9 **About This Computer...** window).

List Manager Limitations

The List Manager is not suitable for displaying large amounts of data. The limiting size for list data is 32KB, and performance degrades well before that limit is reached.

Options For Creating and Managing Lists

You can use the List Manager function `LNew` to create a list, in which case you must provide all the functions necessary to add rows and data to the list, handle mouse and keyboard events, etc. Alternatively, you can use the **list box control** to simplify the matter of list creation and handling.

The first section of this chapter addresses the former method and the subject of lists in general. The second section addresses the list box control, and indicates those areas in the first section which are not relevant when you use list box controls.

Appearance and Features of Lists

Fig 1 shows a dialog with two typical single-column lists. The items in the list on the left are exclusively text items and the items in the list on the right are recorded pictures comprising a graphic and a title string. The list on the left supports the selection of multiple items.

FIG 1 - DIALOG BOX WITH TWO LISTS

To create a list with graphical elements, such as the list at the right at Fig 1, you must write a custom **list definition function** (see below), because the default list definition function only supports the display of text.

Cells, Cell Font, and Cell Highlighting

Cells

A list is a number of items displayed within a rectangle, each item being contained within an invisible rectangular **cell**. Cells may contain different types of data, but all cells within a particular list are of the same size.

Cell Font

By default, lists inherit the font of the graphics port associated with the window or dialog in which they reside.[1] Ordinarily, your text-only lists should use the large or small system font.

Regardless of the font your application uses, if a string is too long to fit in its cell using the current font, the List Manager uses condensed type in an attempt to make it fit (Mac OS 8/9 only). Failing that, the List Manager truncates the string and appends the ellipsis character.

Cell HighLighting

Your application may or may not allow the user to select one or more cells in a list. If your application allows users to select cells, then, when the user selects a cell, the List Manager automatically highlights that cell.

Scroll Bars

Lists may contain scroll bars, which allow you to include more items in a list than can be contained within the list's display rectangle. If a list includes a scroll bar but the number of cells is such that they are all visible, the List Manager disables the scroll bar.

Selection of Cells Using The Mouse

LClick

Your application must call `LClick` whenever a mouse-down occurs in an active list. `LClick` handles all user interaction within the list until the user releases the mouse button. This includes cell highlighting and, when the user drags the mouse outside the list's display rectangle, automatic list scrolling. `LClick` also

[1] In the case of list control boxes, the font may be set using `SetControlFontStyle` or a `'dftb'` resource.

examines the state of the Shift and Command keys, which are central to the process of multiple cell selection in lists.

Multiple Cell Selection Using the Default Cell-Selection Algorithm

The List Manager's cell-selection algorithm allows the user to select a contiguous range of cells, or even several discontiguous ranges of cells, by using the Shift and Command keys in conjunction with the mouse.[2] The following describes the default cell-selection behaviour.

Cell Selection With the Shift Key

The user can extend a selection of just one cell to several contiguous cells by pressing the Shift key and clicking another item. By clicking and dragging with the Shift key down, the user can extend or shrink the range of selected cells. If the cursor is dragged outside the list's display rectangle, the list will scroll so as to enable the user to include cells which were not initially visible.

Cell Selection With the Command Key

A range of cells may be added or deleted from the current selection by pressing the Command key and then dragging the cursor over other cells. The List Manager checks the status of the first cell clicked in so as to determine whether to add or remove selections. All cells in the range over which the cursor passes will be deselected if that first cell is initially selected. On the other hand, all cells in the range over which the cursor passes will be selected if that first cell is initially not selected.

When a cell's selection status is changed by Command-dragging, that selection status remains the same for the duration of the drag, that is, it will not change if the user moves the cursor back over the cell. The effect of the Command key thus differs from that of the Shift key in this respect.

Shift-Clicking — Discontiguous Cells Selected

Discontiguity is lost if the user Shift-clicks a cell after having previously created discontiguous selections. The List Manager selects all cells in the range of the selected cell closest to the top of the list 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 selected cell closest to the bottom of the list.

Customising the Cell-Selection Algorithm

As will be seen, the List Manager's cell-selection algorithm may easily be customised so as to modify its default behaviour. The most common modification is to defeat multiple cell selection, allowing the user to select only one cell.

Selection of Cells Using the Keyboard

Some users prefer to use the keyboard to select cells in lists. Your application should support the following methods of cell selection via the keyboard:

- ***Cell Selection Using Arrow Keys.*** This method involves the use of the Arrow keys to move and extend cell selections.
- ***Type Selection.*** This method involves the user simply typing the text associated with an item. It is thus only relevant to text-only lists (or lists whose items can be identified by text strings).

Cell Selection Using Arrow Keys

The List Manager does not provide any functions to support cell selection by Arrow key. Accordingly, your application must supply all of the necessary code. The following describes what that code should do.

[2] If the user presses both the Shift and Command keys when clicking a cell, the Shift key is ignored.

Moving the Selection Using Arrow Keys

Shift and Command Keys Not Down

When the user presses an Arrow key, and is not at the same time pressing the Shift or Command key, the selection should be moved by one cell.

If the user presses the Up Arrow, for example, your application should respond by selecting the cell which is above the first selected cell and by deselecting all other selected cells. (Of course, if the first selected cell is the topmost cell in the list, your application should respond by simply deselecting all cells other than the first selected cell.) If necessary, your application should then scroll the list to ensure that the newly-selected cell is visible.

Command Key Down

When the user presses an Arrow key while at the same time pressing the Command key, your application should move the first selected cell or the last selected cell (depending on which arrow key is used) as far as it can move in the appropriate direction. For example, in a single-column list, pressing the Up Arrow key should select the first cell in the list, deselect all other cells, and scroll the list, if necessary, to ensure that the newly-selected cell is visible.

Extending the Selection Using Arrow Keys

When the user presses an Arrow key while the Shift key is down, the user is attempting to **extend** the selection. There are two alternative algorithms your application can use to respond to Shift-Arrow key combinations: the **extend algorithm** and the **anchor algorithm**. The easiest one to implement is the extend algorithm.

The Extend Algorithm

Using the extend algorithm, 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-Down Arrow in a single-column list, the application should find the last selected cell and select the cell immediately below it, or, if the user presses Shift-Up Arrow, the application should find the first selected cell and select the cell above it. As always, the list should then be scrolled, if necessary, to make the newly-selected cell visible.

Type Selection

The List Manager does not provide any functions to support type selection, although the Text Utilities provide a data type and functions which support this method of keyboard selection. That said, your application must provide most of the code. The following describes what that code should do.

In a text-only list, when the user types the text of an item in a list, the list should scroll to that cell and select it.

However, rather than requiring the user to type the entire text of the item before searching for a match, your application should repeatedly search for a match as each character is entered. Accordingly, every time the user types a character, your application should add it to a string. If this string is currently two characters long, for example, your application should then walk the cells of the list, comparing these two characters with the first two characters of the text in each cell. If a match is found, that cell should be selected and the list scrolled, if necessary, to make the cell visible.

Your application should automatically reset the internal string to a null string when the user has not pressed a key for a given amount of time. To make your application consistent with other applications and the Finder, this time should be twice the number of ticks contained in the low memory global `KeyThresh`. (The value in `KeyThresh` is set by the user at the "Delay Until Repeat" section of the Keyboard control panel (Mac OS 8/9) and System Preferences/Keyboard (Mac OS X).)

Implementing Type Selection

To implement type selection, your application must store the characters the user has typed and the time when the user last typed a character. The Text Utilities `TypeSelectRecord` structure may be used for this purpose:

```
struct TypeSelectRecord
{
  unsigned long tsrLastKeyTime;  // Time when the last character was typed
  ScriptCode    tsrScript;
  Str63         tsrKeyStrokes;   // Characters typed by the user
};
typedef struct TypeSelectRecord TypeSelectRecord;
```

The Text Utilities function `TypeSelectNewKey` adds a new character to the `tsrKeyStrokes` field.

A global variable of type `SInt16` should be used to store the number of ticks after which type selection must be reset. The low memory accessor function `LMGetKeyThresh` may be used to obtain the value in the low memory global `KeyThresh`.

The Text Utilities function `TypeSelectClear` may be used to reset the `tsrKeyStrokes` and `tsrLastKeyTime` fields of the `TypeSelectRecord` structure to `NULL` and `0` respectively.

`LSearch` should be called to search the list's cells for a match with the specified characters, a universal procedure pointer to a comparison function being passed in the `searchProc` parameter.

Creating, Disposing Of, and Managing Lists

The List Structure

The **list structure**, which the List Manager uses to keep track of information about a list, is central to the creation and management of lists. Although the list structure is not opaque, you are encouraged to access the list structure indirectly using the provided accessor functions because this will give your application greater threading capability on Mac OS X.

Before describing the list structure and its associated accessor functions, however, it is necessary to describe another data type used exclusively by the List Manager, that is, the `Cell` data type.

The Cell Data Type

A cell in a list can be described by the `Cell` data type, which has the same structure as the `Point` data type:

```
typedef Point Cell;
```

The `Cell` data type's fields, however, have a different meaning from those of the `Point` data type. In the `Cell` data type, the `h` field specifies the row number and the `v` field specifies the column number. The first cell in a list is defined as cell (0,0). Fig 2 shows a multi-column list in which each cell's text is set to the coordinates of the cell.

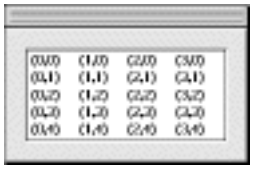

FIG 2 - CELL COORDINATES

The ListRec Structure

The list structure is defined by the `ListRec` data type:

```
struct ListRec
{
  Rect             rView;         // List's display rectangle.
  GrafPtr          port;          // List's graphics port.
  Point            indent;        // Indent distance for drawing.
  Point            cellSize;      // Size in pixels of a cell.
  Rect             visible;       // Boundary of visible cells.
  ControlRef       vScroll;       // Vertical scroll bar.
  ControlRef       hScroll;       // Horizontal scroll bar.
  SInt8            selFlags;      // Selection flags.
  Boolean          lActive;       // true if list is active.
  SInt8            lReserved;     // (Reserved.)
  Sint8            listFlags;     // Automatic scrolling flags.
  long             clikTime;      // TickCount at time of last tick.
  Point            clikLoc;       // Position of last click.
  Point            mouseLoc;      // Current mouse location.
  ListClickLoopUPP lClickLoop;    // Function called by LClick.
  Cell             lastClick;     // Last cell clicked.
  long             refCon;        // For application use.
  Handle           listDefProc;   // List definition function.
  Handle           userHandle;    // For application use.
  ListBounds       dataBounds;    // Boundary of cells allocated.
  DataHandle       cells;         // Cell data.
  short            maxIndex;      // (Used internally.)
  short            cellArray[1];  // Offsets to data.
};

typedef struct ListRec ListRec;
typedef ListRec *ListPtr;
typedef ListPtr *ListHandle;
```

Field Descriptions and Associated Accessor Functions

`rView` The list's visible rectangle (local coordinates). This does not include the area occupied by the list's scroll bars (if any).

Accessor Functions: `GetListViewBounds SetListViewBounds`

`port` The graphics port of the window containing the list. The coordinates of the list's visible rectangle are local to this port.

Accessor Functions: `GetListPort SetListPort`

`indent` The location, relative to the upper left corner of the cell, at which drawing should begin. For example, the default list definition function sets the vertical coordinate of this field to near the bottom of the cell so that characters drawn with QuickDraw's `DrawText` function are centred vertically in the cell.

Accessor Functions: `GetListCellIndent SetListCellIndent`

`cellSize` The size (in pixels) of each cell in the list. For text-only lists, you usually let the List Manager automatically calculate the cell dimensions. In this case, the List Manager adds the ascent, descent and leading of the port's font to arrive at the height of a cell (which works out as 16 pixels for 12-point Charcoal on Mac OS 8/9, for example). You should make the height of your list equal to a multiple of cell height. For cell width, the List Manager divides the width of the list's display rectangle by the number of columns in the list.

Accessor Functions: `GetListCellSize SetListCellSize`

`visible` Specifies those cells in a list that are visible within the `rView` rectangle. The `left` and `top` fields are set by the List Manager to the coordinates of the first visible cell. The `right` and `bottom` fields are set to one greater than the horizontal and vertical coordinates of the last

visible cell. If, for example, the first three columns and six rows are visible (that is, the last visible cell has coordinates (2,5), the List Manager sets the `visible` field to (0,0,3,6).

The List Manager sets the `right` and `bottom` fields to one greater than the horizontal and vertical coordinates of the last visible cell so as to facilitate the use of QuickDraw's `PtInRect` function to determine whether a cell is currently visible. When `PtInRect` is used for this purpose, a `Cell` variable is passed as the first parameter and the `visible` field is passed as the second parameter. When `PtInRect`'s parameters are expressed as cell coordinates, the cells "hang" down and to the right of the mathematical rectangle. Thus, in the above example, if the cell passed as the first parameter to `PtInRect` specifies row 6 or higher or column 3 or higher, `PtInRect` returns `false`.

The fact that the `visible` field is set in this way also means that the number of visible rows and columns may be determined by simply subtracting the value in the `top` field from the value in the `bottom` field (rows) and the value in the `left` field from the value in the `right` field (columns).

Accessor Functions: `GetListVisibleCells SetListVisibleCells`

`vScroll` Handle to the vertical scroll bar (or `NULL` if there is no vertical scroll bar).

Accessor Functions: `GetListVerticalScrollBar SetListVerticalScrollBar`

`hScroll` Handle to the horizontal scroll bar (or `NULL` if there is no horizontal scroll bar).

Accessor Functions: `GetListHorizontalScrollBar SetListHorizontalScrollBar`

`selFlags` The algorithm the List Manager uses to select cells in response to a click in the list.

Accessor Functions: `GetListSelectionFlags SetListSelectionFlags`

`lActive` `true` if a list is active or `false` if it is inactive. (Do not change this field directly. Use `LActivate` to activate or deactivate a list.)

Accessor Functions: `GetListActive LActivate`

`listFlags` Flags indicating whether automatic vertical and horizontal scrolling is enabled. If automatic scrolling is enabled, the list scrolls when the user clicks a cell and then drags the cursor out of the `rView` rectangle. If the list has the associated scroll bar (horizontal or vertical), automatic scrolling is enabled by default. The following constants are used to specify whether horizontal and vertical autoscrolling should be enabled or disabled:

```
lDoVAutoscroll = 2   Allows vertical scrolling.
lDoHAutoscroll = 1   Allows horizontal scrolling.
```

Accessor Functions: `GetListFlags SetListFlags`

`clikTime` The time when the user last clicked the mouse.

Accessor Functions: `GetListClickTime SetListClickTime`

`clikLoc` The local coordinates of the last mouse click.

Accessor Function: `GetListClickLocation`

`mouseLoc` Current location of the cursor in local coordinates.

Accessor Function: `GetListMouseLocation`

`lClickLoop` Contains `NULL` if the default click loop function is to be used. However, your application cab assign a universal procedure pointer to a custom click-loop function to this field. Your application will not need a custom click-loop function unless it needs to perform some special processing while the user drags the cursor.

Accessor Functions: `GetListClickLoop SetListClickLoop`

`lastClick` Cell coordinates of the last click. You can access the value in this field using `LLastClick`.

Accessor Function: `SetListLastClick`

`refCon` For your application's use.

Accessor Functions: `GetListRefCon SetListRefCon`

`listDefProc` Handle to the list definition function code.

Accessor Functions: `GetListDefinition SetListDefinition`

`userHandle` For your application's use. Typically, applications use this field to store a handle to some additional storage associated with a list.

Accessor Functions: `GetListUserHandle SetListUserHandle`

`dataBounds` The total cell dimensions of the list. It is similar to the `visible` field in that its `right` and `bottom` fields are each set to one greater than the horizontal and vertical coordinates of the last cell — except that, in this case, the "last cell" is the last cell in the list, not the last cell in the visible rectangle. For example, if a list contains 5 columns and 12 rows (that is, the last cell in the list has coordinates (4,10)), the `dataBounds` field is set to (0,0,5,12).

Accessor Function: `GetListDataBounds`

`cells` Handle to a relocatable block where cell data is stored. Because of the way this field is defined, no list can contain more than 32,000 bytes of data.

Accessor Functions: `GetListDataHandle`

`cellArray` Offsets to data in the relocatable block specified by the `cells` field. Do not change the `cells` field, or access the information in the `cellArray` field, directly.

The fields of a list structure that you will be most concerned with are the `rView`, `port`, `cellSize`, `visible`, and `dataBounds` fields.

Creating Lists

Creating Lists Which Do Not Use a Custom List Definition Function

If you are creating a list that does not use a custom list definition function, you should use the function `LNew` to create the list:

```
ListHandle  LNew(const Rect *rView,const ListBounds *dataBounds,Point cSize,
                 short theProc,WindowRef theWindow,Boolean drawIt,Boolean hasGrow,
                 Boolean scrollHoriz,Boolean scrollVert);
```

`rView` Rectangle in which to display the list. This rectangle is in the local coordinates of the window passed in the `theWindow` parameter, and does not include the area occupied by the list's scroll bars.

`dataBounds` Initial data bounds. For example, to create a list with 5 columns and 10 rows, set the `left` and `top` fields to (0,0) and the `right` and `bottom` fields to 5 and 10 respectively.

`cSize` Size of each cell. If your application is using the default list definition function and passes (0,0) in this field, the size is calculated automatically.

`theProc` Pass `0` in this parameter to cause the default list definition function to be used. (See also the Carbon Note below.)

`theWindow` Reference to the window in which the list is to be installed.

`drawIt` Specifies whether automatic drawing mode is to be initially enabled. When automatic redrawing is enabled (by setting this parameter to `true`), the list is automatically redrawn whenever it is changed.

This setting can be changed later using `LSetDrawingMode`. If your application chooses to disable automatic drawing mode (for example, for aesthetic reasons while adding rows and columns to a list) it should do so only for short periods of time.

`hasGrow` Specifies whether space should be left for a size box/resize control.

`scrollHoriz` Pass `true` if your list requires a horizontal scroll bar, otherwise pass `false`.

`scrollVert` Pass `true` if your list requires a vertical scroll bar, otherwise pass `false`.

Creating Lists Which Use a Custom List Definition Function

If you are creating a list which uses a custom list definition function, you should use the function `CreateCustomList` to create the list:

```
OSStatus  CreateCustomList(const Rect *rView,const ListBounds *dataBounds,
                           Point cellSize,const ListDefSpec *theSpec,
                           WindowRef theWindow,Boolean drawIt,Boolean hasGrow,
                           Boolean scrollHoriz,Boolean scrollVert,ListHandle *outList);
```

The main difference between `LNew` and `CreateCustomList` is that the former takes the resource ID of a list definition function whereas the latter takes a universal procedure pointer to a list definition function.

> **Carbon Note**
>
> In the Classic API, custom list definition functions are compiled separately as `'CODE'` resources and the resource ID is passed in the `theProc` parameter of `LNew`. In Carbon, custom definition functions (and, indeed, all other custom definition functions) cannot be stored in resources; accordingly, the function `CreateCustomList` was introduced with Carbon to accommodate that situation.

Drawing List Box Frames and Focus Rectangles

List Box Frame

The List Manager does not draw the list box frame around the list. Accordingly, this must be drawn by your application.

Focus Rectangle

In a window with multiple lists, you need to indicate to the user which list is the current list, that is, which list is the target of current mouse and keyboard activity.[3] Accordingly, you should draw a focus rectangle around the current list. (See the list on the left at Fig 1). The focus rectangle should be removed when the window or dialog containing the lists is deactivated.

Disposing of a List

When you are finished with a list, you should dispose of it using `LDispose`, which disposes of the list structure as well as the data associated with the list. `LDispose` does not, however, dispose of any application-specific data you may have stored in a relocatable block specified by the `userHandle` field of the list structure. This should be separately disposed of before the call to `LDispose`.

Adding Rows and Columns to a List

When an application creates a list, it might choose to, for example, pre-allocate the columns it needs and then add rows to the list one by one. It might also create the list and add both rows and columns to it later.

Rows are inserted into a list using `LAddRow` and deleted using `LDelRow`. Columns are inserted in a list using `LAddColumn` and deleted using `LDelColumn`.

Disabling and Enabling the Automatic Drawing Mode

`LSetDrawingMode` should be used to turn off the automatic drawing mode before making changes to a list. After the changes have been made, `LSetDrawingMode` should be called again, this time to turn the automatic drawing mode back on.

[3] A single list in a window should also be outlined with a focus rectangle if keyboard input could have some other effect in the window not related to the list (for example, if the list is in a dialog containing both a list and an edit text item).

InvalWindowRect should be called after the second call to LSetDrawingMode to invalidate the rectangle containing the list and its scroll bars. LUpdate, which should be called when your application receives an update event, will then redraw the list.

Responding to Events in a List

Mouse-Down Events

As previously stated, when a mouse-down event occurs in a list, including in the associated scroll bar areas, your application must call LClick. If the click is outside the list's display rectangle or scroll bars, LClick returns immediately, otherwise it handles all user interaction until the user releases the mouse button. While the mouse button is down, the List Manager performs scrolling as necessary, selects or de-selects cells as appropriate, and adjusts the scroll bars.

Note that LClick returns true if the click was a double click. If the list is in a dialog, your application should respond to a double click in the same way that it would respond to a click on the default (OK) button.

In the case of multiple lists, if the mouse-down occurs inside a non-current list's display rectangle or scroll bar area, your application should call its function for changing the current list.

Key-Down Events

If a key-down event is received, and assuming that your application supports cell selection by Arrow key and/or type selection, your application should call its appropriate functions. In the case of multiple lists, your application should also respond to Tab key presses by changing the current list.

Update Events

If an update event (Classic event model) or kEventWindowDrawContent event type (Carbon event model) is received, your application must call LUpdate to redraw the list. The region specified in the first parameter to the LUpdate call is usually the window's visible region as retrieved by GetPortVisibleRegion.

Your application will also need to draw the list box frame in the correct state (window active or inactive) and, if a focus rectangle is required and the window is active, the focus rectangle.

Activate Events

If an activate event (Classic event model) or kEventWindowActivated or kEventWindowDeactivated event type is received (Carbon event model), your application must call LActivate to activate or deactivate the list as appropriate. Your application will also need to draw the list box frame in the correct state, and either draw or erase the keyboard focus rectangle, depending on whether the window is becoming active or inactive.

If your application supports type selection in a list, it will also need to reset certain type selection variables when the window containing that list is becoming active.

Getting and Setting List Selections

The List Manager provides functions for determining which cells are currently selected and for selecting and deselecting cells. LGetSelect is used to either determine whether a specified cell is selected or to keep advancing from a specified starting cell until the next selected cell is found. LSetSelect is used to select or deselect a specified cell.

LNextCell, which simply advances from one cell in a list to the next, is often used in functions associated with getting and setting list selections.

Scrolling a List

LAutoScroll may be used to scroll the first selected cell to the upper-left corner of the list's display rectangle.

LScroll allows your application to scroll the list by a specified number of rows and/or columns. Typically, you would use LScroll when you want your application to scroll a list just enough so that a certain cell (such as the cell the user has just selected using the an Arrow key or type selection) is visible.

Storing, Adding To, Getting, and Clearing Cell Data

Storing Data

Your application can store data in a cell using LSetCell. LSetCell's parameters include a pointer to 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. The data stored in a cell might be sourced from, for example, a string list resource.

Adding to Data

Your application can append data to a cell using LAddToCell.

Getting Cell Data

LGetCell may be used to copy the contents of a cell into a buffer. LGetCellDataLocation may be used to obtain the address and length of a cell's data. Unlike LGetCell, LGetCellDataLocation does not make a copy of the data, and should thus be used when you want to access, but not manipulate, the data.

Clearing Data

Your application can remove all data from a cell using LClrCell.

Searching a List

Your application can use LSearch to search through a list for a particular item. LSearch takes, as one parameter, a universal procedure pointer to a **match function**. If NULL is specified for this parameter, LSearch searches the list for the first cell whose data matches the specified data, calling the Text Utilities IdenticalString function to compare each cell's data with the specified data until IdenticalString returns 0, indicating that a match has been found.

Custom Match Functions

The default match function is useful for text-only lists. Your application can use a different match function to facilitate searches in other types of lists as long as that function is defined just like IdenticalString.

A common custom match function is one which supports type selection in lists, that is, one which works like the default match function but which allows the cell data to be longer than the data being searched for. For example, a search for the string "be" would match a cell containing the string "Beams".

Changing the Current List

As previously stated, when a window or dialog contains multiple lists, your application should allow the user to change the current list by clicking in one of the non-current lists or by pressing the Tab key or Shift-Tab. In a window with more than two lists, Tab key presses should make the next list in a pre-determined sequence the current list, and Shift-Tab should make the previous list in that sequence the current list. The pre-determined sequence is best implemented using a **linked ring**.

Linked Ring

To create a linked ring, you can use the refCon field of each list structure. Assign the handle to the second list to the refCon field of the first list, assign the handle to the third list to the refCon field of the second list, and so on, until, to close the circle, the handle to the first list is assigned to the refCon field of the last list. Then, in response to a Tab key press in the current list, your application can ascertain the next list in the ring by looking at the current list's refCon field.

Responding to Shift-Tab is a little more complex. The following example function shows how this can be done:

```
ListHandle gCurrentListHdl;

void  doFindPreviousListInRing(void)
{
  ListHandle listHdl;

  listHdl = gCurrentListHdl;

  while((ListHandle) GetListRefCon(listHdl) != gCurrentList)
    listHdl = (ListHandle) GetListRefCon(listHdl);

  gCurrentListHdl = listHdl;
}
```

Customising the Cell-Selection Algorithm

You can modify the algorithm the List Manager uses to select cells in response to mouse clicking and dragging by changing the value in the `selFlags` field of the list structure. (Recall that, by default, mouse clicks deselect all cells and select the current cell, Shift-click and Shift-drag extend the selection as a rectangular range, and Command-click and Command drag toggle selections according to the selection state of the initial cell.)

The bits in the `selFlags` field are represented by the following constants. Those constants, and the effect the values they represent have on the cell-selection algorithm, are as follows:

Constant	Value	Effect
`lOnlyOne`	128	Allow only one cell to be selected at any one time.
`lExtendDrag`	64	Allow the user to select a range of cells by clicking the first cell and dragging to the last cell without necessarily pressing the Shift or Command key. (Ordinarily, dragging in this manner results in only the last cell being selected.)
`lNoDisjoint`	32	Prevent discontiguous selections using the Command key, while still allowing the user to select a contiguous range of cells.
`lNoExtend`	16	Cause all previously selected cells to be deselected when the user Shift-clicks.
`lNoRect`	8	Disable the feature which 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. (With 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.)
`lUseSense`	4	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.)
`lNoNilHilite`	2	Turn off the highlighting of cells which contain no data. (Note that this 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.)

These constants are often used additively. For example, you could make the Shift key work just like the Command key using the following code:

```
SetListSelectionFlags(listHdl,lNoRect + lNoExtend + lUseSense);
```

If your application customises the cell-selection algorithm in lists which allow multiple cell selection, it should make the non-standard behaviour clear to the user. Typically, this is done by displaying explanatory text above the list's display rectangle.

The List Box Control

The list box control reduces the programming effort involved in managing lists. Basically, this control frees your application from the requirement to provide its own functions for attending to mouse and keyboard interaction with the list (except for type selection). To create and manage lists using the list box control, you need to:

- Provide a list box `'CNTL'` resource and a list box description (`'ldes'`) resource (see below).
- Provide functions for storing data in the list's cells and, where required, for adding rows and/or columns.
- Provide a function to support type selection, if required.
- Modify the list's cell selection algorithm, if required.
- Provide a function which searches for, and returns the data in, the selected cell or cells.

The handle to the control is assigned to the `refCon` field of the list structure. This allows a custom list definition function to determine whether the control should be drawn in the activated or deactivated state by looking at the `contrlHilite` field of the control structure.

List Box Variants, Values, Constants, and Resources

Variant and Control Definition ID

The list box CDEF resource ID is 22. The two available variants and their control definition IDs are as follows:

Variant	Var Code	Control Definition ID
List box.	`0`	`352` `kControlListBoxProc`
Autosizing list box.	`1`	`353` `kControlListBoxAutoSizeProc`

Control Values

Control Value	Content
Initial	Resource ID of the `'ldes'` resource holding the list box information. Reset to `0` after creation. An initial value of `0` indicates not to read an `'ldes'` resource. (See The List Box Description Resource, below.)
Minimum	Reserved. Set to `0`.
Maximum	Reserved. Set to `0`.

Control Data Tag Constants

Control Data Tag Constant	Meaning and Data Type Returned or Set
`kControlListBoxListHandleTag`	Gets a handle to a list box. **Data type returned:** `ListHandle`
`kControlListBoxDoubleClickTag`	Checks to see whether the most recent click in a list box was a double click. **Data type returned:** `Boolean`. If `true`, the last click was a double click. If `false`, not.
`kControlListBoxKeyFilterTag`	Gets or sets a key filter function. **Data type returned or set:** `ControlKeyFilterUPP`
`kControlListBoxFontStyleTag`	Gets or sets the font style. **Data type returned or set:** `ControlFontStyleRec`

Control Part Codes

Constant	Value	Description
`kControlListBoxPart`	`24`	Event occurred in a list box.
`kControlListBoxDoubleClickPart`	`25`	Double-click occurred in a list box.

The List Box Description Resource

The list box description (`'ldes'`) resource, which must have resource ID of greater than 127, is used to specify information for a list box. The information is used by The Control Manager to provide additional

information to the relevant list box control. Fig 3 shows the structure of a compiled `'ldes'` resource and such a resource being created using Resorcerer.

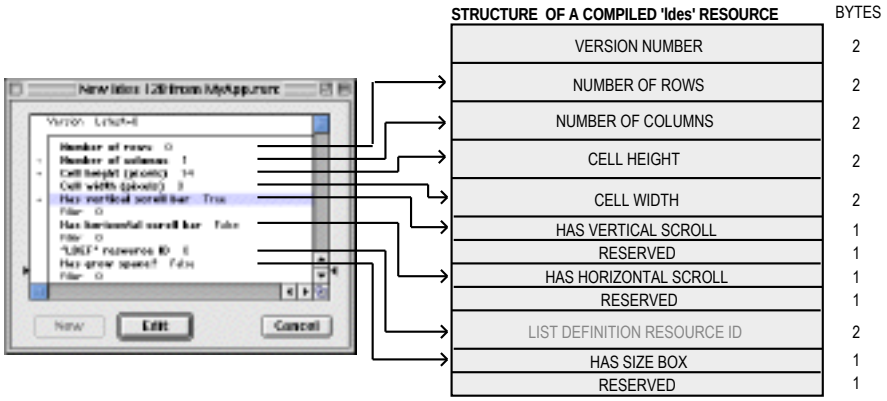

FIG 3 - CREATING AN 'ldes' RESOURCE USING RESORCERER

The following describes the main fields of a compiled `'ldes'` resource:

Field	*Description*
NUMBER OF ROWS	The number of rows in the list box.
NUMBER OF COLUMNS	The number of columns in the list box.
CELL HEIGHT	The height of the list cells. Specify `0` to cause the height to be calculated automatically.
CELL WIDTH	The width of a the list cells. Specify `0` to cause the width to be calculated automatically.
HAS VERTICAL SCROLL BAR	`true` causes the list box to contain a vertical scroll bar.
HAS HORIZONTAL SCROLL BAR	`true` causes the list box to contain a horizontal scroll bar.
RESOURCE ID	The resource ID of the list definition function to use for the list. In Carbon, always set to `0`.
HAS SIZE BOX	`true` causes the List Manager to leave room for, and draw, a size box.

Programmatic Creation

List box controls may be created programmatically using the function `CreateListBoxControl`:

```
OSStatus  CreateListBoxControl(WindowRef window,const Rect *boundsRect,Boolean autoSize,
          SInt16 numRows,SInt16 numColumns,Boolean horizScroll,Boolean vertScroll,
          SInt16 cellHeight,SInt16 cellWidth,Boolean hasGrowSpace,
          const ListDefSpec *listDef,ControlRef *outControl);
```

Custom List Definition Functions

As previously stated, the default list definition function supports the display of unstyled text only. If your application needs to display items graphically, or display more than one type of information in each cell[4], you must create your own list definition (callback) function.

Your custom list definition (callback) function must be declared like this:

```
void  myListDefinition(SInt16 lMessage,Boolean lSelect,Rect *lRect,Cell lCell,
                       SInt16 lDataOffset,SInt16 lDataLen,ListHandle lHandle);
```

[4] For example, the MAC OS 8/9 Finder's **About This Computer...** dialog contains a single-column list of applications currently in use. Each cell in the list contains an icon, the name of the application, the amount of memory in the application partition, and a graphical indication of how much of that memory has been used.

Messages Sent by List Manager

In essence, the sole requirement of your list definition function is to respond appropriately to four types of messages sent to it by the List Manager, and which are received in the `lMessage` parameter. The following constants define the four message types:

Constant	Value	Meaning
`lInitMsg`	`0`	Do any special list initialisation.
`lDrawMsg`	`1`	Draw the cell.
`lHiliteMsg`	`2`	Invert the cell's highlight state.
`lCloseMsg`	`3`	Take any special disposal action.

The `lSelect`, `lRect`, `lCell`, `lDataOffset`, and `lDataLen` parameters, which contain information about the cell affected by the message, pass information to your custom list definition function only when the `lDrawMsg` or `lHiliteMsg` messages are received. The `lSelect` parameter indicates whether the cell should be highlighted. `lRect` and `lCell` provide the cell's rectangle and coordinates. `lDataOffset` and `lDataLen` parameters provide the offset and length of the cell's data referenced by the `cells` field of the list structure.

The Initialisation Message

In response to the `lInitMsg` message, your application might, for example, change the `cellSize` and `indent` fields of the list structure. However, many custom list definition functions do not need to perform any action in response to the initialisation message.

The Draw Message

In response to the `lDrawMsg` message, your custom list definition function must examine the specified cell's data and draw the cell as appropriate, ensuring that the characteristics of the drawing environment are not altered.

The HighLighting Message

In response to the `lHiliteMsg` message, your custom list definition function should highlight the cell's rectangle. The following example shows how this might be done:

```
void  doLDEFHighlight(Rect *cellRect)
{
  UInt8 hiliteVal;

  hiliteVal = LMGetHiliteMode();
  BitClr(&hiliteVal,pHiliteBit);
  LMSetHiliteMode(hiliteVal);

  InvertRect(cellRect);
}
```

Responding to the Close Message

The `lCloseMsg` is sent immediately before the List Manager disposes of the memory occupied by the list. Your custom list definition function needs to respond only if it needs to perform some special processing at that point, such as releasing any additional memory associated with the list.

Main List Manager Constants, Data Types, and Functions

Constants

Masks For listFlags Field of List Structure

```
lDoVAutoscroll      = 2     Allow vertical autoscrolling.
lDoHAutoscroll      = 1     Allow horizontal autoscrolling.
```

Masks For selFlags Field of List Structure

```
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 Function

```
lInitMsg            = 0      Do any special list initialisation.
lDrawMsg            = 1      Draw the cell.
lHiliteMsg          = 2      Invert cell's highlight state.
lCloseMsg           = 3      Take any special disposal action.
```

Control Kind

```
kControlKindListBox = FOUR_CHAR_CODE('lbox')
```

Data Types

```
typedef Point   Cell;
typedef Rect    ListBounds;
typedef char    DataArray[32001];
typedef char    *DataPtr;
typedef DataPtr *DataHandle;
```

List Structure

```
struct ListRec
{
  Rect              rView;         // List's display rectangle.
  GrafPtr           port;          // List's graphics port.
  Point             indent;        // Indent distance for drawing.
  Point             cellSize;      // Size in pixels of a cell.
  Rect              visible;       // Boundary of visible cells.
  ControlRef        vScroll;       // Vertical scroll bar.
  ControlRef        hScroll;       // Horizontal scroll bar.
  SInt8             selFlags;      // Selection flags.
  Boolean           lActive;       // true if list is active.
  SInt8             lReserved;     // (Reserved.)
  Sint8             listFlags;     // Automatic scrolling flags.
  long              clikTime;      // TickCount at time of last tick.
  Point             clikLoc;       // Position of last click.
  Point             mouseLoc;      // Current mouse location.
  ListClickLoopUPP  lClickLoop;    // Function called by LClick.
  Cell              lastClick;     // Last cell clicked.
  long              refCon;        // For application use.
  Handle            listDefProc;   // List definition function.
  Handle            userHandle;    // For application use.
  ListBounds        dataBounds;    // Boundary of cells allocated.
  DataHandle        cells;         // Cell data.
  short             maxIndex;      // (Used internally.)
  short             cellArray[1];  // Offsets to data.
};

typedef struct ListRec ListRec;
typedef ListRec *ListPtr;
typedef ListPtr *ListHandle;
```

Functions

Creating and Disposing of Lists

```
ListHandle      LNew(const Rect *rView,const ListBounds *dataBounds,Point cSize,
                short theProc,WindowRef theWindow,Boolean drawIt,Boolean hasGrow,
                Boolean scrollHoriz,Boolean scrollVert);
OSStatus        CreateCustomList(const Rect *rView,const ListBounds *dataBounds,
                Point cellSize,const ListDefSpec *theSpec,WindowRef theWindow,Boolean drawIt,
                Boolean hasGrow,Boolean scrollHoriz,Boolean scrollVert,ListHandle *outList);
void            LDispose(ListHandle lHandle);
```

Creating List Box Controls

```
OSStatus        CreateListBoxControl(WindowRef window,const Rect *boundsRect,
                Boolean autoSize,SInt16 numRows,SInt16 numColumns,Boolean horizScroll,
                Boolean vertScroll,SInt16 cellHeight,SInt16 cellWidth,Boolean hasGrowSpace,
                const ListDefSpec *listDef,ControlRef *outControl);
```

Adding and Deleting Rows and Columns

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

Determining or Changing a Selection

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

Accessing and Manipulating Data Cells

```
void            LSetCell(const void *dataPtr,short dataLen,Cell theCell,ListHandle lHandle);
void            LAddToCell(const void *dataPtr,short dataLen,Cell theCell,
                ListHandle lHandle);
void            LClrCell(Cell theCell,ListHandle lHandle);
void            LGetCellDataLocation(short *offset,short *len,Cell theCell,
                ListHandle lHandle);
void            LGetCell(void *dataPtr,short *dataLen,Cell theCell,ListHandle lHandle);
```

Responding to Events

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

Modifying a List's Appearance

```
void            LSetDrawingMode(Boolean drawIt,ListHandle lHandle);
void            LDraw(Cell theCell,ListHandle lHandle);
void            LAutoScroll(ListHandle lHandle);
void            LScroll(short dCols,short dRows,ListHandle lHandle);
```

Searching For a List Containing a Particular Item

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

Changing the Size of Cells and Lists

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

Getting Information About Cells

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

List Structure Accessor Functions

```
Rect*           GetListViewBounds(ListRef list,Rect *view);
void            SetListViewBounds(ListRef list,const Rect *view);
CGrafPtr        GetListPort(ListRef list);
void            SetListPort(ListRef list,CGrafPtr port);
Point*          GetListCellIndent(ListRef list,Point *indent);
```

```
void            SetListCellIndent(ListRef list,Point *indent);
Point*          GetListCellSize(ListRef list,Point *size);
void            SetListCellSize(ListRef list,Point *size);
ListBounds *    GetListVisibleCells(ListRef list,ListBounds *visible);
void            SetListVisibleCells(ListRef list,ListBounds *visible);
ControlHandle   GetListVerticalScrollBar(ListRef list);
void            SetListVerticalScrollBar(ListRef list,ControlHandle vScroll);
ControlHandle   GetListHorizontalScrollBar(ListRef list);
void            SetListHorizontalScrollBar(ListRef list,ControlHandle hScroll);
OptionBits      GetListSelectionFlags(ListRef list);
void            SetListSelectionFlags(ListRef list,OptionBits selectionFlags);
Boolean         GetListActive(ListRef list);
OptionBits      GetListFlags(ListRef list);
void            SetListFlags(ListRef list,OptionBits listFlags);
SInt32          GetListClickTime(ListRef list);
void            SetListClickTime(ListRef list,SInt32 time);
Point*          GetListClickLocation(ListRef list,Point *click);
Point*          GetListMouseLocation(ListRef list,Point *mouse);
ListClickLoopUPP GetListClickLoop(ListRef list);
void            SetListClickLoop(ListRef list,ListClickLoopUPP clickLoop);
void            SetListLastClick(ListRef list,Cell *lastClick);
SInt32          GetListRefCon(ListRef list);
void            SetListRefCon(ListRef list,SInt32 refCon);
Handle          GetListUserHandle(ListRef list);
void            SetListUserHandle(ListRef list,Handle userHandle);
Handle          GetListDefinition(ListRef list);
void            SetListDefinition(ListRef list,Handle listDefProc);
ListBounds*     GetListDataBounds(ListRef list,ListBounds *bounds);
DataHandle      GetListDataHandle(ListRef list);
```

Creating and Disposing of Universal Procedure Pointers

```
ListSearchUPP    NewListSearchUPP(ListSearchProcPtr userRoutine);
ListClickLoopUPP NewListClickLoopUPP(ListClickLoopProcPtruserRoutine);
ListDefUPP       NewListDefUPP(ListDefProcPtr userRoutine);
void             DisposeListSearchUPP(ListSearchUPP userUPP);
void             DisposeListClickLoopUPP(ListClickLoopUPP userUPP);
void             DisposeListDefUPP(ListDefUPP userUPP);
```

Application-Defined (Callback) Function

```
void             myListDefinition(short lMessage, Boolean lSelect, Rect *lRect, Cell lCell,
                 short lDataOffset, short lDataLen, ListHandle lHandle);
```

Relevant Text Utilities Data Type and Functions

Data Type

```
struct TypeSelectRecord
{
  unsigned long tsrLastKeyTime;
  ScriptCode    tsrScript;
  Str63         tsrKeyStrokes;
};
typedef struct TypeSelectRecord TypeSelectRecord;
```

Functions

```
void      TypeSelectClear(TypeSelectRecord *tsr);
Boolean   TypeSelectNewKey(const EventRecord *theEvent,TypeSelectRecord *tsr);
```

Demonstration Program Lists Listing

```
// ****************************************************************************************
// Lists.h                                                                  CARBON EVENT MODEL
// ****************************************************************************************
//
// This program allows the user to open a window and a movable modal dialog by choosing the
// relevant items in the Demonstration menu.  The window and the dialog both contain two
// lists.
//
// The cells of one list in the window, and of both lists in the dialog, contain text.  The
// cells of the second list in the window contain icons.
//
// The text lists use the default list definition function.  The list with the icons uses a
// custom list definition function.
//
// The currently active list is indicated by a keyboard focus frame, and can be changed by
// clicking in the non-active list or by pressing the tab key.
//
// The text list in the window uses the default cell-selection algorithm; accordingly,
// multiple cells, including discontiguous multiple cells, may be selected.  The cell-
// selection algorithm for the other lists is customised so as to allow the selection of only
// one cell at a time.
//
// All lists support arrow key selection.  The text list in the window and one of the lists in
// the dialog support type selection.
//
// The window is provided with an "Extract" push button.  When this button is clicked, or when
// the user double clicks in one of the lists, the current selections in the lists are
// extracted and displayed in the bottom half of the window.  In the dialog, the user's
// selections are displayed in static text fields embedded in placards below each list.
//
// The program utilises the following resources:
//
// •  A 'plst' resource.
//
// •  An 'MBAR' resource, and 'MENU' resources for Apple, File, Edit and Demonstration menus
//    (preload, non-purgeable).
//
// •  A'DLOG' resource (purgeable) (initially not visible) and associated 'dlgx', 'dftb', and
//    'DITL' resources (purgeable).
//
// •  'CNTL' resources (purgeable) for various controls in both the window and dialog box,
//    including the list controls for the dialog box.
//
// •  'ldes' resources associated with the list controls for the dialog box.
//
// •  'STR#' resources (purgeable) containing the text strings for the text lists and for the
//    titles of the icons.
//
// •  An icon suite (non-purgeable) containing the icons for icon list.
//
// •  An 'LDEF' resource (preload, locked, non-purgeable) containing the custom list
//    definition function  used by the icon list.
//
// •  'hrct' and 'hwin' (purgeable) resources for balloon help.
//
// •  A 'SIZE' resource with the acceptSuspendResumeEvents, canBackground,
//    doesActivateOnFGSwitch, and isHighLevelEventAware flags set.
//
// ****************************************************************************************

// ..... includes

#include <Carbon.h>

// ..... defines
```

```
#define rMenubar                128
#define mAppleApplication       128
#define  iAbout                 1
#define mFile                   129
#define  iQuit                  12
#define mDemonstration          131
#define  iHandMadeLists         1
#define  iListControlLists      2
#define cExtractButton          128
#define cGroupBox               129
#define cSoftwareStaticText     130
#define cHardwareStaticText     131
#define rTextListStrings        128
#define rIconListIconSuiteBase  128
#define rIconListStrings        129
#define rListsDialog            128
#define  iDateFormatList        4
#define  iWatermarkList         5
#define  iDateFormatStaticText  7
#define  iWatermarkStaticText   9
#define  rDateFormatStrings     130
#define  rWatermarkStrings      131
#define kUpArrow                0x1e
#define kDownArrow              0x1f
#define kTab                    0x09
#define kScrollBarWidth         15
#define kMaxKeyThresh           120
#define kSystemLDEF             0

// ..... typedefs

typedef struct
{
  ListHandle textListHdl;
  ListHandle iconListHdl;
  ControlRef extractButtonHdl;
} docStructure, **docStructureHandle;

typedef struct
{
  RGBColor     backColour;
  PixPatHandle backPixelPattern;
  Pattern      backBitPattern;
} backColourPattern;

// ..... function prototypes

void        main                        (void);
void        doPreliminaries             (void);
OSStatus    appEventHandler             (EventHandlerCallRef,EventRef,void *);
OSStatus    windowEventHandler          (EventHandlerCallRef,EventRef,void *);
void        doAdjustMenus               (void);
void        doMenuChoice                (MenuID,MenuItemIndex);
void        doSaveBackground            (backColourPattern *);
void        doRestoreBackground         (backColourPattern *);
void        doSetBackgroundWhite        (void);

void        doOpenListsWindow           (void);
void        doKeyDown                   (SInt8,EventRecord *);
void        doDrawContent               (WindowRef);
void        doActivateDeactivate        (WindowRef,Boolean);
void        doInContent                 (Point,UInt32);
void        doControlHit                (WindowRef,ControlRef,Point);
ListHandle  doCreateTextList            (WindowRef,Rect,SInt16,SInt16);
void        doAddRowsAndDataToTextList  (ListHandle,SInt16,SInt16);
void        doAddTextItemAlphabetically (ListHandle,Str255);
ListHandle  doCreateIconList            (WindowRef,Rect,SInt16,ListDefUPP);
void        doAddRowsAndDataToIconList  (ListHandle,SInt16);
void        doHandleArrowKey            (SInt8,EventRecord *,Boolean);
```

```
void        doArrowKeyMoveSelection     (ListHandle,SInt8,Boolean);
void        doArrowKeyExtendSelection   (ListHandle,SInt8,Boolean);
void        doTypeSelectSearch          (ListHandle,EventRecord *);
SInt16      searchPartialMatch          (Ptr,Ptr,SInt16,SInt16);
Boolean     doFindFirstSelectedCell     (ListHandle,Cell *);
void        doFindLastSelectedCell      (ListHandle,Cell *);
void        doFindNewCellLoc            (ListHandle,Cell,Cell *,SInt8,Boolean);
void        doSelectOneCell             (ListHandle,Cell);
void        doMakeCellVisible           (ListHandle,Cell);
void        doResetTypeSelection        (void);
void        doRotateCurrentList         (void);
void        doDrawFrameAndFocus         (ListHandle,Boolean);
void        doExtractSelections         (void);
void        doDrawSelections            (Boolean);
void        listDefFunction             (SInt16,Boolean,Rect *,Cell,SInt16,SInt16,ListHandle);
void        doLDEFDraw                  (Boolean,Rect *,Cell,SInt16,ListHandle);
void        doLDEFHighlight             (Rect *);

void        doListsDialog               (void);
Boolean     eventFilter                 (DialogPtr,EventRecord *,SInt16 *);

// ********************************************************************************************
// Lists.c
// ********************************************************************************************

// ..... includes

#include "Lists.h"

// ..... global variables

ListDefUPP        gListDefFunctionUPP;
Boolean           gRunningOnX = false;
backColourPattern gBackColourPattern;

// ********************************************************************************** main

void  main(void)
{
  MenuBarHandle menubarHdl;
  SInt32        response;
  MenuRef       menuRef;
  EventTypeSpec applicationEvents[] = { { kEventClassApplication, kEventAppActivated    },
                                        { kEventClassCommand,     kEventProcessCommand  },
                                        { kEventClassMenu,        kEventMenuEnableItems } };

  // ..... do preliminaries

  doPreliminaries();

  // ..... create universal procedure pointers

  gListDefFunctionUPP = NewListDefUPP((ListDefProcPtr) listDefFunction);

  // ..... set up menu bar and menus

  menubarHdl = GetNewMBar(rMenubar);
  if(menubarHdl == NULL)
    ExitToShell();
  SetMenuBar(menubarHdl);
  DrawMenuBar();

  Gestalt(gestaltMenuMgrAttr,&response);
  if(response & gestaltMenuMgrAquaLayoutMask)
  {
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
    {
      DeleteMenuItem(menuRef,iQuit);
```

```
      DeleteMenuItem(menuRef,iQuit - 1);
      DisableMenuItem(menuRef,0);
    }

    gRunningOnX = true;
  }
  else
  {
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
      SetMenuItemCommandID(menuRef,iQuit,kHICommandQuit);
  }

// ……… install application event handler

  InstallApplicationEventHandler(NewEventHandlerUPP((EventHandlerProcPtr) appEventHandler),
                                 GetEventTypeCount(applicationEvents),applicationEvents,
                                 0,NULL);

  // ……… run application event loop

  RunApplicationEventLoop();
}

// ********** doPreliminaries

void  doPreliminaries(void)
{
  MoreMasterPointers(256);
  InitCursor();
}

// ********** appEventHandler

OSStatus  appEventHandler(EventHandlerCallRef eventHandlerCallRef,EventRef eventRef,
                          void * userData)
{
  OSStatus      result = eventNotHandledErr;
  UInt32        eventClass;
  UInt32        eventKind;
  HICommand     hiCommand;
  MenuID        menuID;
  MenuItemIndex menuItem;
  WindowClass   windowClass;

  eventClass = GetEventClass(eventRef);
  eventKind  = GetEventKind(eventRef);

  switch(eventClass)
  {
    case kEventClassApplication:
      if(eventKind == kEventAppActivated)
        SetThemeCursor(kThemeArrowCursor);
      break;

    case kEventClassCommand:
      if(eventKind == kEventProcessCommand)
      {
        GetEventParameter(eventRef,kEventParamDirectObject,typeHICommand,NULL,
                          sizeof(HICommand),NULL,&hiCommand);
        menuID = GetMenuID(hiCommand.menu.menuRef);
        menuItem = hiCommand.menu.menuItemIndex;
        if((hiCommand.commandID != kHICommandQuit) &&
           (menuID >= mAppleApplication && menuID <= mDemonstration))
        {
          doMenuChoice(menuID,menuItem);
          result = noErr;
        }
      }
```

```
      break;

      case kEventClassMenu:
      if(eventKind == kEventMenuEnableItems)
      {
        GetWindowClass(FrontWindow(),&windowClass);
        if(windowClass == kDocumentWindowClass)
          doAdjustMenus();
        result = noErr;
      }
      break;
  }

  return result;
}

// ********************************************************************** windowEventHandler

OSStatus  windowEventHandler(EventHandlerCallRef eventHandlerCallRef,EventRef eventRef,
                             void* userData)
{
  OSStatus           result = eventNotHandledErr;
  UInt32             eventClass;
  UInt32             eventKind;
  WindowRef          windowRef;
  EventRecord        eventRecord;
  docStructureHandle docStrucHdl;
  ControlRef         controlRef = NULL;
  ControlPartCode    controlPartCode;
  SInt8              charCode;
  UInt32             modifiers;
  Point              mouseLocation;

  eventClass = GetEventClass(eventRef);
  eventKind  = GetEventKind(eventRef);

  switch(eventClass)
  {
    case kEventClassWindow:                                          // event class window
      GetEventParameter(eventRef,kEventParamDirectObject,typeWindowRef,NULL,sizeof(windowRef),
                        NULL,&windowRef);
      switch(eventKind)
      {
        case kEventWindowDrawContent:
          doDrawContent(windowRef);
          result = noErr;
          break;

        case kEventWindowActivated:
          doActivateDeactivate(windowRef,true);
          result = noErr;
          break;

        case kEventWindowDeactivated:
          doActivateDeactivate(windowRef,false);
          result = noErr;
          break;

        case kEventWindowClickContentRgn:
          GetEventParameter(eventRef,kEventParamMouseLocation,typeQDPoint,NULL,
                            sizeof(mouseLocation),NULL,&mouseLocation);
          SetPortWindowPort(FrontWindow());
          GlobalToLocal(&mouseLocation);
          GetEventParameter(eventRef,kEventParamKeyModifiers,typeUInt32,NULL,
                            sizeof(modifiers),NULL,&modifiers);
          doInContent(mouseLocation,modifiers);
          result = noErr;
          break;
```

```
        case kEventWindowClose:
          docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);
          LDispose((*docStrucHdl)->textListHdl);
          LDispose((*docStrucHdl)->iconListHdl);
          DisposeHandle((Handle) docStrucHdl);
          DisposeWindow(windowRef);
          result = noErr;
          break;
      }
      break;

    case kEventClassMouse:                                          // event class mouse
      switch(eventKind)
      {
        case kEventMouseDown:
          GetEventParameter(eventRef,kEventParamMouseLocation,typeQDPoint,NULL,
                            sizeof(mouseLocation),NULL,&mouseLocation);
          SetPortWindowPort(FrontWindow());
          GlobalToLocal(&mouseLocation);
          controlRef = FindControlUnderMouse(mouseLocation,FrontWindow(),&controlPartCode);
          if(controlRef)
          {
            doControlHit(FrontWindow(),controlRef,mouseLocation);
            result = noErr;
          }
          break;
      }
      break;

    case kEventClassKeyboard:                                      // event class keyboard
      switch(eventKind)
      {
        case kEventRawKeyDown:
          GetEventParameter(eventRef,kEventParamKeyMacCharCodes,typeChar,NULL,
                            sizeof(charCode),NULL,&charCode);
          GetEventParameter(eventRef,kEventParamKeyModifiers,typeUInt32,NULL,
                            sizeof(modifiers),NULL,&modifiers);
          eventRecord.what      = keyDown;
          eventRecord.message   = charCode;
          eventRecord.when      = EventTimeToTicks(GetEventTime(eventRef));
          eventRecord.modifiers = modifiers;
          doKeyDown(charCode,&eventRecord);
          result = noErr;
          break;
      }
      break;
  }

  return result;
}

// ********************************************************************************* doAdjustMenus

void  doAdjustMenus(void)
{
  MenuRef menuRef;

  menuRef = GetMenuRef(mDemonstration);

  if(FrontWindow())
    DisableMenuItem(menuRef,1);
  else
    EnableMenuItem(menuRef,1);
}

// ********************************************************************************* doMenuChoice

void  doMenuChoice(MenuID menuID,MenuItemIndex menuItem)
{
```

```
  if(menuID == 0)
    return;

  switch(menuID)
  {
    case mAppleApplication:
      if(menuItem == iAbout)
        SysBeep(10);
      break;

    case mDemonstration:
      if(menuItem == iHandMadeLists)
        doOpenListsWindow();
      else if(menuItem == iListControlLists)
        doListsDialog();
      break;
  }
}

// ********************************************************************** doSaveBackground

void  doSaveBackground(backColourPattern *gBackColourPattern)
{
  GrafPtr      currentPort;
  PixPatHandle backPixPatHdl;

  GetPort(&currentPort);

  GetBackColor(&gBackColourPattern->backColour);
  gBackColourPattern->backPixelPattern  = NULL;

  backPixPatHdl = NewPixPat();
  GetPortBackPixPat(currentPort,backPixPatHdl);

  if((*backPixPatHdl)->patType != 0)
    gBackColourPattern->backPixelPattern = backPixPatHdl;
  else
    gBackColourPattern->backBitPattern = *(PatPtr) (*(*backPixPatHdl)->patData);

  DisposePixPat(backPixPatHdl);
}

// ********************************************************************** doRestoreBackground

void  doRestoreBackground(backColourPattern *gBackColourPattern)
{
  RGBBackColor(&gBackColourPattern->backColour);

  if(gBackColourPattern->backPixelPattern)
    BackPixPat(gBackColourPattern->backPixelPattern);
  else
    BackPat(&gBackColourPattern->backBitPattern);
}

// ********************************************************************** doSetBackgroundWhite

void  doSetBackgroundWhite(void)
{
  RGBColor whiteColour = { 0xFFFF, 0xFFFF, 0xFFFF };
  Pattern  whitePattern;

  RGBBackColor(&whiteColour);
  GetQDGlobalsWhite(&whitePattern);
  BackPat(&whitePattern);
}

// ****************************************************************************************
// WindowList.c
// ****************************************************************************************
```

```
// ..... includes

#include "Lists.h"

// ..... global variables

ListHandle       gCurrentListHdl;
Str255           gStringArray[16];
TypeSelectRecord gTSStruct;
SInt16           gTSResetThreshold;
ListHandle       gTSLastListHit;

extern ListDefUPP        gListDefFunctionUPP;
extern backColourPattern gBackColourPattern;
extern Boolean           gRunningOnX;

// ********************************************************************** doOpenListsWindow

void  doOpenListsWindow(void)
{
  OSStatus           osError;
  Rect               contentRect = { 100,100,502,357 };
  WindowRef          windowRef;
  docStructureHandle docStrucHdl;
  ControlRef         controlRef;
  Str255             software = "\pSoftware:";
  Str255             hardware = "\pHardware:";
  SInt16             fontNum;
  Rect               textListRect, pictListRect;
  ListHandle         textListHdl, iconListHdl;
  EventTypeSpec   windowEvents[] = { { kEventClassWindow,   kEventWindowDrawContent     },
                                     { kEventClassWindow,   kEventWindowActivated       },
                                     { kEventClassWindow,   kEventWindowDeactivated     },
                                     { kEventClassWindow,   kEventWindowClickContentRgn },
                                     { kEventClassWindow,   kEventWindowClose           },
                                     { kEventClassMouse,    kEventMouseDown             },
                                     { kEventClassKeyboard, kEventRawKeyDown            } };

  // ..... open window, attach document structure, set background colour/pattern

  osError = CreateNewWindow(kDocumentWindowClass,kWindowStandardHandlerAttribute,
                            &contentRect,&windowRef);
  if(osError != noErr)
    ExitToShell();

  ChangeWindowAttributes(windowRef,kWindowCloseBoxAttribute,0);
  RepositionWindow(windowRef,NULL,kWindowAlertPositionOnMainScreen);
  SetWTitle(windowRef,"\pHand Made Lists");

  if(!(docStrucHdl = (docStructureHandle) NewHandle(sizeof(docStructure))))
    ExitToShell();
  SetWRefCon(windowRef,(SInt32) docStrucHdl);

  SetPortWindowPort(windowRef);

  SetThemeWindowBackground(windowRef,kThemeBrushDialogBackgroundActive,true);
  if(!gRunningOnX)
    doSaveBackground(&gBackColourPattern);
  else
    doSetBackgroundWhite();

  // ..... create window's controls

  if(!gRunningOnX)
    CreateRootControl(windowRef,&controlRef);

  if(!((*docStrucHdl)->extractButtonHdl = GetNewControl(cExtractButton,windowRef)))
    ExitToShell();
```

```
  if(!(controlRef = GetNewControl(cSoftwareStaticText,windowRef)))
    ExitToShell();
  SetControlData(controlRef,kControlEntireControl,kControlStaticTextTextTag,software[0],
                 &software[1]);

  if(!(controlRef = GetNewControl(cHardwareStaticText,windowRef)))
    ExitToShell();
  SetControlData(controlRef,kControlEntireControl,kControlStaticTextTextTag,hardware[0],
                 &hardware[1]);

  if(!(controlRef = GetNewControl(cGroupBox,windowRef)))
    ExitToShell();

  // ..... set window's font

  GetFNum("\pGeneva",&fontNum);
  TextFont(fontNum);
  TextSize(10);

  // ..... create lists, assign handles to document structure fields

  SetRect(&textListRect,21,26,151,130);
  SetRect(&pictListRect,169,26,236,130);

  textListHdl = doCreateTextList(windowRef,textListRect,1,kSystemLDEF);
  iconListHdl = doCreateIconList(windowRef,pictListRect,1,gListDefFunctionUPP);

  (*docStrucHdl)->textListHdl = textListHdl;
  (*docStrucHdl)->iconListHdl  = iconListHdl;

  // ..... assign handles to list structure refCon fields for linked ring

  SetListRefCon(textListHdl,(SInt32) iconListHdl);
  SetListRefCon(iconListHdl,(SInt32) textListHdl);

  // ..... make text list the current list

  gCurrentListHdl = textListHdl;

  // ..... install window event handler

  InstallWindowEventHandler(windowRef,
                            NewEventHandlerUPP((EventHandlerProcPtr) windowEventHandler),
                            GetEventTypeCount(windowEvents),windowEvents,0,NULL);

  // ..... save and set background colour/pattern, show window

  ShowWindow(windowRef);
}

// ********************************************************************************* doKeyDown

void  doKeyDown(SInt8 charCode,EventRecord * eventStrucPtr)
{
  docStructureHandle docStrucHdl;
  Boolean            allowExtendSelect;

  docStrucHdl = (docStructureHandle) GetWRefCon(FrontWindow());

  if(charCode == kTab)
  {
    doRotateCurrentList();
  }
  else if(charCode == kUpArrow || charCode == kDownArrow)
  {
    if(gCurrentListHdl == (*docStrucHdl)->textListHdl)
      allowExtendSelect = true;
    else
```

```
      allowExtendSelect = false;

    doHandleArrowKey(charCode,eventStrucPtr,allowExtendSelect);
  }
  else
  {
    if(gCurrentListHdl == (*docStrucHdl)->textListHdl)
      doTypeSelectSearch((*docStrucHdl)->textListHdl,eventStrucPtr);
  }
}

// ************************************************************************** doDrawContent

void  doDrawContent(WindowRef windowRef)
{
  docStructureHandle docStrucHdl;
  ListHandle         textListHdl, iconListHdl;
  RgnHandle          visibleRegionHdl = NewRgn();

  SetPortWindowPort(windowRef);

  GetPortVisibleRegion(GetWindowPort(windowRef),visibleRegionHdl);
  UpdateControls(windowRef,visibleRegionHdl);

  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);
  textListHdl = (*docStrucHdl)->textListHdl;
  iconListHdl = (*docStrucHdl)->iconListHdl;

  if(visibleRegionHdl)
  {
    if(gRunningOnX)
    {
      if(IsWindowHilited(windowRef))
        TextMode(srcOr);
      else
        TextMode(grayishTextOr);
    }

    LUpdate(visibleRegionHdl,textListHdl);
    LUpdate(visibleRegionHdl,iconListHdl);
    DisposeRgn(visibleRegionHdl);
  }

  doDrawFrameAndFocus(textListHdl,IsWindowHilited(windowRef));
  doDrawFrameAndFocus(iconListHdl,IsWindowHilited(windowRef));

  doDrawSelections(windowRef == FrontWindow());
}

// ********************************************************************** doActivateDeactivate

void  doActivateDeactivate(WindowRef windowRef,Boolean becomingActive)
{
  GrafPtr            oldPort;
  ControlRef         controlRef;
  docStructureHandle docStrucHdl;
  ListHandle         textListHdl, iconListHdl;
  RgnHandle          visibleRegionHdl = NewRgn();

  GetPort(&oldPort);
  SetPortWindowPort(windowRef);

  if(!gRunningOnX)
  {
    GetRootControl(windowRef,&controlRef);
    if(becomingActive)
      ActivateControl(controlRef);
    else
      DeactivateControl(controlRef);
```

```
  }

  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);
  textListHdl = (*docStrucHdl)->textListHdl;
  iconListHdl = (*docStrucHdl)->iconListHdl;

  if(visibleRegionHdl)
    GetPortVisibleRegion(GetWindowPort(windowRef),visibleRegionHdl);

  if(becomingActive)
  {
    LActivate(true,textListHdl);
    LActivate(true,iconListHdl);

    if(!gRunningOnX)
    {
      TextMode(srcOr);
      if(visibleRegionHdl)
      {
        LUpdate(visibleRegionHdl,textListHdl);
        LUpdate(visibleRegionHdl,iconListHdl);
        DisposeRgn(visibleRegionHdl);
      }
    }

    doDrawFrameAndFocus(textListHdl,true);
    doDrawFrameAndFocus(iconListHdl,true);

    doResetTypeSelection();

    doDrawSelections(true);
  }
  else
  {
    LActivate(false,textListHdl);
    LActivate(false,iconListHdl);

    if(!gRunningOnX)
    {
      TextMode(grayishTextOr);
      if(visibleRegionHdl)
      {
        LUpdate(visibleRegionHdl,textListHdl);
        LUpdate(visibleRegionHdl,iconListHdl);
        DisposeRgn(visibleRegionHdl);
      }
    }

    doDrawFrameAndFocus(textListHdl,false);
    doDrawFrameAndFocus(iconListHdl,false);

    doDrawSelections(false);
  }

  SetPort(oldPort);
}

// ************************************************************************** doInContent

void  doInContent(Point mouseLocation,UInt32 modifiers)
{
  WindowRef          windowRef;
  docStructureHandle docStrucHdl;
  Rect               textListRect, pictListRect;
  Boolean            isDoubleClick;

  windowRef = FrontWindow();
  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);
```

```
  GetListViewBounds((*docStrucHdl)->textListHdl,&textListRect);
  GetListViewBounds((*docStrucHdl)->iconListHdl,&pictListRect);

  if(PtInRect(mouseLocation,&textListRect) || PtInRect(mouseLocation,&pictListRect))
  {
    if((PtInRect(mouseLocation,&textListRect) &&
        gCurrentListHdl != (*docStrucHdl)->textListHdl) ||
       (PtInRect(mouseLocation,&pictListRect) &&
        gCurrentListHdl != (*docStrucHdl)->iconListHdl))
    {
      doRotateCurrentList();
    }

    isDoubleClick = LClick(mouseLocation,modifiers,gCurrentListHdl);
    if(isDoubleClick)
      doExtractSelections();
  }
}

// ************************************************************************** doControlHit

void  doControlHit(WindowRef windowRef,ControlRef controlRef,Point mouseLocation)
{
  docStructureHandle docStrucHdl;

  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

  if(controlRef == (*docStrucHdl)->extractButtonHdl)
  {
    if(TrackControl(controlRef,mouseLocation,NULL))
      doExtractSelections();
  }
  else
  {
    if((mouseLocation.h < 165 && gCurrentListHdl != (*docStrucHdl)->textListHdl) ||
       (mouseLocation.h > 165 && gCurrentListHdl != (*docStrucHdl)->iconListHdl))
    {
      doRotateCurrentList();
    }

    LClick(mouseLocation,0,gCurrentListHdl);
  }
}

// ************************************************************************** doCreateTextList

ListHandle  doCreateTextList(WindowRef windowRef,Rect listRect,SInt16 numCols,SInt16 lDef)
{
  Rect       dataBounds;
  Point      cellSize;
  ListHandle textListHdl;
  Cell       theCell;

  SetRect(&dataBounds,0,0,numCols,0);
  SetPt(&cellSize,0,0);

  listRect.right = listRect.right - kScrollBarWidth;

  textListHdl = LNew(&listRect,&dataBounds,cellSize,lDef,windowRef,true,false,false,true);

  doAddRowsAndDataToTextList(textListHdl,rTextListStrings,15);

  SetPt(&theCell,0,0);
  LSetSelect(true,theCell,textListHdl);

  doResetTypeSelection();

  return textListHdl;
}
```

```
// ************************************************************ doAddRowsAndDataToTextList

void  doAddRowsAndDataToTextList(ListHandle textListHdl,SInt16 stringListID,
                                 SInt16 numberOfStrings)
{
  SInt16 stringIndex;
  Str255 theString;

  for(stringIndex = 1;stringIndex < numberOfStrings + 1;stringIndex++)
  {
    GetIndString(theString,stringListID,stringIndex);
    doAddTextItemAlphabetically(textListHdl,theString);
  }
}

// ************************************************************ doAddTextItemAlphabetically

void  doAddTextItemAlphabetically(ListHandle listHdl,Str255 theString)
{
  Boolean    found;
  ListBounds dataBounds;
  SInt16     totalRows, currentRow, cellDataOffset, cellDataLength;
  DataHandle dataHandle;
  Cell       aCell;

  found = false;

  GetListDataBounds(listHdl,&dataBounds);
  totalRows = dataBounds.bottom - dataBounds.top;
  currentRow = -1;

  while(!found)
  {
    currentRow += 1;
    if(currentRow == totalRows)
      found = true;
    else
    {
      SetPt(&aCell,0,currentRow);
      LGetCellDataLocation(&cellDataOffset,&cellDataLength,aCell,listHdl);

      dataHandle = GetListDataHandle(listHdl);
      MoveHHi((Handle) dataHandle);
      HLock((Handle) dataHandle);

      if(CompareText(theString + 1,((Ptr) (*listHdl)->cells[0] + cellDataOffset),
                     StrLength(theString),cellDataLength,NULL) == -1)
      {
        found = true;
      }

      HUnlock((Handle) dataHandle);
    }
  }

  currentRow = LAddRow(1,currentRow,listHdl);
  SetPt(&aCell,0,currentRow);

  LSetCell(theString + 1,(SInt16) StrLength(theString),aCell,listHdl);
}

// ************************************************************************ doCreateIconList

ListHandle  doCreateIconList(WindowRef windowRef,Rect listRect,SInt16 numCols,
                             ListDefUPP listDefinitionFunctionUPP)
{
  Rect        dataBounds;
  Point       cellSize;
```

```
  ListHandle  iconListHdl;
  Cell        theCell;
  ListDefSpec listDefSpec;

  SetRect(&dataBounds,0,0,numCols,0);
  SetPt(&cellSize,52,52);

  listRect.right = listRect.right - kScrollBarWidth;

  listDefSpec.u.userProc = listDefinitionFunctionUPP;

  CreateCustomList(&listRect,&dataBounds,cellSize,&listDefSpec,windowRef,true,false,false,
                   true,&iconListHdl);

  SetListSelectionFlags(iconListHdl,lOnlyOne);

  doAddRowsAndDataToIconList(iconListHdl,rIconListIconSuiteBase);

  SetPt(&theCell,0,0);
  LSetSelect(true,theCell,iconListHdl);

  return iconListHdl;
}

// ************************************************************* doAddRowsAndDataToIconList

void  doAddRowsAndDataToIconList(ListHandle iconListHdl,SInt16 iconSuiteBase)
{
  ListBounds dataBounds;
  SInt16     rowNumber, suiteIndex, index = 0;
  Handle     iconSuiteHdl;
  Cell       theCell;

  GetListDataBounds(iconListHdl,&dataBounds);
  rowNumber = dataBounds.bottom;

  for(suiteIndex = iconSuiteBase;suiteIndex < (iconSuiteBase + 10);suiteIndex++)
  {
    GetIconSuite(&iconSuiteHdl,suiteIndex,kSelectorAllLargeData);

    rowNumber = LAddRow(1,rowNumber,iconListHdl);
    SetPt(&theCell,0,rowNumber);
    LSetCell(&iconSuiteHdl,sizeof(iconSuiteHdl),theCell,iconListHdl);

    rowNumber += 1;
  }
}

// ************************************************************************** doHandleArrowKey

void  doHandleArrowKey(SInt8 charCode,EventRecord * eventStrucPtr,Boolean allowExtendSelect)
{
  Boolean  moveToTopBottom = false;

  if(eventStrucPtr->modifiers & cmdKey)
    moveToTopBottom = true;

  if(allowExtendSelect && (eventStrucPtr->modifiers & shiftKey))
    doArrowKeyExtendSelection(gCurrentListHdl,charCode,moveToTopBottom);
  else
    doArrowKeyMoveSelection(gCurrentListHdl,charCode,moveToTopBottom);
}

// ************************************************************** doArrowKeyMoveSelection

void  doArrowKeyMoveSelection(ListHandle listHdl,SInt8 charCode,Boolean moveToTopBottom)
{
  Cell currentSelection, newSelection;
```

```
  if(doFindFirstSelectedCell(listHdl,&currentSelection))
  {
    if(charCode == kDownArrow)
      doFindLastSelectedCell(listHdl,&currentSelection);

    doFindNewCellLoc(listHdl,currentSelection,&newSelection,charCode,moveToTopBottom);

    doSelectOneCell(listHdl,newSelection);
    doMakeCellVisible(listHdl,newSelection);
  }
}

// ************************************************************** doArrowKeyExtendSelection

void  doArrowKeyExtendSelection(ListHandle listHdl,SInt8 charCode,Boolean moveToTopBottom)
{
  Cell currentSelection, newSelection;

  if(doFindFirstSelectedCell(listHdl,&currentSelection))
  {
    if(charCode == kDownArrow)
      doFindLastSelectedCell(listHdl,&currentSelection);

    doFindNewCellLoc(listHdl,currentSelection,&newSelection,charCode,moveToTopBottom);

    if(!(LGetSelect(false,&newSelection,listHdl)))
      LSetSelect(true,newSelection,listHdl);

    doMakeCellVisible(listHdl,newSelection);
  }
}

// ************************************************************************ doTypeSelectSearch

void  doTypeSelectSearch(ListHandle listHdl,EventRecord * eventStrucPtr)
{
  Cell          theCell;
  ListSearchUPP searchPartialMatchUPP;

  if((gTSLastListHit != listHdl) || ((eventStrucPtr->when - gTSStruct.tsrLastKeyTime) >=
     gTSResetThreshold) || (StrLength(gTSStruct.tsrKeyStrokes) == 63))
    doResetTypeSelection();

  gTSLastListHit = listHdl;
  gTSStruct.tsrLastKeyTime = eventStrucPtr->when;

  TypeSelectNewKey(eventStrucPtr,&gTSStruct);

  SetPt(&theCell,0,0);

  searchPartialMatchUPP  = NewListSearchUPP((ListSearchProcPtr) searchPartialMatch);

  if(LSearch(gTSStruct.tsrKeyStrokes + 1,StrLength(gTSStruct.tsrKeyStrokes),
             searchPartialMatchUPP,&theCell,listHdl))
  {
    LSetSelect(true,theCell,listHdl);
    doSelectOneCell(listHdl,theCell);
    doMakeCellVisible(listHdl,theCell);
  }

  DisposeListSearchUPP(searchPartialMatchUPP);
}

// ************************************************************************ searchPartialMatch

SInt16  searchPartialMatch(Ptr searchDataPtr,Ptr cellDataPtr,SInt16 cellDataLen,
                           SInt16 searchDataLen)
{
  SInt16 result;
```

```
  if((cellDataLen > 0) && (cellDataLen >= searchDataLen))
    result = IdenticalText(cellDataPtr,searchDataPtr,searchDataLen,searchDataLen,NULL);
  else
    result = 1;

  return result;
}

// ********************************************************************* doFindFirstSelectedCell

Boolean  doFindFirstSelectedCell(ListHandle listHdl,Cell *theCell)
{
  Boolean result;

  SetPt(theCell,0,0);
  result = LGetSelect(true,theCell,listHdl);

  return result;
}

// ********************************************************************** doFindLastSelectedCell

void  doFindLastSelectedCell(ListHandle listHdl,Cell *theCell)
{
  Cell    aCell;
  Boolean moreCellsInList;

  if(doFindFirstSelectedCell(listHdl,&aCell))
  {
    while(LGetSelect(true,&aCell,listHdl))
    {
      *theCell = aCell;
      moreCellsInList = LNextCell(true,true,&aCell,listHdl);
    }
  }
}

// ************************************************************************** doFindNewCellLoc

void  doFindNewCellLoc(ListHandle listHdl,Cell oldCellLoc,Cell *newCellLoc,SInt8 charCode,
                       Boolean moveToTopBottom)
{
  ListBounds dataBounds;
  SInt16     listRows;

  GetListDataBounds(listHdl,&dataBounds);
  listRows = dataBounds.bottom - dataBounds.top;
  *newCellLoc = oldCellLoc;

  if(moveToTopBottom)
  {
    if(charCode == kUpArrow)
      (*newCellLoc).v = 0;
    else if(charCode == kDownArrow)
      (*newCellLoc).v = listRows - 1;
  }
  else
  {
    if(charCode ==  kUpArrow)
    {
      if(oldCellLoc.v != 0)
        (*newCellLoc).v = oldCellLoc.v - 1;
    }
    else if(charCode == kDownArrow)
    {
      if(oldCellLoc.v != listRows - 1)
        (*newCellLoc).v = oldCellLoc.v + 1;
    }
```

```
  }
}

// ************************************************************************** doSelectOneCell

void  doSelectOneCell(ListHandle listHdl,Cell theCell)
{
  Cell    nextSelectedCell;
  Boolean moreCellsInList;

  if(doFindFirstSelectedCell(listHdl,&nextSelectedCell))
  {
    while(LGetSelect(true,&nextSelectedCell,listHdl))
    {
      if(nextSelectedCell.v != theCell.v)
        LSetSelect(false,nextSelectedCell,listHdl);
      else
        moreCellsInList = LNextCell(true,true,&nextSelectedCell,listHdl);
    }

    LSetSelect(true,theCell,listHdl);
  }
}

// ************************************************************************** doMakeCellVisible

void  doMakeCellVisible(ListHandle listHdl,Cell newSelection)
{
  ListBounds visibleRect;
  SInt16     dRows;

  GetListVisibleCells(listHdl,&visibleRect);

  if(!(PtInRect(newSelection,&visibleRect)))
  {
    if(newSelection.v > visibleRect.bottom - 1)
      dRows = newSelection.v - visibleRect.bottom + 1;
    else if(newSelection.v < visibleRect.top)
      dRows = newSelection.v - visibleRect.top;

    LScroll(0,dRows,listHdl);
  }
}

// ************************************************************************** doResetTypeSelection

void  doResetTypeSelection(void)
{
  TypeSelectClear(&gTSStruct);
  gTSLastListHit = NULL;
  gTSResetThreshold = 2 * LMGetKeyThresh();
  if(gTSResetThreshold > kMaxKeyThresh)
    gTSResetThreshold = kMaxKeyThresh;
}

// ************************************************************************** doRotateCurrentList

void  doRotateCurrentList(void)
{
  ListHandle oldListHdl, newListHdl;

  oldListHdl = gCurrentListHdl;

  newListHdl = (ListHandle) GetListRefCon(gCurrentListHdl);
  gCurrentListHdl = newListHdl;

  doDrawFrameAndFocus(oldListHdl,true);
  doDrawFrameAndFocus(newListHdl,true);
}
```

```
// ********************************************************************** doDrawFrameAndFocus

void  doDrawFrameAndFocus(ListHandle listHdl,Boolean inState)
{
  Rect borderRect;

  GetListViewBounds(listHdl,&borderRect);
  borderRect.right += kScrollBarWidth;

  if(!gRunningOnX)
    doRestoreBackground(&gBackColourPattern);
  else
    InvalWindowRect(FrontWindow(),&borderRect);

  DrawThemeFocusRect(&borderRect,false);

  if(inState)
    DrawThemeListBoxFrame(&borderRect,kThemeStateActive);
  else
    DrawThemeListBoxFrame(&borderRect,kThemeStateInactive);

  if(listHdl == gCurrentListHdl)
    DrawThemeFocusRect(&borderRect,inState);

  if(!gRunningOnX)
    doSetBackgroundWhite();
}

// ********************************************************************** doExtractSelections

void  doExtractSelections(void)
{
  docStructureHandle docStrucHdl;
  ListHandle         textListHdl, iconListHdl;
  SInt16             a, cellIndex, offset, dataLen;
  ListBounds         dataBounds;
  Cell               theCell;
  Rect               theRect;

  docStrucHdl = (docStructureHandle) GetWRefCon(FrontWindow());
  textListHdl = (*docStrucHdl)->textListHdl;
  iconListHdl = (*docStrucHdl)->iconListHdl;

  for(a=0;a<16;a++)
    gStringArray[a][0] = 0;

  GetListDataBounds(textListHdl,&dataBounds);

  for(cellIndex = 0;cellIndex < dataBounds.bottom;cellIndex++)
  {
    SetPt(&theCell,0,cellIndex);
    if(LGetSelect(false,&theCell,textListHdl))
    {
      LGetCellDataLocation(&offset,&dataLen,theCell,textListHdl);
      LGetCell(gStringArray[cellIndex] + 1,&dataLen,theCell,textListHdl);
      gStringArray[cellIndex][0] = (SInt8) dataLen;
    }
  }

  SetPt(&theCell,0,0);
  LGetSelect(true,&theCell,iconListHdl);
  GetIndString(gStringArray[15],rIconListStrings,theCell.v + 1);

  SetRect(&theRect,24,181,233,380);
  InvalWindowRect(FrontWindow(),&theRect);
}

// ********************************************************************** doDrawSelections
```

```
void  doDrawSelections(Boolean inState)
{
  Rect        theRect;
  SInt16      a, nextLine = 190;
  CFStringRef stringRef;

  if(inState == kThemeStateActive)
    TextMode(srcOr);
  else
    TextMode(grayishTextOr);

  SetRect(&theRect,22,179,235,382);
  EraseRect(&theRect);

  for(a=0;a<15;a++)
  {
    if(gStringArray[a][0] != 0)
    {
      stringRef = CFStringCreateWithPascalString(NULL,gStringArray[a],
                                                 kCFStringEncodingMacRoman);
      SetRect(&theRect,36,nextLine,240,nextLine + 15);
      DrawThemeTextBox(stringRef,kThemeSmallSystemFont,0,false,&theRect,teJustLeft,
                       NULL);
        nextLine += 12;
    }
  }

  stringRef = CFStringCreateWithPascalString(NULL,gStringArray[15],
                                             kCFStringEncodingMacRoman);
  SetRect(&theRect,170,190,240,205);
  DrawThemeTextBox(stringRef,kThemeSmallSystemFont,0,false,&theRect,teJustLeft,
                   NULL);

  TextMode(srcOr);
}

// ********************************************************************** listDefFunction

void  listDefFunction(SInt16 message,Boolean selected,Rect *cellRect,Cell theCell,
                      SInt16 dataOffset,SInt16 dataLen,ListHandle theList)
{
  switch(message)
  {
    case lDrawMsg:
      doLDEFDraw(selected,cellRect,theCell,dataLen,theList);
      break;

    case lHiliteMsg:
      doLDEFHighlight(cellRect);
      break;
  }
}

// ************************************************************************** doLDEFDraw

void  doLDEFDraw(Boolean selected,Rect *cellRect,Cell theCell,SInt16 dataLen,
                 ListHandle theList)
{
  GrafPtr oldPort;
  Rect    drawRect;
  Handle  iconSuiteHdl;
  Str255  theString;

  GetPort(&oldPort);

  SetPort(GetListPort(theList));

  EraseRect(cellRect);
```

```
  drawRect = *cellRect;

  drawRect.top += 2;
  drawRect.left += 10;
  drawRect.right -= 10;
  drawRect.bottom -= 18;

  if(dataLen == sizeof(Handle))
  {
    LGetCell(&iconSuiteHdl,&dataLen,theCell,theList);

    if(GetListActive(theList))
      PlotIconSuite(&drawRect,kAlignNone,kTransformNone,iconSuiteHdl);
    else
      PlotIconSuite(&drawRect,kAlignNone,kTransformDisabled,iconSuiteHdl);
  }

  GetIndString(theString,129,theCell.v + 1);
  SetRect(&drawRect,drawRect.left - 10,drawRect.top + 36,drawRect.right + 10,
          drawRect.bottom + 16);
  TETextBox(&theString[1],theString[0],&drawRect,teCenter);

  if(selected)
    doLDEFHighlight(cellRect);

  SetPort(oldPort);
}

// ************************************************************************** doLDEFHighlight

void  doLDEFHighlight(Rect *cellRect)
{
  UInt8 hiliteVal;

  hiliteVal = LMGetHiliteMode();
  BitClr(&hiliteVal,pHiliteBit);
  LMSetHiliteMode(hiliteVal);

  InvertRect(cellRect);
}

// ********************************************************************************************
// DialogLists.c                                                          CLASSIC EVENT MODEL
// ********************************************************************************************

// ..................................................................................... includes

#include "Lists.h"

// ************************************************************************** doListsDialog

void  doListsDialog(void)
{
  DialogPtr      dialogPtr;
  GrafPtr        oldPort;
  ModalFilterUPP eventFilterUPP;
  ControlRef     dateFormatControlRef, watermarkControlRef, controlRef;
  ListHandle     dateFormatListHdl, watermarkListHdl;
  SInt16         itemHit;
  Cell           theCell;
  SInt16         dataLen, offset;
  Str255         dateFormatString, watermarkString;
  Boolean        wasDoubleClick = false;

  // ..................................................... explicitly deactivate front window if it exists, create dialog

  if(FrontWindow())
    doActivateDeactivate(FrontWindow(),false);
```

```
if(!(dialogPtr = GetNewDialog(rListsDialog,NULL,(WindowRef) -1)))
  ExitToShell();

GetPort(&oldPort);
SetPortDialogPort(dialogPtr);

// ……… set default push button

SetDialogDefaultItem(dialogPtr,kStdOkItemIndex);

// ……… create universal procedure pointer for event filter function

eventFilterUPP = NewModalFilterUPP((ModalFilterProcPtr) eventFilter);

// ……… add rows to lists, store data in their cells, modify cell selection algorithm

GetDialogItemAsControl(dialogPtr,iDateFormatList,&dateFormatControlRef);
GetControlData(dateFormatControlRef,kControlEntireControl,kControlListBoxListHandleTag,
               sizeof(dateFormatListHdl),&dateFormatListHdl,NULL);

doAddRowsAndDataToTextList(dateFormatListHdl,rDateFormatStrings,17);

SetListSelectionFlags(dateFormatListHdl,lOnlyOne);

SetPt(&theCell,0,0);
LSetSelect(true,theCell,dateFormatListHdl);

GetDialogItemAsControl(dialogPtr,iWatermarkList,&watermarkControlRef);
GetControlData(watermarkControlRef,kControlEntireControl,kControlListBoxListHandleTag,
               sizeof(watermarkListHdl),&watermarkListHdl,NULL);

doAddRowsAndDataToTextList(watermarkListHdl,rWatermarkStrings,12);

SetListSelectionFlags(watermarkListHdl,lOnlyOne);

SetPt(&theCell,0,0);
LSetSelect(true,theCell,watermarkListHdl);

// ……… show dialog and set keyboard focus

ShowWindow(GetDialogWindow(dialogPtr));

SetKeyboardFocus(GetDialogWindow(dialogPtr),dateFormatControlRef,1);

// ……… enter ModalDialog loop

do
{
  ModalDialog(eventFilterUPP,&itemHit);

  if(itemHit == iDateFormatList)
  {
    SetPt(&theCell,0,0);
    LGetSelect(true,&theCell,dateFormatListHdl);
    LGetCellDataLocation(&offset,&dataLen,theCell,dateFormatListHdl);
    LGetCell(dateFormatString + 1,&dataLen,theCell,dateFormatListHdl);
    dateFormatString[0] = (SInt8) dataLen;

    GetDialogItemAsControl(dialogPtr,iDateFormatStaticText,&controlRef);
    SetControlData(controlRef,kControlEntireControl,kControlStaticTextTextTag,
                   dateFormatString[0],&dateFormatString[1]);
    Draw1Control(controlRef);

    GetControlData(dateFormatControlRef,kControlEntireControl,
                   kControlListBoxDoubleClickTag,sizeof(wasDoubleClick),&wasDoubleClick,
                   NULL);
  }
  else if(itemHit == iWatermarkList)
```

```
    {
      SetPt(&theCell,0,0);
      LGetSelect(true,&theCell,watermarkListHdl);
      LGetCellDataLocation(&offset,&dataLen,theCell,watermarkListHdl);
      LGetCell(watermarkString + 1,&dataLen,theCell,watermarkListHdl);
      watermarkString[0] = (SInt8) dataLen;

      GetDialogItemAsControl(dialogPtr,iWatermarkStaticText,&controlRef);
      SetControlData(controlRef,kControlEntireControl,kControlStaticTextTextTag,
                     watermarkString[0],&watermarkString[1]);
      Draw1Control(controlRef);

      GetControlData(watermarkControlRef,kControlEntireControl,
                     kControlListBoxDoubleClickTag,sizeof(wasDoubleClick),&wasDoubleClick,
                     NULL);
    }

  } while(itemHit != kStdOkItemIndex &&  wasDoubleClick == false);

  // ..... clean up

  DisposeDialog(dialogPtr);
  DisposeModalFilterUPP(eventFilterUPP);
  SetPort(oldPort);
}

// ********************************************************************************* eventFilter

Boolean  eventFilter(DialogPtr dialogPtr,EventRecord *eventStrucPtr,SInt16 *itemHit)
{
  Boolean    handledEvent;
  GrafPtr    oldPort;
  SInt8      charCode;
  ControlRef controlRef, focusControlRef;
  ListHandle watermarkListHdl;

  handledEvent = false;
  GetPort(&oldPort);
  SetPortDialogPort(dialogPtr);

  if(eventStrucPtr->what == keyDown)
  {
    charCode = eventStrucPtr->message & charCodeMask;

    if(charCode != kUpArrow && charCode != kDownArrow && charCode != kTab)
    {
      GetDialogItemAsControl(dialogPtr,iWatermarkList,&controlRef);
      GetControlData(controlRef,kControlEntireControl,kControlListBoxListHandleTag,
                     sizeof(watermarkListHdl),&watermarkListHdl,NULL);
      GetKeyboardFocus(GetDialogWindow(dialogPtr),&focusControlRef);
      if(controlRef == focusControlRef)
      {
        doTypeSelectSearch(watermarkListHdl,eventStrucPtr);
        Draw1Control(controlRef);
      }

      handledEvent = true;
    }
  }
  else
  {
    handledEvent = StdFilterProc(dialogPtr,eventStrucPtr,itemHit);
  }

  SetPort(oldPort);
  return handledEvent;
}

// ***********************************************************************************************
```

Demonstration Program Lists Comments

When this program is run, the user should open the window and movable modal dialog by choosing the relevant items in the Demonstration menu. The user should manipulate the lists in the window and dialog, noting their behaviour in the following circumstances:

- Changing the active list (that is, the current target of mouse and keyboard activity) by clicking in the non-active list and by using the Tab key to cycle between the two lists.
- Scrolling the active list using the vertical scroll bars, including dragging the scroll box/scroller and clicking in the scroll arrows and gray areas/track.
- Clicking, and clicking and dragging, in the active list so as to select a particular cell, including dragging the cursor above and below the list to automatically scroll the list to the desired cell.
- Pressing the Up-Arrow and Down-Arrow keys, noting that this action changes the selected cell and, where necessary, scrolls the list to make the newly-selected cell visible.
- In the lists in the window:
  - Double-clicking on a cell in the active list.
  - Pressing the Command-key as well as the Up-Arrow and Down-Arrow keys, noting that, in both the text list and the picture list, this results in the top-most or bottom-most cell being selected.
- In the "Software" list in the window:
  - Shift-clicking and dragging in the list to make contiguous multiple cell selections.
  - Command-clicking and dragging in the list to make discontiguous multiple cell selections, noting the differing effects depending on whether the cell initially clicked is selected or not selected.
  - Shift-clicking outside a block of multiple cell selections, including between two fairly widely separated discontiguous selected cells.
  - Pressing the Shift-key as well as the Up-Arrow and Down-Arrow keys, noting that this results in multiple cell selections.
- When the text list in the window, or the right hand list in the dialog, is the active list, typing the text of a particular cell so as to select that cell by type selection, noting the effects of any excessive delay between keystrokes.

The user should also send the program to the background and bring it to the foreground again, noting the list deactivation/activation effects.

Lists.h

defines

rListsDialog represents the resource ID of the dialog's 'DLOG' resource. CExtractButton and the following three constants represent the resource IDs of the window's controls. The next three constants represent the resource IDs of resources containing the strings for the window's text list, the icon suite for the icon list, and the strings for the icon titles.

rListsDialog represents the resource ID of the dialog's 'DLOG', 'dlgx', and 'dftb' resources. The following four constants represent the item numbers of items in the dialog's 'DITL' resource. The next two constants represent the resource IDs of the 'STR#' resources containing the strings for the dialog's lists.

The next three constants represent the character codes returned by the Up Arrow, Down Arrow, and Tab keys. kScrollBarWidth represents the width of the lists' vertical scroll bars. kMaxKeyThresh is used in the type selection function. kSystemLDEF represents the resource ID of the default list definition function.

typedefs

The type docStructure will be used to store the handles to the two list structures for the window and the reference to the window's push button. The handle to this structure will be stored in the window object.

The backColourPattern data type will be used to save and restore the background colour and pattern.

Lists.c

Lists.c is simply the basic "engine" which supports the demonstration. There is little in this file that has not featured in previous demonstration programs.

main

A universal procedure pointer is created for the custom list definition function used by the second list in the window.

windowEventHandler

When the kEventWindowClickContentRgn event type is received, doInContent is called. The mouse location in local coordinates and the modifier keys are passed in the call.

When the kEventWindowClose event type is received, a handle to the window's document structure is retrieved so as to be able to pass the handles to the window's two list structures in the two calls to LDispose. LDispose disposes of all memory associated with the specified list.

When the kEventRawKeyDown event type is received, doKeyDown is called with the address of a variable of type EventRecord passed in the second parameter. Note that, in the case of this particular event type, the function ConvertEventRefToEventRecord does not "fill in" the event record's what, message, and modifiers fields. Accordingly, in lieu of a call to ConvertEventRefToEventRecord, the character code and modifiers are extracted from the event and a fully fleshed-out event record is constructed by the program prior to the call to doKeyDown.

doSaveBackground, doRestoreBackground, and doSetBackgroundWhite

doSaveBackground and doRestoreBackground save and restore the background colour and the background bit or pixel pattern, and are called only when the program is run on Mac OS 8/9. doSetBackgroundWhite sets the background colour to white and the background pattern to the pattern white.

WindowList.c

WindowList.c contains the functions pertaining to the lists in the window.

Global Variables

gCurrentListHandle will be assigned the handle to the list structure associated with the currently active list in the window. gStringArray will be assigned strings representing the selections from the lists. The next three global variables are associated with the type selection functions.

doOpenListsWindow

doOpenListsWindow creates the window and its controls, and calls the functions which create the two lists for the window.

SetThemeWindowBackground is called to set the window's background colour/pattern and, if the program is running on Mac OS 8/9, doSaveBackground is called to save this background colour/pattern for later use in the function doDrawFrameAndFocus. The call to doSetBackgroundWhite at this point is required only on Mac OS X to ensure that the background within the list frames is drawn in white.

CreateRootControl creates a root control for the window so as to simplify the task of activating and deactivating the window's controls. (This call is not required on Mac OS X because, on Mac OS X, the root control will be created automatically for windows which have at least one control.) The window's remaining controls are then created.

The calls to doCreateTextList and doCreateIconList create the lists. First, the rectangles in which the lists are to be displayed are defined. These are then passed in the calls to doCreateTextList and doCreateIconList. The handles to the list structures returned by these functions are then assigned to the relevant fields of the window's document structure.

The next block assigns the icon list's handle to the refCon field of the text list's list structure and the text list's handle to the refCon field of the icon list's list structure. This establishes the "linked ring" which will be used to facilitate the rotation of the active list via Tab key presses.

The next line establishes the text list as the currently active list.

doKeyDown

The first line gets the handle to the document structure.

If the key pressed was the Tab key, doRotateCurrentList is called to change the currently active list.

If the key pressed was either the Up Arrow or the Down Arrow key, and if the current list is the text list, a variable which specifies whether multiple cell selections via the keyboard are permitted is set to true. If the current list is the icon list, this variable is set to false. This variable is then passed as a parameter in a call to doHandleArrowKey, which further processes the Arrow key event.

If the key pressed was neither the Tab key, the Up Arrow key, or the Down Arrow key, and if the active list is the text list, the event is passed to doTypeSelectSearch (the type selection function) for further processing.

doDrawContent

doDrawContent is called when the kEventWindowDrawContent event type is received. The calls to LUpdate update (that is, redraw) the lists and the calls to doDrawFrameAndFocus draw the focus rectangles in the appropriate state. The call to doDrawSelections simply draws the current list selections in the rectangle at the bottom of the window.

The calls to TextMode if the program is running on Mac OS X are required because of certain machinations in the function doDrawFrameAndFocus.

doActivateDeactivate

doActivateDeactivate is called when the kEventWindowActivated and kEventWindowDeactivated event types are received.

If the program is running on Mac OS 8/9 the root control is activated or deactivated, as appropriate, thus activating or deactivating all the controls in the window. (This is done automatically on Mac OS X.)

If the window is becoming active, the following occurs. For both lists, LActivate is called with true passed in the first parameter so as to highlight the currently selected cells. The calls to TextMode and LUpdate when the program is running on Mac OS 8/9 are required for cosmetic purposes only. LUpdate causes a redraw of all of the list's text in the srcOr mode. The calls to doDrawFrameAndFocus draw the list box frames in the active state and ensure that a keyboard focus frame is redrawn around the currently active list. The call to doResetTypeSelection resets certain variables used by the type selection function. (This latter is necessary because it is possible that, while the application was in the background, the user may have changed the "Delay Until Repeat" setting in the Keyboard control panel (Mac OS 8/9) or System Preferences/Keyboard (Mac OS X), a value which is used in the type selection function.) doDrawSelections redraws the current list selections in the srcOr mode.

Except for the call to doResetTypeSelection, much the same occurs if the window is becoming inactive, except that LActivate removes highlighting from the currently selected cells, LUpdate redraws the lists' text in the grayishTextOr mode (on Mac OS 8/9), doDrawFrameAndFocus removes the keyboard focus frame from the active list and draws the list box frames in the inactive state, and doDrawSelections redraws the current list selections in the grayishTextOr mode.

doInContent

doInContent is called when the kEventWindowClickContentRgn event type is received. Note that the mouse location received in the mouseLocation formal parameter is in local coordinates.

The calls to GetListViewBounds get the lists' display rectangles.

If the mouse click was in one of the list rectangles, and if that rectangle is not the current list's rectangle, doRotateCurrentList is called to change the currently active list. Next, LClick is called to handle all user action until the mouse-button is released. If LClick returns true, a double-click occurred, in which case doExtractSelections is called to extract and display the contents of the currently selected cells.

doControlHit

doControlHit is called when the kEventMouseDown event type is received and a call to FindControlUnderMouse reports that there is a control under the mouse cursor.

If the control is the Extract push button, TrackControl is called to handle user actions until the mouse button is released. If the cursor is still within the control when the mouse button is released, doExtractSelections is called to extract and display the contents of the currently selected cells.

If the control is one of the lists' scroll bars, doRotateCurrentList is called to change the currently active list. (This is necessary because the function doInContent only responds to clicks in the lists' display rectangles, which exclude the scroll bars.) LClick is then called to handle user interaction with the scroll bar.

doCreateTextList

doCreateTextList, supported by the two following functions, creates the text list.

SetRect sets the rectangle which will be passed as the rDataBnds parameter of the LNew call to specify one column and (initially) no rows. SetPt sets the variable that will be passed as the cellSize parameter so as to specify that the List Manager should automatically calculate the cell size. The next line adjusts the received list rectangle to eliminate the area occupied by the vertical scroll bar.

The call to LNew creates the list. The parameters specify that the List Manager is to calculate the cell size, the default list definition function is to be used, automatic drawing mode is to be enabled, no room is to be left for a size box, the list is not to have a horizontal scroll bar, and the list is to have a vertical scroll bar.

The call to doAddRowsAndDataToTextList adds rows to the list and stores data in its cells.

The next two lines set the cell at the topmost row as the initially-selected cell. doResetTypeSelection calls a function which initialises certain variables used by the type selection function. The last line returns the handle to the list.

doAddRowsAndDataToTextList

doAddRowsAndDataToTextList adds rows to the text list and stores data in its cells. The data is retrieved from a 'STR#' resource.

The for loop copies the strings from the specified 'STR#' resource and passes each string as a parameter in a call to doAddTextItemsAlphabetically, which inserts a new row into the list and copies the string to that cell.

Note at this point that the strings in the 'STR#' resource are not arranged alphabetically.

doAddTextItemAlphabetically

doAddTextItemAlphabetically does the heavy work in the process of adding the rows to the text list and storing the text. The bulk of the code is concerned with building the list in such a way that the cells are arranged in alphabetical order.

The first line sets the variable found to false. The next line sets the variable totalRows to the number of rows in the list. (In this program, this is initially 0.) The next line sets the variable currentRow to -1.

The while loop executes until the variable found is set to true.

Within the loop, the first line increments currentRow to 0. The first time this function is called, currentRow will equal totalRows at this point and the loop will thus immediately exit to the first line below the loop. The call to LAddRow at this line adds one row to the list, inserting it before the row specified by currentRow. The list now has one row (cell (0,0)). LSetCell copies the string to this cell. The function then exits, to be re-called by doAddRowsAndDataToTextList for as many times as there are remaining strings.

The second time the function is called, the first line in the while loop again sets currentRow to 0. This time, however, the if block does not execute because totalRows is now 1. Thus SetPt sets the variable aCell to (0,0) and LGetCellDataLocation retrieves the offset and length of the data in cell (0,0). This allows the string in this cell to be alphabetically compared with the "incoming" string using CompareText. If the incoming string is "less than" the string in cell (0,0), CompareText returns -1, in which case:

- The loop exits. LaddRow inserts one row before cell(0,0) and the old cell (0,0) thus becomes cell(0,1). The list now contains two rows.

- SetPt sets cell (0,0) and LSetCell copies the "incoming" string to that cell. The "incoming" string, which was alphabetically "less than" the first string, is thus assigned to the correct cell in the alphabetical sense.

- The function then exits, to be re-called for as many times as there are remaining strings.

If, on the other hand, CompareText returns 0 (strings equal) or 1 ("incoming" string "greater than" the string in cell (0,0), the loop repeats. At the first line in the loop, currentRow is incremented to 1, which is equal to totalRows. Accordingly, the loop exits immediately, LAddRow inserts a row before cell (0,1) (that is, cell (0,1) is created), LSetCell copies the "incoming" string to that cell, and the function exits, to be re-called for as many times as there are remaining strings.

The ultimate result of all this is an alphabetically ordered list.

doCreateIconList

doCreateIconList, supported by the following function, creates the icon list. This list uses a custom list definition function; accordingly, CreateCustomList, rather than LNew, is used to create the list.

SetRect sets the rectangle which will be passed as the dataBounds parameter of the CreateCustomList call to specify one column and (initially) no rows. SetPt sets the variable which will be passed as the cellSize parameter so as to specify that the List Manager should make the cell size of all cells 52 by 52 pixels. The next line adjusts the list rectangle to reflect the area occupied by the vertical scroll bar. The line after that assigns the universal procedure pointer to the custom list definition function to the userProc field of the variable of type ListDefSpec that will be passed in the theSpec parameter of the CreateCustomList call.

The call to CreateCustomList creates the list. The parameters specify that the List Manager is to make all cell sizes 52 by 52 pixels, a custom list definition function is to be used, automatic drawing mode is to be enabled, no room is to be left for a size box, the list is not to have a horizontal scroll bar, and the list is to have a vertical scroll bar.

The next line assigns lOnlyOne to the selFlags field of the list structure, meaning that the List Manager's cell selection algorithm is modified so as to allow only one cell to be selected at any one time.

The call to doAddRowsAndDataToIconList adds rows to the list and stores data in its cells.

The next two lines select the cell at the topmost row as the initially-selected cell. The last line returns the handle to the list.

doAddRowsAndDataToIconList

doAddRowsAndDataToIconList adds ten rows to the icon list and stores a handle to an icon suite in each of the eight cells.

The first two lines set the variable rowNumber to the current number of rows, which is 0.

The for loop executes ten times. Each time through the loop, the following occurs:

- GetIconSuite creates a new icon suite and fills it with icons with the specified resource ID and of the types specified in the last parameter (that is, large icons only).

- LAddRow inserts a new row in the list at the location specified by the variable rowNumber. SetPt sets this cell and LSetCell stores the handle to the icon suite as the cell's data. The last line increments the variable rowNumber, which is passed in the SetPt call.

doHandleArrowKey

doHandleArrowKey further processes Down Arrow and Up Arrow key presses. This is the first of eleven functions dedicated to the handling of key-down events.

Recall that doHandleArrowKey's third parameter (allowExtendSelect) is set to true by the calling function (doKeyDown) only if the text list is the currently active list.

The first line sets the variable moveToTopBottom to false, which can be regarded as the default. At the next two lines, if the Command key was also down at the time of the Arrow key press, this variable is set to true.

If the text list is the currently active list and the Shift key was down, doArrowKeyExtendSelection is called; otherwise, doArrowKeyMoveSelection is called.

doArrowKeyMoveSelection

doArrowKeyMoveSelection further processes those Arrow key presses which occurred when either list was the currently active list but the Shift key was not down. The effect of this function is to deselect all currently selected cells and to select the appropriate cell according to, firstly, which Arrow key was pressed (Up or Down) and, secondly, whether the Command key was down at the same time.

The if statement calls doFindSelectedCell, which searches for the first selected cell in the specified list. That function returns true if a selected cell is found, or false if the list contains no selected cells.

If true is returned by that call, the variable currentSelection will hold the first selected cell. However, this could be changed by the second line within the if block if the key pressed was the Down-Arrow. doFindLastSelectedCell finds the last selected cell (which could, of course, well be the same cell as the first selected cell if only one cell is currently selected). Either way, the variable currentSelection will now hold either the only cell currently selected, the first cell selected (if more than one cell is currently selected and the key pressed was the Up Arrow), or the last cell selected (if more than one cell is currently selected and the key pressed was the Down Arrow).

With that established, doFindNewCellLoc determines the next cell to select, which will depend on, amongst other things, whether the Command key was down at the time of the key press (that is, on whether the moveToTopBottom parameter is true or false). The variable newSelection will contain the results of that determination.

doSelectOneCell then deselects all currently selected cells and selects the cell specified by the variable newSelection.

It is possible that the newly-selected cell will be outside the list's display rectangle. Accordingly, doMakeCellVisible, if necessary, scrolls the list until the newly-selected cell appears at the top or the bottom of the display rectangle.

doArrowKeyExtendSelection

doArrowKeyExtendSelection is similar to the previous function except that it adds additional cells to the currently selected cells. This function is called only when the text list is the currently active list and the Shift key was down at the time of the Arrow key press.

By the fifth line, the variable currentSelection will hold either the only cell currently selected, the first cell selected (if more than one cell is currently selected and the key pressed was the Up Arrow), or the last cell selected (if more than one cell is currently selected and the key pressed was the Down Arrow).

doFindNewCellLoc determines the next cell to select, which will depend on, amongst other things, whether the Command key was down at the time of the key press (that is, on whether the moveToTopBottom parameter is true or false). The variable newSelection will contain the results of that determination. The similarities between this function and doArrowKeyMoveSelection end there.

At the next line, LGetSelect is called to check whether the cell specified by the variable newSelection is selected. If it is not, LSetSelect selects it. (This check by LGetSelect is advisable because, for example, the first-selected cell as this function is entered might be cell (0,0), that is, the very top row. If the Up-Arrow was pressed in this circumstance, and as will be seen, doFindNewCellLoc returns cell (0,0) in the newSelection variable. There is no point in selecting a cell which is already selected.)

It is possible that the newly-selected cell will be outside the list's display rectangle. Accordingly, doMakeCellVisible, if necessary, scrolls the list until the newly-selected cell appears at the top or the bottom of the display rectangle.

doTypeSelectSearch

doTypeSelectSearch is the main type selection function. It is called from doKeyDown whenever the key pressed is not the Tab key, the Up Arrow key or the Down Arrow key.

The global variables gTSStruct, gTSResetThreshold, and gTSLastListHit are central to the operation of doTypeSelectSearch. gTSStruct holds the current type selection search string entered by the user and the time in ticks of the last key press. gTSResetThreshold holds the number of ticks which must elapse before type selection resets, and is dependent on the value the user sets in the "Delay Until Repeat" setting in the Keyboard control panel (Mac OS 8/9) or System Preferences/Keyboard (Mac OS X). gTSLastListHit holds a handle to the last list that type selection affected.

The first block will cause doResetTypeSelection, which resets type selection, to be called if either of the following situations prevail: if the list which is the target of the current key press is not the same as the list which was the target of the previous key press; if a number of ticks since the last key press is greater than the number stored in gTSResetThreshold; if the current length of the type selection string is 63 characters.

The next line stores the handle to the list which is the target of the current key press in gTSLastListHit so as to facilitate the comparison at the first if block the next time the function is called. The next line stores the time of the current key press in gTSLastKeyTime for the same purpose.

The call to TypeSelectNewKey extracts the character code from the message field of the event structure and adds the character to the tsrKeyStrokes field of gTSStruct. That field now holds all the characters received since the last type selection reset.

SetPt sets the variable theCell to represent the first cell in the list. This is passed as a parameter in the LSearch call, and specifies the first cell to examine. LSearch examines this cell and all subsequent cells in an attempt to find a match to the type selection string. If a match exists, the cell in which the first match is found will be returned in theCell parameter, LSearch will return true and the following three lines will execute.

Of those three lines, ordinarily only the call to LSetSelect (which deselects all currently selected cells and selects the specified cell) and the last line (which, if necessary, scrolls the list so that the newly-selected cell is visible in the display rectangle) would be necessary. However, because the function doSelectOneCell has no effect unless there is currently at least one selected cell in the list, the call to doSelectOneCell is included to account for the situation where the user may have deselected all of the text list cells using Command-clicking or dragging.

The actual matching task is performed by the match (callback) function the universal procedure pointer to which is passed in the third parameter to the LSearch call. Note that the default match function has been replaced by the custom callback function doSearchPartialMatch.

doSearchPartialMatch

doSearchPartialMatch is the custom match function called by LSearch, in the previous function, to attempt to find a match to the current type selection string. For the default function to return a match, the type selection string would have to match an entire cell's text. doSearchPartialMatch, however, only compares the characters of the type selection string with the same number of characters in the cell's text. For example, if the type selection string is currently "fr" and a cell with the text "Fractal Painter" exists, doSearchPartialMatch will report a match.

A comparison by IdenticalText (which returns 0 if the strings being compared are equal) is only made if the cell contains data and the length of that data is greater than or equal to the current length of the type selection string. If these conditions do not prevail, doSearchPartialMatch returns 1 (no match found). If these conditions do prevail, IdenticalText is called with, importantly, both the third and fourth parameters set to the current length of the type selection string. IdenticalText will return 0 if the strings match or 1 if they do not match.

doFindFirstSelectedCell

doFindFirstSelectedCell and the following four functions are general utility functions called by the previous Arrow key handling and type selection functions. doFindFirstSelectedCell searches for the first selected cell in a list, returning true if a selected cell is found and providing the cell's coordinates to the calling function.

SetPt sets the starting cell for the LGetSelect call. Since the first parameter in the LGetSelect call is set to true, LGetSelect will continue to search the list until a selected cell is found or until all cells have been examined.

doFindFirstSelectedCell returns true when and if a selected cell is found.

doFindLastSelectedCell

doFindLastSelectedCell finds the last selected cell in a list (which could, of course, also be the first selected cell if only one cell is selected).

If the call to doFindFirstSelectedCell reveals that no cells are currently selected, doFindlastSelectedCell simply returns. If, however, doFindFirstSelectedCell finds a selected cell, that cell is passed as the starting cell in the LGetSelect call.

As an example of how the rest of this function works, assume that the first selected cell is (0,1), and that cell (0,4) is the only other selected cell. LGetSelect examines this cell and returns true, causing the loop to execute. The first line in the while loop thus assigns (0,1) to theCell and the next line increments aCell to (0,2). LGetSelect starts another search using (0,2) as the starting cell. Because cells (0,2) and (0,3) are not selected, LGetSelect advances to cell (0,4) before it returns. Since it has found another selected cell, LGetSelect again returns true, so the loop executes again. aCell now contains (0,4), and the first line in the while loop assigns that to theCell. Once again, LNextCell increments aCell, this time to (0,5).

This time, however, LGetSelect will return false because neither cell (0,5) nor any cell below it is selected. The loop thus terminates, theCell containing (0,4), which is the last selected cell.

doFindNewCellLoc

doFindNewCellLoc finds the new cell to be selected in response to Arrow key presses. That cell will be either one up or one down from the cell specified in the oldCellLoc parameter (if the Command key was not down at the time of the Arrow key press) or the top or bottom cell (if the Command key was down).

The first line gets the number of rows in the list. (Recall that the List Manager sets the dataBounds.bottom coordinate to one more than the vertical coordinate of the last cell.)

If the Command key was down (moveToTopBottom is true) and the key pressed was the Up Arrow, the new cell to be selected is the top cell in the list. If the key pressed was the Down Arrow key, the new cell to be selected is the bottom cell in the list.

If the Command key was not down and the key pressed was the Up Arrow key, and if the first selected cell is the top cell in the list, the new cell to be selected remains as set at the second line in the function; otherwise, the new cell to be selected is set as the cell above the first selected cell. If the key pressed was the Down Arrow key, and if the last selected cell is the bottom cell in the list, the new cell to be selected remains as set at the second line in the function; otherwise, the new cell to be selected is set as the cell below the last selected cell.

doSelectOneCell

doSelectOneCell deselects all cells in the specified list and selects the specified cell.

If no cells in the list are selected, the function returns immediately. Otherwise, the first selected cell is passed as the starting cell in the call to LGetSelect.

The while loop will continue to execute while a selected cell exists between the starting cell specified in the LGetSelect call and the end of the list. Within the loop, if the current LGetSelect starting cell is not the cell specified for selection, that cell is deselected. When the loop exits, LSetSelect selects the cell specified for selection.

Note that defeating the de-selection of the cell specified for selection if it is already selected (the if statement within the while loop) prevents the unsightly flickering which would occur as a result of that cell being deselected inside the loop and then selected again after the loop exits.

doMakeCellVisible

doMakeCellVisible checks whether a specified cell is within the list's display rectangle and, if not, scrolls the list until that cell is visible.

The first line gets a copy of the rectangle that encompasses the currently visible cells. (Note that this rectangle is in cell coordinates.) The if statement tests whether the specified cell is within this rectangle. If it is not, the list is scrolled as follows:

- If the specified cell is "below" the bottom of the display rectangle, the variable dRows is set to the difference between the cell's v coordinate and the value in the bottom field of the display rectangle, plus 1. (Recall that the List Manager sets the bottom field to one greater than the v coordinate of the last visible cell.)

- If the specified cell is "above" the top of the display rectangle, the variable dRows is set to the difference between the cell's v coordinate and the value in the top field of the display rectangle.

With the number of cells to scroll, and the direction to scroll, established, LScroll is called to effect the scroll.

doResetTypeSelection

doResetTypeSelection resets the global variables which are central to the operation of the type selection function doTypeSelectSearch.

The first line resets the tsrKeyStrokes and tsrLastKeyTime fields of gTSStruct to NULL and 0 respectively. The next line sets the variable which holds the handle to the list which is the target of the current key press to NULL. The next line sets the variable which holds the type selection reset threshold to twice the value stored in the low memory global variable KeyThresh. However, if this value is greater than the value represented by the constant kMaxKeyThresh, the variable is made equal to kMaxKeyThresh.

doRotateCurrentList

doRotateCurrentList rotates the currently active list in response to the Tab key and to mouse-downs in the non-active list.

The first line saves the handle to the currently active list. The next line retrieves the handle to the new list to be activated from the refCon field of the currently active list's list structure. The third line makes the new list the currently active list.

The last two lines cause the keyboard focus frame to be erased from the previously current list, the list box frame to be drawn around the previously current list, and the keyboard focus frame to be drawn around the new current list.

doDrawFrameAndFocus

doDrawFrameAndFocus is called by doDrawContent, doActivateDeactivate, and doRotateCurrentList to draw or erase the keyboard focus frame from the specified list, and to draw the list box frame in either the activated or deactivated state.

The second and third lines get the list's rectangle from the rView field of the list structure and expand it to the right by the width of the scroll bar.

The machinations at the next four lines are for cosmetic purposes only. If the program is running on Mac OS 8/9, the current background colour and pattern will be white, so the saved background colour/pattern must be restored before the first call to DrawThemeFocusRect, which erases the keyboard focus frame to the background colour/pattern.

Depending on the value received in the inState formal parameter, the list box frame is drawn in either the activated or deactivated state. If the specified list is the current list, DrawThemeFocusRect is called again, this time to draw the keyboard focus frame.

If the program is running on Mac OS 8/9, the last two lines reset the backgound colour/pattern to white.

doExtractSelections

doExtractSelections is called when the user clicks the Extract push button or double clicks an item in a list.

The first block gets the handles to the lists. The next two lines initialise the Str255 array that will be used to hold the extracted strings.

The next block copies the data from the selected cells in the text list to the Str255 array. The for loop is traversed once for each cell in the list. SetPt increments the v coordinate of the variable theCell. If the specified cell is selected (LGetSelect), LGetCellDataLocation is called to get the length of the data in the cell, and LGetCell is called to copy the cell's data into an element of the Str255 array.

The next block gets the selected cell in the icon list, retrieves the related string from the specified STR# resource, and assigns it to the 15th element of the Str255 array. SetPt sets the starting cell for the LGetSelect search.

The last two lines force a kEventWindowDrawContent event, which will cause the function doDrawSelections to draw the contents of the Str255 array in the group box at the bottom of the window.

doDrawSelections

doDrawSelections is called by doDrawContent and doActivateDeactivate to draw the contents of the Str255 array "filled in" by the function doExtractSelections.

listDefFunction

listDefFunction the custom list definition (callback) function used by the window's icon list.

The List Manager sends a list definition function four types of messages in the message parameter. Only two of these are relevant to this list definition function. listDefFunction calls the appropriate function to handle each message type.

doLDEFDraw

doLDEFDraw handles the lDrawMsg message, which relates to a specific cell.

The first two lines save the current graphics port and set the graphics port to the port in which the list is drawn.

EraseRect erases the cell rectangle. The next line gets a copy of the 52 pixel by 52 pixel cell rectangle. The next four lines adjust this rectangle to the size of a 32 by 32 pixel icon.

The if statement checks whether the cell's data is 4 bytes long (the size of a handle). If it is, LGetCell is called to get the cells's data into the variable iconSuiteHdl and PlotIconSuite is called to

draw the icon within the specified rectangle. If the list is active, kTransformNone is passed in the transform parameter, otherwise kTransformDisabled is passed. This latter causes the icon to be drawn in the disabled (dimmed) state.

GetIndString is then called to get the string corresponding to the icon. The rectangle used to draw the icon is adjusted and passed, together with the string, in a call to TETextBox. TETextBox draws the string, with centre justification, underneath the icon.

If the lDrawMsg message indicated that the cell was selected, the cell highlighting function is called. The previously saved graphics port is then restored

doLDEFHighlight

doLDEFHighlight handles the lHiliteMsg message and may also be called from doLDEFDraw.

A copy of the value in the low memory global HiliteMode is acquired, BitClr is called to clear the highlight bit, and HiliteMode is set to this new value. The last line highlights the cell.

DialogList.c

doListsDialog contains the main functions pertaining to the lists in the movable modal dialog.

doListsDialog

doListsDialog creates a movable modal dialog using 'DLOG', 'dlgx', 'dftb', and 'DITL' resources. The 'DITL' resource contains, amongst other items, two list controls. Each list control is supported by an 'ldes' resource. Both 'ldes' resources specify no rows, one column, a cell height of 14 pixels, a vertical scroll bar, and the system LDEF. The 'dftb' resource specifies the small system font for the list controls.

At the first block, the window, if open, is explicitly deactivated. The dialog is then created. At the next block, the Dialog Manager is told which items are the default and Cancel items.

A custom event filter function is used. The call to NewModalFilterProc creates the associated universal procedure pointer.

At the next block, and for each list control, the handle to the list control is obtained, the handle to the associated list structure is obtained, the function doAddRowsAndDataToTextList is called to add the specified number of rows and the data to the list's cells, the cell-selection algorithm is customised to allow the selection of one cell only, and the first cell is selected.

ShowWindow is then called to display the dialog. The call to SetKeyboardFocus sets the keyboard focus to the "Date Format" list.

Within the do-while loop, ModalDialog retains control until an enabled item is hit. If the push buttons are hit, or if the last click in one of the list boxes was a double-click, the loop exits.

If the item hit is the "Date Format" list, SetPt sets the variable theCell to represent the first cell in the list. This is passed as a parameter in the LGetSelect call, which searches the list until it finds a cell that is selected. LGetDataLocation is called to get the length of the data in that cell and LGetCell is called to copy the data (a string) to a local Str255 variable. At the next block, a reference to the static text control associated with this list is obtained and its text is set with the string obtained by LGetCell. Draw1Control is then called to draw the static text field control with this newly-set text.

The last action is to check whether the last click in the list box was a double-click. If the last click was a double-click, the variable wasDoubleClick is set to true, causing the loop to exit.

The same general procedure is followed in the event of a hit on the "Watermark" list.

When the OK push button is hit, or one of the lists has been double-clicked, the dialog and the universal procedure pointer are disposed of.

eventFilter

A custom event filter function is used to intercept keyDown events so as to support type selection in the "Watermark" list.

If the event is a keyDown event, the character code is extracted from the event structure's message field.

If the key hit was not the Up-Arrow, Down-Arrow, or tab key, the following occurs. GetDialogItemAsControl is called to get the reference to the "Watermark" list control, GetControlData is called to get the handle to the associated list, and GetKeyboardFocus is called to get the reference to the control with keyboard

focus. If the "Watermark" list control currently has the focus, the function doTypeSelectSearch is called (to handle type selection) and Draw1Control is called on the list control to ensure that the type-selected item is highlighted. handledEvent is then set to true to inform ModalDialog that the filter function handled the event.

Apart from supporting type-selection in the "Watermark" list, this arrangement means that the only keyDown events received by ModalDialog in respect of the "Date Format" list will be Up-Arrow, Down-Arrow, and tab key events.