18 Version 1.2 (Frozen)

LISTS AND CUSTOM LIST DEFINITION FUNCTIONS

Includes Demonstration Program ListsPascal

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, those 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 procedure** (see below), because the default list definition procedure only supports the display of text.

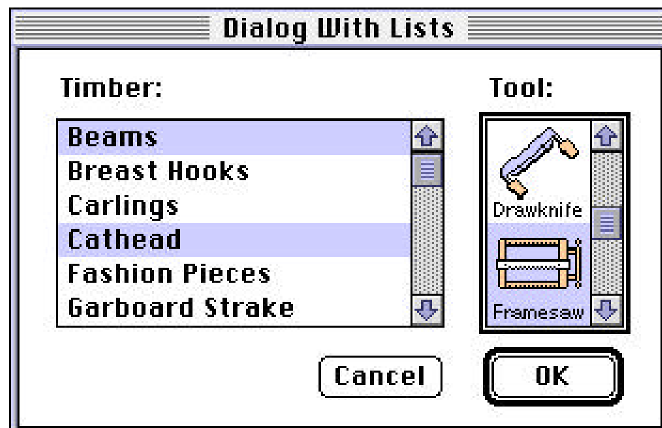


FIG 1 - DIALOG BOX WITH TWO LISTS

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.

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:

---

[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.

- **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`	`integer`	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`	`longint`	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.

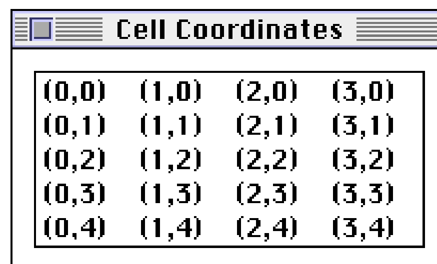


FIG 2 - COORDINATES OF CELLS

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

The ListRec Data Type

The list record is defined by the ListRec data type:

```
type

  ListRec = record
      rView:         Rect;
      port:          GrafPtr;
      indent:        Point;
      cellSize:      Point;
      visible:       ListBounds;
      vScroll:       ControlRef;
      hScroll:       ControlRef;
      selFlags:      SInt8;
      lActive:       boolean;
      lReserved:     SInt8;
      listFlags:     SInt8;
      clikTime:      longint;
      clikLoc:       Point;
      mouseLoc:      Point;
      lClickLoop:    ListClickLoopUPP;
      lastClick:     Cell;
      refCon:        longint;
      listDefProc:   Handle;
      userHandle:    Handle;
      dataBounds:    ListBounds;
      cells:         DataHandle;
      maxIndex:      integer;
      cellArray:     array [0..0] of integer;
  end;

  ListPtr = ^ListRec;
  ListHandle = ^ListPtr;
  ListRef = 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 procedure 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 `nil` if the list does not have a vertical scroll bar.

`hScroll` A handle to the horizontal scroll bar, or `nil` 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 `nil` 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 procedure.

`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:

```
type
  DataArray = packed array [0..32000] of char;
```

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`:

```
function LNew(var rView: Rect; var dataBounds: ListBounds; cSize: Point;
              theProc: integer; theWindow: WindowRef; drawIt: boolean;
              hasGrow: boolean; scrollHoriz: boolean; scrollVert: boolean): ListRef;
```

`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 procedure, 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 procedure to use for the list. To use the default list definition procedure, 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.

`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.

---

[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).

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 nil 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:

```
gCurrentListHdl : ListHandle;

procedure DoFindPreviousListInRing;

var
listHdl : ListHandle;

begin
listHdl := gCurrentListHdl;

while(ListHandle(listHdl^^.refCon) <> gCurrentList) do
  listHdl := ListHandle(listHdl^^.refCon);

gCurrentListHdl := listHdl;
end;
```

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 Procedures

As previously stated, the default list definition procedure 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 procedure. After writing a list definition procedure, you must compile it as a resource of type `'LDEF'` and store it in the resource fork of the application that uses the procedure.

Your custom list definition procedure must be defined like this:

```
procedure ListDef(lMessage: integer; lSelect: boolean; var lRect: Rect; lCell: Cell;
                  lDataOffset: integer; lDataLen: integer; lHandle: ListRef);
```

[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.

Messages Sent by List Manager

In essence, the sole requirement of your list definition procedure 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 procedure 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 procedure 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 procedures do not need to perform any action in response to the `lInitMsg` message.

Responding to the Draw Message

The list definition procedure 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 procedure 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:

```
procedure DoLDEFHighlight(var cellRect : Rect);

var
hiliteVal : ByteParameter;

begin
hiliteVal := LMGetHiliteMode;
BitClr(Ptr(@hiliteVal), pHiliteBit);
LMSetHiliteMode(hiliteVal);

InvertRect(cellRect);
end;
```

Responding to the Close Message

The List Manager sends your list definition procedure the `lCloseMsg` immediately before disposing of the memory occupied by list. Your list definition procedure 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 Procedure

```
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

```
type
  ListRef = ListHandle;
  Cell = Point;
  ListBounds = Rect;
  DataArray = packed array [0..32000] of CHAR;
  DataPtr = ^DataArray;
  DataHandle = ^DataPtr;

function ListSearch(aPtr: Ptr; bPtr: Ptr; aLen: integer; bLen: integer): integer;
```

List Record

```
  ListRec = record
    rView:         Rect;
    port:          GrafPtr;
    indent:        Point;
    cellSize:      Point;
    visible:       ListBounds;
    vScroll:       ControlRef;
    hScroll:       ControlRef;
    selFlags:      SInt8;
    lActive:       boolean;
    lReserved:     SInt8;
    listFlags:     SInt8;
    clikTime:      longint;
    clikLoc:       Point;
    mouseLoc:      Point;
    lClickLoop:    ListClickLoopUPP;
    lastClick:     Cell;
    refCon:        longint;
    listDefProc:   Handle;
    userHandle:    Handle;
    dataBounds:    ListBounds;
    cells:         DataHandle;
    maxIndex:      integer;
    cellArray:     array [0..0] of integer;
  end;

  ListPtr = ^ListRec;
  ListHandle = ^ListPtr;
  ListRef = ListHandle;
```

Routines

Creating and Disposing of Lists

```
function   LNew(var rView: Rect; var dataBounds: ListBounds; cSize: Point; theProc: integer;
           theWindow: WindowRef; drawIt: boolean; hasGrow: boolean; scrollHoriz: boolean;
           scrollVert: boolean): ListRef;
procedure  LDispose(lHandle: ListRef);
```

Adding and Deleting Rows and Columns

```
function   LAddColumn(count: integer; colNum: integer; lHandle: ListRef): integer;
function   LAddRow(count: integer; rowNum: integer; lHandle: ListRef): integer;
procedure  LDelColumn(count: integer; colNum: integer; lHandle: ListRef);
procedure  LDelRow(count: integer; rowNum: integer; lHandle: ListRef);
```

Determining or Changing a Selection

```
function   LGetSelect(next: boolean; var theCell: Cell; lHandle: ListRef): boolean;
procedure  LSetSelect(setIt: boolean; theCell: Cell; lHandle: ListRef);
```

Accessing and Manipulating Data Cells

```
procedure  LSetCell(dataPtr: UNIV Ptr; dataLen: integer; theCell: Cell; lHandle: ListRef);
procedure  LAddToCell(dataPtr: UNIV Ptr; dataLen: integer; theCell: Cell; lHandle: ListRef);
procedure  LClrCell(theCell: Cell; lHandle: ListRef);
procedure  LGetCell(dataPtr: UNIV Ptr; var dataLen: integer; theCell: Cell;
           lHandle: ListRef);
procedure  LGetCellDataLocation(var offset: integer; var len: integer; theCell: Cell; lHandle:
ListRef);
```

Responding to Events

```
function   LClick(pt: Point; modifiers: integer; lHandle: ListRef): boolean;
procedure  LUpdate(theRgn: RgnHandle; lHandle: ListRef);
procedure  LActivate(act: boolean; lHandle: ListRef);
```

Modifying a List's Appearance

```
procedure  LDraw(theCell: Cell; lHandle: ListRef);
procedure  LSetDrawingMode(drawIt: boolean; lHandle: ListRef);
procedure  LScroll(dCols: integer; dRows: integer; lHandle: ListRef);
procedure  LAutoScroll(lHandle: ListRef);
```

Searching For a List Containing a Particular Item

```
function   LSearch(dataPtr: UNIV Ptr; dataLen: integer; searchProc: ListSearchUPP;
           var theCell: Cell; lHandle: ListRef): boolean;
```

Changing the Size of Cells and Lists

```
procedure  LSize(listWidth: integer; listHeight: integer; lHandle: ListRef);
procedure  LCellSize(cSize: Point; lHandle: ListRef);
```

Getting Information About Cells

```
function   LNextCell(hNext: boolean; vNext: boolean; var theCell: Cell; lHandle: ListRef):
           boolean;
procedure  LRect(var cellRect: Rect; theCell: Cell; lHandle: ListRef);
function   LLastClick(lHandle: ListRef): Cell;
```

Demonstration Program

```
1  { ######################################################################################
2  // ListsPascal.p
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 procedure.
12 //
13 // The picture list uses a custom list definition procedure.  The source code for the
14 // custom list definition procedure is at the file LDEFPascal.p in the LDEFPascal 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 procedure
50 //    used by the picture list.
51 //
52 // •  A 'SIZE' resource with the acceptSuspendResumeEvents, doesActivateOnFGSwitch, and
53 //    is32BitCompatible flags set.
54 //
55 // ###################################################################################### }
56
57 program ListsPascal(input, output);
58
59 { ........................................................ include the following Universal Interfaces }
60
61 uses
62
63   Windows, Fonts, Menus, TextEdit, Quickdraw, Dialogs, QuickdrawText, Processes, Types,
64   Memory, Events, TextUtils, ToolUtils, OSUtils, Devices, Lists, LowMem, SegLoad, Sound;
65
66 { ...................................................................... define the following constants }
67
68 const
69
70 mApple = 128;
```

```
71    iAbout = 1;
72   mFile = 129;
73    iQuit = 11;
74   mDemonstration = 131;
75    iDialog = 1;
76
77   rMenubar = 128;
78   rWindow = 128;
79   rDialog = 129;
80    iOK = 1;
81    iCancel = 2;
82    iUserItemText = 3;
83    iUserItemPict = 4;
84   rListCellStrings = 128;
85   rListCellPicts = 128;
86   rListCellPictTitles = 129;
87
88   kUpArrow = $1e;
89   kDownArrow = $1f;
90   kTab = $09;
91   kScrollBarWidth = 15;
92   kMaxKeyThresh = 120;
93
94   kSystemLDEF = 0;
95   kCustomLDEF = 128;
96
97   kMaxLong = $7FFFFFFF;
98
99   { .................................................................................... user-defined types }
100
101  type
102
103  ListsRec = record
104    textListHdl : ListRef;
105    pictListHdl : ListRef;
106    end;
107
108  ListsRecPtr = ^ListsRec;
109  ListsRecHandle = ^ListsRecPtr;
110
111  { ...................................................................................... global variables }
112
113  var
114
115  gDone : boolean;
116  gInBackground : boolean;
117  gWindowPtr : WindowPtr;
118  gCurrentListHdl : ListRef;
119  gTSString : string;
120  gTSResetThreshold : integer;
121  gTSLastKeyTime : longint;
122  gTSLastListHit : ListRef;
123
124  menubarHdl : Handle;
125  menuHdl : MenuHandle;
126  eventRec : EventRecord;
127
128  { ####################################################################### DoInitManagers }
129
130  procedure DoInitManagers;
131
132    begin
133    MaxApplZone;
134    MoreMasters;
135
136    InitGraf(@qd.thePort);
137    InitFonts;
138    InitWindows;
139    InitMenus;
140    TEInit;
141    InitDialogs(nil);
142
143    InitCursor;
144    FlushEvents(everyEvent, 0);
145    end;
146      {of procedure DoInitManagers}
147
```

```
148  { ############################################################ DoDrawDialogDefaultButton }
149
150  procedure DoDrawDialogDefaultButton(theDialogPtr : DialogPtr);
151
152    var
153    oldPort : WindowPtr;
154    oldPenState : PenState;
155    itemType : integer;
156    itemHandle : Handle;
157    itemRect : Rect;
158    buttonOval : integer;
159
160    begin
161    GetPort(oldPort);
162    GetPenState(oldPenState);
163
164    GetDialogItem(theDialogPtr, iOK, itemType, itemHandle, itemRect);
165    SetPort(ControlHandle(itemHandle)^^.contrlOwner);
166    InsetRect(itemRect, -4, -4);
167    buttonOval := (itemRect.bottom - itemRect.top) div 2 + 2;
168
169    if (ControlHandle(itemHandle)^^.contrlHilite = 255) then
170      PenPat(qd.gray)
171    else
172      PenPat(qd.black);
173
174    PenSize(3, 3);
175    FrameRoundRect(itemRect, buttonOval, buttonOval);
176
177    SetPenState(oldPenState);
178    SetPort(oldPort);
179    end;
180      {of procedure DoDrawDialogDefaultButton}
181
182  { ########################################################### DoAddRowsAndDataToPictList }
183
184  procedure DoAddRowsAndDataToPictList(pictListHdl : ListRef; pictListID : integer);
185
186    var
187    rowNumber, pictIndex : integer;
188    pictureHdl : PicHandle;
189    theCell : Cell;
190
191    begin
192    rowNumber := pictListHdl^^.dataBounds.bottom;
193
194    for pictIndex := pictListID to (pictListID + 5) do
195      begin
196      pictureHdl := GetPicture(pictIndex);
197
198      rowNumber := LAddRow(1, rowNumber, pictListHdl);
199      SetPt(theCell, 0, rowNumber);
200      LSetCell(@pictureHdl, sizeof(PicHandle), theCell, pictListHdl);
201
202      rowNumber := rowNumber + 1;
203      end;
204    end;
205      {of procedure DoAddRowsAndDataToPictList}
206
207  { #################################################################### DoCreatePictList }
208
209  function DoCreatePictList(theDialogPtr : DialogPtr; listRect : Rect;
210                          numCols, lDef : integer) : ListRef;
211
212    var
213    dataBounds : Rect;
214    cellSize : Point;
215    pictListHdl : ListRef;
216    theCell : Cell;
217
218    begin
219    SetRect(dataBounds, 0, 0, numCols, 0);
220    SetPt(cellSize, 48, 48);
221
222    listRect.right := listRect.right - kScrollBarWidth;
223
224    pictListHdl := LNew(listRect, dataBounds, cellSize, lDef, theDialogPtr, true,
```

```
225                                 false, false, true);
226
227   pictListHdl^^.selFlags := lOnlyOne;
228
229   DoAddRowsAndDataToPictList(pictListHdl, rListCellPicts);
230
231   SetPt(theCell, 0, 0);
232   LSetSelect(true, theCell, pictListHdl);
233
234   DoCreatePictList := pictListHdl;
235   end;
236     {of function DoCreatePictList}
237
238 { ######################################################### DoAddTextItemAlphabetically }
239
240 procedure DoAddTextItemAlphabetically(listHdl : ListRef; theString : string);
241
242   var
243   found : boolean;
244   totalRows, currentRow, cellDataOffset, cellDataLength : integer;
245   aCell : Cell;
246
247   begin
248   found := false;
249
250   totalRows := listHdl^^.dataBounds.bottom - listHdl^^.dataBounds.top;
251   currentRow := -1;
252
253   while not (found) do
254     begin
255     currentRow := currentRow + 1;
256     if (currentRow = totalRows) then
257       found := true
258     else begin
259       SetPt(aCell, 0, currentRow);
260       LGetCellDataLocation(cellDataOffset, cellDataLength, aCell, listHdl);
261
262       MoveHHi(Handle(listHdl^^.cells));
263       HLock(Handle(listHdl^^.cells));
264
265       if (IUMagPString(Ptr(longint(@theString) + 1),
266             (Ptr(longint(@listHdl^^.cells) + cellDataOffset)),
267                 integer(theString[0]), cellDataLength, nil) = -1) then
268         begin
269         found := true;
270         end;
271
272       HUnlock(Handle(listHdl^^.cells));
273       end;
274     end;
275
276   currentRow := LAddRow(1, currentRow, listHdl);
277   SetPt(aCell, 0, currentRow);
278
279   LSetCell((Ptr(longint(@theString) + 1)), integer(theString[0]), aCell, listHdl);
280   end;
281     {of procedure DoAddTextAlphabetically}
282
283 { ########################################################## DoAddRowsAndDataToTextList }
284
285 procedure DoAddRowsAndDataToTextList(textListHdl : ListRef; stringListID : integer);
286
287   var
288   stringIndex : integer;
289   theString : string;
290
291   begin
292   for stringIndex := 1 to 15 do
293     begin
294     GetIndString(theString, stringListID, stringIndex);
295     DoAddTextItemAlphabetically(textListHdl, theString);
296     end;
297   end;
298     {of procedure DoAddRowsAndDataToTextList}
299
300 { ################################################################ DoResetTypeSelection }
301
```

```
302  procedure DoResetTypeSelection;
303
304    begin
305    gTSString[0] := char(0);
306    gTSLastListHit := nil;
307    gTSLastKeyTime := 0;
308    gTSResetThreshold := 2 * LMGetKeyThresh;
309    if (gTSResetThreshold > kMaxKeyThresh) then
310      gTSResetThreshold := kMaxKeyThresh;
311    end;
312      {of procedure DoResetTypeSelection}
313
314  { ##################################################################### DoCreateTextList }
315
316  function DoCreateTextList(theDialogPtr : DialogPtr; listRect : Rect;
317                    numCols, lDef : integer) : ListRef;
318
319    var
320    dataBounds : Rect;
321    cellSize : Point;
322    textListHdl : ListRef;
323    theCell : Cell;
324
325    begin
326    SetRect(dataBounds, 0, 0, numCols, 0);
327    SetPt(cellSize, 0, 0);
328
329    listRect.right := listRect.right - kScrollBarWidth;
330
331    textListHdl := LNew(listRect, dataBounds, cellSize, lDef, theDialogPtr,
332                             true, false, false, true);
333
334    DoAddRowsAndDataToTextList(textListHdl, rListCellStrings);
335
336    SetPt(theCell, 0, 0);
337    LSetSelect(true, theCell, textListHdl);
338
339    DoResetTypeSelection;
340
341    DoCreateTextList := textListHdl;
342    end;
343      {of function DoCreateTextList}
344
345  { ######################################################################## DoAdjustMenus }
346
347  procedure DoAdjustMenus;
348
349    var
350    fileMenuHdl, demoMenuHdl : MenuHandle;
351
352    begin
353    fileMenuHdl := GetMenuHandle(mFile);
354    demoMenuHdl := GetMenuHandle(mDemonstration);
355
356    if (WindowPeek(FrontWindow)^.windowKind = dialogKind) then
357      begin
358      DisableItem(fileMenuHdl, 0);
359      DisableItem(demoMenuHdl, 0);
360      end
361    else begin
362      EnableItem(fileMenuHdl, 0);
363      EnableItem(demoMenuHdl, 0);
364      end;
365
366    DrawMenuBar;
367    end;
368      {of procedure DoAdjustMenus}
369
370  { ############################################################## DoCreateDialogWithLists }
371
372  procedure DoCreateDialogWithLists;
373
374    var
375    modalDlgPtr : DialogPtr;
376    listsRecHdl : ListsRecHandle;
377    fontNum, itemType : integer;
378    itemHdl : Handle;
```

```
379     itemRect : Rect;
380     textListHdl, pictListHdl : ListRef;
381
382     begin
383     modalDlgPtr := GetNewDialog(rDialog, nil, WindowPtr(-1));
384     if (modalDlgPtr = nil) then
385       ExitToShell;
386
387     listsRecHdl := ListsRecHandle(NewHandle(sizeof(ListsRec)));
388     if (listsRecHdl = nil) then
389       ExitToShell;
390     SetWRefCon(modalDlgPtr, longint(listsRecHdl));
391
392     SetPort(modalDlgPtr);
393
394     GetFNum('Chicago', fontNum);
395     TextFont(fontNum);
396     TextSize(12);
397
398     GetDialogItem(modalDlgPtr, iUserItemText, itemType, itemHdl, itemRect);
399     textListHdl := DoCreateTextList(modalDlgPtr, itemRect, 1, kSystemLDEF);
400
401     GetDialogItem(modalDlgPtr, iUserItemPict, itemType, itemHdl, itemRect);
402     pictListHdl := DoCreatePictList(modalDlgPtr, itemRect, 1, kCustomLDEF);
403
404     listsRecHdl^^.textListHdl := textListHdl;
405     listsRecHdl^^.pictListHdl := pictListHdl;
406
407     textListHdl^^.refCon := longint(pictListHdl);
408     pictListHdl^^.refCon := longint(textListHdl);
409
410     gCurrentListHdl := textListHdl;
411
412     ShowWindow(modalDlgPtr);
413     DoAdjustMenus;
414
415     end;
416       {of procedure DoCreateDialogWithLists}
417
418   { ########################################################################## DoMenuChoice }
419
420   procedure DoMenuChoice(menuChoice : longint);
421
422     var
423     menuID, menuItem : integer;
424     itemName : string;
425     daDriverRefNum : integer;
426
427     begin
428     menuID := HiWord(menuChoice);
429     menuItem := LoWord(menuChoice);
430
431     if (menuID = 0) then
432       Exit(DoMenuChoice);
433
434     case (menuID) of
435
436       mApple: begin
437         if (menuItem = iAbout) then
438           SysBeep(10)
439         else begin
440           GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
441           daDriverRefNum := OpenDeskAcc(itemName);
442           end;
443         end;
444
445       mFile: begin
446         if (menuItem = iQuit) then
447           gDone := true;
448         end;
449
450       mDemonstration: begin
451         if (menuItem = iDialog) then
452           begin
453           SetPort(gWindowPtr);
454           EraseRect(gWindowPtr^.portRect);
455           DoCreateDialogWithLists;
```

```
456          end;
457        end;
458      end;
459        {of case statement}
460
461    HiliteMenu(0);
462    end;
463      {of procedure DoMenuChoice}
464
465  { ################################################################## DoDisplaySelections }
466
467  procedure DoDisplaySelections;
468
469    var
470    listsRecHdl : ListsRecHandle;
471    textListHdl, pictListHdl : ListRef;
472    nextLine, cellIndex : integer;
473    theCell : Cell;
474    theString : string;
475    offset, dataLen : integer;
476    ignored : boolean;
477
478    begin
479    nextLine := 15;
480    listsRecHdl := ListsRecHandle(GetWRefCon(FrontWindow));
481    textListHdl := listsRecHdl^^.textListHdl;
482    pictListHdl := listsRecHdl^^.pictListHdl;
483
484    HideWindow(FrontWindow);
485    SetPort(gWindowPtr);
486
487    MoveTo(10, nextLine);
488    DrawString('INGREDIENTS:');
489    MoveTo(120, nextLine);
490    DrawString('DICE WITH:');
491
492    for cellIndex := 0 to (textListHdl^^.dataBounds.bottom - 1) do
493      begin
494      SetPt(theCell, 0, cellIndex);
495      if (LGetSelect(false, theCell, textListHdl)) then
496        begin
497        LGetCellDataLocation(offset, dataLen, theCell, textListHdl);
498        LGetCell(Ptr(longint(@theString) + 1), dataLen, theCell, textListHdl);
499        theString[0] := char(dataLen);
500
501        nextLine := nextLine + 15;
502        MoveTo(10, nextLine);
503        DrawString(theString);
504        end;
505      end;
506
507    SetPt(theCell, 0, 0);
508    ignored := LGetSelect(true, theCell, pictListHdl);
509    GetIndString(theString, rListCellPictTitles, theCell.v + 1);
510    MoveTo(120, 30);
511    DrawString(theString);
512    end;
513      {of procedure DoDisplaySelections}
514
515  { ############################################################### DoDrawActiveListBorder }
516
517  procedure DoDrawActiveListBorder(listHdl : ListRef);
518
519    var
520    oldPenState : PenState;
521    borderRect : Rect;
522
523    begin
524    GetPenState(oldPenState);
525    PenSize(2, 2);
526
527    borderRect := listHdl^^.rView;
528    borderRect.right := borderRect.right + kScrollBarWidth;
529    InsetRect(borderRect, -4, -4);
530
531    if ((listHdl = gCurrentListHdl) and listHdl^^.lActive) then
532      PenPat(qd.black)
```

```
533     else
534       PenPat(qd.white);
535
536     FrameRect(borderRect);
537
538     SetPenState(oldPenState);
539     end;
540       {of procedure DoDrawActiveListBorder}
541
542   { ################################################################## DoDrawListsBorders }
543
544   procedure DoDrawListsBorders(textListHdl, pictListHdl : ListRef);
545
546     var
547     oldPenState : PenState;
548     borderRect : Rect;
549
550     begin
551     GetPenState(oldPenState);
552     PenSize(1, 1);
553
554     borderRect := textListHdl^^.rView;
555     InsetRect(borderRect, -1, -1);
556     FrameRect(borderRect);
557
558     borderRect := pictListHdl^^.rView;
559     InsetRect(borderRect, -1, -1);
560     FrameRect(borderRect);
561
562     SetPenState(oldPenState);
563     end;
564       {of procedure DoDrawListsBorders}
565
566   { ################################################################## DoRotateCurrentList }
567
568   procedure DoRotateCurrentList;
569
570     var
571     myWindowPtr : WindowPtr;
572     oldListHdl, newListHdl : ListRef;
573
574     begin
575     myWindowPtr := FrontWindow;
576     if (WindowPeek(myWindowPtr)^.windowKind <> dialogKind) then
577       Exit(DoRotateCurrentList);
578
579     oldListHdl := gCurrentListHdl;
580     newListHdl := ListRef(gCurrentListHdl^^.refCon);
581     gCurrentListHdl := newListHdl;
582
583     DoDrawActiveListBorder(oldListHdl);
584     DoDrawActiveListBorder(newListHdl);
585     end;
586       {of procedure DoRotateCurrentList}
587
588   { ##################################################################### DoFindNewCellLoc }
589
590   procedure DoFindNewCellLoc(listHdl : ListRef; oldCellLoc : Cell; var newCellLoc : Cell;
591                 charCode : UInt8; moveToTopBottom : boolean);
592
593     var
594     listRows : integer;
595
596     begin
597     listRows := listHdl^^.dataBounds.bottom - listHdl^^.dataBounds.top;
598     newCellLoc := oldCellLoc;
599
600     if (moveToTopBottom) then
601       begin
602       if (charCode = kUpArrow) then
603         newCellLoc.v := 0
604       else if (charCode = kDownArrow) then
605         newCellLoc.v := listRows - 1;
606       end
607     else begin
608       if (charCode =  kUpArrow) then
609         begin
```

```
610        if (oldCellLoc.v <> 0) then
611          newCellLoc.v := oldCellLoc.v - 1;
612        end
613      else if (charCode = kDownArrow) then
614        begin
615        if (oldCellLoc.v <> listRows - 1) then
616          newCellLoc.v := oldCellLoc.v + 1;
617        end;
618      end;
619    end;
620      {of procedure DoFindNewCellLoc}
621
622  { ############################################################ DoFindFirstSelectedCell }
623
624  function DoFindFirstSelectedCell(listHdl : ListRef; var theCell : Cell) : boolean;
625
626    var
627    result : boolean;
628
629    begin
630    SetPt(theCell, 0, 0);
631    result := LGetSelect(true, theCell, listHdl);
632
633    DoFindFirstSelectedCell := result;
634    end;
635      {of function DoFindFirstSelectedCell}
636
637  { ############################################################# DoFindLastSelectedCell }
638
639  procedure DoFindLastSelectedCell(listHdl : ListRef; var theCell : Cell);
640
641    var
642    aCell : Cell;
643    moreCellsInList : boolean;
644
645    begin
646    if (DoFindFirstSelectedCell(listHdl, aCell)) then
647      begin
648      while (LGetSelect(true, aCell, listHdl)) do
649        begin
650        theCell := aCell;
651        moreCellsInList := LNextCell(true, true, aCell, listHdl);
652        end;
653      end;
654    end;
655      {of procedure DoFindLastSelectedCell}
656
657  { ################################################################## DoMakeCellVisible }
658
659  procedure DoMakeCellVisible(listHdl : ListRef; newSelection : Cell);
660
661    var
662    visibleRect : Rect;
663    dRows : integer;
664
665    begin
666    visibleRect := listHdl^^.visible;
667
668    if not(PtInRect(newSelection, visibleRect)) then
669      begin
670      if (newSelection.v > visibleRect.bottom - 1) then
671        dRows := newSelection.v - visibleRect.bottom + 1
672      else if (newSelection.v < visibleRect.top) then
673        dRows := newSelection.v - visibleRect.top;
674
675      LScroll(0, dRows, listHdl);
676      end;
677    end;
678      {of procedure DoMakeCellVisible}
679
680  { #################################################################### DoSelectOneCell }
681
682  procedure DoSelectOneCell(listHdl : ListRef; theCell : Cell) ;
683
684    var
685    nextSelectedCell : Cell;
686    moreCellsInList : boolean;
```

```
687
688    begin
689    if (DoFindFirstSelectedCell(listHdl, nextSelectedCell)) then
690      begin
691      while(LGetSelect(true, nextSelectedCell, listHdl)) do
692        begin
693        if (nextSelectedCell.v <> theCell.v) then
694          LSetSelect(false, nextSelectedCell, listHdl)
695        else
696          moreCellsInList := LNextCell(true, true, nextSelectedCell, listHdl);
697        end;
698
699      LSetSelect(true, theCell, listHdl);
700      end;
701    end;
702      {of procedure DoSelectOneCell}
703
704  { ################################################################### DoSearchPartialMatch }
705
706  function DoSearchPartialMatch(searchDataPtr, cellDataPtr : Ptr;
707                     cellDataLen, searchDataLen : integer ) : integer;
708
709    var
710    result : integer;
711
712    begin
713    if ((cellDataLen > 0) and (cellDataLen >= searchDataLen)) then
714      result := IUMagIDString(cellDataPtr, searchDataPtr, searchDataLen, searchDataLen)
715    else
716      result := 1;
717
718    DoSearchPartialMatch := result;
719    end;
720      {of function DoSearchPartialMatch}
721
722  { ###################################################################### DoTypeSelectSearch }
723
724  procedure DoTypeSelectSearch( listHdl : ListRef; var theEvent : EventRecord);
725
726    var
727    newChar : char;
728    theCell : Cell;
729
730    begin
731    newChar := chr(BAnd(theEvent.message, charCodeMask));
732
733    if ((gTSLastListHit <> listHdl) or ((theEvent.when - gTSLastKeyTime) >=
734       gTSResetThreshold) or (integer(gTSString[0]) = 255)) then
735      DoResetTypeSelection;
736
737    gTSLastListHit := listHdl;
738    gTSLastKeyTime := theEvent.when;
739
740    gTSString[0] := char(integer(gTSString[0]) + 1);
741    gTSString[integer(gTSString[0])] := newChar;
742
743    SetPt(theCell, 0, 0);
744
745    if (LSearch(Ptr(longint(@gTSString) + 1), integer(gTSString[0]), @DoSearchPartialMatch,
746                    theCell, listHdl)) then
747      begin
748      LSetSelect(true, theCell, listHdl);
749      DoSelectOneCell(listHdl, theCell);
750      DoMakeCellVisible(listHdl, theCell);
751      end;
752    end;
753      {of procedure DoTypeSelectSearch}
754
755  { ############################################################ DoArrowKeyExtendSelection }
756
757  procedure DoArrowKeyExtendSelection(listHdl : ListRef; charCode : UInt8;
758                       moveToTopBottom : boolean);
759
760    var
761    currentSelection, newSelection : Cell;
762
763    begin
```

```
764    if (DoFindFirstSelectedCell(listHdl, currentSelection)) then
765      begin
766      if (charCode = kDownArrow) then
767        DoFindLastSelectedCell(listHdl, currentSelection);
768
769      DoFindNewCellLoc(listHdl, currentSelection, newSelection, charCode,
770                                 moveToTopBottom);
771
772      if not (LGetSelect(false, newSelection, listHdl)) then
773        LSetSelect(true, newSelection, listHdl);
774
775      DoMakeCellVisible(listHdl, newSelection);
776      end;
777    end;
778      {of procedure DoArrowKeyExtendSelection}
779
780  { ############################################################# DoArrowKeyMoveSelection }
781
782  procedure DoArrowKeyMoveSelection(listHdl : ListRef; charCode : UInt8;
783                      moveToTopBottom : boolean);
784
785    var
786    currentSelection, newSelection : Cell;
787
788    begin
789    if (DoFindFirstSelectedCell(listHdl, currentSelection)) then
790      begin
791      if (charCode = kDownArrow) then
792        DoFindLastSelectedCell(listHdl, currentSelection);
793
794      DoFindNewCellLoc(listHdl, currentSelection, newSelection, charCode,
795                                 moveToTopBottom);
796
797      DoSelectOneCell(listHdl, newSelection);
798      DoMakeCellVisible(listHdl, newSelection);
799      end;
800    end;
801      {of procedure DoArrowKeyMoveSelection}
802
803  { #################################################################### DoHandleArrowKey }
804
805  procedure DoHandleArrowKey(charCode : UInt8; var theEvent : EventRecord;
806                      allowExtendSelect : boolean);
807
808    var
809    moveToTopBottom : boolean;
810
811    begin
812    moveToTopBottom := false;
813
814    if (BAnd(theEvent.modifiers, cmdKey) <> 0) then
815      moveToTopBottom := true;
816
817    if (allowExtendSelect and (BAnd(theEvent.modifiers, shiftKey) <> 0)) then
818      DoArrowKeyExtendSelection(gCurrentListHdl, charCode, moveToTopBottom)
819    else
820      DoArrowKeyMoveSelection(gCurrentListHdl, charCode, moveToTopBottom);
821    end;
822      {of procedure DoHandleArrowKey}
823
824  { ################################################################### DoItemHitInDialog }
825
826  procedure DoItemHitInDialog(myDialogPtr : DialogPtr; itemHit : integer);
827
828    var
829    listsRecHdl : ListsRecHandle;
830
831    begin
832    if ((itemHit = iOK) or (itemHit = iCancel)) then
833      begin
834      if (itemHit = iOK) then
835        DoDisplaySelections;
836
837      listsRecHdl := ListsRecHandle(GetWRefCon(myDialogPtr));
838
839      LDispose(listsRecHdl^^.textListHdl);
840      LDispose(listsRecHdl^^.pictListHdl);
```

```
841      DisposeHandle(Handle(listsRecHdl));
842      DisposeDialog(myDialogPtr);
843
844      DoAdjustMenus;
845      end;
846    end;
847      {of procedure DoItemHitInDialog}
848
849  { ########################################################################## DoInContent }
850
851  procedure DoInContent(var theEvent : EventRecord);
852
853    var
854    oldPort : GrafPtr;
855    listsRecHdl : ListsRecHandle;
856    textListHdl, pictListHdl : ListRef;
857    textListRect, pictListRect, gCurrentListRect : Rect;
858    mouseXY : Point;
859    isDoubleClick : boolean;
860    theDialogPtr : DialogPtr;
861    itemHit : integer;
862
863    begin
864    GetPort(oldPort);
865
866    listsRecHdl := ListsRecHandle(GetWRefCon(FrontWindow));
867    textListHdl := listsRecHdl^^.textListHdl;
868    pictListHdl := listsRecHdl^^.pictListHdl;
869
870    textListRect := listsRecHdl^^.textListHdl^^.rView;
871    pictListRect := listsRecHdl^^.pictListHdl^^.rView;
872    gCurrentListRect := gCurrentListHdl^^.rView;
873    textListRect.right := textListRect.right + kScrollBarWidth;
874    pictListRect.right := pictListRect.right + kScrollBarWidth;
875    gCurrentListRect.right := gCurrentListRect.right + kScrollBarWidth;
876
877    mouseXY := theEvent.where;
878    GlobalToLocal(mouseXY);
879
880    if ((PtInRect(mouseXY, textListRect) and (gCurrentListHdl <> textListHdl)) or
881       (PtInRect(mouseXY, pictListRect) and (gCurrentListHdl <> pictListHdl))) then
882      begin
883      DoRotateCurrentList;
884      end
885    else if (PtInRect(mouseXY, gCurrentListRect)) then
886      begin
887      SetPort(gCurrentListHdl^^.port);
888      isDoubleClick := LClick(mouseXY, theEvent.modifiers, gCurrentListHdl);
889      if (isDoubleClick) then
890        DoItemHitInDialog(FrontWindow, iOK);
891      end
892    else begin
893      if (DialogSelect(theEvent, theDialogPtr, itemHit)) then
894        DoItemHitInDialog(theDialogPtr, itemHit);
895      end;
896
897    SetPort(oldPort);
898
899    end;
900      {of procedure DoInContent}
901
902  { ##################################################################### DoActivateDialog }
903
904  procedure DoActivateDialog(myWindowPtr : WindowPtr; becomingActive : Boolean);
905
906    var
907    listRecsHdl : ListsRecHandle;
908    textListHdl, pictListHdl : ListRef;
909    itemType : integer;
910    itemHdl : Handle;
911    itemRect : Rect;
912
913    begin
914    listRecsHdl := ListsRecHandle(GetWRefCon(myWindowPtr));
915    textListHdl := listRecsHdl^^.textListHdl;
916    pictListHdl := listRecsHdl^^.pictListHdl;
917
```

```
918     if (becomingActive) then
919       begin
920       GetDialogItem(DialogPtr(myWindowPtr), iOK, itemType, itemHdl, itemRect);
921       HiliteControl(ControlHandle(itemHdl), 0);
922       GetDialogItem(DialogPtr(myWindowPtr), iCancel, itemType, itemHdl, itemRect);
923       HiliteControl(ControlHandle(itemHdl), 0);
924       DoDrawDialogDefaultButton(myWindowPtr);
925
926       LActivate(true, textListHdl);
927       LActivate(true, pictListHdl);
928
929       DoDrawActiveListBorder(gCurrentListHdl);
930       DoResetTypeSelection;
931       end
932     else begin
933       GetDialogItem(DialogPtr(myWindowPtr), iOK, itemType, itemHdl, itemRect);
934       HiliteControl(ControlHandle(itemHdl), 255);
935       GetDialogItem(DialogPtr(myWindowPtr), iCancel, itemType, itemHdl, itemRect);
936       HiliteControl(ControlHandle(itemHdl), 255);
937       DoDrawDialogDefaultButton(myWindowPtr);
938
939       LActivate(false, textListHdl);
940       LActivate(false, pictListHdl);
941
942       DoDrawActiveListBorder(gCurrentListHdl);
943       end;
944     end;
945       {of procedure DoActivateDialog}
946
947   { ########################################################################## DoOSEvent }
948
949   procedure DoOSEvent(var theEvent : EventRecord);
950
951     begin
952     case BAnd(BSR(theEvent.message, 24), $000000FF) of
953
954       suspendResumeMessage:
955         begin
956         gInBackground := BAnd(theEvent.message, resumeFlag) = 0;
957         if (WindowPeek(FrontWindow)^.windowKind = dialogKind) then
958           DoActivateDialog(FrontWindow, not (gInBackground));
959         end;
960
961       mouseMovedMessage:
962         begin
963         end;
964       end;
965         {of case statement}
966     end;
967       {of procedure DoOSEvent}
968
969   { ######################################################################### DoActivate }
970
971   procedure DoActivate(var theEvent : EventRecord);
972
973     var
974     myWindowPtr : WindowPtr;
975     becomingActive : boolean;
976
977     begin
978     myWindowPtr := WindowPtr(theEvent.message);
979     becomingActive := (BAnd(theEvent.modifiers, activeFlag) = activeFlag);
980
981     if (WindowPeek(myWindowPtr)^.windowKind = dialogKind) then
982       DoActivateDialog(myWindowPtr, becomingActive);
983     end;
984       {of procedure DoActivate}
985
986   { ###################################################################### DoUpdateLists }
987
988   procedure DoUpdateLists(myWindowPtr : WindowPtr);
989
990     var
991     listsRecHdl : ListsRecHandle;
992     textListHdl, pictListHdl : ListRef;
993
994     begin
```

```
995   listsRecHdl := ListsRecHandle(GetWRefCon(myWindowPtr));
996
997   textListHdl := listsRecHdl^^.textListHdl;
998   pictListHdl := listsRecHdl^^.pictListHdl;
999
1000  SetPort(textListHdl^^.port);
1001
1002  LUpdate(textListHdl^^.port^.visRgn, textListHdl);
1003  LUpdate(pictListHdl^^.port^.visRgn, pictListHdl);
1004
1005  DoDrawListsBorders(textListHdl, pictListHdl);
1006  DoDrawActiveListBorder(textListHdl);
1007  DoDrawActiveListBorder(pictListHdl);
1008  end;
1009    {of procedure DoUpdateLists}
1010
1011 { ############################################################################ DoUpdate }
1012
1013 procedure DoUpdate(var theEvent : EventRecord);
1014
1015   var
1016   myWindowPtr : WindowPtr;
1017
1018   begin
1019   myWindowPtr := WindowPtr(theEvent.message);
1020
1021   BeginUpdate(myWindowPtr);
1022
1023   if (WindowPeek(myWindowPtr)^.windowKind = dialogKind) then
1024     begin
1025     UpdateDialog(myWindowPtr, myWindowPtr^.visRgn);
1026     DoDrawDialogDefaultButton(myWindowPtr);
1027     DoUpdateLists(myWindowPtr);
1028     end;
1029
1030   EndUpdate(myWindowPtr);
1031
1032   end;
1033     {of procedure DoUpdate}
1034
1035 { ########################################################################### DoKeyDown }
1036
1037 procedure DoKeyDown(charCode : UInt8; var theEvent : EventRecord);
1038
1039   var
1040   listsRecHdl : ListsRecHandle;
1041   allowExtendSelect : boolean;
1042
1043   begin
1044   if (WindowPeek(FrontWindow)^.windowKind = dialogKind) then
1045     begin
1046     listsRecHdl := ListsRecHandle(GetWRefCon(FrontWindow));
1047
1048     if (charCode = kTab) then
1049       DoRotateCurrentList
1050     else if ((charCode = kUpArrow) or (charCode = kDownArrow)) then
1051       begin
1052       if (gCurrentListHdl = listsRecHdl^^.textListHdl) then
1053         allowExtendSelect := true
1054       else
1055         allowExtendSelect := false;
1056       DoHandleArrowKey(charCode, theEvent, allowExtendSelect);
1057       end
1058     else begin
1059       if (gCurrentListHdl = listsRecHdl^^.textListHdl) then
1060         DoTypeSelectSearch(listsRecHdl^^.textListHdl, theEvent);
1061       end;
1062     end;
1063   end;
1064     {of procedure DoKeyDown}
1065 { ######################################################################### DoMouseDown }
1066
1067 procedure DoMouseDown(var theEvent : EventRecord);
1068
1069   var
1070   partCode : integer;
1071   myWindowPtr : WindowPtr;
```

```
1072
1073    begin
1074    partCode := FindWindow(theEvent.where, myWindowPtr);
1075
1076    case (partCode) of
1077
1078      inMenuBar: begin
1079        DoAdjustMenus;
1080        DoMenuChoice(MenuSelect(theEvent.where));
1081        end;
1082
1083      inSysWindow: begin
1084        SystemClick(theEvent, myWindowPtr);
1085        end;
1086
1087      inContent: begin
1088        if (myWindowPtr <> FrontWindow) then
1089          begin
1090          if (WindowPeek(FrontWindow)^.windowKind = dialogKind) then
1091             SysBeep(10)
1092          else SelectWindow(myWindowPtr);
1093          end
1094        else begin
1095          if (WindowPeek(FrontWindow)^.windowKind = dialogKind) then
1096            DoInContent(theEvent);
1097          end;
1098        end;
1099
1100      inDrag: begin
1101        if ((WindowPeek(FrontWindow)^.windowKind = dialogKind) and
1102           (WindowPeek(myWindowPtr)^.windowKind <> dialogKind)) then
1103          begin
1104          SysBeep(10);
1105          Exit(DoMouseDown);
1106          end;
1107        DragWindow(myWindowPtr, theEvent.where, qd.screenBits.bounds);
1108        end;
1109      end;
1110        {of statement}
1111    end;
1112      {of procedure DoMouseDown}
1113
1114  { ############################################################################## DoEvents }
1115
1116  procedure DoEvents(var theEvent : EventRecord);
1117
1118    var
1119    charCode : UInt8;
1120
1121    begin
1122    case (theEvent.what) of
1123
1124      mouseDown: begin
1125        DoMouseDown(theEvent);
1126        end;
1127
1128      keyDown, autoKey: begin
1129        charCode := UInt8(BAnd(theEvent.message, charCodeMask));
1130        if (BAnd(theEvent.modifiers, cmdKey) <> 0) then
1131          begin
1132          DoAdjustMenus;
1133          DoMenuChoice(MenuKey(char(charCode)));
1134          end;
1135        DoKeyDown(charCode, theEvent);
1136        end;
1137
1138      updateEvt: begin
1139        DoUpdate(theEvent);
1140        end;
1141
1142      activateEvt: begin
1143        DoActivate(theEvent);
1144        end;
1145
1146      osEvt: begin
1147        DoOSEvent(theEvent);
1148        HiliteMenu(0);
```

```
1149        end;
1150      end;
1151        {of case statement}
1152    end;
1153      {of procedure DoEvents}
1154
1155  { ################################################################# start of main program }
1156
1157  begin
1158
1159    { ............................................................... initialise managers }
1160
1161    DoInitManagers;
1162
1163    { ........................................................... set up menu bar and menus }
1164
1165    menubarHdl := GetNewMBar(rMenubar);
1166    if (menubarHdl = nil) then
1167      ExitToShell;
1168    SetMenuBar(menubarHdl);
1169    DrawMenuBar;
1170
1171    menuHdl := GetMenuHandle(mApple);
1172    if (menuHdl = nil) then
1173      ExitToShell
1174    else
1175      AppendResMenu(menuHdl, 'DRVR');
1176
1177    { ........................................................................ open window }
1178
1179    gWindowPtr := GetNewWindow(rWindow, nil, WindowPtr(-1));
1180    if (gWindowPtr = nil) then
1181        ExitToShell;
1182
1183    SetPort(gWindowPtr);
1184    TextSize(10);
1185
1186    { .................................................................... enter eventLoop }
1187
1188    gDone := false;
1189
1190    while not (gDone) do
1191      begin
1192      if (WaitNextEvent(everyEvent, eventRec, kMaxLong, nil)) then
1193        DoEvents(eventRec);
1194      end;
1195
1196  end.
1197
1198  { ####################################################################################### }
1199
1200  { #######################################################################################
1201  // LDEFPascal.p          Custom List Definition Procedure for Lists Demonstration Program
1202  // #####################################################################################
1203  //
1204  // The default list definition procedure supports the display of unstyled text only. The
1205  // default list definition procedure is used by the text list (the list at the left of
1206  // the dialog box) in the Lists demonstration program.
1207  //
1208  // The list at the right of the dialog box in the Lists demonstration program displays
1209  // icons.  This custom list definition procedure is used by that list.
1210  //
1211  // ###################################################################################### }
1212
1213  unit LDEFPascal;
1214
1215  { ............................................................... unit interface section }
1216
1217  interface
1218
1219  { ........................................... include the following Universal Interfaces }
1220
1221  uses
1222
1223    Quickdraw, QuickdrawText, Types, Events, ToolUtils, OSUtils, Lists, LowMem;
1224
1225  { ................................................................. procedure interfaces }
```

```
1226
1227  {$MAIN}
1228  procedure main(message : integer; selected : Boolean; var cellRect : Rect; theCell : Cell;
1229              dataOffset : integer; dataLen : integer; theList : ListHandle);
1230
1231  procedure DoLDEFDraw(selected : Boolean;  var cellRect : Rect; theCell : Cell;
1232              dataLen : integer; theList : ListHandle);
1233
1234  procedure DoLDEFHighlight(var cellRect : Rect);
1235
1236  { ..................................................................... unit implementation section }
1237
1238  implementation
1239
1240  { ################################################################################## main }
1241
1242  procedure main(message : integer; selected : Boolean; var cellRect : Rect; theCell : Cell;
1243              dataOffset : integer; dataLen : integer; theList : ListHandle);
1244
1245    begin
1246    case (message) of
1247      lDrawMsg:
1248        begin
1249        DoLDEFDraw(selected, cellRect, theCell, dataLen, theList);
1250        end;
1251
1252      lHiliteMsg:
1253        begin
1254        DoLDEFHighlight(cellRect);
1255        end;
1256      end;
1257        {of case statement}
1258    end;
1259      {of procedure main}
1260
1261  { ############################################################################ DoLDEFDraw }
1262
1263  procedure DoLDEFDraw(selected : Boolean;  var cellRect : Rect; theCell : Cell;
1264              dataLen : integer; theList : ListHandle);
1265
1266    var
1267    oldPort : GrafPtr;
1268    oldClip : RgnHandle;
1269    oldPenState : PenState;
1270    drawRect : Rect;
1271    pictureHdl : PicHandle;
1272
1273    begin
1274    GetPort(oldPort);
1275    SetPort(theList^^.port);
1276
1277    oldClip := NewRgn;
1278    GetClip(oldClip);
1279
1280    GetPenState(oldPenState);
1281    PenNormal;
1282
1283    EraseRect(cellRect);
1284
1285    drawRect := cellRect;
1286
1287    if (dataLen = sizeof(PicHandle)) then
1288      begin
1289      LGetCell(@pictureHdl, dataLen, theCell, theList);
1290      DrawPicture(pictureHdl, drawRect);
1291      end;
1292
1293    if (selected) then
1294      DoLDEFHighlight(cellRect);
1295
1296    SetPort(oldPort);
1297
1298    SetClip(oldClip);
1299    DisposeRgn(oldClip);
1300    SetPenState(oldPenState);
1301    end;
1302      {of procedure DoLDEFDraw}
```

```
1303
1304  { ###################################################################### DoLDEFHighlight }
1305
1306  procedure DoLDEFHighlight(var cellRect : Rect);
1307
1308    var
1309    hiliteVal : ByteParameter;
1310
1311    begin
1312    hiliteVal := LMGetHiliteMode;
1313    BitClr(Ptr(@hiliteVal), pHiliteBit);
1314    LMSetHiliteMode(hiliteVal);
1315
1316    InvertRect(cellRect);
1317    end;
1318      {of procedure DoLDEFHighlight}
1319
1320  end.
1321    {of unit LDEFPascal}
1322
1323  { ####################################################################################### }
```

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 b 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 t 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 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 bot the text list and the picture list, this results in the top-most or bottom-most cell bein selected.
- When the text list is the active list, typing the text of a particular cell so as to sele 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 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.

The constant declaration block

Lines 70-75 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 88-90 define constants relating to character codes returned by the Up Arrow, Down Arrow, and Tab keys. Lines 91-92 define constants used in the type selection routines. Lines 94-95 defines constants for the resource IDs of default and custom list definition procedures.

The type declaration block

Lines 103-109 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.

The variable declaration block

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.

The procedure DoDrawDialogDefaultButton

DoDrawDialogDefaultButton draws the bold outline around the default (OK) button in the dialog box.

The procedure DoAddRowsAndDataToPictList

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

Line 192 sets the variable rowNumber to the current number of rows, which is 0.

The loop entered at Line 194 executes 6 times Each time through the loop, the following occurs:

- A picture resource is read in from a 'PICT' resource (Line 196).
- Line 198 inserts a new row in the list at the location specified by the variable rowNumber. Line 199 sets this cell and Line 200 stores the handle to the recorded picture as the cell's data. Line 202 increments the variable rowNumber.

The function DoCreatePictList

DoCreatePictList, supported by the following procedure (DoAddRowsAndDataToPictList), creates the picture list.

Line 219 sets the rectangle which will be passed as the rDataBnds parameter of the LNew call to specify one column and (initially) no rows. Line 220 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 222 adjusts the list rectangle to reflect the area occupied by the vertical scroll bar.

The call to LNew at Line 224 creates the list. The parameters specify that the List Manager is to make all cell sizes 48 by 48 pixels, a custom list definition procedure 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 227 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 229 calls an application-defined function which adds rows to the list and stores data in its cells.

Lines 231-232 selects the cell at the topmost row as the initially-selected cell. Line 234 returns the handle to the list.

The procedure DoAddTextItemAlphabetically

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

Line 248 sets the variable found to false. Line 250 sets the variable totalRows to the number rows in the list. (In this program, this is initially 0.) Line 251 sets the variable currentl to -1. The loop entered at Line 253 executes until the variable found is set to true.

Within the loop, Line 255 increments currentRow to 0. The first time this function is called, currentRow will equal totalRows at this point (Lines 256-257) and the loop will thus immediatel exit to Line 276. Line 276 adds one row to the list, inserting it before the row specified by currentRow. The list now has one row (cell (0,0)). Line 279 copies the string to this cell. ' function then exits, to be called another 14 times by DoAddRowsAndDataToTextList.

The second time the function is called, Line 255 again sets currentRow to 0. This time, howev Line 256 does not execute because totalRows is now 1. Thus Line 259 sets the variable aCell t (0,0) and LGetCellDataLocation is then called at Line 260 to retrieve the offset and length of the data in cell (0,0). This allows the string in this cell to be alphabetically compared witl the "incoming" string (Line 265). If the incoming string is "less than" the string in cell (0,0), IUMagPString returns -1, in which case:

- The loop exits to Line 276. Line 276 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 277 sets cell (0,0) and Line 279 copies the "incoming" string to that cell. The "incoming" string, which was alphabetically "less than" the first string, is thus assigne 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 255, currentRow is incremented to 1, which is equal to totalRows. Accordingly, the loop exits immediately, Line 276 inserts a row before cell (0,1) (that is, cell (0,1) is created), Line 279 copies the "incoming" string to tl 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.

The procedure DoAddRowsAndDataToTextList

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

The loop at Lines 292-296 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 int the list and copies the string to that cell.

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

The procedure DoResetTypeSelection

DoResetTypeSelection resets the global variables which are central to the operation of the typ selection function doTypeSelectSearch.

Line 305, in effect, makes the type selection string an empty string. Line 306 sets the varial which holds the handle to the list which is the target of the current key press to nil. Line : sets the variable which holds the number of ticks since the last key press to 0. Line 308 set: the variable which holds the type selection reset threshold to twice the value stored in the l memory global variable KeyThresh. However, if this value is greater than the value represente by the constant kMaxKeyThresh, the variable is made equal to kMaxKeyThresh (Lines 309-310).

The procedure DoCreateTextList

DoCreateTextList, supported by two previous procedures, creates the text list.

Line 326 sets the rectangle which will be passed as the rDataBnds parameter of the LNew call t specify one column and (initially) no rows. Line 327 sets the variable which will be passed a: the cellSize parameter so as to specify that the List Manager should automatically calculate tl cell size. Line 329 adjusts the list rectangle to reflect the area occupied by the vertical scroll bar.

The call to LNew at Line 331 creates the list. The parameters specify that the List Manager i: to calculate the cell size, the default list definition procedure 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 334 calls an application-defined procedure which adds rows to the list and stores data in its cells.

Lines 336-337 selects the cell at the topmost row as the initially-selected cell. Line 339 calls an application-defined function which initialises certain variables used by the type selection routines. Line 341 returns the handle to the list.

The procedure DoAdjustMenus

DoAdjustMenus enables and disables menus as appropriate.

The procedure DoCreateDialogWithLists

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

Line 383 creates a dialog from the specified resource. Line 387 allocates a relocatable block for the lists record and assigns the handle to this record to the refCon field of the dialog's window record. Line 392 sets the dialog's graphics port as the current port and Lines 394-396 set the font for this port as 12 point Chicago.

The calls to GetDialogItem at Lines 398 and 401 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 399 and 402 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 procedure 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 404-405).

Line 407 assigns the picture list's handle to the refCon field of the text list's list record and Line 408 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 410 establishes the text list as the currently active list.

Line 412 un-hides the dialog box and Line 413 disables the File and Demonstration menus to accord with user-interface guidelines for the display of a movable modal dialog.

The procedure DoMenuChoice

DoMenuChoice handles menu choices. Note that, at Lines 451-455, 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.

The procedure 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 480-482 get the handles to the lists. Lines 484-485 hide the dialog box and set the window's graphics port as the current port. Lines 487-490 draw the list titles in the window.

Lines 492-505 get the data from the selected cells in the text list and display it in the window. Line 492 sets up a loop which will be traversed once for each cell in the list. Line 494 increments the v coordinate of the variable theCell. If the specified cell is selected (Line 495), LGetCellDataLocation is called to get the length of the data in the cell (Line 497), LGetCell is called to get the cell's data into a Str255 variable (Line 498), the length byte of this variable is set (Line 499), and the string is drawn in the window (Lines 502-503).

Lines 507-511 gets the selected cell in the picture list and displays the title of the selected picture. Line 507 sets the starting cell for the LGetSelect search initiated at Line 508. 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 509-511).

The procedure DoDrawActiveListBorder

DoDrawActiveListBorder draws and erases the 2-pixel-wide border which identifies the 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.

The procedure DoDrawListsBorders

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

The procedure DoRotateCurrentList

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

Line 579 saves the handle to the currently active list. Line 580 retrieves the handle to the n list to be activated from the refCon field of the currently active list's list record. Line 58 makes the new list the currently active list. Lines 583-584 erase the 2-pixel-wide border arou the previously active list and draw the border around the new active list.

The procedure DoFindNewCellLoc

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

Line 597 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 600) and the key pressed was the Up Arrow (Line 602, the new cell to be selected is the top cell in the list (Line 603). If the key pressed was the Down Arrow key, the new cell to be selected is the bottom cell in the list (Lines 604-605).

If the Command key was not down and the key pressed was the Up Arrow key (Lines 607-608), and i the first selected cell is the top cell in the list, the new cell to be selected remains as se at Line 598; otherwise, the new cell to be selected is set as the cell above the first selecte cell (Lines 610-611). If the key pressed was the Down Arrow key (Line 613), and if the last selected cell is the bottom cell in the list, the new cell to be selected remains as set at Li 598; otherwise, the new cell to be selected is set as the cell below the last selected cell (Lines 615-616).

The procedure DoFindFirstSelectedCell

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

Line 630 sets the starting cell for the LGetSelect call at Line 631. Since the first paramete 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.

The procedure DoFindLastSelectedCell

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

If the call to DoFindFirstSelectedCell at Line 646 reveals that no cells are currently selecte DoFindlastSelectedCell simply returns. If, however, DoFindFirstSelectedCell finds a selected cell, that cell is passed as the starting cell in the LGetSelect call at Line 648.

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 648, LGetSelect examines this cell and returns true, causing the loop to execute. Line 650 thus assigns (0,1) to theCel and Line 651 increments aCell to (0,2). LGetSelect starts another search using (0,2) as the starting cell. Because cells (0,2) and (0,3) are not selected, LGetSelect advances to cell (0, before it returns. Since it has found another selected cell, LGetSelect again returns true, s the loop executes again. aCell now contains (0,4), and Line 650 assigns that to theCell. Onc again, Line 651 increments aCell, this time to (0,5).

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

The procedure 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 666 gets a copy of the rectangle which encompasses the currently visible cells. (Note that his rectangle is in cell coordinates.) Line 668 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 670-671). (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 672-673).

With the number of cells to scroll, and the direction to scroll, established, LScroll is called at Line 675 to effect the scroll.

The procedure 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 689). Otherwise, the first selected cell is passed as the starting cell in the call to LGetSelect at Line 691.

The loop entered at Line 691 will continue to execute while a selected cell exists between the starting cell specified in the LGetSelect call and the end of the list. Within the loop, if the current LGetSelect starting cell is not the cell specified for selection, that cell is deselected (Lines 693-694). When the loop exits, Line 699 selects the cell specified for selection.

Note that defeating the de-selection of the cell specified for selection if it is already selected (Line 693) 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.

The function 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 "ba" and a cell with the text "Banana" 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 713). If these conditions do not prevail, DoSearchPartialMatch returns 1 (no match found). If these conditions do prevail, IUMagIDString is called (Line 714) 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.

The procedure 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 731 extracts the character code from the message field of the event record.

Lines 733-735 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 737 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 733 next time the function is called. Line 738 stores the time of the current key press in gTSLastKeyTime for the same purpose. Line 740 increments the length byte of the type selection string and Line 741 adds the received character to the type selection string. That string now holds all the characters received since the last type selection reset.

Line 743 sets the variable theCell to represent the first cell in the list. This is passed as parameter in the LSearch call at Line 745, 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 748 (which deselects all currently selected cells and selects the specified cell) and Line 750 (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 749 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.

The procedure DoArrowKeyExtendSelection

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

After Lines 764-767 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 769 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 772 calls LGetSelect to check whether the cell specified by the variable newSelection is selected. If it is not, Line 773 selects it. (This check by LGetSelect is advisable because, for example, the first-selected cell as this function is entered might be cell (0,0), that is, the very top row. If the Up-Arrow was pressed in this circumstance, and as will be seen, DoFindNewCellLoc (Line 769) 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 775 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.

The procedure DoArrowKeyMoveSelection

DoArrowKeyMoveSelection further processes those Arrow key presses which occurred when either list was the 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 789 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 792 if the key pressed was the Down-Arrow. Line 792 calls an application-defined function which finds the last selected cell (which could, of course, well be the same cell as the first selected cell if only one cell is currently selected). Either way, the variable currentSelection will now hold either the only cell currently selected, the first cell selected (if more than one cell is currently selected and the key pressed was the Up Arrow), or the last cell selected (if more than one cell is currently selected and the key pressed was the Down Arrow).

With that established, Line 794 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 797 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 798 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.

The procedure DoHandleArrowKey

DoHandleArrowKey further processes Down Arrow and Up Arrow key presses.

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 812 sets the variable moveToTopBottom to false, which can be regarded as the default. If the Command key was also down at the time of the Arrow key press, this variable is set to true (Lines 814-815).

At Lines 817-818, if the text list is the currently active list, and if the Shift key was down, the application-defined procedure DoArrowKeyExtendSelection is called; otherwise, the application-defined procedure DoArrowKeyMoveSelection is called.

The procedure 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 832), and if the button was the OK button (or the user double clicked on a cell in the active list) (Line 834), an application-defined function is called to draw the current list selections in the window (Line 835). In addition, the list records are disposed of (Lines 837-840), the lists record is disposed of (Line 841), and the dialog is disposed of (Line 842).

Line 844 enables the File and Demonstration menus which, in accordance with human interface guidelines, are disabled while the movable dialog box is open.

The procedure DoInContent

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

Line 864 saves the pointer to the current graphics port. Lines 866-868 get the handles to the two lists. Lines 870-872 get copies of the lists' display rectangles. Since these rectangles do not include the scroll bars, Lines 873-875 expand them to the right encompass the scroll bar area. Lines 877-878 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 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 880-884).

If the mouse click was in the currently active list (Line 885), the current graphics port is set to that associated with the window in which the list resides (Line 887) before the call to LClick at Line 888. 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 889-890).

If the click was not in the display rectangle plus scroll bar area of the active list (Line 892), DialogSelect is called at Line 893 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.)

The procedure DoActivateDialog

DoActivateDialog further processes the activate event.

Lines 914-916 get the handles to the two list records.

If the dialog box is becoming active (Line 918), the OK and Cancel buttons are highlighted and made active (Lines 920-924) and the two lists are activated (Lines 926-927). (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 929) and an application-defined function is called to reset certain variables used in the type selection routines (Line 930). (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 932), the OK and Cancel buttons are unhighlighted and made inactive (Lines 933-937) and the two lists are deactivated (Lines 939-940. (Deactivating the lists causes the selected cells to be unhighlighted and the scroll bars to b hidden.) In addition, the two-pixel-wide border around the active list is erased (Line 942).

The procedure 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.

The procedure DoActivate

DoActivate handles activate events, and is concerned only with activate events in the dialog b The function determines whether the window in question is to be activated or deactivated (Line 979) and, if the window is the dialog box (Line 981), passes that determination as a parameter an application-defined function which further processes the event (Line 982).

The procedure DoUpdateLists

DoUpdateLists updates the lists in the dialog box.

Line 995 gets the handle to the lists record, allowing Lines 997-998 to retrieve the handles t the list records. Lines 1002-1003 then call LUpdate to redraw those parts of the lists which need updating and to update the scroll bars if necessary.

Line 1005 calls an application-defined function which draws the one-pixel outline around each list. Lines 1006-1007 call, for each list, an application defined function which either draws erases (as appropriate) the two-pixel-wide active list border.

The procedure DoUpdate

DoUpdate handles update events. Between the usual calls to BeginUpdate and EndUpdate, and if window being updated is the dialog box (Line 1023), UpdateDialog is called to redraw the dialo box (Line 1025), an application-defined function is called to draw the bold outline around the default (OK) button (Line 1026), and an application-defined function is called to update the lists (Line 1027).

The procedure DoKeyDown

DoKeyDown further processes key-down and auto-key events, and is concerned only with key-down auto-key events in the dialog box (Line 1044).

Line 1046 gets the handle to the lists record (note the plural) which, as will be seen, is sto in the refCon field of the dialog box's window record. (The lists record stores the handles t 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 1048-1049).

If the key pressed was either the Up Arrow or the Down Arrow key (Line 1050), and if the curre list is the text list (Line 1052), a variable which specifies whether multiple cell selections via the keyboard are permitted is set to true (Line 1053). If the current list is the picture list, this variable is set to false (Line 1054). This variable is then passed as a parameter a call to an application-defined procedure which further processes the Arrow key event (Line 1056).

If the key pressed was neither the Tab key, the Up Arrow key, or the Down Arrow key (Line 1058), and if the active list is the text list (Line 1059), the event is passed to an application-defined type selection procedure for further processing (Line 1060).

The procedures DoMouseDown and DoEvents

DoMouseDown further processes mouse-down events. Note that, if the event is in the content region of the active window (Line 1087), and if that window is the dialog box (Line 1090), the application-defined function DoInContent is called (Line 1096).

DoEvents performs initial event handling.

The main program block

The main function initialises the system software managers (Line 1161), sets up the menus (Lines 1165-1175), opens a window and sets the text size for that window (Lines 1179-1184), and enters the main event loop (Lines 1188-1194).

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

Custom List Definition Procedure

The procedure main

The List Manager sends a list definition procedure four types of messages in the message parameter. The main function calls the appropriate function to handle each message type.

The procedure DoLDEFDraw

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

Lines 1274-1275 save the current drawing environment and set the graphics port. Line 1281 sets the pen size, mode and pattern to the defaults. Line 1283 erases the cell rectangle.

Lines 1285 gets a copy of the 48 pixel by 48 pixel cell rectangle.

Line 1287 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 1289 to get the cells's data into the variable pictureHdl and DrawPicture is called at Line 1290 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 function is called (Lines 1293-1294).

Lines 1296-1300 restore the saved drawing environment.

The procedure DoLDEFHighlight

DoLDEFHighlight handles the lHiliteMsg message. Lines 1312-1314 will cause the highlight colour to be used if this is possible. (A copy of the value at the low memory global HiliteMode is acquired, BitClr is called to clear the highlight bit, and HiliteMode is set to this new value.) Either way, Line 1316 will either highlight the cell or, on a black and white display, simply invert its pixels.

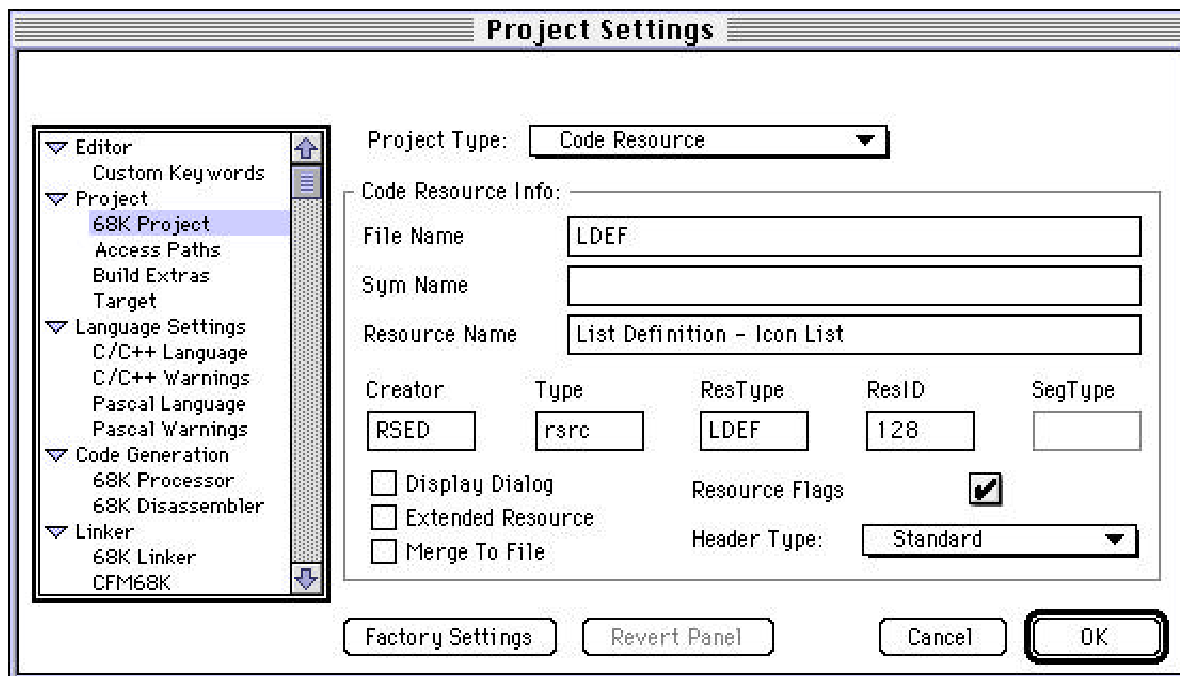
Creating the LDEF Resource

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.