20

LISTS AND CUSTOM LIST DEFINITION FUNCTIONS

Includes 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 Finder's 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 Apple menu), or which display more than one type of information in each item (as does the list in the About This Computer... window).

List Manager Limitations

Although the List Manager can handle small, simple lists effectively, it is not suitable for displaying large amounts of data such as, for example, that used by a spreadsheet application. The List Manager cannot maintain lists whose data occupies more than 32 KB of memory.

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 application-defined 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.

> ***Historical Note***
>
> The list box control was introduced with Mac OS 8 and the Appearance Manager.

Appearance and Features of Lists

Fig 1 shows a dialog box 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.

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.

FIG 1 - DIALOG BOX WITH TWO LISTS

Cells, Cell Font, and Cell Highlighting

Cells

A list is a series of items displayed within a rectangle. Each item is contained within an invisible rectangular **cell**. All cells within a list are of the same size, but cells may contain different types of data.

Cell Font

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

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

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 effort to make it fit. If the string is still too long, 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 a vertical scroll bar (see Fig 1), a horizontal scroll bar, or both. By using scroll bars, you can include more items in a list than can fit within the list's display rectangle, and the user can then scroll the list to view multiple items. If a list includes a scroll bar but the number of cells is such that they are all visible, the List Manager automatically 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 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 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

To add or remove a range of cells from the current selection, the user can press the Command key and then drag the cursor over the other cells. The List Manager determines whether to add or remove selections in a range of cells by checking the status of the first cell clicked in. If that cell is initially selected, then Command-dragging deselects all cells in the range over which the cursor passes. If, on the other hand, that cell is initially not selected, Command-dragging selects all cells in the range over which the cursor passes.

Once the user changes a cell's selection status by Command-dragging over a cell, the selection status of the cell stays the same for the duration of the drag even if the user

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

moves the cursor back over that cell. The effect of the Command key thus differs from that of the Shift key in this respect.

Shift-Clicking — Discontiguous Cells Selected

If the user Shift-clicks a cell after having created discontiguous selection ranges, the discontiguity is lost. The List Manager selects all cells in the range of the first selected cell (that is, 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 last selected cell (that is, 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 selection of cells using the keyboard in two ways:

- ***Cell Selection Using Arrow Keys.*** Your application should support the use of the Arrow keys to move and extend cell selections.

- ***Type Selection.*** If your application uses text-only lists (or lists whose items can be identified by text strings), your application should allow the user to select an item by simply typing the text associated with that item. This method of cell selection is known as **type selection**.

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

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 user is attempting to move the selection 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 the Command key is down, 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 of the Up Arrow key should select the first cell in the list and

deselect all other cells. Once again, your application should 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 different 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

In a text-only list, when the user types the text of an item in a list, your application should respond by scrolling to the cell containing that text and selecting 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 or 120 ticks, whichever is the greater.[3]

Implementing Type Selection

To implement type selection, your application must keep a record of the characters the user has typed, the time when the user last typed a character, the amount of time which must elapse since that last character was typed before the type selection string is reset, and which list the last typed character affected. The following shows the variables you might use for this purpose, together with their usage:

Variable Name	Type	Usage
gTSString	Str255	Stores the string which represents current status of the type selection.
gTSThresh	short	Stores the number of ticks after which type selection resets. For example, if the user types "abcde'" but waits for more than gTSThresh, before typing "f", the application should set gTSString to "f", not "abcdef".
gTSElapse	long	Stores the time in ticks of the last key-down.
gTSLastListHit	ListHandle	Stores the list affected by the last typed character.

[3] The value in KeyThresh is set by the user at the "Delay Until Repeat" section of the Keyboard control panel. Call LMGetKeyThresh to obtain this value.

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. In most cases, your application can get or set information in a list structure using List Manager functions.

Before describing the list structure, 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

Each cell in a list can be described by a data structure of type Cell, 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.

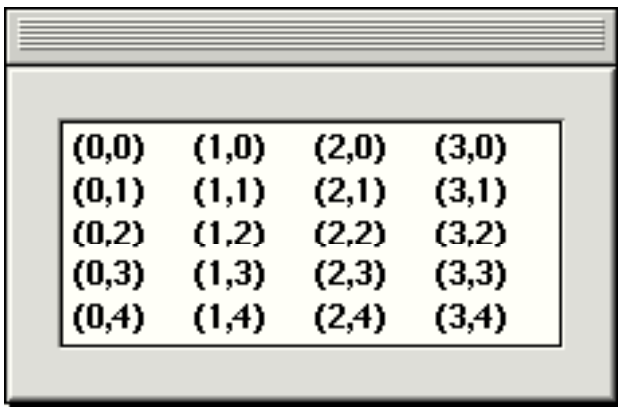

FIG 2 - COORDINATES OF CELLS

The ListRec Data Type

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

rView — Specifies the list's display rectangle in the local coordinates of the graphics port specified by the port field (see below). Note that the display rectangle does not include the area occupied by a list's scroll bars.

port — The graphics port of the window containing the list.

indent — Indicates 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.

cellSize — Specifies 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 determines the vertical size of a cell by adding the ascent, descent and leading of the port's font (which works out as 16 pixels for 12-point Charcoal for example). You should make the height of your list equal to a multiple of this height. The default horizontal size of a cell is determined by dividing the width of the list's display rectangle by the number of columns in the list.

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

The List Manager sets the 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. Recall from Chapter 12 — Drawing With QuickDraw that the mathematical borders of a rectangle are infinitely thin and that the displayed rectangle of pixels "hangs" down and to the right of the mathematical rectangle. When PtInRect's parameters are expressed as cell coordinates, it is the *cells* which "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 5 or higher or column 2 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).

vScroll — A handle to the vertical scroll bar, or NULL if the list does not have a vertical scroll bar.

hScroll — A handle to the horizontal scroll bar, or NULL if the list does not have a horizontal scroll bar.

selFlags — Specifies the algorithm the List Manager uses to select cells in response to a click in the list.

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.

listFlags — Indicates whether automatic vertical and horizontal scrolling is enabled. If automatic scrolling is enabled, then a list scrolls when the user clicks a cell and then drags the cursor out of the rectangle specified by the rView field. By default, the List Manager enables automatic scrolling if the list has the associated scroll bar (horizontal or vertical). The following constants define bits in this field which determine whether horizontal or vertical autoscrolling are enabled:

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

clikTime — Indicates the time when the user last clicked the mouse.

clikLoc — Indicates the local coordinates of the last mouse click.

mouseLoc — Indicates the current location of the cursor in local coordinates. Ordinarily you would use the Event Manager's GetMouse function to obtain this information, but this field may be more convenient to access from within a click-loop function (see below).

lClickLoop — Contains a universal procedure pointer to a click-loop function continually called by LClick, or NULL if the default click loop function is to be used. Your application may place a universal procedure pointer to a custom click-loop function in this field.

It is unlikely that your application will need to define its own click-loop function because the List Manager's default click-loop function uses a rather robust algorithm to respond to mouse clicks. Your application needs a custom function only if it needs to perform some special processing while the user drags the cursor after clicking in a list.

lastClick — Indicates the cell coordinates of the last click. You can access the value in this field using LLastClick. If your application depends on the accuracy of the information in this field and the clikTime and clikLoc fields, and if your application treats keyboard selection of list items identically to mouse selection of list items, then it should update the values of these fields after highlighting a cell in response to a keyboard event.

refCon — For your application's use.

listDefProc — Contains a handle to the code used by the list definition function.

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

dataBounds — Specifies the total cell dimensions of the list, including cells which are not visible. 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 display rectangle. For example, if a list contains 4 columns and 10 rows (that is, the last cell in the list has coordinates (3,9)), the List Manager sets the dataBounds field to (0,0,4,10).

cells — Contains a handle to a relocatable block used to store cell data. The handle is defined like this:

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

Because of the way the cells field is defined, therefore, no list can contain more than 32,000 bytes of data.

cellArray — Used to store offsets to data in the relocatable block specified by the cells field. Your application should not change the cells field directly or access the information in the cellArray field directly. The List Manager provides functions for manipulating the information in the list.

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

Creating a List

LNew

You create a list using LNew:

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

rView — The rectangle in which to display the list, in local coordinates. (Does not include the area taken up by the list's scroll bars.)

dataBounds — The initial data bounds for the list. Set the left and top fields to (0,0) and the right and bottom fields to kInitialColumns and kInitialRows, to create a list with kInitialColumns columns and kInitialRows rows.

cSize — The size of each cell in the list. If your application specifies (0,0) and is using the default list definition function, the List Manager computes the size automatically, setting the v field to the sum of the ascent, descent, and leading of the current font and the h field using the following formula:

```
cSize.h = (rView.right - rView.left) / (dataBounds.right - dataBounds.left)
```

theProc — The resource ID of the list definition function to use for the list. To use the default list definition function, specify 0.

theWindow — Pointer to the window in which to install the list.

drawIt — Indicates whether automatic drawing mode is initially enabled. When automatic redrawing is enabled (by setting this parameter to true), the list is automatically redrawn whenever a change is made to it.

You can later change this setting 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 — Indicates whether space should be left for a size box.

scrollHoriz — Specify true if your list requires a horizontal scroll bar, otherwise specify false.

scrollVert — Specify true if your list requires a vertical scroll bar, otherwise specify false.

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.[4] 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 box 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.

InvalRect 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 box, 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 application-defined function for changing the current list.

[4] 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 box containing both a list and an editable text item).

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 application-defined 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 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 from the colour graphics port's visRgn field.

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 a window containing a list is activated or deactivated, 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 application-defined 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 box 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

Your application can use the refCon field of each list structure to create the linked ring. The refCon field of the first list is assigned the handle to the second list, the refCon field of the second list is assigned the handle to the third list, and so on, until the refCon field of the last list is assigned the handle to the first list. Then, in response to a Tab key press in the current list, your application can determine the next list in the sequence by looking at the current list's refCon field.

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

```
ListHandle    gCurrentListHdl;

void  doFindPreviousListInRing(void)
{
   ListHandle    listHdl;

   listHdl = gCurrentListHdl;

   while((ListHandle) (*listHdl)->refCon != gCurrentList)
      listHdl = (ListHandle) (*listHdl)->refCon;
```

```
    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 the 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:

```
(*listHdl)->selFlags = 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 application-defined functions for storing data in the list's cells and, where required, for adding rows and/or columns.

- Provide an application-defined function to support type selection, if required.
- Modify the list's cell selection algorithm, if required.
- Provide an application-defined 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.
kControlListBoxLDEFTag	Sets the 'LDEF' resource to be used to draw a list box's contents. This is useful for creating a list box without an 'ldes' resource. **Data type set:** SInt16
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

You can use a list box description resource to specify information in a list box. A list box description resource is a resource of type 'ldes'. All list box description resources must have resource ID of greater than 127. The Control Manager uses the information you specify to provide additional information to the corresponding list box control. Fig 6 shows the structure of a compiled 'ldes' resource.

	BYTES
VERSION NUMBER	2
NUMBER OF ROWS	2
NUMBER OF COLUMNS	2
CELL HEIGHT	2
CELL WIDTH	2
HAS VERTICAL SCROLL	1
RESERVED	1
HAS HORIZONTAL SCROLL	1
RESERVED	1
LIST DEFINITION RESOURCE ID	2
HAS SIZE BOX	1
RESERVED	1

FIG 3 - STRUCTURE OF A COMPILED LIST BOX DESCRIPTIC

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

Field	Description
NUMBER OF ROWS	An integer specifying the number of rows in the list box.
NUMBER OF COLUMNS	An integer specifying the number of columns in the list box.
CELL HEIGHT	An integer specifying the height of a list item. If 0 is specified, the list item height is automatically calculated.
CELL WIDTH	An integer specifying the width of a list item. If 0 is specified, the list item width is automatically calculated.
HAS VERTICAL SCROLL BAR	A Boolean value that indicates whether the list box should contain a vertical scroll bar. If true, the list box contains a vertical scroll bar. If false, no vertical scroll bar.
HAS HORIZONTAL SCROLL BAR	A Boolean value that indicates whether the list should contain a horizontal scroll bar. Specify true if your list requires a horizontal scroll bar. Specify false otherwise.
RESOURCE ID	The resource ID of the list definition function to use for the list. To use the default list definition function, which supports the display of unstyled text, specify a resource ID of 0.
HAS SIZE BOX	A Boolean value that indicates whether the List Manager should leave room for a size box. If true, a size box will be drawn. If false, a size box will not be drawn.

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[5], you must create your own list definition function. After

5 For example, the Finder's About This Computer... dialog box 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.

writing a list definition function, you must compile it as a resource of type 'LDEF' and store it in the resource fork of the application that uses the function.

Your custom list definition function must be defined like this:

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

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 pass information to your list definition function only when the value in the lMessage parameter contains either the lDrawMsg or lHiliteMsg constants. These parameters provide information about the cell affected by the message. The selected parameter indicates whether the cell should be highlighted. The lRect and lCell parameters indicate the cell's rectangle and coordinates. The lDataOffset and lDataLen parameters specify the offset and length of the cell's data within the relocatable block referenced by the cells field of the list structure.

Responding to the Initialisation Message

The List Manager automatically allocates memory for a list and fills out the fields of a list structure before calling your list definition function with an lInitMsg message. Your application might respond to the initialisation message by changing, say, the cellSize and indent fields of the list structure. However, many list definition functions do not need to perform any action in response to the lInitMsg message.

Responding to the Draw Message

The list definition function must respond to the draw message by examining the specified cell's data and drawing the cell as appropriate, ensuring that the characteristics of the drawing environment are not altered.

Responding to the HighLighting Message

Virtually every list definition function should respond to the lHiliteMsg message in the same way, that is, by highlighting 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 List Manager sends your list definition function the lCloseMsg immediately before disposing of the memory occupied by list. Your list definition function needs to respond only if it needs to perform some special processing before a list is disposed of, such as releasing memory associated with the list that would not be released by LDispose.

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.

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);
void            LDispose(ListHandle lHandle);
```

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);
```

Demonstration Program

```
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
// List.h
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
//
// 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 box both
// contain two lists.
//
// The cells of one list in the window, and of both lists in the dialog box, 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 box 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 box, the
// user's selections are displayed in static text fields embedded in placards below each
// list.
//
// The program utilises the following resources:
//
// •   An 'MBAR' resource, and 'MENU' resources for Apple, File, Edit and Demonstration
//     menus (preload, non-purgeable).
//
// •   A 'WIND' resource (purgeable) (initially not visible).
//
// •   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, doesActivateOnFGSwitch, and
//     is32BitCompatible flags set.
//
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

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

#include <Appearance.h>
#include <ControlDefinitions.h>
#include <Devices.h>
#include <LowMem.h>
#include <Sound.h>
#include <ToolUtils.h>
```

```
//
.....................................................................................................................................
..................................... defines

#define rMenubar                    128
#define mApple                        128
#define  iAbout                     1
#define mFile                         129
#define  iQuit                      11
#define mDemonstration              131
#define  iHandMadeLists             1
#define  iListControlLists        2
#define rListsWindow                128
#define  cExtractButton             128
#define  cGroupBox1                    129
#define  cGroupBox2                    130
#define  cBalloonHelpStaticText   131
#define  cSoftwareStaticText      132
#define  cHardwareStaticText      133
#define  rTextListStrings           128
#define  rIconListIconSuiteBase   128
#define  rIconListStrings           129
#define rListsDialog                128
#define  iDateFormatList            5
#define  iWatermarkList                6
#define  iDateFormatStaticText    8
#define  iWatermarkStaticText     10
#define  rDateFormatStrings          130
#define  rWatermarkStrings        131
#define kUpArrow                       0x1e
#define kDownArrow                     0x1f
#define kTab                           0x09
#define kScrollBarWidth                15
#define kMaxKeyThresh                  120
#define kSystemLDEF                    0
#define kCustomLDEF                    128
#define MAXLONG                           0x7FFFFFFF

//
.....................................................................................................................................
..................................... typedefs

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

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

//
.....................................................................................................................................
..... function prototypes

void                main                                (void);
void                doInitManagers                   (void);
void                doEvents                            (EventRecord *);
void                doAdjustMenus                    (void);
void                doMenuChoice                     (SInt32);
void                doGetDepthAndDevice           (void);
void                doSaveBackground                 (backColourPattern *);
void                doRestoreBackground           (backColourPattern *);
void                doSetBackgroundWhite          (void);
void                doOpenListsWindow                (void);
void                doKeyDown                           (SInt8,EventRecord *);
void                doUpdate                            (EventRecord *);
void                doActivate                          (EventRecord *);
void                doActivateWindow                 (WindowPtr,Boolean);
void                doInContent                         (EventRecord *);
ListHandle          doCreateTextList                 (DialogPtr,Rect,SInt16,SInt16);
void                doAddRowsAndDataToTextList (ListHandle,SInt16,SInt16);
void                doAddTextItemAlphabetically   (ListHandle,Str255);
ListHandle          doCreateIconList                 (DialogPtr,Rect,SInt16,SInt16);
```

```
void                    doAddRowsAndDataToIconList  (ListHandle,SInt16);
void                    doHandleArrowKey                  (SInt8,EventRecord *,Boolean);
void                    doArrowKeyMoveSelection           (ListHandle,SInt8,Boolean);
void                    doArrowKeyExtendSelection     (ListHandle,SInt8,Boolean);
void                    doTypeSelectSearch                (ListHandle,EventRecord *);
pascal 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                  (void);
void                    doListsDialog                         (void);
pascal Boolean eventFilter                                (DialogPtr,EventRecord *,SInt16 *);

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
// Lists.c
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

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

#include "List.h"

//
..........................................................................................................................................................................
................. global variables

ListSearchUPP          gSearchPartialMatchUPP;
SInt16                 gPixelDepth;
Boolean                    gIsColourDevice    = false;
Boolean                    gDone;
Boolean                    gInBackground;
backColourPattern    gBackColourPattern;

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ main

void  main(void)
{
    Handle             menubarHdl;
    MenuHandle     menuHdl;
    EventRecord     eventStructure;

    //
..........................................................................................................................................................................
... initialise managers

    doInitManagers();

    // ..................................................................................................................................................................
create routine descriptor

    gSearchPartialMatchUPP = NewListSearchProc((ProcPtr) searchPartialMatch);

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

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

    menuHdl = GetMenuHandle(mApple);
    if(menuHdl == NULL)
        ExitToShell();
    else
        AppendResMenu(menuHdl,'DRVR');

    // .......................... get pixel depth and whether colour device for certain Appearance functions

    doGetDepthAndDevice();
```

```
  //
..............................................................................................................................................................................
.............. enter eventLoop

  gDone = false;

  while(!gDone)
  {
    if(WaitNextEvent(everyEvent,&eventStructure,MAXLONG,NULL))
      doEvents(&eventStructure);
  }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doInitManagers

void  doInitManagers(void)
{
  MaxApplZone();
  MoreMasters();

  InitGraf(&qd.thePort);
  InitFonts();
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs(NULL);

  InitCursor();
  FlushEvents(everyEvent,0);

  RegisterAppearanceClient();
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doEvents

void  doEvents(EventRecord *eventStrucPtr)
{
  WindowPtr             windowPtr;
  SInt16                partCode;
  docStructureHandle    docStrucHdl;
  SInt8                 charCode;

  switch(eventStrucPtr->what)
  {
    case mouseDown:
      partCode = FindWindow(eventStrucPtr->where,&windowPtr);

      switch(partCode)
      {
        case inMenuBar:
          doAdjustMenus();
          doMenuChoice(MenuSelect(eventStrucPtr->where));
          break;

        case inContent:
          if(windowPtr != FrontWindow())
            SelectWindow(windowPtr);
          else
            doInContent(eventStrucPtr);
          break;

        case inDrag:
          DragWindow(windowPtr,eventStrucPtr->where,&qd.screenBits.bounds);
          break;

        case inGoAway:
          docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);
          LDispose((*docStrucHdl)->textListHdl);
          LDispose((*docStrucHdl)->iconListHdl);
          DisposeHandle((Handle) docStrucHdl);
          DisposeWindow(windowPtr);
          break;
      }
      break;

    case keyDown:
    case autoKey:
      charCode = eventStrucPtr->message & charCodeMask;
      if((eventStrucPtr->modifiers & cmdKey) != 0)
```

```
      {
        doAdjustMenus();
        doMenuChoice(MenuEvent(eventStrucPtr));
      }
      if(FrontWindow())
        doKeyDown(charCode,eventStrucPtr);
      break;

    case updateEvt:
      doUpdate(eventStrucPtr);
      break;

    case activateEvt:
      doActivate(eventStrucPtr);
      break;

    case osEvt:
      switch((eventStrucPtr->message >> 24) & 0x000000FF)
      {
        case suspendResumeMessage:
          if(FrontWindow())
          {
            gInBackground = (eventStrucPtr->message & resumeFlag) == 0;
            doActivateWindow(FrontWindow(),!gInBackground);
          }
          break;
      }
      HiliteMenu(0);
      break;
  }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doAdjustMenus

void  doAdjustMenus(void)
{
  MenuHandle demoMenuHdl;

  demoMenuHdl = GetMenuHandle(mDemonstration);

  if(FrontWindow())
    DisableItem(demoMenuHdl,1);
  else
    EnableItem(demoMenuHdl,1);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doMenuChoice

void  doMenuChoice(SInt32 menuChoice)
{
  SInt16  menuID, menuItem;
  Str255  itemName;
  SInt16  daDriverRefNum;

  menuID = HiWord(menuChoice);
  menuItem = LoWord(menuChoice);

  if(menuID == 0)
    return;

  switch(menuID)
  {
    case mApple:
      if(menuItem == iAbout)
        SysBeep(10);
      else
      {
        GetMenuItemText(GetMenuHandle(mApple),menuItem,itemName);
        daDriverRefNum = OpenDeskAcc(itemName);
      }
      break;

    case mFile:
      if(menuItem == iQuit)
        gDone = true;
      break;

    case mDemonstration:
      if(menuItem == iHandMadeLists)
```

```
            doOpenListsWindow();
        else if(menuItem == iListControlLists)
            doListsDialog();
        break;
    }

    HiliteMenu(0);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doGetDepthAndDevice

void doGetDepthAndDevice(void)
{
    GDHandle deviceHdl;

    deviceHdl = LMGetMainDevice();
    gPixelDepth = (*(*deviceHdl)->gdPMap)->pixelSize;
    if(BitTst(&(*deviceHdl)->gdFlags,gdDevType))
        gIsColourDevice = true;
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doSaveBackground

void  doSaveBackground(backColourPattern *gBackColourPattern)
{
    GrafPtr currentPort;

    GetPort(&currentPort);

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

    if((**((CGrafPtr) currentPort)->bkPixPat).patType != 0)
        gBackColourPattern->backPixelPattern = ((CGrafPtr) currentPort)->bkPixPat;
    else
        gBackColourPattern->backBitPattern =
                                         *(PatPtr) (*(**((CGrafPtr) currentPort)->bkPixPat).patData);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ 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 };

    RGBBackColor(&whiteColour);
    BackPat(&qd.white);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
// WindowList.c
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

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

#include "List.h"

//
.............................................................................................................................................................................
.................. global variables

ListHandle    gCurrentListHdl;
Str255        gTSString;
SInt16        gTSResetThreshold;
SInt32        gTSLastKeyTime;
```

```
ListHandle    gTSLastListHit;
Str255        gStringArray[16];

extern ListSearchUPP         gSearchPartialMatchUPP;
extern SInt16                gPixelDepth;
extern Boolean               gIsColourDevice;
extern backColourPattern gBackColourPattern;

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doOpenListsWindow

void  doOpenListsWindow(void)
{
  WindowPtr                  windowPtr;
  docStructureHandle docStrucHdl;
  ControlHandle            controlHdl;
  Str255                     software = "\pSoftware:";
  Str255                     hardware = "\pHardware:";
  Str255                     balloon  = "\pBalloon help is available";
  SInt16                     fontNum;
  Rect                       textListRect, pictListRect;
  ListHandle                 textListHdl, iconListHdl;

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

  if(!(windowPtr = GetNewCWindow(rListsWindow,NULL,(WindowPtr) -1)))
      ExitToShell();

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

  SetPort(windowPtr);

  SetThemeWindowBackground(windowPtr,kThemeBrushDialogBackgroundActive,true);
  doSaveBackground(&gBackColourPattern);

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

  CreateRootControl(windowPtr,&controlHdl);

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

  if(!(controlHdl = GetNewControl(cGroupBox1,windowPtr)))
    ExitToShell();
  if(!(controlHdl = GetNewControl(cBalloonHelpStaticText,windowPtr)))
    ExitToShell();
  SetControlData(controlHdl,kControlNoPart,kControlStaticTextTextTag,balloon[0],
                       (Ptr) &balloon[1]);

  if(!(controlHdl = GetNewControl(cSoftwareStaticText,windowPtr)))
    ExitToShell();
  SetControlData(controlHdl,kControlNoPart,kControlStaticTextTextTag,software[0],
                       (Ptr) &software[1]);

  if(!(controlHdl = GetNewControl(cHardwareStaticText,windowPtr)))
    ExitToShell();
  SetControlData(controlHdl,kControlNoPart,kControlStaticTextTextTag,hardware[0],
                       (Ptr) &hardware[1]);

  if(!(controlHdl = GetNewControl(cGroupBox2,windowPtr)))
    ExitToShell();

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

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

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

  SetRect(&textListRect,20,32,150,136);
  SetRect(&pictListRect,171,32,238,136);

  textListHdl = doCreateTextList(windowPtr,textListRect,1,kSystemLDEF);
  iconListHdl = doCreateIconList(windowPtr,pictListRect,1,kCustomLDEF);
```

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

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

  (*textListHdl)->refCon = (SInt32) iconListHdl;
  (*iconListHdl)->refCon = (SInt32) textListHdl;

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

  gCurrentListHdl = textListHdl;

  //
..........................................................................................................................................................................
.......................... show window

  ShowWindow(windowPtr);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ 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);
  }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doUpdate

void  doUpdate(EventRecord *eventStrucPtr)
{
  WindowPtr           windowPtr;
  Rect                theRect;
  docStructureHandle  docStrucHdl;
  ListHandle          textListHdl, iconListHdl;

  windowPtr = (WindowPtr) eventStrucPtr->message;
  SetPort(windowPtr);

  BeginUpdate(windowPtr);

  doRestoreBackground(&gBackColourPattern);
  SetRect(&theRect,20,188,237,381);
  EraseRect(&theRect);
  UpdateControls(windowPtr,windowPtr->visRgn);
  doSetBackgroundWhite();

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

  LUpdate(windowPtr->visRgn,textListHdl);
  LUpdate(windowPtr->visRgn,iconListHdl);

  doDrawFrameAndFocus(textListHdl,((WindowPeek) windowPtr)->hilited);
  doDrawFrameAndFocus(iconListHdl,((WindowPeek) windowPtr)->hilited);
```

```
  doDrawSelections();

  EndUpdate(windowPtr);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doActivate

void  doActivate(EventRecord *eventStrucPtr)
{
  WindowPtr  windowPtr;
  Boolean    becomingActive;

  windowPtr = (WindowPtr) eventStrucPtr->message;
  becomingActive = ((eventStrucPtr->modifiers & activeFlag) == activeFlag);
  doActivateWindow(windowPtr,becomingActive);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doActivateWindow

void  doActivateWindow(WindowPtr windowPtr,Boolean becomingActive)
{
  ControlHandle         controlHdl;
  docStructureHandle docStrucHdl;
  ListHandle            textListHdl, iconListHdl;

  GetRootControl(windowPtr,&controlHdl);

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

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

    SetThemeTextColor(kThemeTextColorListView,gPixelDepth,gIsColourDevice);
    LUpdate(windowPtr->visRgn,textListHdl);
    LUpdate(windowPtr->visRgn,iconListHdl);

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

    doResetTypeSelection();

    doRestoreBackground(&gBackColourPattern);
    ActivateControl(controlHdl);
    doSetBackgroundWhite();

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

    SetThemeTextColor(kThemeTextColorDialogInactive,gPixelDepth,gIsColourDevice);
    LUpdate(windowPtr->visRgn,textListHdl);
    LUpdate(windowPtr->visRgn,iconListHdl);

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

    doRestoreBackground(&gBackColourPattern);
    DeactivateControl(controlHdl);
    doSetBackgroundWhite();

    doDrawSelections();
  }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doInContent

void  doInContent(EventRecord *eventStrucPtr)
{
  WindowPtr             windowPtr;
  docStructureHandle docStrucHdl;
  ListHandle            textListHdl, iconListHdl;
```

```
    Rect                    textListRect, pictListRect;
    Point                   mouseXY;
    ControlHandle       controlHdl;
    Boolean                 isDoubleClick;

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

    textListRect = (*(*docStrucHdl)->textListHdl)->rView;
    pictListRect = (*(*docStrucHdl)->iconListHdl)->rView;
    textListRect.right += kScrollBarWidth;
    pictListRect.right += kScrollBarWidth;

    SetPort(windowPtr);
    mouseXY = eventStrucPtr->where;
    GlobalToLocal(&mouseXY);

    if(PtInRect(mouseXY,&textListRect) || PtInRect(mouseXY,&pictListRect))
    {
        if((PtInRect(mouseXY,&textListRect) && gCurrentListHdl != textListHdl) ||
            (PtInRect(mouseXY,&pictListRect) && gCurrentListHdl != iconListHdl))
        {
            doRotateCurrentList();
        }

        isDoubleClick = LClick(mouseXY,eventStrucPtr->modifiers,gCurrentListHdl);
        if(isDoubleClick)
            doExtractSelections();
    }
    else if(FindControl(mouseXY,windowPtr,&controlHdl))
  {
        if(TrackControl(controlHdl,mouseXY,NULL))
        {
            if(controlHdl == (*docStrucHdl)->extractButtonHdl)
                doExtractSelections();
        }
    }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doCreateTextList

ListHandle  doCreateTextList(WindowPtr windowPtr,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,windowPtr,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;
   SInt16  totalRows, currentRow, cellDataOffset, cellDataLength;
   Cell       aCell;

   found = false;

   totalRows = (*listHdl)->dataBounds.bottom - (*listHdl)->dataBounds.top;
   currentRow = -1;

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

         MoveHHi((Handle) (*listHdl)->cells);
         HLock((Handle) (*listHdl)->cells);

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

         HUnlock((Handle)(*listHdl)->cells);
      }
   }

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

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

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doCreateIconList

ListHandle  doCreateIconList(WindowPtr windowPtr,Rect listRect,SInt16 numCols,SInt16 lDef)
{
   Rect           dataBounds;
   Point          cellSize;
   ListHandle     iconListHdl;
   Cell           theCell;

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

   listRect.right = listRect.right - kScrollBarWidth;

   iconListHdl = LNew(&listRect,&dataBounds,cellSize,lDef,windowPtr,true,false,false,true);

   (*iconListHdl)->selFlags = lOnlyOne;

   doAddRowsAndDataToIconList(iconListHdl,rIconListIconSuiteBase);

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

   return(iconListHdl);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doAddRowsAndDataToIconList

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

```
    rowNumber = (*iconListHdl)->dataBounds.bottom;

    for(suiteIndex = iconSuiteBase;suiteIndex < (iconSuiteBase + 8);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)
{
    SInt8          newChar;
    Cell           theCell;

    newChar = eventStrucPtr->message & charCodeMask;

    if((gTSLastListHit != listHdl) || ((eventStrucPtr->when - gTSLastKeyTime) >=
         gTSResetThreshold) || (StrLength(gTSString) == 255))
```

```
      doResetTypeSelection();

    gTSLastListHit = listHdl;
    gTSLastKeyTime = eventStrucPtr->when;

    gTSString[0] = (SInt8) (StrLength(gTSString) + 1);
    gTSString[StrLength(gTSString)] = newChar;

    SetPt(&theCell,0,0);

    if(LSearch((Ptr) gTSString+1,StrLength(gTSString),gSearchPartialMatchUPP,&theCell,
                listHdl))
    {
      LSetSelect(true,theCell,listHdl);
      doSelectOneCell(listHdl,theCell);
      doMakeCellVisible(listHdl,theCell);
    }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ searchPartialMatch

pascal 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)
{
    SInt16  listRows;

    listRows = (*listHdl)->dataBounds.bottom - (*listHdl)->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)
{
    Rect     visibleRect;
    SInt16   dRows;

    visibleRect = (*listHdl)->visible;

    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)
{
    gTSString[0] = 0;
    gTSLastListHit = NULL;
    gTSLastKeyTime = 0;
    gTSResetThreshold = 2 * LMGetKeyThresh();
    if(gTSResetThreshold > kMaxKeyThresh)
        gTSResetThreshold = kMaxKeyThresh;
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doRotateCurrentList

void  doRotateCurrentList(void)
{
    ListHandle    oldListHdl, newListHdl;

    oldListHdl = gCurrentListHdl;
    newListHdl = (ListHandle) (*gCurrentListHdl)->refCon;
    gCurrentListHdl = newListHdl;
```

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

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doDrawFrameAndFocus

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

    doRestoreBackground(&gBackColourPattern);

    borderRect = (*listHdl)->rView;
    borderRect.right += kScrollBarWidth;

    DrawThemeFocusRect(&borderRect,false);

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

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

    doSetBackgroundWhite();
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doExtractSelections

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

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

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

    for(cellIndex=0;cellIndex < (*textListHdl)->dataBounds.bottom;cellIndex++)
    {
        SetPt(&theCell,0,cellIndex);
        if(LGetSelect(false,&theCell,textListHdl))
        {
            LGetCellDataLocation(&offset,&dataLen,theCell,textListHdl);
            LGetCell((Ptr) 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,20,188,237,381);
    InvalRect(&theRect);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doDrawSelections

void  doDrawSelections(void)
{
    SInt16  a, nextLine = 193;

    for(a=0;a<15;a++)
    {
        if(gStringArray[a][0] != 0)
        {
            MoveTo(36,nextLine += 12);
            DrawString(gStringArray[a]);
        }
    }
```

```
    MoveTo(170,205);
    DrawString(gStringArray[15]);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
// DialogLists.c
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

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

#include "List.h"

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doListsDialog

void  doListsDialog(void)
{
    DialogPtr               dialogPtr;
    GrafPtr                 oldPort;
    ModalFilterUPP eventFilterUPP;
    ControlHandle       dateFormatControlHdl, watermarkControlHdl, controlHdl;
    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())
        doActivateWindow(FrontWindow(),false);

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

    GetPort(&oldPort);
    SetPort(dialogPtr);

    // ....................................................................................................................................... set default and cancel
push buttons

    SetDialogDefaultItem(dialogPtr,kStdOkItemIndex);
    SetDialogCancelItem(dialogPtr,kStdCancelItemIndex);

    // .............................................................................................. create routine descriptor for event filter function

    eventFilterUPP = NewModalFilterProc((ProcPtr) eventFilter);

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

    GetDialogItemAsControl(dialogPtr,iDateFormatList,&dateFormatControlHdl);
    GetControlData(dateFormatControlHdl,kControlNoPart,kControlListBoxListHandleTag,
                             sizeof(dateFormatListHdl),(Ptr) &dateFormatListHdl,NULL);

    doAddRowsAndDataToTextList(dateFormatListHdl,rDateFormatStrings,17);

    (*dateFormatListHdl)->selFlags = lOnlyOne;

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

    GetDialogItemAsControl(dialogPtr,iWatermarkList,&watermarkControlHdl);
    GetControlData(watermarkControlHdl,kControlNoPart,kControlListBoxListHandleTag,
                             sizeof(watermarkListHdl),(Ptr) &watermarkListHdl,NULL);

    doAddRowsAndDataToTextList(watermarkListHdl,rWatermarkStrings,12);

    (*watermarkListHdl)->selFlags = lOnlyOne;

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

    // .......................................................................................................................................... show dialog and set
keyboard focus

    ShowWindow(dialogPtr);
```

```
  SetKeyboardFocus(dialogPtr,watermarkControlHdl,1);
  SetKeyboardFocus(dialogPtr,dateFormatControlHdl,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((Ptr) dateFormatString + 1,&dataLen,theCell,dateFormatListHdl);
      dateFormatString[0] = (SInt8) dataLen;

      GetDialogItemAsControl(dialogPtr,iDateFormatStaticText,&controlHdl);
      SetControlData(controlHdl,kControlNoPart,kControlStaticTextTextTag,
                     dateFormatString[0],(Ptr) &dateFormatString[1]);
      Draw1Control(controlHdl);

      GetControlData(dateFormatControlHdl,kControlNoPart,kControlListBoxDoubleClickTag,
                     sizeof(wasDoubleClick),(Ptr) &wasDoubleClick,NULL);
    }
    else if(itemHit == iWatermarkList)
    {
      SetPt(&theCell,0,0);
      LGetSelect(true,&theCell,watermarkListHdl);
      LGetCellDataLocation(&offset,&dataLen,theCell,watermarkListHdl);
      LGetCell((Ptr) watermarkString + 1,&dataLen,theCell,watermarkListHdl);
      watermarkString[0] = (SInt8) dataLen;

      GetDialogItemAsControl(dialogPtr,iWatermarkStaticText,&controlHdl);
      SetControlData(controlHdl,kControlNoPart,kControlStaticTextTextTag,
                     watermarkString[0],(Ptr) &watermarkString[1]);
      Draw1Control(controlHdl);

      GetControlData(watermarkControlHdl,kControlNoPart,kControlListBoxDoubleClickTag,
                     sizeof(wasDoubleClick),(Ptr) &wasDoubleClick,NULL);
    }


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

  // 
...........................................................................................................................................................................
.................................. clean up

  DisposeDialog(dialogPtr);
  DisposeRoutineDescriptor(eventFilterUPP);

  SetPort(oldPort);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ eventFilter

pascal Boolean  eventFilter(DialogPtr dialogPtr,EventRecord *eventStrucPtr,
                            SInt16 *itemHit)
{
  Boolean        handledEvent;
  GrafPtr        oldPort;
  SInt8          charCode;
  ControlHandle  controlHdl, focusControlHdl;
  ListHandle     watermarkListHdl, focusListHdl;
  Rect           listRect;

  handledEvent = false;

  if((eventStrucPtr->what == updateEvt) &&
     ((WindowPtr) eventStrucPtr->message != dialogPtr))
  {
    doUpdate(eventStrucPtr);
  }
  else if((eventStrucPtr->what == updateEvt) &&
          ((WindowPtr) eventStrucPtr->message == dialogPtr))
```

```
    {
        if(((WindowPeek) dialogPtr)->hilited)
        {
            GetPort(&oldPort);
            SetPort(dialogPtr);

            GetKeyboardFocus(dialogPtr,&focusControlHdl);
            GetControlData(focusControlHdl,kControlNoPart,kControlListBoxListHandleTag,
                                sizeof(focusListHdl),(Ptr) &focusListHdl,NULL);
            listRect = (*focusListHdl)->rView;
            listRect.right += kScrollBarWidth;
            DrawThemeFocusRect(&listRect,true);

            SetPort(oldPort);
        }
    }
    else
    {
        GetPort(&oldPort);
        SetPort(dialogPtr);

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

            if(charCode != kUpArrow && charCode != kDownArrow && charCode != kTab)
            {
                GetDialogItemAsControl(dialogPtr,iWatermarkList,&controlHdl);
                GetControlData(controlHdl,kControlNoPart,kControlListBoxListHandleTag,
                            sizeof(watermarkListHdl),(Ptr) &watermarkListHdl,NULL);
                GetKeyboardFocus(dialogPtr,&focusControlHdl);
                if(controlHdl == focusControlHdl)
                {
                    doTypeSelectSearch(watermarkListHdl,eventStrucPtr);
                    Draw1Control(controlHdl);
                }

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

        SetPort(oldPort);
    }

    return(handledEvent);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
```

Demonstration Program Comments

When this program is run, the user should open the window and movable modal dialog box by choosing the relevant items in the Demonstration menu. The user should manipulate the lists in the window and dialog box, 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 and clicking in the scroll arrows and gray areas.
- 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 text 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.

List.h

#define

rListsWindow represents the resource ID of the window's 'WIND' resource. The following six constants represent the resource IDs of the window's controls. The next three constants represent the resource IDs of 'STR#' resource containing the strings for the window's text list, the icon suite for the icon list, and the 'STR#' resource containing 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 and kCustomLDEF represent the resource IDs of the default and custom list definition functions.

#typedef

A variables of type docStructure which will be used to store the handles to the two list structures for the window and the handle to the window's push button. The handle to this structure will be assigned to the refCon field of the dialog box's window structure.

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 which has not featured in previous demonstration programs.

main

A routine descriptor is created for the match function used by LSearch in the application's type selection function.

doEvents

In the inGoAway case in the mouseDown case, 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. DisposeHandle then disposes of the window's document structure and DisposeWindow removes the window from the screen, removes it from the window list and discards all its data storage.

doSaveBackground, doRestoreBackground, and doSetBackgroundWhite

doSaveBackground and doRestoreBackground are essentially the same as the functions introduced at Chapter 12 — Drawing With QuickDraw for saving and restoring the drawing environment. In this case, the saving and restoring is limited to the background colour and the background bit or pixel pattern. 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. The next four global variables are associated with the type selection functions. gStringArray will be assigned strings representing the selections from the lists.

doOpenListsWindow

doOpenListsWindow creates the window and its controls, and calls the application-defined functions which creates the two lists for the window.

The call to GetNewCWindow creates the window. The call to NewHandle creates a block for the window's document structure. SetWRefcon attaches the document structure to the window. SetThemeWindowBackground is called to set the window's background colour/pattern and doSaveBackground is called to save this background colour pattern for later use.

CreateRootControl creates a root control for the window so as to simplify the task of activating and deactivating the window's controls. In the rest of this block, the window's push button, group box, and static text field controls are created, and the latter's text is set.

At the next block, the window's font and font size is set to the small system font so that the text in the lists will appear in this font.

The lists are then created. First, the rectangles in which the lists are to be displayed are defined. These are then passed in the calls to the application-defined functions which create the lists. 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 picture 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 penultimate line establishes the text list as the currently active list. ShowWindow is then called to display the window.

doKeyDown

The first line gets the handle to the document structure.

If the key pressed was the Tab key, an application-defined function 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 an application-defined function which further processes the Arrow key event (Line 321).

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 an application-defined type selection function for further processing.

doUpdate

doUpdate handles update events. Between the usual calls to BeginUpdate and EndUpdate, the rectangle in which the current list selections are drawn is erased, the list are updated (that is, redrawn), an application-defined function is called to draw the focus rectangles in the appropriate state, and an application-defined function is called to draw the current list selections in the rectangle at the bottom of the window.

doActivateWindow

doActivateWindow activates and deactivates the content area of the window.

GetRootControl gets the handle to the window's root control. The next three lines get the handles to the list structures.

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 LUpdate are necessary for Appearance purposes. Immediately prior to these calls, the window's text colour is set to that for list views. LUpdate causes a redraw of all of the list's text in that colour. 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, a value which is used in the type selection function.) ActivateControl is called on the root control to activate all the window's controls and redraw them in that state. doDrawSelections redraws the current list selections in the colour set by SetThemeTextColour.

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 a dimmed colour, doDrawFrameAndFocus removes the keyboard focus frame from the active list and draws the list box frames in the inactive state, DeactivateControl draws the controls in the inactive mode, and doDrawSelections redraws the current list selections in the colour set by SetThemeTextColour.

doInContent

doInContent further processes mouse-down events in the content region of the window.

In the first block, handles to the two lists are retrieved. The first three lines of the next block get copies of the lists' display rectangles. Since these rectangles do not include the scroll bars, they are then expanded to the right to encompass the scroll bar areas.

The next block converts the mouse coordinates of the mouse-down to local coordinates required by the following call to PtInRect.

If the mouse click was in one of the list rectangles, and if that rectangle is not the current list's rectangle, the application-defined function which changes the currently active list is called. 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 an application-defined function is called to extract the contents of the currently selected cells.

If the mouse-down event was not within one of the list rectangles, FindControl is called to determine if there is an enabled control under the mouse cursor. If so, 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, and if the control is the window's single push button, an application defined function is called to extract the contents of the currently selected cells.

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 next line calls an application-defined function which 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 an application-defined 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 an application-defined function 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 another 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.

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 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 call to LNew 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 next line calls an application-defined function which 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 8 rows to the icon list and stores a handle to an icon suite in each of the 8 cells.

The first line sets the variable rowNumber to the current number of rows, which is 0.

The for loop executes 8 times. Each time through the loop, the following occurs:

- GetIconSuite creates a new icon family 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 if the Shift key was down, the application-defined function doArrowKeyExtendSelection is called; otherwise, the application-defined function 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 an application-defined function 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 a key-down or auto-key event is received and the key pressed is not the Tab key, the Up Arrow key or the Down Arrow key.

The global variables gTSString, gTSResetThreshold, gTSLastKeyTime, and gTSLastListHit are central to the operation of doTypeSelectSearch. gTSString holds the current type selection search string entered by the user. 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" section of the Keyboard control panel. gTSLastKeyTime holds the time in ticks of the last key press. gTSLastListHit holds a handle to the last list that type selection affected.

The first line extracts the character code from the message field of the event structure.

The next block will cause the application-defined function 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 255 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 next two lines increment the length byte of the type selection string and add the received character to the type selection string. That string 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 application-defined 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 "be" and a cell with the text "Beams" 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 which 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, in effect, makes the type selection string an empty string. 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 number of ticks since the last key press to 0. 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 doUpdate, doActivateWindow, 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 first call to DrawThemeFocusRect erases the keyboard focus frame, if it exists.

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.

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 which 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, retrieve the related string from the specified STR# resource, and assign it to the 15th element of the Str255 array. SetPt sets the starting cell for the LGetSelect search.

The last two lines force an update 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 doUpdate and DoActivateWindow to draw the contents of the Str255 array "filled in" by the function doExtractSelections.

DialogList.c

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

doListsDialog

doListsDialog creates a movable modal dialog box 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 routine descriptor.

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 application-defined 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.

ShowDialog is then called to display the dialog. The first call to SetKeyboardFocus is made to force the "Watermark" list's scroll bar to be drawn. The second call is to set 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 handle to the static text field 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.

doFixKeyboardFocusArea is called for the "Watermark" list for reasons previously explained. It accounts for the situation where the mouse-down changed the current list to the "Date Format" list as well as making a selection within that list.

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 or Cancel push button is hit, or one of the lists has been double-clicked, the dialog and the routine descriptor are disposed of.

eventFilter

A custom event filter function is used for two reasons:

- To ensure that the keyboard focus frame is redrawn after an overlaying balloon is removed.
- To intercept keyDown events so as to support type selection in the "Watermark" list.

If the event is an update event for the dialog, and if the application is not currently in the background, the following occurs. GetKeyboardFocus is called to determine which list control currently has the keyboard focus. GetControlData is called to get the handle to the associated list structure. The list's rectangle is then obtained from that list structure, adjusted by the width of the scroll bar and passed in the call to DrawThemeFocusRect to draw the keyboard focus frame around that 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 handle to the "Watermark" list control, GetControlData is called to get the handle to the associated list, and GetKeyboardFocus is called to get the handle to the control with keyboard focus. If the "Watermark" list control currently has the focus, the application-defined 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.)

LDEF.c

LDEF.c is the custom list definition function (LDEF) used by the window's icon list.

main

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. The main function 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 colour graphics port and set the colour 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 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.

When this program is run under a version of the Mac OS earlier than Mac OS 8.5, you may notice the following anomaly.

The width of the list box frame (drawn by the CDEF) is two pixels. The width of the keyboard focus frame is three pixels, that is, one pixel outside the list box frame. When the keyboard focus is moved to the other list, the keyboard focus frame is erased from the previous list and the CDEF redraws the list box frame. The problem is that the keyboard focus frame is erased to white, not to the background colour/pattern of the dialog box proper. The result is a one-pixel-wide white frame around the list box frame.

To compensate for this anomaly, proceed as follows:

In Lists.h, add this function prototype:

```
void  doFixKeyboardFocusArea(DialogPtr, ListHandle);
```

In DialogLists.c, add this function, which simply erases the keyboard focus frame to the correct background colour/pattern and redraws the list box frame:

```
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doFixKeyboardFocusArea

void  doFixKeyboardFocusArea(DialogPtr dialogPtr, ListHandle listHdl)
{
GrafPtr oldPort;
Rect    listRect;

GetPort(&oldPort);
SetPort(dialogPtr);

listRect = (*listHdl)->rView;
listRect.right += kScrollBarWidth;
DrawThemeFocusRect(&listRect,false);
DrawThemeListBoxFrame(&listRect,true);

SetPort(oldPort);
}
```

In DialogLists.c, in the function doListsDialog, after the calls to SetKeyboardFocus, add:

```
doFixKeyboardFocusArea(dialogPtr,watermarkListHdl);
```

In DialogLists.c, in the function doListsDialog, after the first call to Draw1Control, add:

```
doFixKeyboardFocusArea(dialogPtr,watermarkListHdl);
```

In DialogLists.c, in the function doListsDialog, after the second call to Draw1Control, add:

```
doFixKeyboardFocusArea(dialogPtr,dateFormatListHdl);
```

doFixKeyboardFocus obtains the rectangle for the specified list, expands it to accommodate the width of the scroll bar, and erases the keyboard focus frame. Because the background colour/pattern is that of the dialog proper when this function is called, the erasure is to the correct colour/pattern. Because the erasure also erases the list box frame, DrawThemeListBoxFrame is called to redraw that frame.