Barking Spider Software, Inc.
List Control
for
Windows

Barking Spider Software, Inc. Beaverton, Oregon
(503) 629-8236 VOICE/FAX/AUTO (503) 324-0187 VOICE

Introduction

What is the Barking Spider List Control?

The List Control is part of a small, fast Windows DLL that removes the hassles of adding a list display to your product. Using the DLL's powerful exported API, the programmer has complete control over bitmaps, text, spacing, and more.

The Barking Spider List Control currently requires the following execution environment:

Windows 3.1 or greater in standard or enhanced mode.

To program for the List Control your chosen language must be able to:

Access DLL API's
Process Windows messages

NOTE: We will be releasing Visual Basic and C++ interfaces

What is in this package?

Pertaining to the List Control, this package contains:

The Tree/List Control DLL.
The export library for the Tree/List Control DLL.

This manual.

BSCORE.H - The core C language header file. This file contains the constants, structures, notification messages, and error codes return from the exported API.

BSLIST.H - The List Control C language header file. This file contains the function declarations needed to interface with the DLL.

Listdem1 - List demo program 1. This program places the List Control window on it's main window. This program interfaces with the DLL using the DLL's API.

Listdem2 - List demo program 2. This program places the List Control window in a modal dialog. This program shows how to use the List Control with a dialog box editor using the Custom Control interface and the exported API.

What can you do with the List Control?

The List Control allows you to create and modify visual displays of lists.

List modification APIs:
- List item addition
- List item insertion
- List item deletion
- Remove all list items
- Add/change user data

Visual modification APIs:
- Set bitmaps
- Set icons
- Change bitmaps
- Change icons
- Set bitmap/icon spacing
- Set font
- Change list item text

In addition, the List Control simplifies user/tree interaction. All painting and scrolling tasks are handled by the control. Using a simple messaging scheme, the application is notified whenever a list item is selected or double clicked. Keyboard interactions with the List Control produce the same messages as the equivalent mouse interactions.

Additional List Control features:
- Drag and Drop encapsulation

System color change synchronization
Programmer defined data store for each list item
User or control owned strings
Multiple bitmaps per line
Complete list item hit test notification
List item deletion callback

Getting Started

What do you install?

To install the Barking Spider List Control run the install batch file provided.
Copy the Tree/List Control DLL to a directory in your path or into your project's executable directory.
Copy the List Control import library and header files, **BSCORE.H** and **BSLIST.H**, into the appropriate directory(s) on your hard disk.

Programming Overview

This overview is provided as an aid to programmers who are new to using the Barking Spider Software List Control. It provides a brief overview of interaction with the List Control. For more complete information on APIs used in the demos please refer to the API Reference in this manual. For a more indepth technical overview of the Tree Control's operations and APIs please refer to the **BSCORE.H** and **BSTREE.H** header files included in your package. Also, running the demo,
treedem3.exe, will display the List Control and Tree Control APIs along with descriptions.

What will the control display?

The control will display a graphic representation of the list described to it by the application. The display is a linear presentation of list items.

How do I create the tree display window?

The tree display creation step allows the programmer to specify display

placement, size and window styles. This step does not create a list do be displayed, it creates the frame and scroll bars, and readies the control's engine for processing.

The list display can be created by:

Calling the DLL's **BSL_CreateListBox** exported API (NOTE: See **Listdem1** for sample code)

OR

Embedding a List Control in a dialog box with a dialog editor. (NOTE: This uses the Custom Control information embedded in the DLL. See **Listdem2** for sample code)

You must save the window handle of the newly created list display window for later API calls. Because the List Control can have multiple active instances, the window handle is used to identify particular display windows.

How do I construct a list?

The list to be displayed can be created using the APIs **BSL_AddListItems** and/or **BSL_InsertListItems. BSL_AddListItems** appends list items at the end of the list. **BSL_InsertListItems** will insert list items at the given index. Both of these APIs require a pointer to an array of structures that define each list item being added.

Communications from the control

Communication with the List Control is simple. All of the user input is filtered by the control and notification messages are sent to the parent window of the control.

When an event occurs that the application needs to know about, the Control issues a notification message. The lParam of the message contains a pointer to a **SELECT_NOTIF** structure. This structure conatins a pointer to the affected list item and flags that describe the state of the list item.

The three List Notification messages and their causes:

WM_BST_SELECT_NOTIF. This message is triggered by:

- A single left mouse button click while over a list item.
- Highlight movement with Up arrows, Down arrows, Home, End, Page Up, Page Down, Ctrl Up arrows, or Ctrl Down arrows.

WM_BST_SELCET_NOTIF_DBLCLK. This message is triggered by:
- A left mouse button double click while over a list item. NOTE: A **WM_BST_SELECT_NOTIF** is sent on the first click.
- A Carriage Return while a list item is selected.

WM_BST_DO_DRAG. This message is triggered by:
- Depressing and holding the left mouse button over a list item, while moving the mouse a predetermined distance.

Demo API Reference

NOTE: The Demo API Reference is not a complete listing of the List Control APIs. The APIs listed immediately below were chosen to be documented because they are critical to the List Control's operation in the demos. A complete API reference is supplied with each purchase of the Tree/List Control.

BSL_AddListItems ( )
BST_CreateListBox ( )
BST_ResetList ( )
BST_SetBitmapSpace ( )

Complete API Listing:

BSL_AddListItems ( )
BSL_ChangeListItemText ( )
BSL_ChangeUserData ( )
BSL_ConvertPointToSelectNotif ( )
BSL_CreateListBox ( )
BSL_DeleteListItem ( )
BSL_DragAcceptFiles ( )
BSL_GetCurSel ( )
BSL_GetListItem ( )
BSL_GetListItemCount ( )
BSL_GetVersion ( )
BSL_InsertListItems ( )
BSL_ResetList ( )
BSL_SetBitmap ( )
BSL_SetBitmapAndActiveBitmap ( )
BSL_SetBitmapSpace ( )
BSL_SetCurSel ( )
BSL_SetDeleteListItemCallBack ( )
BSL_SetFont ( )

BSL_SetIcon ( )
BSL_SetXSpaceBeforeText ( )

****

BSL_AddListItems ( )

short _export FAR PASCAL **BSL_AddListItems** (
HWND hwndList,
WORD wNodeDefCount,
LP_TREE_NODE_DEF lpTreeNodeDef);

Description

This API allows the application to add one or more list items to the given list.

Arguments

HWND hwndList

This argument indicates which list control will create the new list items and append them to the end of the list. This is the window handle that was returned to the application after calling **BSL_CreateListBox ( )**. **BSL_CreateListBox ( )** creates an empty list.

WORD wNodeDefCount

wNodeDefCount contains the number of list items to be added. In other words, wNodeDefCount is the number of **TREE_NODE_DEF** array elements in the **TREE_NODE_DEF** array pointed to by lpTreeNodeDef.

LP_TREE_NODE_DEF lpTreeNodeDef

lpTreeNodeDef is a pointer to an array of **TREE_NODE_DEF**s that describe each of the list items to be added.

If the list already has list items, then the new list items are appended to end of the list.

While adding list items to the list control, the list control could receive either a memory allocation failure or realize that the total number of list items in the list

has exceeded the maximum number of list items allowed. If either of these conditions occur then the list items that were added to the list control prior to the problem, are NOT removed from the list. The application will receive error messages if there is a memory allocation problem or the maximum number of list items is exceeded. The application can determine what list items did not make it into the list by looking at the value of the lpTreeNode member of each **TREE_NODE_DEF** array starting from the beginning of the array. When a **TREE_NODE_DEF**'s lpTreeNode member is found that has a value of zero, then that **TREE_NODE_DEF** and all subsequent **TREE_NODE_DEF**s in the **TREE_NODE_DEF** array are not used to create new list items.

Comments

Where do **TREE_NODE** pointers come from? There are several ways to retrieve **TREE_NODE** pointers.

1) **TREE_NODE** pointers are supplied to the application by the list control. They are returned to the application via the lpTreeNode member of the **TREE_NODE_DEF** structure, when the application successfully calls **BSL__AddListItems ( )** or **BSL_InsertListItems ( )**, which add list items to the given list. The application can store these **TREE_NODE** pointers for future references. Via the **TREE_NODE** pointers, the index where the list item is position in the list can be accessed.

2) The application will receive the pointer to the list control owned **SELECT_NOTIF** structure as the result of a notification of an event. From the **SELECT_NOTIF** structure, the **TREE_NODE** pointer to the list item involved in the event is available.

3) Calling **BSL_GetListItem ( )** will return the TREE_NODE pointer of the list item specified by the given index.

4) Calling **BSL_ConvertPointToSelectNotif ( )** will return a pointer to the list control owned SELECT_NOTIF structure which in turn, the pointer to the list item that lies under the coordinate supplied by the application can be retrieved.

5) **BSL_SetDeleteNodeCallBack ( )** allows the application to register a callback with the given list control. The callback will be called everytime a list item is deleted from the list. List items can be deleted from the list with two list control export APIs or with the Windows API **DestroyWindow ( )**. The two list control exported APIs are:

BSL_DeleteListItem ( )
BSL_ResetList ( )

Return Codes

BST_NO_ERROR
BST_ERR_MEMORY_ALLOC_FAILED
BST_ERR_LEVEL_LIMIT_EXCEEDED
BST_ERR_TOO_MANY_NODES
BST_ERR_ONLY_ONE_ROOT_ALLOWED
BST_ERR_INVALID_PARENT_FOR_INSERTION
BSL_ERR_INVALID_INDEX

****

BSL_CreateListBox ( )

```
HWND _export FAR PASCAL BSL_CreateListBox (
        HANDLE hInstance,
        HWND hwndApp,
        int x, int y,
        int nWidth, int nHeight,
        DWORD dwStyle,
        DWORD dwExStyle);
```

Description

Creates an empty list. **BSL_CreateListBox ( )** ORs the style bits specified in dwStyle to the list control's required styles and calls the Windows API, **CreateWindowEx ( )**. In effect, **BSL_CreateListBox** calls ( ) **CreateWindowEx ( )** as:

```
CreateWindowEx ( dwExStyle,
          "BST_Tree",
          "",
          dwStyle | dwTreeControlStyles,
          x,
          y,
          nWidth,
          nHeight,
```

```
hwndApp,
NULL,
hInstance,
NULL);
```

Once the list is created then list items can be added by calling the exported APIs:

BSL_AddListItems ( )
BSL_InsertListItems ( )

Arguments

HANDLE hInstance

Instance associated with the creation of the list control window.

HWND hwndApp

Window handle of the parent window that is creating the list control.

int x

X location of the upper left corner of the list control in client area coordinates of the parent window.

int y

Y location of the upper left corner of the list control in client area coordinates of the parent window.

int nWidth

Width of the list control in device (pixel) units.

int nHeight

Height of the list control in device (pixel) units.

DWORD dwStyle

Application requested **CreateWindowEx ( )** styles.

DWORD dwExStyle

Application requested **CreateWindowEx ( )** extended styles.

Comments

The successful return value from this call, which is a window handle, will be used as the list control identifier for applying the list control exported APIs. Most of the list control APIs require the list control window handle.

Return Codes

If successful, **BSL_CreateListBox ( )** will return the window handle of the newly created list control. If failure, then a NULL will be returned.

****

BSL_ResetList ( )

void _export FAR PASCAL **BSL_ResetList** (
HWND hwndList);

Description

Remove all of the list items in the specified list control but do not destroy the list control.

Arguments

HWND hwndList

This argument specifies the list control that will destroy all of it's list items. This list control handle can still be used in any of the APIs such as **BSL_AddListItems ( )**. It does not invalidate the list control window handle.

Comments

When a list item is deleted, it's **TREE_NODE** structure is no longer valid.

If a notification of a list item's deletion is desired, then the application can use the list control exported API, **BSL_SetDeleteNodeCallBack** ( ), to register a callback

function that the list control will call just before deletion of the list item. If the application has assigned a pointer to dynamically allocated memory in the lpUserData member of the list item, it is the responsibility of the application to free this memory.

Return Codes

BST_NO_ERROR

****

BSL_SetBitmapSpace ( )

short _export FAR PASCAL **BSL_SetBitmapSpace** (
HWND hwndList, short nBitmap,
short nWidth,
short nHeight,
BOOL bCenterBitmap);

Description

Define the list control's maximum width and height (in pixels) of the bitmap space, identified by the argument nBitmap, for all items in the list. This API will reserve space before the beginning of the list's text for the drawing of the bitmap identified by nBitmap. If the bitmap/icon handle associated with bitmap space is NULL, the empty bitmap space will still be represented. If the next thing after the empty bitmap space is the list item text, then the list control shifts the text left until it is butted against a non empty bitmap space or the lines.

Bitmap spaces offer the application the ability to fine tune each bitmap position and to define each bitmap hit test area. Hit testing is not performed on the bitmap but on the bitmap space.

The dimensions of a bitmap space are defined globally for all tree nodes. The reason for this is to keep column alignment of the bitmaps. This is visually pleasing and offers consistency with hit testing.

Arguments

HWND hwndList

This argument specifies the list control in which to reserve the bitmap space. This space will be used to draw a bitmap. This space is before the start of the list item's text, if it has text.

short nBitmap

Identifies the bitmap space. This is a zero based index into a MAX_BITMAPS size array where each member of the array is a structure. This structure has two members which hold the width and height of the bitmap space. The width and height are defined in device coordinates (pixels).

short nWidth

nWidth is the width, in pixels, of the reserved bitmap space.

short nHeight

nHeight is the height, in pixels, of the reserved bitmap space.

BOOL bCentered

If set to TRUE then center the bitmap/icon in the bitmap space.

Comments

The bitmap/icon will be painted in the reserved space centered between the top and bottom boundaries. If bCentered is set to TRUE then the bitmap/icon will be centered between the left and right boundaries. If bCentered is set to FALSE then the bitmap/icon will be left justified in the bitmap space. If the bitmap is larger than the width and/or the height, then the bitmap will be clipped. If the icon is larger than the bitmap space there will be no clipping since the Windows API, DrawIcon(), does not provide it. Therefore, if an icon is going to be associated with a bitmap space, make the width and height of the bitmap space at least as wide and tall as the values returned from GetSystemMetrics (SM_CXICON) and GetSystemMetrics (SM_CXICON).

If either nHeight or nWidth is zero, then there is no bitmap space.

Remember, that the bitmap space definitions, 0 thru MAX_BITMAPS-1, are global to all list items in the given list control.

Return Codes

BST_NO_ERROR