A Virtual List Box Implementation with VLIST

Kyle Marsh
Microsoft Developer Network Technology Group

Abstract

The standard list box control in Microsoft® Windows™ can contain a limited number of items. Applications may need to implement list boxes that exceed this limitation. This article introduces a custom control called VLIST, which is a *virtual list box* capable of displaying millions of items. Part 1 of the article provides instructions for using the VLIST virtual list box, and documents the styles and messages that VLIST supports. Part 2 describes how VLIST was implemented, and discusses the problems encountered and decisions made during the implementation; readers who would like to modify VLIST or implement their own virtual list box may find this section interesting. For a working example of VLIST, see the Microsoft Developer Network CDthe topic listing in the Index window was implemented with VLIST.

Introduction

The standard list box control in Microsoft® Windows™ version 3.1 has a number of limitations:

It can display a maximum of 32,767 items.

It is limited to 32K of item data. Therefore, list boxes that store strings can contain a maximum of 5440 items, and list boxes that don't store strings (owner-drawn list boxes implemented without the LBS_HASSTRINGS style) can contain a maximum of 8160 items.

It is limited to 64K of string data. Therefore, a list box with an average string length of 30 bytes per item can store a maximum of 2184 items.

Future versions of Windows may support standard list boxes that stretch these limits; however, an application designed for Windows version 3.1 must use a custom control for this purpose. The custom control discussed in this article is called a *virtual list box* and can contain millions of items.

Is a Virtual List Box Necessary?

Before you decide to use a list box with many (say, more than a thousand) items, you should consider the following two issues:

Appropriateness. Is a list box the best way to display these items? If the list has 10,000 items and only 20 items are visible, the user can see only a small percentage of the list (0.2 percent) at one time. To get from the top of the list to the bottom of the list, the user must page down 500 times. The user can jump to the top or bottom of the list with one action, but finding a particular item in the list is usually a time-consuming task. You should explore better ways to present the data before deciding to use a list box.

Overhead. How much overhead is required to fill the list box? Consider a database with 100,000 names. To add all the names to the list box, the application must read through the entire data file, extract and format the names, and add them to the list box one name at a time. The list box, in turn, must allocate memory for each new item and copy the data into this memory. The user is usually interested in a small percentage of the items, and effort spent retrieving and storing all items is wasted overhead. A large amount of data can result in such a huge overhead that performance

becomes unacceptable.

If you decide that a list box is the best way to present the data, a virtual list box will help reduce the amount of overhead.

A virtual list box appears to contain many items, but actually contains only enough items to fill the list box display. When the user scrolls the list box up or down one line, the virtual list box loads the next or previous item to be displayed. When the user scrolls the list box up or down one page, the virtual list box loads the previous or next page of items. A virtual list box that loads and unloads items in this manner is also called a *paging list box* (or *swapping list box*).

Advantages of the Virtual List Box

A virtual list box has three advantages over the Windows standard list box control:

Unlimited size. A virtual list box can accommodate up to four billion items. For example, you can create a virtual list box that contains all of the names in the phone book. Of course, paging through a few million names is probably the last thing a user wants to do, so a virtual list box with that many items is just as inappropriate as a standard list box.

Small overhead. The only overhead required involves displaying the current page to the user. By keeping overhead to a minimum, a virtual list box can appear to contain many thousands of items without incurring a performance penalty. Instead of reading hundreds or thousands of records and putting them into a list box, the application reads only a few records at a time. For example, a multiuser database application strives to keep locked records at a minimum. In this case, a virtual list box is more efficient than a standard list box, even when displaying a small number of items.

Reasonable memory requirements. A virtual list box does not require a large amount of memory because it loads a limited number of items at one time. Even if a large amount of memory is available, either physically or through Windows virtual memory modes (enhanced mode and Windows NT™), you should keep memory usage to a minimum to improve overall system performance.

Disadvantages of the Virtual List Box

A virtual list box does have two disadvantages:

Slower scrolling speed. A standard list box stores all items in memory and is ready to display them instantly. A virtual list box must add new items and remove old items as it scrolls the list. Depending on how complicated the logic is for retrieving the new items, scrolling may result in a noticeable performance degradation.

Increased coding requirements. In a standard list box, the application usually scans items once, as they are added to the list. In a virtual list box, the application must be able to scan items forward and backward from any given starting point in the list.

Part 1. Using VLIST

The VLIST custom control implements a single-selection, single-column virtual list box. VLIST manages functions such as filling and scrolling the list box, but relies on the application to retrieve items for the list box. This leads to a control situation that differs considerably from the control situation for a standard list box. With a standard list box, the application adds and removes items as needed. When items are added, the list can be scrolled without any further information from the application. With

VLIST, the application sends the VLIST control a message indicating that it is ready to receive messages. From that point on, the VLIST control sends messages to the application to request information.

VLIST requires two types of information:

A long integer indicating the number of items in the VLIST control.

An item to be used by the VLIST control.

How Many Items?

To control the scroll box (also known as the scrollbar thumb), VLIST must know the number of items in the list. A standard list box obtains the number of items in the list by counting items as they are added to the list. Applications that use the VLIST control, however, may not be able to obtain this information easily. Without this information, VLIST is unable to control the scroll range and cannot position the scroll box correctly.

For example, let's assume that the VLIST control contains customer names that start with *M*,*N*, or *O*. Most databases cannot tell the application how many names fit this criterion without actually counting the names. Counting could be a time-consuming operation, so the application may decide to tell VLIST that it does not know how many items are going to be in the list.

If an application cannot determine the number of items in the virtual list box, VLIST places the scroll box in the middle of the scroll bar. This allows the user to scroll up or down the list one page or line at a time using the scroll bar or scroll arrows. The user can also move the scroll box to a specific position in the scroll bar. In this case, VLIST asks the application for the item closest to the specified location and positions the list accordingly. The application may or may not be able to provide the information; in either case, the scroll box returns to the middle of the scroll bar. Because the placement of the scroll box should represent the current position in the list, placing the scroll box in the middle of the bar is likely to mislead and confuse the user. Whenever possible, an application should determine the number of items in the virtual list box to avoid this situation.

VLIST can be modified to present a different appearance when the number of items in the list is not known. For example, VLIST could have buttons for line up, line down, page up, and page down instead of a scroll bar (see Figure 1).

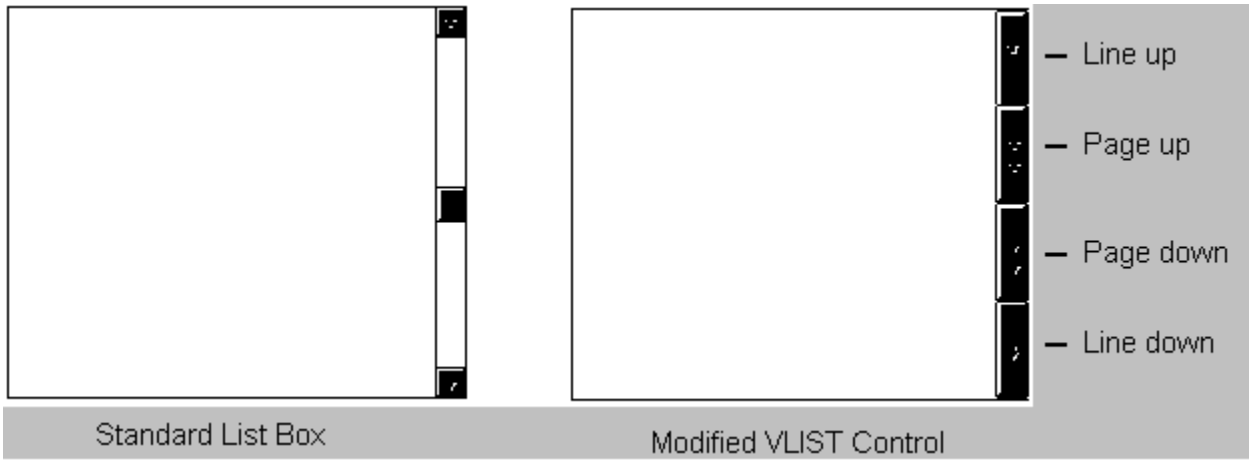

Figure 1. Customizing the VLIST Control

This new interface solves the scroll box placement problem by omitting the scroll box. Because the control does not look like a standard list box, the user does not expect it to act like one. The trade-off is the additional documentation this interface would require. (On the other hand, a scroll bar that does not operate correctly also requires additional documentation.)

What Item?

A standard list box identifies an individual item by its index number. The item at the top of the list is item 0, the next item is item 1, and so on. In some cases, it would be better if the application used a 32-bit data value instead of the index number to identify an item. For example, in a list of customer names that begin with *M*, it may be impractical for the application to get item number 1000. The application can start at the first item and get the next item until it reaches item number 1000, but it may incur a severe performance penalty in doing so. In this case, it would be easier for the application to get the item based on a 32-bit data value associated with an item. If this data value is the record identifier for the item in the database, the application can go directly to the record associated with this item. For convenience, VLIST can use either the item's index number or its 32-bit data value (which is supplied by the application) to identify the itemthe application makes this choice. (See "The Item Identifier" section, later in this article.)

Using the index number

How an application identifies an item affects how VLIST responds to certain actions. When an application uses the item's index number as an identifier, VLIST can use the index number to position the scroll box accurately. For example, when item number 50 is displayed in the first line of a list box that contains 100 items, VLIST can place the scroll box in the middle of the scroll bar. The index number also allows VLIST to respond to a user's request to jump to a particular location in the list. If there are 100 items in the list and the user wants to see 75 percent through the list, VLIST asks the application for item number 75.

Using the counter

When the application requests that VLIST use the 32-bit data value to identify an item, VLIST cannot use the item index to position the scroll box, but will use its own counter instead. This counter is initialized when the top or bottom of the list is displayed. When the user scrolls up or down the list, VLIST increases or decreases the counter as needed. This gives a good approximation of where the scroll box should be located. However, when the list moves without scrolling (for example, in response to a search), VLIST places the scroll box in the middle of the scroll bar because it does not know where to position it. As discussed previously, this placement gives the user incorrect feedback about the actual list position and must be documented.

An application that uses VLIST should use index numbers to identify items whenever possible. When the user requests to jump to a particular location, for example, by moving the scroll box 75 percent of the way down the scroll bar, VLIST asks the application for the item that is 75 percent of the way through the list. If the application cannot fulfill this request, VLIST will not move the list to jump to the user-requested location. For example, many databases cannot easily go to the record that is 75 percent of the way through a file, so database applications may not be able to comply with the user's request. All applications should be able to respond to the 0 percent (top of list) and 100 percent (bottom of list) requests.

Communications Between VLIST and the Application

The VLIST control communicates with the application through messages. The application starts the communication by sending an initialization message to VLIST. From that point on, both VLIST and the application communicate through messages. The application sends its messages to the VLIST window; VLIST sends its messages to its parent window. When VLIST is used in a dialog box, VLIST sends its messages to the dialog manager, which passes them to the dialog procedure.

The Item Identifier

Some messages accept or return a particular item. As discussed in the "What Item?" section, an item can be identified either through an index number or through a 32-bit data value supplied by the application. By default, the index number is used unless the application:

Specifies the VLB_USEDATAVALUES style, which forces the use of data values.

Returns VLB_ERR in the **nStatus** field of the **VLBSTRUCT** structure in response to a VLB_RANGE message. (See the next section for a discussion of **VLBSTRUCT**.) VLIST uses VLB_RANGE to obtain the number of items in the list. If the number of items is not known, both VLIST and the application use 32-bit data values to identify items.

The Virtual List Box Structure

Because messages are limited to two parameters, *wParam* and *lParam*, and VLIST needs to send and receive more information than can fit within these parameters, VLIST defines a structure called **VLBSTRUCT** to pass information to and from the application. **VLBSTRUCT** has the following format:

```
typedef struct _VLBStruct {
   int   nStatus;
   LONG  lData;
   LONG  lIndex;
   LPSTR lpTextPointer;
   LPSTR lpFindString;
} VLBSTRUCT;
```

where:

nStatus is set by the application to indicate the status of the request by VLIST. Values are VLB_OK (successful), VLB_ERR (an error occurred), and VLB_ENDOFFILE (end of file was reached).

lData is the 32-bit data value associated with the item. This value is used both by VLIST, as it makes a request, and by the application, as it fulfills a request. VLIST always associates the data value with the item in the list, so there is no need to send another message to make this association.

lIndex is the logical index for the item. This is a 32-bit data value because the virtual list box can contain more than 32,767 items. If the application cannot identify the item index, this number has no meaning and is ignored by VLIST.

lpTextPointer is a pointer to the text string for the item. This value is filled in by the application as it passes an item to VLIST. If the list box is an owner-drawn list box and does not have the VLBS_HASSTRINGS style, this value has no meaning and is ignored by VLIST.

lpFindString is a pointer to a string to search for in the list box. This value is filled in by VLIST when it sends a VLB_FINDSTRING, VLB_FINDSTRINGEXACT, or VLB_SELECTSTRING

message to the application. This value has no meaning at other times and is ignored by VLIST.

VLIST Styles

The VLBS_ styles described in this section have the same meanings as their LBS_ counterparts in Windows version 3.1, except for VLBS_NOINTEGRALHEIGHT, which VLIST treats differently. VLIST also adds a new style called VLBS_USEDATAVALUES for identifying items. For more information on the LBS_ counterparts of these styles, see the "List Box Controls" technical article.

VLBS_HASSTRINGS

Specifies that the VLIST control contains items consisting of strings. VLIST maintains the memory and pointers for the strings so the application can use the VLB_GETTEXT message to retrieve the text for a particular item. By default, all VLIST controls except those that are owner-drawn have this style. An application can create an owner-drawn VLIST control either with or without this style.

VLBS_NOINTEGRALHEIGHT

Specifies that the size of VLIST is exactly the size specified by the application when it created the VLIST control. Normally, VLIST sizes a control so that it does not display partial items. Please note that VLIST does not support the display of partial items even when this style is used. If a VLIST control has the VLBS_NOINTEGRALHEIGHT style, VLIST leaves the space for a partial line blank.

VLBS_NOREDRAW

Specifies that the appearance of the VLIST control is not updated when changes are made. This style can be changed at any time by sending a WM_SETREDRAW message.

VLBS_NOTIFY

Specifies that VLIST notifies the application with an input message whenever the user clicks or double-clicks an item.

VLBS_OWNERDRAWFIXED

Specifies that the owner of VLIST is responsible for drawing the contents of the VLIST control and that the items in the control are of the same height. The owner window receives a WM_MEASUREITEM message when the list box is created and a WM_DRAWITEM message when a visual aspect of the list box has changed.

VLBS_USEDATAVALUES

Specifies that VLIST always uses an item's 32-bit data value to identify the item. The index number is ignored.

Note This style uses a bit that the standard Windows list box does not use, so it may conflict with a new list box style in the future.

VLBS_USETABSTOPS

Allows VLIST to recognize and expand tab characters when drawing its strings. The default tab positions are 32 dialog box units.

Note A dialog box unit is a horizontal or vertical distance. One horizontal dialog box unit is equal to one-fourth of the current dialog box base width unit. The dialog box base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the current dialog box base units in pixels.

VLBS_WANTKEYBOARDINPUT

Specifies that the owner of the VLIST control receives WM_VKEYTOITEM or WM_CHARTOITEM messages whenever the user presses a key and the list box has the input focus. This allows an application to perform special processing on the keyboard input. If a list box has the VLBS_HASSTRINGS style, the list box can receive WM_VKEYTOITEM messages but not WM_CHARTOITEM messages. If a list box does not have the VLBS_HASSTRINGS style, the list box can receive WM_CHARTOITEM messages as well as WM_VKEYTOITEM messages.

VLIST Messages

VLIST supports the VLB_ messages described below. Most of these are equivalents of the LB_ messages used for standard list boxes.

Note VLIST also accepts the LB_ messages, but these messages treat the virtual control as a standard list box and may cause unpredictable behavior in VLIST.

VLB_FINDITEM

```
wParam = 0;
lpvlb = (LPVLBSTRUCT) lParam;
```

VLIST sends this message to the application to request the item specified in *lpvlb*. If the application can locate the item, it returns VLB_OK in *lpvlb->nStatus* and the item in *lpvlb*. Otherwise, it returns VLB_ERR in *lpvlb->nStatus*.

VLB_FINDPOS

```
wParam = 0;
lpvlb = (LPVLBSTRUCT) lParam;
```

VLIST sends this message to the application to request the item that is located a specified percentage down the list. The percentage is specified in the *lpvlp->lIndex* and *lpvlb->lData* fields. If the application can locate the item, it returns VLB_OK in *lpvlb->nStatus* and the item in *lpvlb*. Otherwise, it returns VLB_ERR in *lpvlb->nStatus*.

VLB_FINDSTRING, VLB_FINDSTRINGEXACT, VLB_SELECTSTRING

```
wParam = 0;
lpvlb = (LPVLBSTRUCT) lParam;
```

The application sends VLIST these messages to find the string specified in *lpvlb->lpszFindString*:

VLB_FINDSTRING searches for an item that begins with the characters in the specified string.

VLB_FINDSTRINGEXACT searches for an item that matches the specified string.

VLB_SELECTSTRING searches for an item that begins with the characters in the specified string and selects that item.

To locate the string, VLIST sends the message on to the application with the same *wParam* and *lParam* values. The application is free to implement any search technique it needs at this point (that is, the application can deviate from the descriptions of VLB_FINDSTRING, VLB_FINDSTRINGEXACT, and VLB_SELECTSTRING above). If the application locates the requested item, it returns VLB_OK in *lpvlb->nStatus*, and VLIST returns the item identifier to the application. (If the message is VLB_SELECTSTRING, VLIST also selects the item.) If the application cannot find the requested item, it returns a status of VLB_ERR in *lpvlb->nStatus*, and VLIST returns VLB_ERR to the application.

VLB_FIRST

```
wParam = 0;
lpvlb = (LPVLBSTRUCT) lParam;
```

VLIST sends this message to the application to request the first item in the list. If the application can locate the item, it returns VLB_OK in *lpvlb->nStatus*

revised and the item in *lpvlb*. Otherwise, it returns VLB_ERR in *lpvlb->nStatus*.

VLB_GETCOUNT

```
wParam = 0;
lParam = 0L;
```

The application sends this message to VLIST to request the number of items in the virtual list box. VLIST returns the number of items in the virtual list, or 1L if it does not know the number.

VLB_GETCURSEL

```
wParam = 0;
lParam = 0L;
```

The application sends this message to VLIST to request the ID of the currently selected item. VLIST returns the item identifier for the selected item, or 1L if no item is currently selected.

VLB_GETHORIZONTALEXTENT

```
wParam = 0;
lParam = 0L;
```

The application sends this message to VLIST to request the horizontal extent for the virtual list box. VLIST returns the horizontal extent of the list box in pixels.

VLB_GETITEMDATA

```
wParam = 0;
lpvlb = (LPVLBSTRUCT) lParam;
```

The application sends this message to VLIST to retrieve the 32-bit data value associated with the item specified in *lpvlb*. If the item is not currently displayed in the list box, VLIST sends the VLB_GETITEMDATA message to the application to request the data value. If the application locates the item, it returns VLB_OK in *lpvlb->nStatus* and the data value in *lpvlb->lData*; if it cannot locate the item, it returns VLB_ERR in *lpvlb->nStatus*. VLIST returns the 32-bit data value of the specified item, or VLB_ERR if it cannot obtain this value.

VLB_GETITEMHEIGHT

```
wParam = 0;
lParam = 0L;
```

The application sends this message to VLIST to request the height of items in the list box. VLIST returns the height in pixels.

VLB_GETITEMRECT

```
wParam = 0;
lpvlb = (LPVLBSTRUCT) lParam;
```

The application sends this message to VLIST to request the client rectangle for the item specified in *lpvlb*. If the item is visible, VLIST returns VLB_OK. If the item is not currently visible, VLIST returns VLB_ERR and a rectangle with 1 values for all corner coordinates.

VLB_GETLINES

```
wParam = 0;
lParam = 0L;
```

The application sends this message to VLIST to request the number of visible lines in the list box. VLIST returns the number of visible items in the list.

VLB_GETTEXT

```
wParam = 0;
lpvlb = (LPVLBSTRUCT) lParam;
```

The application sends this message to VLIST to retrieve the string for the item specified in *lpvlb*. If the specified item is not visible, VLIST sends this message to the application to request the text. VLIST returns the string length in bytes and copies the string to *lpvlb->lpszText*. If VLIST cannot obtain the string for the specified item, it returns VLB_ERR. The location that *lpvlb->lpszText* points to must be large enough to hold the string and terminating null character. You can use VLB_GETTEXTLEN to obtain the length of the string before copying it. If the application locates the item, it returns VLB_OK in *lpvlb->nStatus* and the string in *lpvlb->lpszText*. If it cannot locate the item, it returns VLB_ERR in *lpvlb->nStatus*.

VLB_GETTEXTLEN

```
wParam = 0;
lpvlb = (LPVLBSTRUCT) lParam;
```

The application sends this message to VLIST to request the string length for the item specified in *lpvlb*. If the specified item is not visible, VLIST sends this message to the application to request the string length. VLIST returns the string length, or VLB_ERR if it cannot obtain the string for the specified item. If the application locates the item, it returns VLB_OK in *lpvlb->nStatus* and the string in *lpvlb->lpszText*. If the application cannot locate the item, it returns VLB_ERR in *lpvlb->nStatus*.

VLB_GETTOPINDEX

```
wParam = 0;
lParam = 0L;
```

The application sends this message to VLIST to request the first visible item in the list. VLIST returns the item identifier for this item.

VLB_INITIALIZE

```
lpvlb = (LPVLBSTRUCT) lParam;
```

The application sends this message to VLIST to initialize the list box. When VLIST receives this message, it begins managing the virtual list box functions. VLIST returns VLB_OK.

VLB_LAST

```
wParam = 0;
lpvlb = (LPVLBSTRUCT) lParam;
```

VLIST sends this message to the application to request the last item in the list. If the application can locate the item, it returns VLB_OK in *lpvlb->nStatus* and the item in *lpvlb*. Otherwise, it returns VLB_ERR in *lpvlb->nStatus*.

VLB_NEXT

```
wParam = 0;
lpvlb = (LPVLBSTRUCT) lParam;
```

VLIST sends this message to the application to request the item after the item specified in *lpvlb*. If the application can locate the item, it returns VLB_OK in *lpvlb->nStatus* and the item in *lpvlb*. Otherwise, it returns VLB_ERR in *lpvlb->nStatus*.

VLB_PAGEDOWN

```
nAdjust = wParam;
lParam  = 0L;
```

The application sends this message to VLIST to scroll the virtual list box down one page. A page is the number of visible lines. The application can adjust the number of items scrolled by specifying the number of lines to add to the page size. A negative value causes the virtual list box to scroll fewer than the visible number of lines; a positive value causes the virtual list box to scroll more than the visible number of lines. The message returns VLB_ERR if an error occurs.

VLB_PAGEUP

```
nAdjust = wParam;
lParam = 0L;
```

The application sends this message to VLIST to scroll the virtual list box up one page. A page is the number of visible lines. The application can adjust the number of items scrolled by specifying the number of lines to add to the page size. A negative value causes the virtual list box to scroll fewer than the visible number of lines; a positive value causes the virtual list box to scroll more than the visible number of lines. The message returns VLB_ERR if an error occurs.

VLB_PREV

```
wParam = 0;
lpvlb = (LPVLBSTRUCT) lParam;
```

VLIST sends this message to the application to request the item before the item specified in *lpvlb*. If the application can locate the item, it returns VLB_OK in *lpvlb->nStatus* and the item in *lpvlb*. Otherwise, it returns VLB_ERR in *lpvlb->nStatus*.

VLB_RANGE

```
wParam = 0;
lpvlb = (LPVLBSTRUCT) lParam;
```

After receiving the VLB_INITIALIZE, VLB_UPDATEPAGE, and VLB_RESETCONTENT messages, VLIST sends this message to the application to request the number of items in the virtual list box. If the application knows the number of items, it returns VLB_OK in *lpvlb->nStatus* and the number of items in *lpvlb->lIndex*. If the application does not know the number of items in the virtual list box, it returns VLB_ERR in *lpvlb->nStatus.*

VLB_RESETCONTENT

```
wParam = 0;
lParam = 0L;
```

The application sends this message to VLIST to request that VLIST reset the contents of the list box. In response, VLIST sends a VLB_RANGE message to its parent window and displays the top of the virtual list. VLIST returns VLB_OK.

VLB_SETCURSEL

```
nOption = wParam;
lpvlb   = (LPVLBSTRUCT) lParam;
```

The application sends this message to VLIST to select an item and scroll it into view if necessary. The *nOption* parameter specifies how the target item is determined and can have the following values:

VLB_FINDITEMFind the item specified in *lpvlb*.

VLB_FIRSTFind the first item in the list.

VLB_LASTFind the last item in the list. The last item is displayed at the bottom of the list box.

VLB_NEXTFind the item after the item specified in *lpvlb*. If the *lpvlb* item is at the bottom of the list box, VLIST scrolls the list down one line so the target item is at the bottom of the list box. If the specified item is not visible, VLIST positions the list so the target item is at the top of the list box.

VLB_PREVFind the item before the item specified in *lpvlb*. If the *lpvlb* item is at the top of the list box, VLIST scrolls the list up one line so the target item is at the top of the list box. If the specified item is not visible, VLIST positions the list so the target item is at the top of the list box.

VLIST returns VLB_OK if it can locate the target item, or VLB_ERR if it cannot.

VLB_SETHORIZONTALEXTENT

```
wParam = nHorizontalExtent;
lParam = 0L;
```

The application sends this message to VLIST to set the horizontal scroll range to the given value. This message does not return a value.

VLB_SETITEMDATA

```
wParam = 0;
lpvlb = (LPVLBSTRUCT) lParam;
```

The application sends this message to VLIST to set the 32-bit data value of the item specified in *lpvlb* to the value in *lpvlb->lData*. If the item is not currently visible, the data value is not set. This message returns VLB_ERR if an error occurs.

VLB_SETITEMHEIGHT

```
wParam  = 0;
lParam = MAKELPARAM(cyItem, 0);
```

The application sends this message to VLIST to set the line height of items in the list box to the given value. The message returns VLB_ERR if the height is invalid.

VLB_SETTABSTOPS

```
wParam = (WPARAM) cTabs;
lParam = (LPARAM) (int FAR*) lpTabs;
```

The application sends this message to set the tab-stop positions in the virtual list box. (The virtual list box must be created with the VLBS_USETABSTOPS style.)

If the *cTabs* parameter is zero and the *lpTabs* parameter is NULL, the default tab stop is two dialog box units.

If *cTabs* is 1, the list box has tab stops separated by the distance specified by *lpTabs*.

If *lpTabs* points to more than a single value, a tab stop is set for each value in *lpTabs*, up to the number specified by *cTabs*.

A dialog box unit is a horizontal or vertical distance. One horizontal dialog box unit is equal to one-fourth of the current dialog box base width unit. The dialog box base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the current dialog box base units in pixels. For more information on dialog units, see the "Ask Dr. GUI #5" technical article. The message returns nonzero if all the tabs were set; otherwise, the return value is zero.

VLB_SETTOPINDEX

```
wParam = (WPARAM) index;
lParam = 0L;
```

The application sends this message to VLIST to ensure that the item specified in *lpvlb* is visible. VLIST positions the list so that either the specified item appears at the top of the list box or the maximum scroll range has been reached. The message returns VLB_ERR if an error occurs.

VLB_UPDATEPAGE

```
wParam = 0;
lParam = 0L;
```

The application sends this message to VLIST to update the contents of the virtual list box. VLIST does not add or delete items in the same way a standard list box does. To add, remove, or update the items in the virtual list, the application must use VLB_UPDATEPAGE. The message returns VLB_ERR if an error occurs.

Sample Dialog Procedure

In the following code, the **DIALOGSMsgProc** function shows the dialog procedure for a dialog box that contains a VLIST control. The virtual list box has 10,000 items in the format:

nnnn Item

where *nnnn* is the item number.

```
BOOL FAR PASCAL DIALOGSMsgProc(HWND hWndDlg, UINT Message, WPARAM
                               wParam, LPARAM lParam)
{
 int i,j;
 HWND                  hwndList;
 static char           szText[128];
 LPVLBSTRUCT           lpvlbInStruct;

 switch(Message)
   {
    case WM_INITDIALOG:
         cwCenter(hWndDlg, 0);
         /* Initialize working variables                    */
         hwndList = GetDlgItem(hWndDlg, 101);
         SendMessage(hwndList, VLB_INITIALIZE, 0, 0L);
         break; /* End of WM_INITDIALOG                      */

    case WM_CLOSE:
         /* Closing the Dialog behaves the same as Cancel   */
         PostMessage(hWndDlg, WM_COMMAND, IDCANCEL, 0L);
         break; /* End of WM_CLOSE                           */

    case WM_COMMAND:
         switch(wParam)
           {
            case IDOK:
                 EndDialog(hWndDlg, FALSE);
                 break;
           }
         break;    /* End of WM_COMMAND                     */

    case VLB_PREV:
         lpvlbInStruct = (LPVLBSTRUCT)lParam;

         if ( lpvlbInStruct->lIndex > 0 ) {
            lpvlbInStruct->nStatus = VLB_OK;
            lpvlbInStruct->lIndex--;
            lpvlbInStruct->lData = lpvlbInStruct->lIndex;
            sprintf(szText,"%ld Item",lpvlbInStruct->lIndex);
            lpvlbInStruct->lpTextPointer = szText;
            return TRUE;
         }
         else {
            lpvlbInStruct->nStatus = VLB_ENDOFFILE;
            return TRUE;
         }
         break;

    case VLB_FINDPOS:
```

```
      lpvlbInStruct = (LPVLBSTRUCT)lParam;

      if ( lpvlbInStruct->lIndex == 0L ) {
         goto First;
      }
      else if ( lpvlbInStruct->lIndex == 100L ) {
         goto Last;
      }
      else {
         lpvlbInStruct->lIndex = lpvlbInStruct->lData*1000L;
         lpvlbInStruct->nStatus = VLB_OK;
         sprintf(szText,"%ld Item",lpvlbInStruct->lIndex);
         lpvlbInStruct->lpTextPointer = szText;
         return TRUE;
      }
   break;

case VLB_FINDITEM:
      lpvlbInStruct = (LPVLBSTRUCT)lParam;

      lpvlbInStruct->lIndex = lpvlbInStruct->lData;
      lpvlbInStruct->nStatus = VLB_OK;
      sprintf(szText,"%ld Item",lpvlbInStruct->lIndex);
      lpvlbInStruct->lpTextPointer = szText;

      return TRUE;

   break;

case VLB_FINDSTRING:
case VLB_FINDSTRINGEXACT:
case VLB_SELECTSTRING:
   {
      _fstrcpy(szText,lpvlbInStruct->lpFindString);
      lpvlbInStruct->lIndex = atol(szText);
      sprintf(szText,"%ld Item",lpvlbInStruct->lIndex);
      lpvlbInStruct->lpTextPointer = szText;
      lpvlbInStruct->lData = lpvlbInStruct->lIndex;
      lpvlbInStruct->nStatus = VLB_OK;
      return TRUE;
   }
   break;

case VLB_RANGE:
      lpvlbInStruct = (LPVLBSTRUCT)lParam;

      lpvlbInStruct->lIndex = 100000L;
      lpvlbInStruct->nStatus = VLB_OK;
      return TRUE;

   break;

case VLB_NEXT:
      lpvlbInStruct = (LPVLBSTRUCT)lParam;

      if ( lpvlbInStruct->lIndex < 99999L ) {
         lpvlbInStruct->nStatus = VLB_OK;
```

```
            lpvlbInStruct->lIndex++;
            lpvlbInStruct->lData = lpvlbInStruct->lIndex;
            sprintf(szText,"%ld Item",lpvlbInStruct->lIndex);
            lpvlbInStruct->lpTextPointer = szText;
            return TRUE;
         }
         else {
            lpvlbInStruct->nStatus = VLB_ENDOFFILE;
            return TRUE;
         }
       break;

    case VLB_FIRST:
First:
         lpvlbInStruct = (LPVLBSTRUCT)lParam;

         lpvlbInStruct->nStatus = VLB_OK;
         lpvlbInStruct->lIndex = 0L;
         sprintf(szText,"%ld Item",lpvlbInStruct->lIndex);
         lpvlbInStruct->lpTextPointer = szText;
         lpvlbInStruct->lData = lpvlbInStruct->lIndex;
         return TRUE;

    case VLB_LAST:
Last:
         lpvlbInStruct = (LPVLBSTRUCT)lParam;

         lpvlbInStruct->nStatus = VLB_OK;
         lpvlbInStruct->lIndex = 99999L;
         sprintf(szText,"%ld Item",lpvlbInStruct->lIndex);
         lpvlbInStruct->lpTextPointer = szText;
         lpvlbInStruct->lData = lpvlbInStruct->lIndex;
         return TRUE;

       break;

    case VLB_GETITEMDATA:
         lpvlbInStruct = (LPVLBSTRUCT)lParam;

         lpvlbInStruct->nStatus = VLB_OK;
         lpvlbInStruct->lData = lpvlbInStruct->lIndex;
         return TRUE;
       break;

    default:
        return FALSE;
   }
 return TRUE;
}
```

Part 2. Implementing VLIST

This section describes the implementation of VLIST for those of you who are interested in its history. This information is useful for readers who would like to modify VLIST or create their own virtual list boxes.

Two Methods for Implementing a Virtual List Box

Ideally, a virtual list box should look and behave exactly like a standard list box, and should support the same styles as a standard list box (owner-drawn, multiple or extended selection, multiple column, and so on). You can implement a virtual list box in two ways:

1. **Write the virtual list box from scratch.**

   The disadvantage of this method is the tremendous effort required to duplicate the functionality of a standard list box. Each new feature that Windows adds to the standard list box must be duplicated in the virtual list box. The advantage of this method is that it yields a virtual list box without compromisesyou can modify the behavior of the list box because the code is custom.

2. **Write a virtual list box based on a standard Windows list box.**

   This method requires less effort than method 1 because you can use most of the functionality of the standard list box as is. New features added to the standard list box need not be added to the virtual list box. However, this method may require some compromisesbecause the behavior of the base list box is under the control of Windows, modifying the behavior for a particular virtual list box may not be possible.

The following sections describe the implemention of a virtual list box (called VLIST) based on a Windows list box (method 2 above).

Two Windows

The VLIST virtual list box control consists of two windows:

The first window is the main window for the control and provides the border for the virtual list box.

The second window is a standard Windows list box, but is created without a border. This window is a child of the main window and covers the main window's client area completely.

These two windows appear together to form a control that looks like a single standard list box. (See Figure 2.)

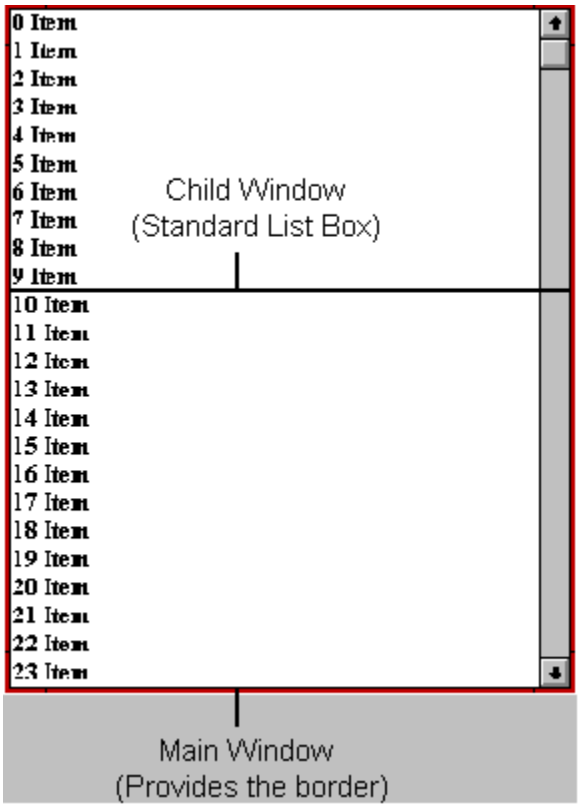

Figure 2. The two windows in VLIST

Why Two Windows?

Using two windows allows VLIST to be packaged as a separate window class and as a custom control, so a developer can use the virtual list box as easily as any of the standard controls provided with Windows.

> Using a separate window class allows you to use the window's extra bytes to store a pointer to a structure that contains the information for each virtual list box instance. This is necessary because a virtual list box needs to store its own control information.
>
> Packaging the virtual list box as a custom control lets you access it through the Dialog Editor.

You could also implement VLIST by subclassing a standard list box, but that would mean that an application that uses the virtual list box would have to create a standard list box, and then subclass it. If the virtual list box is written from scratch, it needs only one window.

Creating the VLIST Control

An application that uses VLIST creates the VLIST main window by calling the **CreateWindow** or **DialogBox** function. When the main window is created, VLIST must do some initialization processing of its own by processing the WM_NCCREATE and WM_CREATE messages.

WM_NCCREATE

When VLIST gets the WM_NCCREATE message from Windows, it locally allocates its information structure, stores the pointer to the structure in the window's extra bytes, and then ensures that the main window does not have vertical or horizontal scroll bars.

```
LONG VLBNcCreateHandler( HWND hwnd, LPCREATESTRUCT lpcreateStruct)
{
  PVLBOX pVLBox;

  //
  // Allocate storage for the VLBOX structure.
  //
  pVLBox = (PVLBOX) LocalAlloc(LPTR, sizeof(VLBOX));

  if (!pVLBox)
      // Error, no memory
      return((long)NULL);

  SetWindowWord(hwnd, 0, (WPARAM)(WORD)pVLBox);

  pVLBox->styleSave = lpcreateStruct->style;

  //
  // Make sure that there are no scroll bar styles.
  //
  SetWindowLong(hwnd, GWL_STYLE,
                (LPARAM)(DWORD)( pVLBox->styleSave &
                ~WS_VSCROLL & ~WS_HSCROLL));


  return((LONG)(DWORD)(WORD)hwnd);

}
```

WM_CREATE

When VLIST receives the WM_CREATE message from Windows, it initializes the variables in the information structure, and determines the height of the font used and the number of lines in the list box. VLIST then ensures that the style it will use to create the child window list box is acceptable. The child window list box:

Must not have a scroll bar (do not use WS_VSCROLL or WS_HSCROLL).

Must not have a border (do not use WS_BORDER or WS_THICKFRAME).

Must not have the following styles (VLIST does not support these):

LBS_SORT: The order of the virtual list box is controlled by the application that uses it.

LBS_MULTIPLESEL, LBS_EXTENDEDSEL, LBS_OWNERDRAWVARIABLE, LBS_MULTICOLUMN: These styles are not supported in this example, although it is possible to add support for these.

Must have the LBS_WANTKEYBOARDINPUT style. VLIST will need to get keyboard events that are sent to the list box so it can react to scrolling keystrokes.

Must have the LBS_NOINTEGRALHEIGHT style. VLIST needs to size the child window so that the list box fills the main window's client area completely and the scroll bar appears to be a part of the main window. If you do not use this style, Windows shrinks the child window to fit only visible lines and leaves a blank space between the bottom of the scroll bar and the main window's border.

Note VLIST cannot display partially visible lines even if you use LBS_NOINTEGRALHEIGHT, because the way a standard list box scrolls these lines is incompatible with the way VLIST manages scrolling. In a standard list box, when the list is scrolled up, the selected item is moved down. When the selection reaches a partially visible item, Windows forces that item to be fully visible by scrolling the list box up the necessary height. This scrolling causes problems for VLIST, because VLIST needs to control scrolling itself. If VLIST were written from scratch, it could mimic the standard list box behavior and could select partially visible items during scrolling.

Must have the LBS_NOTIFY style, so VLIST can react to changes in the selected item in the child window list box.

After processing the child window's style, VLIST calls **CreateWindow** to create the child window, and then subclasses the window to handle focus and scrolling functionality.

Note Don't confuse subclassing the standard list box with implementing VLIST with a subclass. In this program, VLIST (not the application) subclasses the standard list box it uses. The application does not need to subclass VLIST.

```
LONG VLBCreateHandler( PVLBOX pVLBox, HWND hwnd, LPCREATESTRUCT
                       lpcreateStruct)
{
  LONG           windowStyle = pVLBox->styleSave;
  RECT           rc;
  TEXTMETRIC     TextMetric;
  HDC            hdc;

  //
  // Initialize variables.
  //
  pVLBox->hwnd            = hwnd;
  pVLBox->hwndParent      = lpcreateStruct->hwndParent;
  pVLBox->nId             = (int) lpcreateStruct->hMenu;
  pVLBox->hInstance       = lpcreateStruct->hInstance;
  pVLBox->nvlbRedrawState = 1;
  pVLBox->lNumLogicalRecs = -2L;
  pVLBox->lSelItem        = -1L;
  pVLBox->wFlags          = 0;

  //
  // Enforce the use of 32-bit data values.
  //
  if ( windowStyle & VLBS_USEDATAVALUES )
      pVLBox->wFlags  |= USEDATAVALUES;
  else
      pVLBox->wFlags &= ~USEDATAVALUES;

  //
  // Determine if this VLB is storing strings.
  //
  pVLBox->wFlags |= HASSTRINGS;
  if ((windowStyle & VLBS_OWNERDRAWFIXED )
```

```
   && (!(windowStyle & VLBS_HASSTRINGS)))
   pVLBox->wFlags &= ~HASSTRINGS;

//
// Get the font height and number of lines.
//
hdc = GetDC(hwnd);
GetTextMetrics(hdc, &TextMetric);
ReleaseDC(hwnd,hdc);
pVLBox->nchHeight = TextMetric.tmHeight;
GetClientRect(hwnd,&rc);
pVLBox->nLines = ((rc.bottom - rc.top) / pVLBox->nchHeight);

//
// Remove borders and scroll bars.
//
windowStyle = windowStyle & ~WS_BORDER & ~WS_THICKFRAME;
windowStyle = windowStyle & ~WS_VSCROLL & ~WS_HSCROLL;

//
// Remove standard list box we don't support.
//
windowStyle = windowStyle & ~LBS_SORT;
windowStyle = windowStyle & ~LBS_MULTIPLESEL;
windowStyle = windowStyle & ~LBS_OWNERDRAWVARIABLE;
windowStyle = windowStyle & ~LBS_MULTICOLUMN;
windowStyle = windowStyle & ~VLBS_USEDATAVALUES;

//
// Add list box styles we must have.
//
windowStyle = windowStyle | LBS_WANTKEYBOARDINPUT;
windowStyle = windowStyle | LBS_NOINTEGRALHEIGHT;
windowStyle = windowStyle | LBS_NOTIFY;

//
// Create the list box window.
//
pVLBox->hwndList =
  CreateWindowEx((DWORD)0L,
                 (LPSTR)"LISTBOX",(LPSTR)NULL,
                 windowStyle | WS_CHILD,
                 0,
                 0,
                 rc.right,
                 rc.bottom,
                 pVLBox->hwnd,
                 (HMENU)VLBLBOXID,
                 pVLBox->hInstance,
                 NULL);

if (!pVLBox->hwndList)
    return((LONG)-1L);

//
```

```
  // Subclass the list box.
  //
  pVLBox->lpfnLBWndProc = (WNDPROC)SetWindowLong(pVLBox->hwndList,
                           GWL_WNDPROC,
                            (LONG)(WNDPROC)LBSubclassProc);

  return((LONG)(DWORD)(WORD)hwnd);
}
```

Destroying the Virtual List Box

When the VLIST main window is destroyed, VLIST must free its information structure. This processing occurs when VLIST receives the WM_NCDESTROY message.

```
void VLBNcDestroyHandler(HWND hwnd,  PVLBOX pVLBox, WPARAM wParam,
                         LPARAM lParam)
{

  if (pVLBox)
      LocalFree((HANDLE)pVLBox);

  //
  // In case rogue messages float through after we free
  // the pVLBox, set the handle in the window structure to
  // FFFF, and test for this value at the top of the VLIST WndProc.
  //
  SetWindowWord(hwnd, 0, (WPARAM)-1);

  DefWindowProc(hwnd, WM_NCDESTROY, wParam, lParam);
}
```

Sizing the Virtual List Box

Each time the virtual list box is resized (that is, whenever the main window receives a WM_SIZE message), the child window list box must also be resized. When the main window receives this message, VLIST:

Determines the line height. If VLIST is an owner-drawn list box, the application may resize the height of each line with a VLB_SETITEMHEIGHT message. In this case, the virtual list box must be resized with the new line height even if the size of the window itself does not change, because a new line height may increase or decrease the number of lines that fit into the list box.

Resizes the main window to hold only fully visible lines, if the virtual list box has the VLBS_NOINTEGRALHEIGHT style.

Resizes the child window to completely fill the main window's client area.

Calculates a new number of visible lines.

Refreshes the child window if the virtual list box contained data before the resizing took place. If the list box is now large enough to hold all of the items in the virtual list box, VLIST displays all of the items; otherwise, VLIST redraws the current display.

```
void VLBSizeHandler( PVLBOX pVLBox, int nItemHeight)
```

```
//
// Recalculates the size of the internal control in response to
// the resizing of the virtual list box window.
//
{
  HDC             hdc;
  TEXTMETRIC      TextMetric;
  RECT            rcWindow;
  RECT            rcClient;
  HANDLE          hOldFont;

  //
  // Set the line height.
  //
  if ( nItemHeight ) {
    pVLBox->nchHeight = nItemHeight;
  }
  else if ((pVLBox->styleSave & VLBS_OWNERDRAWFIXED) ) {
    pVLBox->nchHeight = (int) SendMessage(pVLBox->hwndList,
                                          LB_GETITEMHEIGHT, 0,0L);
  }
  else {
    hdc = GetDC(pVLBox->hwndList);
    if (pVLBox->hFont)
       hOldFont = SelectObject(hdc, pVLBox->hFont);
    GetTextMetrics(hdc, &TextMetric);
    pVLBox->nchHeight = TextMetric.tmHeight;
    if (pVLBox->hFont)
       SelectObject(hdc, hOldFont);
    ReleaseDC(pVLBox->hwndList,hdc);
  }

  //
  // Get the main window's client area.
  //
  GetClientRect(pVLBox->hwnd,&rcClient);

  //
  // Is there a window?
  // Does the list box have an integral height?
  //
  if (pVLBox->hwnd && !(pVLBox->styleSave&VLBS_NOINTEGRALHEIGHT) ) {
      //
      // If it needs adjusting...
      //
      if (rcClient.bottom % pVLBox->nchHeight) {
          //
          // ...adjust it.
          //
             GetWindowRect(pVLBox->hwnd,&rcWindow);
             SetWindowPos(pVLBox->hwnd, NULL, 0, 0,
                 rcWindow.right - rcWindow.left,
                 ((rcClient.bottom/pVLBox->nchHeight)
                     *pVLBox->nchHeight)+((rcWindow.bottom-
                     rcWindow.top) - (rcClient.bottom)),
```

```
                    SWP_NOACTIVATE | SWP_NOMOVE | SWP_NOZORDER);
        }
    }

    //
    // Now adjust the child window list box to fill the new
    // main window's client area.
    //
    if ( pVLBox->hwndList ) {
        GetClientRect(pVLBox->hwnd,&rcClient);
        SetWindowPos(pVLBox->hwndList, NULL, 0, 0,
           rcClient.right+(GetSystemMetrics(SM_CXBORDER)*2),
           rcClient.bottom+(GetSystemMetrics(SM_CXBORDER)*2),
           SWP_NOACTIVATE | SWP_NOMOVE | SWP_NOZORDER);
    }

    //
    // Calculate the number of lines.
    //
    pVLBox->nLines = rcClient.bottom / pVLBox->nchHeight;

    //
    // If there is stuff already in the list box, update
    // the display (there may be more or fewer items now).
    //
    if ( pVLBox->lNumLogicalRecs != -2L ) {
        if ( pVLBox->lNumLogicalRecs <= pVLBox->nLines ) {
           VLBFirstPage(pVLBox);
        }
        else if ( pVLBox->wFlags & USEDATAVALUES ) {
            VLBFindPage(pVLBox, SendMessage(pVLBox->hwndList,
                                LB_GETITEMDATA, 0, 0L), FALSE );
        }
        else {
            VLBFindPage(pVLBox, pVLBox->lToplRecNum, FALSE);
        }

    }

}
```

Setting the Font for the Virtual List Box

When VLIST gets a WM_SETFONT message, it sets the child window's font to a new value by sending a WM_SETFONT message to the child window list box. VLIST then resizes the list box based on this new font.

```
void VLBSetFontHandler( PVLBOX pVLBox, HANDLE hFont, BOOL fRedraw)
{
  pVLBox->hFont = hFont;

  SendMessage(pVLBox->hwndList, WM_SETFONT, (WPARAM)hFont,
              (LPARAM)FALSE);
```

```
  VLBSizeHandler(pVLBox, 0);

  if (fRedraw)
    {
      InvalidateRect(pVLBox->hwnd, NULL, TRUE);
    }
}
```

Managing the Scroll Bars

Windows manages the vertical scroll bar in a standard list box. If the list box is not large enough to accommodate all of the items, Windows gives the list box a vertical scroll bar. When the list box is able to display all of the items, Windows removes the vertical scroll bar. Thus, Windows frees the application from having to manage the scroll bar manually.

Note Windows version 3.1 provides a new style called LBS_DISABLENOSCROLL, which disables the scroll bars instead of removing them. Our discussion on adding and removing scroll bars in this section also applies to enabling and disabling scroll bars.

VLIST manages the vertical scroll bar itself. Because there are only enough items to fill the child window list box, Windows always removes the vertical scroll bar from the list box. To avoid this, VLIST must ensure that the child window list box is created without the WS_VSCROLL and WS_HSCROLL styles. (The standard list box does not distinguish between vertical and horizontal scroll bar styles; if either style is specified, the other is assumed.) VLIST calls the **ShowScrollBar** function to add or remove the vertical scroll bar as necessary. Because the standard list box does not think that it has scroll bars to manage, it does not interfere with VLIST.

```
void UpdateVLBWindow( PVLBOX pVLBox, LPRECT lpRect)
{
    int   nPos;

    if ( pVLBox->lNumLogicalRecs == -1L )
       SetScrollPos(pVLBox->hwndList, SB_VERT, 50, TRUE);
    else {
       if ( pVLBox->lNumLogicalRecs <= pVLBox->nLines ) {
          if ( pVLBox->styleSave & VLBS_DISABLENOSCROLL )
             EnableScrollBar(pVLBox->hwndList, SB_VERT,
                             ESB_DISABLE_BOTH);
          else
             ShowScrollBar(pVLBox->hwndList, SB_VERT, FALSE);
       }
       else {
           if ( pVLBox->styleSave & VLBS_DISABLENOSCROLL )
              EnableScrollBar(pVLBox->hwndList, SB_VERT,
                              ESB_ENABLE_BOTH);
           else
              ShowScrollBar(pVLBox->hwndList, SB_VERT, TRUE);
           if ( pVLBox->lToplIndex >=
                (pVLBox->lNumLogicalRecs-pVLBox->nLines) ) {
                nPos = 100;
           }
           else if (pVLBox->lToplIndex == 0L) {
```

```
                nPos = 0;
            }
            else {
                nPos = (int) ((pVLBox->lToplIndex*100L) /
                       (pVLBox->lNumLogicalRecs-pVLBox->nLines+1));
            }
            SetScrollPos(pVLBox->hwndList, SB_VERT, nPos, TRUE);
        }

    }

}
```

Because the child window list box is created without the WS_HSCROLL and WS_VSCROLL styles, VLIST must also provide an interface between the application and the horizontal scroll bar management features of the child window list box. For this purpose, VLIST implements the VLB_GETHORIZONTALEXTENT and VLB_SETHORIZONTALEXTENT messages. When a horizontal scroll bar is added or removed from the child window list box, VLIST adjusts the number of visible lines in the virtual list box accordingly.

Controlling the Selection

In a standard list box, Windows keeps track of selected items. In a virtual list box, VLIST tracks selected items itself. When the selected item is scrolled out of view in the child window list box, VLIST removes the selected state. When the selected item scrolls back into view in the child window list box, VLIST sets it as the selected item. Because this implementation does not support extended or multiple selection list boxes, VLIST only needs to track the selected item and know if there is no selection. VLIST could support a multiple selection style by keeping a list of selected items.

Scrolling the Virtual List Box

The main function of VLIST is to scroll the virtual list box contentsscrolling gives VLIST the appearance that it contains more items than it actually stores. As VLIST scrolls, it adds items to, and deletes items from, the child window list box. There are five possible scrolling actions:

Scroll down a line.

Get the item after the last item in the child window list box. If this item exists, remove the item at the top of the list and add the new item to the bottom of the list.

Scroll up a line.

Get the item before the first item in the child window list box. If this item exists, remove the item at the bottom of the list and add the new item to the top of the list.

Scroll down a page.

Get the item after the last item in the child window list box. While this item exists, or until the number of lines displayed in the list box has been scrolled, remove the item at the top of the list and add the new item to the bottom of the list. We selected this implementation instead of asking the parent window for one page at a time for two reasons:

It simplifies the communication between the application and the virtual list box. Only next item and previous item messages are needed, so the application has fewer messages it must

implement handlers for.

It frees the application from having to determine when a page is full. For example, let's assume that the list box is positioned so that it is half a page from the bottom of the list, and VLIST asks the application to send the next page. The application must place the bottom half of the current page's items at the top of the last page, and then fill the last page with the rest of the items. Because VLIST fills the pages, the application does not have to deal with this extra complexity.

Scroll up a page.

Get the item before the first item in the child window list box. While this item exists, or until the number of lines displayed in the list box has been scrolled, remove the item at the bottom of the list and add the new item to the top.

Scroll to a particular position.

The virtual list box can scroll to four positions:

To a particular item. For a given item identifier (the item's index number or a 32-bit data value; see the "What Item?" section earlier in this article), VLIST requests the item from the application, and then positions the list box so that the given item is the first visible item.

To a location in the list. This is very similar to scrolling to a particular item, but in this case a percentage through the list is used. VLIST requests the item at the specified location in the list from the application, and then positions the list so that the item closest to that percentage is the first visible item.

To the top of the list. The first item appears as the first visible item.

To the bottom of the list. The last item in the list appears as the last visible item.

In each case, VLIST asks the parent window to find the item. If the item exists, VLIST:

1. Removes the selection, if one exists.
2. Clears the list box.
3. Adds the first item.
4. Adds the appropriate number of items to fill the list box (less one for the given item) or until the bottom of the list is reached.

Hocus Pocus Focus Smocus

When the selection state changes, the focus rectangle issue rears its ugly head. Standard single-selection list boxes draw a focus rectangle on a selected item. When the selection is removed with LB_SETCURSEL, the focus rectangle stays on the item. When a selected item is scrolled out of view, the focus rectangle goes with it, but the list still has focus. VLIST mimics this behavior. However, when a selected item is scrolled out of view and VLIST sets the current selection to 1 with the LB_SETCURSEL message, the child window list box draws a focus rectangle around the item that previously had the selection. To avoid this, VLIST sets the focus back to its main window when it sets the selected item to 1. The main window sends any keyboard messages it receives to the child window list box and sets the focus back to the child window list box. When the child window list box receives the focus from Windows, and there is no selection or the selection is not visible, it sets the focus to the main window. The main window keeps the focus if there is no selection or if the current selection is not visible. These steps give the virtual list box the appearance of behaving the same as a standard list box.

Keyboard Messages

When a standard list box receives a WM_VSCROLL message to page the list box up or down, it scrolls (up or down) by the number of visible lines. When a standard list box receives a PageUp or PageDown keystroke, it scrolls the list box (up or down) one line less than the number of visible lines. The VLIST implements the varying scroll amounts in the same way.

Minimizing Flicker

Applications use the WM_SETREDRAW message to minimize the flicker associated with list box updates. This message causes the list box to delay updating the display so that all changes can be performed at once. VLIST also uses WM_SETREDRAW when it scrolls to a new position or when it processes multiple updates, for example, when paging up and down the list box. In these cases, VLIST behaves like a standard list box.

Single-line scrolling requires a different technique. Each time the standard list box processes a WM_SETREDRAW message to turn on the redraw flag, it invalidates its window and causes the background to be erased. When an item is added to, or deleted from, a list box, Windows invalidates the list box in the same way. When VLIST scrolls one line at a time, it deletes an item and then adds a new item. As a result, the list box ensures that flicker occurs. To stop the flicker, VLIST must update the list box display itself. First, VLIST removes and adds the items. Then, it uses the **ValidateRect** function to tell Windows not to paint the now invalidated area. Finally, VLIST scrolls the list box window itself and tells Windows to repaint the new (top or bottom) line. This technique eliminates flicker associated with single-line scrolling.

The following sample code illustrates this procedure:

```
//
// Remove the top string.
//
SendMessage(pVLBox->hwndList, LB_DELETESTRING, 0, 0L);

//
// Add the new line.
//
SendMessage(pVLBox->hwndList, LB_ADDSTRING, 0, (LPARAM)
            pVLBox->vlbStruct.lpTextPointer);

//
// Tell Windows not to paint the whole list box.
//
ValidateRect(pVLBox->hwndList, NULL);

//
// Scroll the window up.
//
ScrollWindow(pVLBox->hwndList, 0, (-1)*pVLBox->nchHeight,
              NULL, NULL);

//
// Now tell Windows that the bottom line needs fixing.
//
SendMessage(pVLBox->hwndList, LB_GETITEMRECT,
```

```
            pVLBox->nLines-1, (LPARAM)(LPRECT)&UpdRect);
InvalidateRect(pVLBox->hwndList, &UpdRect, TRUE);
UpdateWindow(pVLBox->hwndList);
```