18 Version 1.1

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, cannot present items in columns as well as rows, and are not suited to the presentation of graphics (such as icons) as items. 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, may be 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 Macintosh...** from the 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, more generally, 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 Macintosh...** 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.

A further minor limitation is that the List Manager expects all cells to be equal in size.

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

Lists inherit the font of the graphics port associated with the window or dialog box in which they reside. Ordinarily, your text-only lists should use the system font (Chicago) with a size of 12 points.

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 an 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 and Size Boxes

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.

Size Box

Your application can specify whether the List Manager should leave room for a size box, although your application is responsible for drawing the grow icon within that box. Usually, size boxes are useful only for lists that are at the bottom of windows which contain them.

When you include a size box, your application should ensure that the user cannot shrink the window so much that the list is no longer visible.

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.[1] The following describes the default cell-selection behaviour.[2]

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 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. Probably the most common modification is to defeat multiple cell selection, allowing the user to select only one cell.

---

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

[2] The default behaviour is somewhat complex and is probably best explored by experimenting with the text-only list in the demonstration program. That list uses the default cell-selection algorythm.

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 routines 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.

Creating, Disposing Of, and Managing Lists

The List Record

The **list record**, 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 record using List Manager routines.

Before describing the list record, 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.

[3] The value in **KeyThresh** is set by the user at the "Delay Until Repeat" section of the Keyboard control panel.

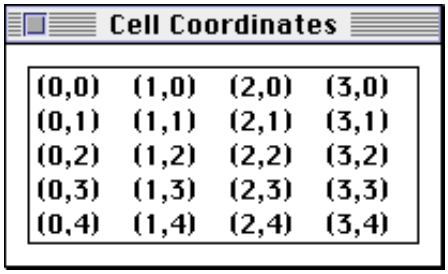


FIG 2 - COORDINATES OF CELLS

The ListRec Data Type

The list record 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 lClikLoop;     // Routine 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,*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** procedure 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 Chicago, 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** routine 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 10 — Basic 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** routine to obtain this information, but this field may be more convenient to access from within a click-loop procedure (see below).

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

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

`lastClik` 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], *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 routines for manipulating the information in the list.

The fields of a list record 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`.

```
ListRef LNew(const RectrView, 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`, `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. (Recall that the List Manager does not draw the grow icon. That is the responsibility of your application)

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 Borders Around the List

One-Pixel-Wide Border. The List Manager does not draw a border around the list. Accordingly, a one-pixel-wide border should be drawn by your application. This should be one pixel outside the rectangle stored in the **rView** field of the list record.

Two-Pixel-Wide Border. 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] The convention is to draw a 2-pixel-wide border around the current list, with one pixel of white space separating it from the one-pixel-wide border (see the list on the right at Fig 1). The outline should be removed when the window or dialog box containing the lists is deactivated.

Creating Lists in Dialog Boxes

List are often used in dialog boxes. Because the Control Manager does not define a control for lists, you must define a list in a dialog item list as a user item.

Disposing of a List

When you are finished with a list, you should dispose of it using **LDispose**, which disposes of the list record 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 record. 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.

---

[4] A single list in a window should also be outlined with a 2-pixel-wide outline 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).

`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 routine for changing the current list.

Key-Down Events

If a key-down event is received, and assuming that your application supports cell selection by Arrow key and/or type selection, your application should call its appropriate application-defined routines. 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 graphics port's `visRgn` field.

Your application will also need to call its application-defined routines for drawing the one-pixel-wide list border and, in the case of a window with multiple lists, the two-pixel-wide border around the current list.

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. In addition, if the window contains multiple lists, the two-pixel wide border around the current list should be erased when the window is being deactivated and drawn when the window is being activated.

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 activated.

Getting and Setting List Selections

The List Manager provides routines 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 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** routine (old name **IUMagIDString**) 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 **UMagIDString**.

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 record 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 record. (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.

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 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 message,Boolean selected,Rect *cellRect,Cell theCell,
                     SInt16 dataOffset,SInt16 dataLen,ListHandle theList);
```

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 **message** 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 **selected**, **cellRect**, **theCell**, **dataOffset**, and **dataLen** parameters pass information to your list definition function only when the value in the **message** 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 **cellRect** and **theCell** parameters indicate the cell's rectangle and coordinates. The **dataOffset** and **dataLen** parameters specify the offset and length of the cell's data within the relocatable block referenced by the **cells** field of the list record.

Responding to the Initialisation Message

The List Manager automatically allocates memory for a list and fills out the fields of a list record 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 record. 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 code shows a response which is compatible with all Macintosh models, including those which do not support Color QuickDraw:

[5]For example, the Finder's **About This Macintosh...** 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.

```
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 Routines

Constants

Masks For `listFlags` Field of List Record

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

Masks For `selFlags` Field of List Record

```
lOnlyOne       = -128  Allow only one item to be selected at once.
lExtendDrag    = 64    Enable multiple item selection without Shift.
lNoDisjoint    = 32    Prevent discontiguous selections.
lNoExtend      = 16    Reset list before responding to Shift-click.
lNoRect        = 8     Shift-drag selects items passed by cursor.
lUseSense      = 4     Allow use of Shift key to deselect items.
lNoNilHilite   = 2     Disable highlighting of empty cells.
```

Messages to List Definition 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 ListHandle ListRef;

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

typedef pascal short (*SearchProcPtr) (Ptr aPtr, Ptr bPtr, short aLen, short bLen);
```

List Record

```
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 lClikLoop;     // Routine 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,*ListPtr,**ListHandle;
```

Routines

Creating and Disposing of Lists

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

Adding and Deleting Rows and Columns

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

Determining or Changing a Selection

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

Accessing and Manipulating Data Cells

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

Responding to Events

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

Modifying a List's Appearance

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

Searching For a List Containing a Particular Item

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

Changing the Size of Cells and Lists

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

Getting Information About Cells

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

Demonstration Program

```
1  // ##############################################################################################
2  // Lists.c
3  // ##############################################################################################
4  //
5  // This program allows the user to open a dialog box by choosing the Dialog With Lists
6  // item in the Demonstration menu.
7  //
8  // The dialog box contains two lists.  The cells of one list contain text.  The cells of
9  // the other list contain icon-like pictures and their titles.
10 //
11 // The text list uses the default list definition function.
12 //
13 // The picture list uses a custom list definition function.  The source code for the
14 // custom list definition function is at the file LDEF.c in the LDEF folder.
15 //
16 // The currently active list is outlined by a two-pixel-wide border.  The currently
17 // active list can be changed by clicking in the non-active list or by pressing the tab
18 // key.
19 //
20 // The text list uses the default cell-selection algorithm; accordingly, multiple cells,
21 // including discontiguous multiple cells, may be selected.  The picture list also
22 // supports arrow key selection (of single or multiple cells) and type selection.
23 //
24 // The constant lOnlyOne is assigned to the selFlags field of the picture list's list
25 // record.  Accordingly, the selection of multiple items is not possible in this list.
26 // Arrow key selection (of single cells) is, however, supported.
27 //
28 // When the dialog is dismissed by clicking on the OK button, or by double-clicking on a
29 // cell in the active list, the user's selections are displayed in a window opened by the
30 // program at program launch.  (Note that the use of the Return, Enter, Esc and
31 // Command-period keys as alternatives to clicking the OK and Cancel buttons in the
32 // dialog box is not supported in this program.)
33 //
34 // The program utilises the following resources:
35 //
36 // • An 'MBAR' resource, and 'MENU' resources for Apple, File, Edit and Demonstration
37 //   menus (preload, non-purgeable).
38 //
39 // • A 'WIND' resource (purgeable) (initially visible) for the window in which the
40 //   user's selections are displayed.
41 //
42 // • A'DLOG' resource (purgeable) and associated 'DITL' resource (purgeable) for the
43 //   dialog box.
44 //
45 // • 'STR#' resources (purgeable) containing the text strings for the text list.
46 //
47 // • 'PICT' resources (non-purgeable) containing the images for the picture list.
48 //
49 // • An 'LDEF' resource (non-purgeable) containing the custom list definition function
50 //   used by the picture list.
51 //
52 // • A 'SIZE' resource with the acceptSuspendResumeEvents, doesActivateOnFGSwitch, and
53 //   is32BitCompatible flags set.
54 //
55 // ##############################################################################################
56
57 // ..... includes
58
59 #include <Fonts.h>
60 #include <Menus.h>
61 #include <TextEdit.h>
62 #include <Dialogs.h>
63 #include <SegLoad.h>
64 #include <ToolUtils.h>
65 #include <Devices.h>
66 #include <Lists.h>
```

```
67  #include <LowMem.h>
68
69  // ..... defines
70
71  #define mApple              128
72  #define  iAbout             1
73  #define mFile               129
74  #define  iQuit              11
75  #define mDemonstration      131
76  #define  iDialog            1
77  #define rMenubar            128
78  #define rWindow             128
79  #define rDialog             129
80  #define  iOK                1
81  #define  iCancel            2
82  #define  iUserItemText      3
83  #define  iUserItemPict      4
84  #define rListCellStrings    128
85  #define rListCellPicts      128
86  #define rListCellPictTitles 129
87  #define kUpArrow            0x1e
88  #define kDownArrow          0x1f
89  #define kTab                0x09
90  #define kScrollBarWidth     15
91  #define kMaxKeyThresh       120
92  #define kSystemLDEF         0
93  #define kCustomLDEF         128
94  #define MAXLONG             0x7FFFFFFF
95
96  // ..... typedefs
97
98  typedef struct
99  {
100   ListHandle textListHdl;
101   ListHandle pictListHdl;
102 } ListsRec, **ListsRecHandle;
103
104 // ..... global variables
105
106 Boolean     gDone;
107 Boolean     gInBackground;
108 WindowPtr   gWindowPtr;
109 ListHandle gCurrentListHdl;
110 Str255      gTSString;
111 SInt16      gTSResetThreshold;
112 SInt32      gTSLastKeyTime;
113 ListHandle gTSLastListHit;
114
115 // ..... function prototypes
116
117 void          main                        (void);
118 void          doInitManagers              (void);
119 void          doEvents                    (EventRecord *);
120 void          doMouseDown                 (EventRecord *);
121 void          doKeyDown                   (SInt8,EventRecord *);
122 void          doUpdate                    (EventRecord *);
123 void          doUpdateLists               (WindowPtr);
124 void          doActivate                  (EventRecord *);
125 void          doActivateDialog            (WindowPtr,Boolean);
126 void          doOSEvent                   (EventRecord *);
127 void          doInContent                 (EventRecord *);
128 void          doItemHitInDialog           (DialogPtr,SInt16);
129 void          doCreateDialogWithLists     (void);
130 ListHandle    doCreateTextList            (DialogPtr,Rect,SInt16,SInt16);
131 void          doAddRowsAndDataToTextList(ListHandle,SInt16);
132 void          doAddTextItemAlphabetically(ListHandle,Str255);
133 ListHandle    doCreatePictList            (DialogPtr,Rect,SInt16,SInt16);
134 void          doAddRowsAndDataToPictList(ListHandle,SInt16);
135 void          doHandleArrowKey            (SInt8,EventRecord *,Boolean);
136 void          doArrowKeyMoveSelection     (ListHandle,SInt8,Boolean);
137 void          doArrowKeyExtendSelection (ListHandle,SInt8,Boolean);
138 void          doTypeSelectSearch          (ListHandle,EventRecord *);
139 pascal SInt16doSearchPartialMatch         (Ptr,Ptr,SInt16,SInt16);
140 Boolean       doFindFirstSelectedCell     (ListHandle,Cell *);
141 void          doFindLastSelectedCell      (ListHandle,Cell *);
142 void          doFindNewCellLoc            (ListHandle,Cell,Cell *,SInt8,Boolean);
143 void          doSelectOneCell             (ListHandle,Cell);
```

```
void          doMakeCellVisible          (ListHandle,Cell);
void          doResetTypeSelection       (void);
void          doRotateCurrentList        (void);
void          doDrawListsBorders         (ListHandle,ListHandle);
void          doDrawActiveListBorder     (ListHandle);
void          doDisplaySelections        (void);
void          doAdjustMenus              (void);
void          doMenuChoice               (SInt32);
void          doDrawDialogDefaultButton (DialogPtr);

// ############################################################################## main

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

  // ..... initialise managers

  doInitManagers();

  // ..... 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');

  // ..... open window

  if(!(gWindowPtr = GetNewWindow(rWindow,NULL,(WindowPtr)-1)))
      ExitToShell();

  SetPort(gWindowPtr);
  TextSize(10);

  // ..... enter eventLoop

  gDone = false;

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

// ########################################################################## doInitManagers

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

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

  InitCursor();
  FlushEvents(everyEvent,0);
}

// ############################################################################## doEvents

void  doEvents(EventRecord *eventRecPtr)
{
```

```
221     SInt8 charCode;
222
223     switch(eventRecPtr->what)
224     {
225       case mouseDown:
226         doMouseDown(eventRecPtr);
227         break;
228
229       case keyDown:
230       case autoKey:
231         charCode = eventRecPtr->message & charCodeMask;
232         if((eventRecPtr->modifiers & cmdKey) != 0)
233         {
234           doAdjustMenus();
235           doMenuChoice(MenuKey(charCode));
236         }
237         doKeyDown(charCode,eventRecPtr);
238         break;
239
240       case updateEvt:
241         doUpdate(eventRecPtr);
242         break;
243
244       case activateEvt:
245         doActivate(eventRecPtr);
246         break;
247
248       case osEvt:
249         doOSEvent(eventRecPtr);
250         HiliteMenu(0);
251         break;
252     }
253   }
254
255   // ############################################################################## doMouseDown
256
257   void doMouseDown(EventRecord *eventRecPtr)
258   {
259     SInt16    partCode;
260     WindowPtrwindowPtr;
261
262     partCode = FindWindow(eventRecPtr->where,&windowPtr);
263
264     switch(partCode)
265     {
266       case inMenuBar:
267         doAdjustMenus();
268         doMenuChoice(MenuSelect(eventRecPtr->where));
269         break;
270
271       case inSysWindow:
272         SystemClick(eventRecPtr,windowPtr);
273         break;
274
275       case inContent:
276         if(windowPtr != FrontWindow())
277         {
278           if(((WindowPeek) (FrontWindow()))->windowKind == dialogKind)
279             SysBeep(10);
280           else
281             SelectWindow(windowPtr);
282         }
283         else
284         {
285           if(((WindowPeek) (FrontWindow()))->windowKind == dialogKind)
286             doInContent(eventRecPtr);
287         }
288         break;
289
290       case inDrag:
291         if((((WindowPeek) FrontWindow())->windowKind == dialogKind) &&
292            (((WindowPeek) windowPtr)->windowKind != dialogKind))
293         {
294           SysBeep(10);
295           return;
296         }
297         DragWindow(windowPtr,eventRecPtr->where,&qd.screenBits.bounds);
```

```
298         break;
299       }
300     }
301
302     // ############################################################################## doKeyDown
303
304     void  doKeyDown(SInt8 charCode,EventRecord *eventRecPtr)
305     {
306       ListsRecHandle listsRecHdl;
307       Boolean        allowExtendSelect;
308
309       if(((WindowPeek) FrontWindow())->windowKind == dialogKind)
310       {
311         listsRecHdl = (ListsRecHandle) GetWRefCon(FrontWindow());
312
313         if(charCode == kTab)
314           doRotateCurrentList();
315         else if(charCode == kUpArrow || charCode == kDownArrow)
316         {
317           if(gCurrentListHdl == (*listsRecHdl)->textListHdl)
318             allowExtendSelect = true;
319           else
320             allowExtendSelect = false;
321           doHandleArrowKey(charCode,eventRecPtr,allowExtendSelect);
322         }
323         else
324         {
325           if(gCurrentListHdl == (*listsRecHdl)->textListHdl)
326             doTypeSelectSearch((*listsRecHdl)->textListHdl,eventRecPtr);
327         }
328       }
329     }
330
331     // ############################################################################### doUpdate
332
333     void  doUpdate(EventRecord *eventRecPtr)
334     {
335       WindowPtrwindowPtr;
336
337       windowPtr = (WindowPtr) eventRecPtr->message;
338
339       BeginUpdate(windowPtr);
340
341       if(((WindowPeek) windowPtr)->windowKind == dialogKind)
342       {
343         UpdateDialog(windowPtr,windowPtr->visRgn);
344         doDrawDialogDefaultButton(windowPtr);
345         doUpdateLists(windowPtr);
346       }
347
348       EndUpdate(windowPtr);
349
350     }
351
352     // ############################################################################ doUpdateLists
353
354     void  doUpdateLists(WindowPtr windowPtr)
355     {
356       ListsRecHandle listsRecHdl;
357       ListHandle     textListHdl, pictListHdl;
358
359       listsRecHdl = (ListsRecHandle) GetWRefCon(windowPtr);
360
361       textListHdl = (*listsRecHdl)->textListHdl;
362       pictListHdl = (*listsRecHdl)->pictListHdl;
363
364       SetPort((*textListHdl)->port);
365
366       LUpdate((*(*textListHdl)->port).visRgn,textListHdl);
367       LUpdate((*(*pictListHdl)->port).visRgn,pictListHdl);
368
369       doDrawListsBorders(textListHdl,pictListHdl);
370       doDrawActiveListBorder(textListHdl);
371       doDrawActiveListBorder(pictListHdl);
372     }
373
374     // ############################################################################## doActivate
```

```
375
376 void  doActivate(EventRecord *eventRecPtr)
377 {
378   WindowPtrwindowPtr;
379   Boolean  becomingActive;
380
381   windowPtr = (WindowPtr) eventRecPtr->message;
382   becomingActive = ((eventRecPtr->modifiers & activeFlag) == activeFlag);
383
384   if(((WindowPeek) windowPtr)->windowKind == dialogKind)
385     doActivateDialog(windowPtr,becomingActive);
386 }
387
388 // ########################################################################## doActivateDialog
389
390 void  doActivateDialog(WindowPtr windowPtr,Boolean becomingActive)
391 {
392   ListsRecHandle listRecsHdl;
393   ListHandle     textListHdl, pictListHdl;
394   SInt16         itemType;
395   Handle         itemHdl;
396   Rect           itemRect;
397
398   listRecsHdl = (ListsRecHandle) GetWRefCon(windowPtr);
399   textListHdl = (*listRecsHdl)->textListHdl;
400   pictListHdl = (*listRecsHdl)->pictListHdl;
401
402   if(becomingActive)
403   {
404     GetDialogItem((DialogPtr) windowPtr,iOK,&itemType,&itemHdl,&itemRect);
405     HiliteControl((ControlHandle) itemHdl,0);
406     GetDialogItem((DialogPtr) windowPtr,iCancel,&itemType,&itemHdl,&itemRect);
407     HiliteControl((ControlHandle) itemHdl,0);
408     doDrawDialogDefaultButton(windowPtr);
409
410     LActivate(true,textListHdl);
411     LActivate(true,pictListHdl);
412
413     doDrawActiveListBorder(gCurrentListHdl);
414     doResetTypeSelection();
415   }
416   else
417   {
418     GetDialogItem((DialogPtr) windowPtr,iOK,&itemType,&itemHdl,&itemRect);
419     HiliteControl((ControlHandle) itemHdl,255);
420     GetDialogItem((DialogPtr) windowPtr,iCancel,&itemType,&itemHdl,&itemRect);
421     HiliteControl((ControlHandle) itemHdl,255);
422     doDrawDialogDefaultButton(windowPtr);
423
424     LActivate(false,textListHdl);
425     LActivate(false,pictListHdl);
426
427     doDrawActiveListBorder(gCurrentListHdl);
428   }
429 }
430
431 // ############################################################################## doOSEvent
432
433 void doOSEvent(EventRecord *eventRecPtr)
434 {
435   switch((eventRecPtr->message >> 24) & 0x000000FF)
436   {
437     case suspendResumeMessage:
438       gInBackground = (eventRecPtr->message & resumeFlag) == 0;
439       if(((WindowPeek) (FrontWindow()))->windowKind == dialogKind)
440         doActivateDialog(FrontWindow(),!gInBackground);
441       break;
442
443     case mouseMovedMessage:
444       break;
445   }
446 }
447
448 // ############################################################################ doInContent
449
450 void  doInContent(EventRecord *eventRecPtr)
451 {
```

```
  GrafPtr        oldPort;
  ListsRecHandle listsRecHdl;
  ListHandle     textListHdl, pictListHdl;
  Rect           textListRect, pictListRect, gCurrentListRect;
  Point          mouseXY;
  Boolean        isDoubleClick;
  DialogPtr      dialogPtr;
  SInt16         itemHit;

  GetPort(&oldPort);

  listsRecHdl = (ListsRecHandle) GetWRefCon(FrontWindow());
  textListHdl = (*listsRecHdl)->textListHdl;
  pictListHdl = (*listsRecHdl)->pictListHdl;

  textListRect = (*(*listsRecHdl)->textListHdl)->rView;
  pictListRect = (*(*listsRecHdl)->pictListHdl)->rView;
  gCurrentListRect = (*gCurrentListHdl)->rView;
  textListRect.right += kScrollBarWidth;
  pictListRect.right += kScrollBarWidth;
  gCurrentListRect.right += kScrollBarWidth;

  mouseXY = eventRecPtr->where;
  GlobalToLocal(&mouseXY);

  if((PtInRect(mouseXY,&textListRect) && gCurrentListHdl != textListHdl) ||
     (PtInRect(mouseXY,&pictListRect) && gCurrentListHdl != pictListHdl))
  {
    doRotateCurrentList();
  }
  else if(PtInRect(mouseXY,&gCurrentListRect))
  {
    SetPort((*gCurrentListHdl)->port);
    isDoubleClick = LClick(mouseXY,eventRecPtr->modifiers,gCurrentListHdl);
    if(isDoubleClick)
      doItemHitInDialog(FrontWindow(),iOK);
  }
  else
  {
    if(DialogSelect(eventRecPtr,&dialogPtr,&itemHit))
      doItemHitInDialog(dialogPtr,itemHit);
  }

  SetPort(oldPort);
}

// ########################################################################## doItemHitInDialog

void  doItemHitInDialog(DialogPtr dialogPtr,SInt16 itemHit)
{
  ListsRecHandle listsRecHdl;

  if(itemHit == iOK || itemHit == iCancel)
  {
    if(itemHit == iOK)
     doDisplaySelections();

    listsRecHdl = (ListsRecHandle) GetWRefCon(dialogPtr);

    LDispose((*listsRecHdl)->textListHdl);
    LDispose((*listsRecHdl)->pictListHdl);
    DisposeHandle((Handle) listsRecHdl);
    DisposeDialog(dialogPtr);

    doAdjustMenus();
  }
}

// ############################################################### doCreateDialogWithLists

void  doCreateDialogWithLists(void)
{
  DialogPtr      modalDlgPtr;
  ListsRecHandle listsRecHdl;
  SInt16         fontNum, itemType;
  Handle         itemHdl;
  Rect           itemRect;
```

```
529     ListHandle      textListHdl, pictListHdl;
530
531     if(!(modalDlgPtr = GetNewDialog(rDialog,NULL,(WindowPtr) -1)))
532       ExitToShell();
533
534     if(!(listsRecHdl = (ListsRecHandle) NewHandle(sizeof(ListsRec))))
535       ExitToShell();
536     SetWRefCon(modalDlgPtr,(SInt32) listsRecHdl);
537
538     SetPort(modalDlgPtr);
539
540     GetFNum("\pChicago",&fontNum);
541     TextFont(fontNum);
542     TextSize(12);
543
544     GetDialogItem(modalDlgPtr,iUserItemText,&itemType,&itemHdl,&itemRect);
545     textListHdl = doCreateTextList(modalDlgPtr,itemRect,1,kSystemLDEF);
546
547     GetDialogItem(modalDlgPtr,iUserItemPict,&itemType,&itemHdl,&itemRect);
548     pictListHdl = doCreatePictList(modalDlgPtr,itemRect,1,kCustomLDEF);
549
550     (*listsRecHdl)->textListHdl = textListHdl;
551     (*listsRecHdl)->pictListHdl = pictListHdl;
552
553     (*textListHdl)->refCon = (SInt32) pictListHdl;
554     (*pictListHdl)->refCon = (SInt32) textListHdl;
555
556     gCurrentListHdl = textListHdl;
557
558     ShowWindow(modalDlgPtr);
559     doAdjustMenus();
560   }
561
562   // ########################################################################## doCreateTextList
563
564   ListHandle  doCreateTextList(DialogPtr dialogPtr,Rect listRect,SInt16 numCols,SInt16 lDef)
565   {
566     Rect            dataBounds;
567     Point           cellSize;
568     ListHandle      textListHdl;
569     Cell            theCell;
570
571     SetRect(&dataBounds,0,0,numCols,0);
572     SetPt(&cellSize,0,0);
573
574     listRect.right = listRect.right - kScrollBarWidth;
575
576     textListHdl = LNew(&listRect,&dataBounds,cellSize,lDef,dialogPtr,true,false,false,true);
577
578     doAddRowsAndDataToTextList(textListHdl,rListCellStrings);
579
580     SetPt(&theCell,0,0);
581     LSetSelect(true,theCell,textListHdl);
582
583     doResetTypeSelection();
584
585     return(textListHdl);
586   }
587
588   // ############################################################## doAddRowsAndDataToTextList
589
590   void  doAddRowsAndDataToTextList(ListHandle textListHdl,SInt16 stringListID)
591   {
592     SInt16 stringIndex;
593     Str255 theString;
594
595     for(stringIndex=1;stringIndex<16;stringIndex++)
596     {
597       GetIndString(theString,stringListID,stringIndex);
598       doAddTextItemAlphabetically(textListHdl,theString);
599     }
600   }
601
602   // ########################################################## doAddTextItemAlphabetically
603
604   void  doAddTextItemAlphabetically(ListHandle listHdl,Str255 theString)
605   {
```

```
606     Boolean found;
607     SInt16 totalRows, currentRow, cellDataOffset, cellDataLength;
608     Cell    aCell;
609
610     found = false;
611
612     totalRows = (*listHdl)->dataBounds.bottom - (*listHdl)->dataBounds.top;
613     currentRow = -1;
614
615     while(!found)
616     {
617       currentRow += 1;
618       if(currentRow == totalRows)
619         found = true;
620       else
621       {
622         SetPt(&aCell,0,currentRow);
623         LGetCellDataLocation(&cellDataOffset,&cellDataLength,aCell,listHdl);
624
625         MoveHHi((Handle) (*listHdl)->cells);
626         HLock((Handle) (*listHdl)->cells);
627
628         if(IUMagPString((Ptr) theString + 1,((Ptr) (*listHdl)->cells[0] + cellDataOffset),
629                         Length(theString),cellDataLength,NULL) == -1)
630         {
631           found = true;
632         }
633
634         HUnlock((Handle)(*listHdl)->cells);
635       }
636     }
637
638     currentRow = LAddRow(1,currentRow,listHdl);
639     SetPt(&aCell,0,currentRow);
640
641     LSetCell((Ptr) theString + 1,(SInt16) Length(theString),aCell,listHdl);
642   }
643
644   // ########################################################################## doCreatePictList
645
646   ListHandle  doCreatePictList(DialogPtr dialogPtr,Rect listRect,SInt16 numCols,SInt16 lDef)
647   {
648     Rect        dataBounds;
649     Point       cellSize;
650     ListHandle pictListHdl;
651     Cell        theCell;
652
653     SetRect(&dataBounds,0,0,numCols,0);
654     SetPt(&cellSize,48,48);
655
656     listRect.right = listRect.right - kScrollBarWidth;
657
658     pictListHdl = LNew(&listRect,&dataBounds,cellSize,lDef,dialogPtr,true,false,false,true);
659
660     (*pictListHdl)->selFlags = lOnlyOne;
661
662     doAddRowsAndDataToPictList(pictListHdl,rListCellPicts);
663
664     SetPt(&theCell,0,0);
665     LSetSelect(true,theCell,pictListHdl);
666
667     return(pictListHdl);
668   }
669
670   // ############################################################## doAddRowsAndDataToPictList
671
672   void  doAddRowsAndDataToPictList(ListHandle pictListHdl,SInt16 pictListID)
673   {
674     SInt16    rowNumber, pictIndex;
675     PicHandlepictureHdl;
676     Cell      theCell;
677
678     rowNumber = (*pictListHdl)->dataBounds.bottom;
679
680     for(pictIndex=pictListID;pictIndex<(pictListID + 6);pictIndex++)
681     {
682       pictureHdl = GetPicture(pictIndex);
```

```
683
684       rowNumber = LAddRow(1,rowNumber,pictListHdl);
685       SetPt(&theCell,0,rowNumber);
686       LSetCell(&pictureHdl,sizeof(PicHandle),theCell,pictListHdl);
687
688       rowNumber += 1;
689     }
690   }
691
692   // ########################################################################## doHandleArrowKey
693
694   void  doHandleArrowKey(SInt8 charCode,EventRecord *eventRecPtr,Boolean allowExtendSelect)
695   {
696     BooleanmoveToTopBottom = false;
697
698     if(eventRecPtr->modifiers & cmdKey)
699       moveToTopBottom = true;
700
701     if(allowExtendSelect && (eventRecPtr->modifiers & shiftKey))
702       doArrowKeyExtendSelection(gCurrentListHdl,charCode,moveToTopBottom);
703     else
704       doArrowKeyMoveSelection(gCurrentListHdl,charCode,moveToTopBottom);
705   }
706
707   // ############################################################### doArrowKeyMoveSelection
708
709   void  doArrowKeyMoveSelection(ListHandle listHdl,SInt8 charCode,Boolean moveToTopBottom)
710   {
711     Cell  currentSelection, newSelection;
712
713     if(doFindFirstSelectedCell(listHdl,&currentSelection))
714     {
715       if(charCode == kDownArrow)
716         doFindLastSelectedCell(listHdl,&currentSelection);
717
718       doFindNewCellLoc(listHdl,currentSelection,&newSelection,charCode,moveToTopBottom);
719
720       doSelectOneCell(listHdl,newSelection);
721       doMakeCellVisible(listHdl,newSelection);
722     }
723   }
724
725   // ############################################################## doArrowKeyExtendSelection
726
727   void  doArrowKeyExtendSelection(ListHandle listHdl,SInt8 charCode,Boolean moveToTopBottom)
728   {
729     Cell  currentSelection, newSelection;
730
731     if(doFindFirstSelectedCell(listHdl,&currentSelection))
732     {
733       if(charCode == kDownArrow)
734         doFindLastSelectedCell(listHdl,&currentSelection);
735
736       doFindNewCellLoc(listHdl,currentSelection,&newSelection,charCode,moveToTopBottom);
737
738       if(!(LGetSelect(false,&newSelection,listHdl)))
739         LSetSelect(true,newSelection,listHdl);
740
741       doMakeCellVisible(listHdl,newSelection);
742     }
743   }
744
745   // ########################################################################## doTypeSelectSearch
746
747   void  doTypeSelectSearch(ListHandle listHdl,EventRecord *eventRecPtr)
748   {
749     SInt8 newChar;
750     Cell  theCell;
751
752     newChar = eventRecPtr->message & charCodeMask;
753
754     if((gTSLastListHit != listHdl) || ((eventRecPtr->when - gTSLastKeyTime) >=
755        gTSResetThreshold) || (Length(gTSString) == 255))
756       doResetTypeSelection();
757
758     gTSLastListHit = listHdl;
759     gTSLastKeyTime = eventRecPtr->when;
```

```
760
761     gTSString[0] = (SInt8) (Length(gTSString) + 1);
762     gTSString[Length(gTSString)] = newChar;
763
764     SetPt(&theCell,0,0);
765
766     if(LSearch((Ptr) gTSString+1,Length(gTSString),&doSearchPartialMatch,&theCell,listHdl))
767     {
768       LSetSelect(true,theCell,listHdl);
769       doSelectOneCell(listHdl,theCell);
770       doMakeCellVisible(listHdl,theCell);
771     }
772   }
773
774   // ########################################################################## doSearchPartialMatch
775
776   pascal SInt16  doSearchPartialMatch(Ptr searchDataPtr,Ptr cellDataPtr,SInt16 cellDataLen,
777                                       SInt16 searchDataLen)
778   {
779     SInt16 result;
780
781     if((cellDataLen > 0) && (cellDataLen >= searchDataLen))
782       result = IUMagIDString(cellDataPtr,searchDataPtr,searchDataLen,searchDataLen);
783     else
784       result = 1;
785
786     return(result);
787   }
788
789   // ########################################################################## doFindFirstSelectedCell
790
791   Boolean  doFindFirstSelectedCell(ListHandle listHdl,Cell *theCell)
792   {
793     Boolean result;
794
795     SetPt(theCell,0,0);
796     result = LGetSelect(true,theCell,listHdl);
797
798     return(result);
799   }
800
801   // ########################################################################## doFindLastSelectedCell
802
803   void  doFindLastSelectedCell(ListHandle listHdl,Cell *theCell)
804   {
805     Cell    aCell;
806     Boolean moreCellsInList;
807
808     if(doFindFirstSelectedCell(listHdl,&aCell))
809     {
810       while(LGetSelect(true,&aCell,listHdl))
811       {
812         *theCell = aCell;
813         moreCellsInList = LNextCell(true,true,&aCell,listHdl);
814       }
815     }
816   }
817
818   // ########################################################################## doFindNewCellLoc
819
820   void  doFindNewCellLoc(ListHandle listHdl,Cell oldCellLoc,Cell *newCellLoc,SInt8 charCode,
821                          Boolean moveToTopBottom)
822   {
823     SInt16 listRows;
824
825     listRows = (*listHdl)->dataBounds.bottom - (*listHdl)->dataBounds.top;
826     *newCellLoc = oldCellLoc;
827
828     if(moveToTopBottom)
829     {
830       if(charCode == kUpArrow)
831         (*newCellLoc).v = 0;
832       else if(charCode == kDownArrow)
833         (*newCellLoc).v = listRows - 1;
834     }
835     else
836     {
```

```
837       if(charCode ==  kUpArrow)
838       {
839         if(oldCellLoc.v != 0)
840           (*newCellLoc).v = oldCellLoc.v - 1;
841       }
842       else if(charCode == kDownArrow)
843       {
844         if(oldCellLoc.v != listRows - 1)
845           (*newCellLoc).v = oldCellLoc.v + 1;
846       }
847     }
848   }
849
850   // ############################################################################## doSelectOneCell
851
852   void  doSelectOneCell(ListHandle listHdl,Cell theCell)
853   {
854     Cell    nextSelectedCell;
855     BooleanmoreCellsInList;
856
857     if(doFindFirstSelectedCell(listHdl,&nextSelectedCell))
858     {
859       while(LGetSelect(true,&nextSelectedCell,listHdl))
860       {
861         if(nextSelectedCell.v != theCell.v)
862           LSetSelect(false,nextSelectedCell,listHdl);
863         else
864           moreCellsInList = LNextCell(true,true,&nextSelectedCell,listHdl);
865       }
866
867       LSetSelect(true,theCell,listHdl);
868     }
869   }
870
871   // ############################################################################## doMakeCellVisible
872
873   void  doMakeCellVisible(ListHandle listHdl,Cell newSelection)
874   {
875     Rect    visibleRect;
876     SInt16 dRows;
877
878     visibleRect = (*listHdl)->visible;
879
880     if(!(PtInRect(newSelection,&visibleRect)))
881     {
882       if(newSelection.v > visibleRect.bottom - 1)
883         dRows = newSelection.v - visibleRect.bottom + 1;
884       else if(newSelection.v < visibleRect.top)
885         dRows = newSelection.v - visibleRect.top;
886
887       LScroll(0,dRows,listHdl);
888     }
889   }
890
891   // ############################################################################## doResetTypeSelection
892
893   void  doResetTypeSelection(void)
894   {
895     gTSString[0] = 0;
896     gTSLastListHit = NULL;
897     gTSLastKeyTime = 0;
898     gTSResetThreshold = 2 * LMGetKeyThresh();
899     if(gTSResetThreshold > kMaxKeyThresh)
900       gTSResetThreshold = kMaxKeyThresh;
901   }
902
903   // ############################################################################## doRotateCurrentList
904
905   void  doRotateCurrentList(void)
906   {
907     WindowPtr       windowPtr;
908     ListHandle      oldListHdl, newListHdl;
909
910     windowPtr = FrontWindow();
911     if(((WindowPeek) windowPtr)->windowKind != dialogKind)
912       return;
913
```

```
914     oldListHdl = gCurrentListHdl;
915     newListHdl = (ListHandle) (*gCurrentListHdl)->refCon;
916     gCurrentListHdl = newListHdl;
917
918     doDrawActiveListBorder(oldListHdl);
919     doDrawActiveListBorder(newListHdl);
920   }
921
922   // ########################################################################## doDrawListsBorders
923
924   void  doDrawListsBorders(ListHandle textListHdl,ListHandle pictListHdl)
925   {
926     PenState oldPenState;
927     Rect     borderRect;
928
929     GetPenState(&oldPenState);
930     PenSize(1,1);
931
932     borderRect = (*textListHdl)->rView;
933     InsetRect(&borderRect,-1,-1);
934     FrameRect(&borderRect);
935
936     borderRect = (*pictListHdl)->rView;
937     InsetRect(&borderRect,-1,-1);
938     FrameRect(&borderRect);
939
940     SetPenState(&oldPenState);
941   }
942
943   // ############################################################### doDrawActiveListBorder
944
945   void  doDrawActiveListBorder(ListHandle listHdl)
946   {
947     PenState oldPenState;
948     Rect     borderRect;
949
950     GetPenState(&oldPenState);
951     PenSize(2,2);
952
953     borderRect = (*listHdl)->rView;
954     borderRect.right += kScrollBarWidth;
955     InsetRect(&borderRect,-4,-4);
956
957     if(listHdl == gCurrentListHdl  && (*listHdl)->lActive)
958       PenPat(&qd.black);
959     else
960       PenPat(&qd.white);
961
962     FrameRect(&borderRect);
963
964     SetPenState(&oldPenState);
965   }
966
967   // ########################################################################## doDisplaySelections
968
969   void  doDisplaySelections(void)
970   {
971     ListsRecHandle listsRecHdl;
972     ListHandle     textListHdl, pictListHdl;
973     SInt16         nextLine = 15, cellIndex;
974     Cell           theCell;
975     Str255         theString;
976     SInt16         offset, dataLen;
977
978     listsRecHdl = (ListsRecHandle) GetWRefCon(FrontWindow());
979     textListHdl = (*listsRecHdl)->textListHdl;
980     pictListHdl = (*listsRecHdl)->pictListHdl;
981
982     HideWindow(FrontWindow());
983     SetPort(gWindowPtr);
984
985     MoveTo(10,nextLine);
986     DrawString("\pTIMBER:");
987     MoveTo(120,nextLine);
988     DrawString("\pTOOL:");
989
990     for(cellIndex=0;cellIndex < (*textListHdl)->dataBounds.bottom;cellIndex++)
```

```
991     {
992       SetPt(&theCell,0,cellIndex);
993       if(LGetSelect(false,&theCell,textListHdl))
994       {
995         LGetCellDataLocation(&offset,&dataLen,theCell,textListHdl);
996         LGetCell((Ptr) theString + 1,&dataLen,theCell,textListHdl);
997         theString[0] = (SInt8) dataLen;
998
999         MoveTo(10,nextLine += 15);
1000        DrawString(theString);
1001      }
1002    }
1003
1004    SetPt(&theCell,0,0);
1005    LGetSelect(true,&theCell,pictListHdl);
1006    GetIndString(theString,rListCellPictTitles,theCell.v + 1);
1007    MoveTo(120,30);
1008    DrawString(theString);
1009  }
1010
1011  // ############################################################################## doAdjustMenus
1012
1013  void  doAdjustMenus(void)
1014  {
1015    MenuHandle fileMenuHdl, demoMenuHdl;
1016
1017    fileMenuHdl = GetMenuHandle(mFile);
1018    demoMenuHdl = GetMenuHandle(mDemonstration);
1019
1020    if(((WindowPeek) FrontWindow())->windowKind == dialogKind)
1021    {
1022      DisableItem(fileMenuHdl,0);
1023      DisableItem(demoMenuHdl,0);
1024    }
1025    else
1026    {
1027      EnableItem(fileMenuHdl,0);
1028      EnableItem(demoMenuHdl,0);
1029    }
1030
1031    DrawMenuBar();
1032  }
1033
1034  // ################################################################################ doMenuChoice
1035
1036  void  doMenuChoice(SInt32 menuChoice)
1037  {
1038    SInt16 menuID, menuItem;
1039    Str255 itemName;
1040    SInt16 daDriverRefNum;
1041
1042    menuID = HiWord(menuChoice);
1043    menuItem = LoWord(menuChoice);
1044
1045    if(menuID == 0)
1046      return;
1047
1048    switch(menuID)
1049    {
1050      case mApple:
1051        if(menuItem == iAbout)
1052          SysBeep(10);
1053        else
1054        {
1055          GetMenuItemText(GetMenuHandle(mApple),menuItem,itemName);
1056          daDriverRefNum = OpenDeskAcc(itemName);
1057        }
1058        break;
1059
1060      case mFile:
1061        if(menuItem == iQuit)
1062          gDone = true;
1063        break;
1064
1065      case mDemonstration:
1066        if(menuItem == iDialog)
1067        {
```

```
1068            SetPort(gWindowPtr);
1069            EraseRect(&gWindowPtr->portRect);
1070            doCreateDialogWithLists();
1071          }
1072          break;
1073      }
1074
1075      HiliteMenu(0);
1076    }
1077
1078    // ############################################################## doDrawDialogDefaultButton
1079
1080    void  doDrawDialogDefaultButton(DialogPtr dialogPtr)
1081    {
1082      WindowPtroldPort;
1083      PenState oldPenState;
1084      SInt16    itemType;
1085      Handle    itemHandle;
1086      Rect      itemRect;
1087      SInt16    buttonOval;
1088
1089      GetPort(&oldPort);
1090      GetPenState(&oldPenState);
1091
1092      GetDialogItem(dialogPtr,iOK,&itemType,&itemHandle,&itemRect);
1093      SetPort((*(ControlHandle) itemHandle)->contrlOwner);
1094      InsetRect(&itemRect,-4,-4);
1095      buttonOval = (itemRect.bottom - itemRect.top) / 2 + 2;
1096
1097      if((*(ControlHandle) itemHandle)->contrlHilite == 255)
1098        PenPat(&qd.gray);
1099      else
1100        PenPat(&qd.black);
1101
1102      PenSize(3,3);
1103      FrameRoundRect(&itemRect,buttonOval,buttonOval);
1104
1105      SetPenState(&oldPenState);
1106      SetPort(oldPort);
1107    }
1108
1109    // ############################################################################################
1110
1111    // ############################################################################################
1112    // LDEF.c                Custom List Definition Function for Lists Demonstration Program
1113    // ############################################################################################
1114    //
1115    // The default list definition function supports the display of unstyled text only. The
1116    // default list definition function is used by the text list (the list at the left of
1117    // the dialog box) in the Lists demonstration program.
1118    //
1119    // The list at the right of the dialog box in the Lists demonstration program displays
1120    // icons.  This custom list definition function is used by that list.
1121    //
1122    // ############################################################################################
1123
1124    // ..... includes
1125
1126    #include <ToolUtils.h>
1127    #include <Lists.h>
1128    #include <LowMem.h>
1129
1130    // ..... function prototypes
1131
1132    pascal void  main           (SInt16,Boolean,Rect *,Cell,SInt16,SInt16,ListHandle);
1133    void         doLDEFDraw     (Boolean,Rect *,Cell,SInt16,ListHandle);
1134    void         doLDEFHighlight(Rect *);
1135
1136    // ############################################################################## main
1137
1138    pascal void main(SInt16 message,Boolean selected,Rect *cellRect,Cell theCell,
1139                     SInt16 dataOffset,SInt16 dataLen,ListHandle theList)
1140    {
1141      switch(message)
1142      {
1143        case lDrawMsg:
1144          doLDEFDraw(selected,cellRect,theCell,dataLen,theList);
```

```
1145        break;
1146
1147      case lHiliteMsg:
1148        doLDEFHighlight(cellRect);
1149        break;
1150    }
1151  }
1152
1153  // ############################################################################## doLDEFDraw
1154
1155  void  doLDEFDraw(Boolean selected,Rect *cellRect,Cell theCell,SInt16 dataLen,
1156                   ListHandle theList)
1157  {
1158    GrafPtr   oldPort;
1159    RgnHandleoldClip;
1160    PenState oldPenState;
1161    Rect      drawRect;
1162    PicHandlepictureHdl;
1163
1164    GetPort(&oldPort);
1165    SetPort((*theList)->port);
1166
1167    oldClip = NewRgn();
1168    GetClip(oldClip);
1169
1170    GetPenState(&oldPenState);
1171    PenNormal();
1172
1173    EraseRect(cellRect);
1174
1175    drawRect = *cellRect;
1176
1177    if(dataLen == sizeof(PicHandle))
1178    {
1179      LGetCell((Ptr) &pictureHdl,&dataLen,theCell,theList);
1180      DrawPicture(pictureHdl,&drawRect);
1181    }
1182
1183    if(selected)
1184      doLDEFHighlight(cellRect);
1185
1186    SetPort(oldPort);
1187
1188    SetClip(oldClip);
1189    DisposeRgn(oldClip);
1190    SetPenState(&oldPenState);
1191  }
1192
1193  // ############################################################################ doLDEFHighlight
1194
1195  void  doLDEFHighlight(Rect *cellRect)
1196  {
1197    UInt8 hiliteVal;
1198
1199    hiliteVal = LMGetHiliteMode();
1200    BitClr(&hiliteVal,pHiliteBit);
1201    LMSetHiliteMode(hiliteVal);
1202
1203    InvertRect(cellRect);
1204  }
1205
1206  // ##############################################################################################
```

Demonstration Program Comments

When this program is run, the user should open the dialog box by choosing the Dialog With Lists item in the Demonstration menu. With the dialog open, the user should manipulate the two lists in the 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.
- Shift-clicking and dragging in the text list to make contiguous multiple cell selections. (Note that the picture list does not allow multiple cell selections.)
- Command-clicking and dragging in the text 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 in the text list outside a block of multiple cell selections, including between two fairly widely separated discontiguous selected cells.
- Double-clicking on a cell in the active list.
- 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.
- Pressing the Shift-key as well as the Up-Arrow and Down-Arrow keys, noting that this results in multiple cell selections in the text list (but not in the picture 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.
- When the text list 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.

When the dialog is dismissed by either clicking on the OK button or double-clicking a cell in the active list, the user should note that the text or picture title of the selected cells are displayed in a window opened by the program.

#define

Lines 71-76 define constants relating to menu IDs and menu items. Lines 77-86 define constants relating to menu bar, window, dialog, string and picture resources, and to dialog box items. Lines 87-89 define constants relating to character codes returned by the Up Arrow, Down Arrow, and Tab keys. Line 91 defines a constant used in the type selection routines. Lines 92-93 defines constants for the resource IDs of default and custom list definition functions.

#typedef

Lines 98-102 define a data type which will be used to store the handles to the two list records associated with the two lists created by the program. As will be seen, the handle to this record will be assigned to the refCon field of the dialog box's window record.

Global Variables

gDone controls program termination. gInBackground relates to foreground/background switching. gWindowPtr will be assigned the pointer to the window opened by the program. gCurrentListHandle will be assigned the handle to the list record associated with the currently active list. The remaining four global variables are associated with the type selection routines.

main

The main function initialises the system software managers (Line 164), sets up the menus (Lines 168-178), opens a window and sets the text size for that window (Lines 182-186), and enters the main event loop (Lines 190-196).

Note that error handling here and in other areas of the program is somewhat rudimentary in that the program simply terminates.

doEvents, doMouseDown

doEvents performs initial event handling.

doMouseDown further processes mouse-down events. Note that, if the event is in the content region of the active window (Line 283), and if that window is the dialog box (Line 285), the application-defined function doInContent is called (Line 286).

doKeyDown

doKeyDown further processes key-down and auto-key events, and is concerned only with key-dow and auto-key events in the dialog box (Line 309).

Line 311 gets the handle to the lists record (note the plural) which, as will be seen, is stored in the refCon field of the dialog box's window record. (The lists record stores the handles to the list records associated with the two lists contained in the dialog box.)

If the key pressed was the Tab key, an application-defined function is called to change the currently active list (Lines 313-314).

If the key pressed was either the Up Arrow or the Down Arrow key (Line 315), and if the current list is the text list (Line 317), a variable which specifies whether multiple cell selections via the keyboard are permitted is set to true (Line 318). If the current list is the picture list, this variable is set to false (Lines 319-320). This variable is then pass as a parameter in a call to an application-defined function which further processes the Arro key event (Line 321).

If the key pressed was neither the Tab key, the Up Arrow key, or the Down Arrow key (Line 323), and if the active list is the text list (Line 325), the event is passed to an application-defined type selection function for further processing (Line 326).

doUpdate

doUpdate handles update events. Between the usual calls to BeginUpdate and EndUpdate, and i the window being updated is the dialog box (Line 341), UpdatetDialog is called to redraw the dialog box (Line 343), an application-defined function is called to draw the bold outline around the default (OK) button (Line 344), and an application-defined function is called to update the lists (Line 345).

doUpdateLists

doUpdateLists updates the lists in the dialog box.

Line 359 gets the handle to the lists record, allowing Lines 361-362 to retrieve the handles to the list records. Lines 366-367 then call LUpdate to redraw those parts of the lists whi need updating and to update the scroll bars if necessary.

Line 369 calls an application-defined function which draws the one-pixel outline around each list. Lines 370-371 call, for each list, an application defined function which either draws or erases (as appropriate) the two-pixel-wide active list border.

doActivate

doActivate handles activate events, and is concerned only with activate events in the dialog box. The function determines whether the window in question is to be activated or deactivat (Line 382) and, if the window is the dialog box (Line 384), passes that determination as a parameter in an application-defined function which further processes the event (Line 385).

doActivateDialog

doActivateDialog further processes the activate event.

Lines 398-400 get the handles to the two list records.

If the dialog box is becoming active (Line 402), the OK and Cancel buttons are highlighted and made active (Lines 404-408) and the two lists are activated (Lines 410-411). (Activating the lists causes previously selected cells to be highlighted and the scroll bars to be shown.) In addition, the two-pixel-wide border is drawn around the active list (Line 413) and an application-defined function is called to reset certain variables used in the type selection routines (Line 414). (This latter is necessary because it is possible that, while the program was in the background, the user changed the "Delay Until Repeat" setting using the Keyboard control panel, a value which is used by the type selection routines.)

If the dialog box is being deactivated (Line 416), the OK and Cancel buttons are unhighlighted and made inactive (Lines 418-422) and the two lists are deactivated (Lines 424-425. (Deactivating the lists causes the selected cells to be unhighlighted and the scroll bars to be hidden.) In addition, the two-pixel-wide border around the active list is erased (Line 427).

doOSEvent

doOSEvent handles operating system events. Recall that the acceptSuspendResumeEvents and doesActivateOnFGSwitch flags in the program's 'SIZE' resource are set. Accordingly, when a suspend/resume event is received when the dialog box is the front window, doActivateDialog is called to ensure that the dialog box is activated on receipt of a resume event.

doInContent

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

Line 461 saves the pointer to the current graphics port. Lines 463-465 get the handles to the two lists. Lines 467-469 get copies of the lists' display rectangles. Since these rectangles do not include the scroll bars, Lines 470-472 expand them to the right to encompass the scroll bar area. Lines 474-475 convert the mouse coordinates to local coordinates to facilitate comparisons with the adjusted list display rectangles.

If the mouse click was in the text list's rectangle and the text list is not the currently active list, or if the mouse click was in the picture list's rectangle and the picture list is not the current list, the application-defined function which changes the active list is called (Lines 477-480).

If the mouse click was in the currently active list (Line 482), the current graphics port is set to that associated with the window in which the list resides (Line 484) before the call to LClick at Line 485. If a click is outside a list's display rectangle and scroll bar, LClick returns immediately, otherwise it handles all user action until the mouse-button is released. In addition, LClick returns true if a double-click occurred. In this program, if a double-click occurred, an application-defined function is called to perform the same action as would apply if the user had clicked the dialog box's OK button (Lines 486-487).

If the click was not in the display rectangle plus scroll bar area of the currently active list (Line 489), DialogSelect is called at Line 491 to determine whether the click was on an enabled item, that is, on either the OK or the Cancel button. If it was, an application-defined function is called to handle that situation.

(As an aside, note that the dialog box contains a user item associated with each list, that the user item rectangles encompass both the list and its scroll bar, that the user item rectangles are retrieved and used to specify the list display rectangles when the lists are created, and that the user items are not activated. An alternative to the foregoing approach to determining whether the mouse-down occurred in a list would be to activate/deactivate the user items along with the dialog's buttons and rely on the DialogSelect call to establish whether the mouse-down occurred in an active list.)

doItemHitInDialog

doItemHitInDialog handles mouse-down events which occur in the dialog box's buttons. It is also called when the user double clicks on a cell in the active list.

If the item clicked was one of the two buttons (Line 504), and if the button was the OK button (or the user double clicked on a cell in the currently active list) (Line 506), an application-defined function is called to draw the current list selections in the window (Line 507). In addition, the list records are disposed of (Lines 511-512), the lists record is disposed of (Line 518), and the dialog is disposed of (Line 514).

Line 516 enables the File and Demonstration menus which, in accordance with user interface guidelines, are disabled while the movable dialog box is open.

doCreateDialogWithLists

doCreateDialogWithLists creates the dialog box and initiates the creation of the associated lists.

Line 531 creates a dialog from the specified resource. Lines 534-536 allocate a relocatable block for the lists record and assign the handle to this record to the refCon field of the dialog's window record. Line 538 sets the dialog's graphics port as the current port and Lines 540-542 set the font for this port as 12 point Chicago.

The calls to GetDialogItem at Lines 544 and 547 are made simply to retrieve the two user item rectangles which will eventually be passed as the rView parameter in the LNew calls which create the lists.

Lines 545 and 548 call the application-defined functions which creates the text list and the picture list. The last three parameters in the function call specify the display rectangle, the number of columns and the resource ID of the list definition function to be used by the list.

The returned handles to the two newly-created lists are assigned to the appropriate fields of the lists record (Lines 550-551).

Line 553 assigns the picture list's handle to the refCon field of the text list's list record and Line 554 assigns the text list's handle to the refCon field of the picture list's list record. This establishes the "linked ring" which will be used to facilitate the rotation of the active list via Tab key presses.

Line 556 establishes the text list as the currently active list.

Line 558 un-hides the dialog box and Line 559 disables the File and Demonstration menus to accord with user-interface guidelines for the display of a movable modal dialog.

doCreateTextList

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

Line 571 sets the rectangle which will be passed as the rDataBnds parameter of the LNew call to specify one column and (initially) no rows. Line 572 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. Line 574 adjusts the list rectangle to reflect the area occupied by the vertical scroll bar.

The call to LNew at Line 576 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.

Line 578 calls an application-defined function which adds rows to the list and stores data in its cells.

Lines 580-581 select the cell at the topmost row as the initially-selected cell. Line 583 calls an application-defined function which initialises certain variables used by the type selection routines. Line 585 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 loop at Lines 595-599 copies 16 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 that the strings are not arranged alphabetically in the 'STR#' resource.

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 way that the cells are arranged in alphabetical order.

Line 610 sets the variable found to false. Line 612 sets the variable totalRows to the number of rows in the list. (In this program, this is initially 0.) Line 613 sets the variable currentRow to -1. The loop entered at Line 615 executes until the variable found is set to true.

Within the loop, Line 617 increments currentRow to 0. The first time this function is called, currentRow will equal totalRows at this point (Lines 618-619) and the loop will thus immediately exit to Line 638. Line 638 adds one row to the list, inserting it before the row specified by currentRow. The list now has one row (cell (0,0)). Line 641 copies the string to this cell. The function then exits, to be called another 14 times by doAddRowsAndDataToTextList.

The second time the function is called, Line 617 again sets currentRow to 0. This time, however, Line 619 does not execute because totalRows is now 1. Thus Line 622 sets the variable aCell to (0,0) and LGetCellDataLocation is then called at Line 623 to retrieve 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 (Line 628). If the incoming string is "less than" the string in cell (0,0), IUMagPString returns -1, in which case:

- The loop exits to Line 638. Line 638 inserts one row before cell(0,0) and the old cell (0,0) thus becomes cell(0,1). The list now contains two rows.

- Line 639 sets cell (0,0) and Line 641 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 called another 13 times by doAddRowsAndDataToTextList.

If, on the other hand, IUMagPString returns 0 (strings equal) or 1 ("incoming" string "greater than" the string in cell (0,0), the loop repeats. At Line 622, currentRow is incremented to 1, which is equal to totalRows. Accordingly, the loop exits immediately, Line 638 inserts a row before cell (0,1) (that is, cell (0,1) is created), Line 641 copies the "incoming" string to that cell, and the function exits, to be called another 13 times by doAddRowsAndDataToTextList.

During the next 13 calls to this function, 13 rows are inserted into the list at a point dependent on the value of the "incoming" string. The ultimate result is an alphabetically ordered list of 15 rows.

doCreatePictList

doCreatePictList, supported by the following function (doAddRowsAndDataToPictList), creates the picture list.

Line 653 sets the rectangle which will be passed as the rDataBnds parameter of the LNew call to specify one column and (initially) no rows. Line 654 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 48 by 48 pixels. Line 656 adjusts the list rectangle to reflect the area occupied by the vertical scroll bar.

The call to LNew at Line 658 creates the list. The parameters specify that the List Manager is to make all cell sizes 48 by 48 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.

Line 660 assigns lOnlyOne to the selFlags field of the list record, 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.

Line 662 calls an application-defined function which adds rows to the list and stores data in its cells.

Lines 664-665 selects the cell at the topmost row as the initially-selected cell. Line 667 returns the handle to the list.

doAddRowsAndDataToPictList

doAddRowsAndDataToPictList adds 6 rows to the picture list and stores a handle to a recorded picture in each of the 6 cells.

Line 678 sets the variable rowNumber to the current number of rows, which is 0.

The loop entered at Line 680 executes 6 times Each time through the loop, the following occurs:

- A picture resource is read in from a 'PICT' resource (Line 682).

- Line 684 inserts a new row in the list at the location specified by the variable rowNumber. Line 685 sets this cell and Line 686 stores the handle to the recorded picture as the cell's data. Line 688 increments the variable rowNumber.

doHandleArrowKey

doHandleArrowKey further processes Down Arrow and Up Arrow key presses. This is the first o
11 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.

Line 696 sets the variable moveToTopBottom to false, which can be regarded as the default. the Command key was also down at the time of the Arrow key press, this variable is set to tr (Lines 698-699).

At Lines 701-702, 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 (Lines 703-704).

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.

Line 713 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 Line 716 if the key pressed was the Down-Arrow. Li 716 calls an application-defined function which finds the last selected cell (which could, o 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, Line 718 calls an application-defined function which 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.

Line 720 then calls an application-defined function which 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, Line 721 calls an application-defined function which, 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 t currently active list and the Shift key was down at the time of the Arrow key press.

After Lines 731-734 execute, 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).

Line 736 calls the application-defined function which 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.

Line 738 calls LGetSelect to check whether the cell specified by the variable newSelection i selected. If it is not, Line 739 selects it. (This check by LGetSelect is advisable becaus for example, the first-selected cell as this function is entered might be cell (0,0), that i the very top row. If the Up-Arrow was pressed in this circumstance, and as will be seen, doFindNewCellLoc (Line 736) 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, Line 741 calls an application-defined function which, 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.

Line 752 extracts the character code from the message field of the event record.

Lines 754-756 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 he 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.

Line 758 stores the handle to the list which is the target of the current key press in gTSLastListHit so as to facilitate the comparison at Line 754 next time the function is called. Line 759 stores the time of the current key press in gTSLastKeyTime for the same purpose. Line 761 increments the length byte of the type selection string and Line 762 adds the received character to the type selection string. That string now holds all the characters received since the last type selection reset.

Line 764 sets the variable theCell to represent the first cell in the list. This is passed as a parameter in the LSearch call at Line 766, 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 Line 768 (which deselects all currently selected cells and selects the specified cell) and Line 770 (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, Line 769 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 callback function at the third parameter to the LSearch call. Note that the default callback function has been replaced by the custom callback function doSearchPartialMatch.

doSearchPartialMatch

doSearchPartialMatch is the custom callback function used 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 IUMagIDString (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 (Line 781). If these conditions do not prevail, doSearchPartialMatch returns 1 (no match found). If these conditions do prevail, IUMagIDString is called (Line 782) with, importantly, both the third and fourth parameters set to the current length of the type selection string. IUMagIDString 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 a
providing the cell's coordinates to the calling function.

Line 795 sets the starting cell for the LGetSelect call at Line 801. 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
the first selected cell if only one cell is selected).

If the call to doFindFirstSelectedCell at Line 808 reveals that no cells are currently
selected, doFindlastSelectedCell simply returns. If, however, doFindFirstSelectedCell finds
selected cell, that cell is passed as the starting cell in the LGetSelect call at Line 810.

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. At Line 810, LGetSelect examine
this cell and returns true, causing the loop to execute. Line 812 thus assigns (0,1) to
theCell and Line 813 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 t
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 Line 812 assigns th
to theCell. Once again, Line 813 increments aCell, this time to (0,5).

This time, however, LGetSelect will return false because neither cell (0,5) nor any cell bel
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 (
the Command key was not down at the time of the Arrow key press) or the top or bottom cell (
the Command key was down).

Line 825 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 (Line 828) and the key pressed was the Up Arrow (Line 830, the n
cell to be selected is the top cell in the list (Line 831). If the key pressed was the Down
Arrow key, the new cell to be selected is the bottom cell in the list (Lines 832-833).

If the Command key was not down and the key pressed was the Up Arrow key (Lines 835-837), an
if the first selected cell is the top cell in the list, the new cell to be selected remains
set at Line 826; otherwise, the new cell to be selected is set as the cell above the first
selected cell (Lines 839-840). If the key pressed was the Down Arrow key (Line 842), and if
the last selected cell is the bottom cell in the list, the new cell to be selected remains a
set at Line 826; otherwise, the new cell to be selected is set as the cell below the last
selected cell (Lines 844-845).

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 (Line 857). Otherwis
the first selected cell is passed as the starting cell in the call to LGetSelect at Line 859

The loop entered at Line 859 will continue to execute while a selected cell exists between t
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 (Lines 861-862). When the loop exits, Line 867 selects the cell specified for
selection.

Note that defeating the de-selection of the cell specified for selection if it is already
selected (Line 861) 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.

Line 878 gets a copy of the rectangle which encompasses the currently visible cells. (Note that his rectangle is in cell coordinates.) Line 880 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 (Lines 882-883). (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 (Lines 884-885).

With the number of cells to scroll, and the direction to scroll, established, LScroll is called at Line 887 to effect the scroll.

doResetTypeSelection

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

Line 895, in effect, makes the type selection string an empty string. Line 896 sets the variable which holds the handle to the list which is the target of the current key press to NULL. Line 897 sets the variable which holds the number of ticks since the last key press to 0. Line 898 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 (Lines 899-900).

doRotateCurrentList

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

Line 914 saves the handle to the currently active list. Line 915 retrieves the handle to the new list to be activated from the refCon field of the currently active list's list record. Line 916 makes the new list the currently active list. Lines 918-919 erase the 2-pixel-wide border around the previously active list and draw the border around the new active list.

doDrawListsBorders

doDrawListsBorders draws the 1-pixel-wide border around each list. The list's display rectangle is copied, expanded by 1 pixel all round, and then drawn.

doDrawActiveListBorder

doDrawActiveListBorder draws and erases the 2-pixel-wide border which identifies the currently active list to the user. The list's display rectangle (which does not include the scroll bar area) is copied, expanded to the right by the scroll bar width, and drawn with a pen pattern of either black or white depending on whether the target list is, or is not, both the current list and currently active.

doDisplaySelections

doDisplaySelections is called when the user dismisses the dialog by either clicking on the OK button or double clicking an item in a list. It displays the user's list selections in the window opened by the program.

Lines 978-980 get the handles to the lists. Lines 982-983 hide the dialog box and set the window's graphics port as the current port. Lines 985-988 draw the list titles in the window.

Lines 990-1002 get the data from the selected cells in the text list and display it in the window. Line 990 sets up a loop which will be traversed once for each cell in the list. Line 992 increments the v coordinate of the variable theCell. If the specified cell is selected (Line 993), LGetCellDataLocation is called to get the length of the data in the cell (Line 995), LGetCell is called to get the cell's data into a Str255 variable (Line 996), the length byte of this variable is set (Line 1002), and the string is drawn in the window (Lines 999-1000).

Lines 1004-1008 get the selected cell in the picture list and display the title of the selected picture. Line 1004 sets the starting cell for the LGetSelect search initiated at Line 1005. The cell identified by LGetSelect is used to index a string in the picture titles 'STR#' resource, which is then read in and drawn (Lines 1006-1008).

doAdjustMenus

doAdjustMenus and doMenuChoice enable and disable menus as appropriate and handle menu choices. Note that, at Lines 1165-1072, choosing the item in the Demonstration menu causes the window to be erased and the function which creates the dialogs and lists to be called.

doDrawDialogDefaultButton

doDrawDialogDefaultButton draws the bold outline around the default (OK) button in the dialo box.

Custom List Definition Function

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.

Lines 1164-1165 save the current drawing environment and set the graphics port. Line 1171 sets the pen size, mode and pattern to the defaults. Line 1173 erases the cell rectangle.

Lines 1175 gets a copy of the 48 pixel by 48 pixel cell rectangle.

Line 1177 checks whether the cell's data is 4 bytes long (the size of a handle to a picture record). If it is, LGetCell is called at Line 1179 to get the cells's data into the variabl pictureHdl and DrawPicture is called at Line 1180 to draw the picture. (Recall that the 'PICT' resources have been made non-purgeable. Hence there are no calls to HNoPurge and HPurge.)

If the lDrawMsg message indicated that the cell was selected, the cell highlighting functior is called (Lines 1183-1184).

Lines 1186-1190 restore the saved drawing environment.

doLDEFHighlight

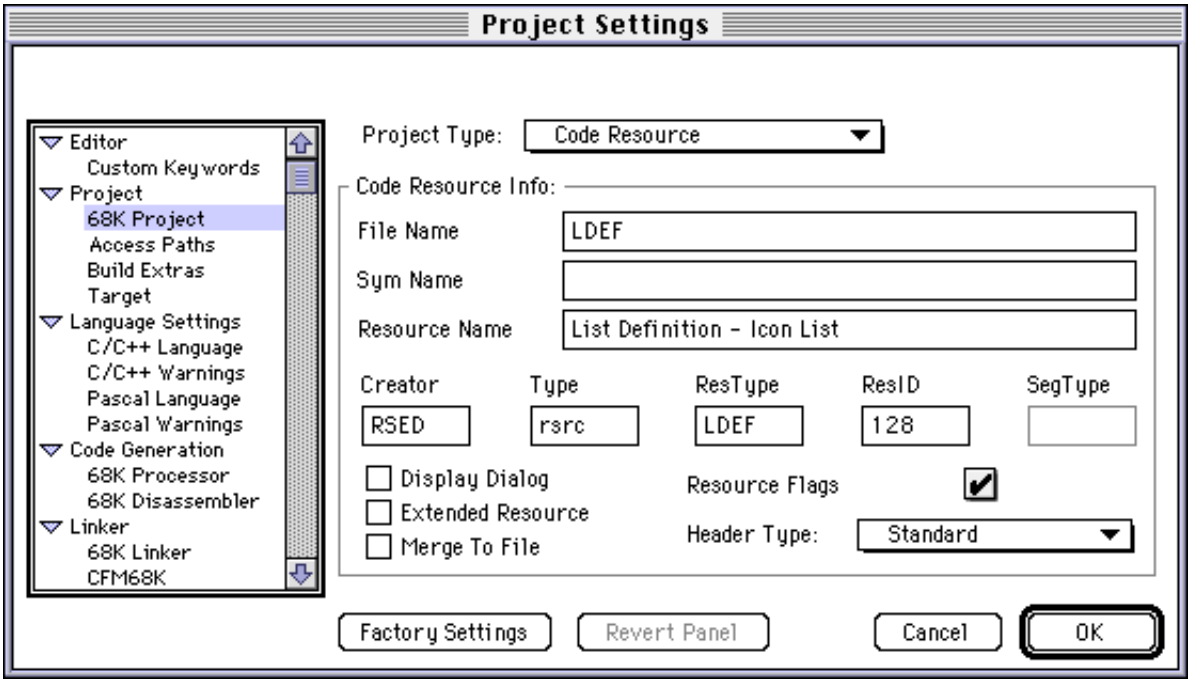
doLDEFHighlight handles the lHiliteMsg message. Lines 1199-1201 will cause the highlight colour to be used if this is possible. (A copy of the value in the low memory global HiliteMode is acquired, BitClr is called to clear the highlight bit, and HiliteMode is set t this new value.) Either way, Line 1203 will either highlight the cell or, on a black and white display, simply invert its pixels.

Creating the LDEF Resource

Creating the LDEF resource means creating a code resource. The Code Resource Projects section of the Creating Mac OS Projects chapter of the CodeWarrior manual Targetting Mac OS is therefore relevant. In brief, to create an LDEF resource using source code such as that at Lines 1111-1206:

- Create a new project in the normal way, adding the source code file and the library MacOS.lib to the project.

- Choose **Project Settings** from the **Edit** menu. Then click 68K Project to bring up the project settings panel. Set the Project Type to Code Resource, enter a File Name and Resource Name as required, enter LDEF as the ResType, enter the ResID (resource ID number) as required, set the Header Type as Standard, and set the Resource Flags Locked and Preload. The project panel should then appear as shown in the Project Settings window below. Note that entering a Resource Name is optional.



- Click on 68K Processor to bring up the processor settings panel. In the Code Model pop-up menu, choose Small.

- Click on 68K Linker to bring up the linker settings panel. Select the Link Single Segment checkbox.

- Click on OK and then choose **Make** from the **Project** menu. The code resource is built and saved to the project folder.

- Within ResEdit, open the project folder. Then open the code resource file (titled LDEF, or whatever was entered in the File Name field in the Project Preferences panel). A ResEdit window opens showing the '**LDEF**' resource icon. Open your program's resource file within ResEdit and copy the '**LDEF**' resource to it.