DragAndDrop – Win32 Drag and Drop Windows library

DragAndDrop
A C Library for Drag-And-Drop support for Windows 32 bit Win32 applications
Version. 1.0 August 27 2009
Anders Karlsson MySQL AB / Sun Microsystems

Table of contents

Introduction

Drag and Drop has been part of Windows since Windows -95 or so. The 1990's was the heyday for Object Orientation, and what was more natural than for Microsoft to come up with some cool generic OO Infrastructure technology. Com was born. And as COM came at the same time as Drag-and-Drop, why not implement the latter using the former technology?

There is a good reason why not. Drag-and-Drop is close to the core functions used by many, if not most, Windows applications. Fact is, it's being forced on us, we have to supported editors dropping text on us, Windows explorer dropping files on us etc. The problem is that the low lever part of Windows is a clean C-API, just like most other modern Operating Systems. Which is not to say that C is a particularly good language to write applications in. But C has the advantage of being of a such a low level nature, that nearly any other layer, applications, servers, application servers, infrastructure layers, old school languages (Fortran, Cobol, Pascal, you name it). Basically, C is a good language for such things, as it easily lets itself reside below other layers of any kind.

This is not so with something like COM. COM is Object Oriented, and as such supports inheritance, interfaces and stuff like that. But it doesn't easily lend itself below some other layer, such as an OO layer with different OO attributes, or below a layer that is not OO at all. And Microsoft knew this. And Microsoft had non-OO languages to support, Like VB and stuff like that. And C++, which is OO, but with some different OO concepts from COM. So Microsoft did this halfway. The did COM OO, but they actually implemented it in C. Sounds Crazy? Nah, this is how much C code works anyway, but COM contains much more complexity than usual, and makes many assumptions and shortcuts on it's way to an OO layer. And to enable Microsoft to implement an OO C++ layer on top of COM, at the same time as a not-so OO VB can work with it. So why make it OO at all? Beats me.

One things Microsoft has hardly ever done is to tell you the details of those layer. Well, COM is there, and it is well described, but cutting to the cheese is simple. Like it you want this:

I have a text in one Window in my application, I just want to drag it to another place in my application and drop it there!

What is needed for this? One call that says: *Here is some stuff to be dropped*. And I have another place where I say: *Here you may drop stuff*. And that's it.

As you can see, these two concepts are a bit different, dragging is an action by the user tat is easy to control, it happens at some point that the user clicks something to drag and we handle it. The second is more different, as in this case, what is dragged might not even be from my own application, it just an outside event, as far as my application goes. So why not handle that the way we handle all other things? If I move a mouse over a window, I get a notification message, right? So why can't I get a notification message when a mouse move over my windows and is dragging something when it does, such as some text? And the answer is, this is not how Drag-and-Drop works. No, you are supposed to implement a bunch of COM objects, 3 or 4 of them. For no good reason, if you ask me. And I am a pure Win32 developer by the way. Straight C, nuthin' else.

So based on my own knowledge (to a small extent), on the http://www.catch22.net/ website, that contains loads of good stuff for C and C++ developers, that want to stick with the clean Win32 AP, so as not to get messed up with too many DLLs, MFS versions and stuff like that, and on the work of Davide Chiodi, referenced in the Catch22 article on Ole Drag and Drop, I now have my own version up.

Using my, C level library, built on top of COM, you only have to do these things:

- Your application has to register itself for OLE, or if you want to, there is a call to initialize this library, that will do that for you.
- Call a function to register a Window in your application to be the receiver of something that is dropped. Again, this is just one C function.
- Call a function to start dragging something destined for somewhere else. No COM object involved, no nothing, just one C function to call.
- And then, when something is dropped, one of two, well known, things happen (which depends on how you registered to receive dropped objects).
    - A callback you provided when you registered is called. Just like. With information on what was dropped, the format of it and where.
    - Alternatively Message is sent. This is a message of your choice. And again, to the message is information on what was being dropped, the format of it and where it was dropped.

If you are writing application code in plain C for Windows, I think you have been looking for something like this at one time or another.

The library provided has a few more tricks up it's sleeve, but actually, the cases where you just want to drag an text or be the receiver of some text or a file, or even when working with custom format, this above will be OK. But the library also allows you to work with the

lower level object, should you so wish, to get some more functionality and control. This is easy to figure out just by looking at how the above mentioned functions do stuff.

What's in the project

Not that much really:

- A Visual C++ (7.0 right now) project for all this.
- A simple test and development application that allows you to test and play with the library and the functions.
- The library source, consisting of one C source file and one include file.

Using the library

These are the basic steps:

Initialize OLE and the library

Somewhere in WinMain or at least before doing anything with the library or OLE in general, you need to initialize stuff. This is done by calling just one simple function:

```
MyDragDropInit(NULL);
```

Register to receive a drop

This is what you do to ensure that your callback or message is received whenever an object is dropped on the object of your choice. This also involved creating an OLE object that is the target of this drop. Usually the way you would do this, is to let the library keep track this, but you need to ensure that you have a reference to this object you have something to get rid of when it's no longer needed, such as when the window is closed.

A window procedure for a non-modal window that is the target for a drag and drop operation may then look like this:

```
{
CLIPFORMAT cf;
static PMYDROPTARGET pTarget = NULL;

switch(uMsg)
   {
   case WM_INITDIALOG:
      cf = CF_TEXT;
      pTarget = MyRegisterDragDrop(hDlg, &cf, 1, WM_NULL,
        MyDropProc, NULL);
      return TRUE;

   case WM_COMMAND:
      switch(wParam)
         {
         case IDOK:
            g_hWndDrop = NULL;
            MyRevokeDragDrop(pTarget);
            DestroyWindow(hDlg);
            return TRUE;
         }
      break;
   }
return FALSE;
}
```

In the above example, I use a callback to receive messages, in this case that is the *MyDropProc* argument to *MyRegisterDragDrop*.

Start dragging

When this happens really depends on your application. In the sample code, I have two examples, one using a static control, and one using an edit control. I will describe the latter here, and it is REAL easy.

I have overloaded the edit control in the dialog, so I can handle a few messages there:

```
   case WM_INITDIALOG:
       pProc = (WNDPROC) (LONG_PTR) GetWindowLongPtr(GetDlgItem(hDlg,
IDC_EDIT_DRAGTEXT), GWL_WNDPROC);
      if(pProc != NULL)
         {
          SetWindowLongPtr(GetDlgItem(hDlg, IDC_EDIT_DRAGTEXT),
GWL_USERDATA, (LONG) (LONG_PTR) pProc);
          SetWindowLongPtr(GetDlgItem(hDlg, IDC_EDIT_DRAGTEXT), GWL_WNDPROC,
(LONG) (LONG_PTR) EditLinkProc);
         }
```

Now the edit control (with ID *IDC_EDIT_DRAGTEXT*) will receive message before they are handled by the default dialog. The only message I worry about I *WM_LBUTTONDOWN* which is what I get when the user clicks with the left mouse button in the control. As this may indicate that the control is receiving focus, I ignore this message if the control doesn't already **have** focus, and then I start dragging the text, like this:

```
   if(hWnd == GetFocus())
      {
      GetWindowText(hWnd, szBuf, sizeof(szBuf));
      MyDragDropText(szBuf);
      return 0;
      }
```

In this case, I am using the most simple construct, which is the *MyDragDropText* function, which will only drag one object, a text string, in one format, *CF_TEXT*. This function also manages the Global memory handles used by OLE. For a more complex example, see the example code.

Receive a dropped object

This is the final step, and this is a result of the call to *MyRegisterDragDrop* in the Register to receive a drop section above. As I said, I have two options here, either a callback is called, or I receive a message, the result is similar, really, as far as I can see, it's more a matter of taste which variation to use, but I will shortly describe the callback method here.

In the call to *MyRegisterDragDrop* above, I passed the name of a function, *MyDropProc* which is the user defined callback to call when a drop occurs. This function has the following prototype:

```
static DWORD MyDropProc(CLIPFORMAT cf, HGLOBAL hData, HWND hWnd,
  DWORD dwKeyState, POINTL pt, void *pUserData);
```

And this doesn't look too bad. Now it's just a question of handling the handle for whichever format I got back. As you saw above though, I can register to receive several different formats, so what am I getting here? You get the first one on the registered formats that match. If data in two possible formats where dropped, the first one in the target is still the one that will match, and only that.

And this still means I can get different formats when called, depending on what was dropped! But in this example, I am only handling one format, the standard and most common CF_TEXT format. So MyDropProc may, in this example, look like this:

```
static DWORD MyDropProc(CLIPFORMAT cf, HGLOBAL hData, HWND hWnd,
  DWORD dwKeyState, POINTL pt, void *pUserData)
   {
   DWORD dwEffect = DROPEFFECT_NONE;
   void *pData;
```

```
    if(cf == CF_TEXT)
       {
       pData = GlobalLock(hData);
       SetWindowText(hWnd, (LPTSTR) pData);
       GlobalUnlock(hData);
       dwEffect = DROPEFFECT_COPY;
       }
    return dwEffect;
    }
```

In this example, I set the window text of the window that is the registered receiver of the drop, and if this is a normal window, the *caption* will be set. If the target is, say, an edit control, the data is pasted to this.

Reference

Types

MYDROPDATA

This is a struct that is sent as the *wParam* of a message when a message is used for a drop target:

```
typedef struct tagMYDROPDATA
    {
    CLIPFORMAT cf;
    POINTL pt;
    DWORD dwKeyState;
    HGLOBAL hData;
    } MYDROPDATA, *PMYDROPDATA;
```

Note that the user defined data pointer is not part of this struct, instead, this is passed in the *lParam* to the message.

PMYDROPSOURCE

This is an opaque pointer to a drop source created by the library. The contents are local to the library itself.

PMYDROPTARGET

This is an opaque pointer to a drop target created by the library. The contents of the struct are local to the library.

MYDDCALLBACK

A pointer to the callback function used when callback are used as the target for a drop operation.

```
typedef DWORD (*MYDDCALLBACK)(CLIPFORMAT cf, HGLOBAL hData, HWND hWnd,
DWORD dwKeyState, POINTL pt,
  void *pUserData);
```

See MyRegisterDragDrop for more information.

Functions

MyDragDropInit

This must be the first function called in the application. It initializes the library and OLE. The passed heap HANDLE may be NULL, in which case the library will use the default process heap.

Arguments:

Name	Type	Comments

Name	Type	Comments
hHeap	HANDLE	A handle to a memory heap that the library will use, if NULL, the library will use the default process heap.

Returns:
Nothing

CreateMyDropSource

Create a source for a drag and drop operation. This is the most advanced of the functions for this purpose, and allows multiple, different, formats and well as both left and right dragging.

Arguments:

Name	Type	Comments
bRightClick	BOOL	Flag if this if the dragging is using the right button.
pFormat	CLIPFORMAT *	An array of clipformats, such as CF_TEXT, CF_HDROP or a format created with *RegisterClipboardFormat()*
phData	HGLOBAL *	An array of globals handles with data, each element in the array is mapped to the format in the *pFormat* argument array.
lFmt	ULONG	The number of elements in the two arrays *pFormat* and *phData*

Returns:
PMYDROPSOURCE – A pointer to the drop source, used by subsequent calls to the library.

CreateMyDropSourceText

Create a text source for a drag and drop operation. This is a simplified version of CreateMyDropSource that only allows dragging of one text object. The text object is passed as a string and the Global handle is managed by the library itself.

Arguments:

Name	Type	Comments
bRightClick	BOOL	Flag if this if the dragging is using the right button.
pText	LPCTSTR	A pointer to the text to drag,

Returns:
PMYDROPSOURCE – A pointer to the drop source, used by subsequent calls to the library.

MyDragDropSourceEx

Start dragging a MYDROPSOURCE object created with CreateMyDropSource or CreateMyDropSourceText.

Arguments:

Name	Type	Comments
pDropSrc	PMYDROPSOURCE	The source to be dragged.
dwOKEffect	DWORD	Allowed effects for dragging operations. See the DoDragDrop() function in the Win32 API for possible values.
pdwEffect	DWORD *	A pointer to a DWORD to receive the used effect. Again see the DrDragDrop() function in the Win32 API.

Returns:

DWORD - The result of the drag operation, actually the result of the called DoDragDrop().

MyDragDropSource

Start dragging a MYDROPSOURCE object created with CreateMyDropSource or CreateMyDropSourceText. Simplified version of MyDragDropSourceEx. The allowed Drop effect is always DROPEFFECT_COPY and the actual used Drop effect is ignored.

Arguments:

Name	Type	Comments
pDropSrc	PMYDROPSOURCE	The source to be dragged.

Returns:

DWORD - The result of the drag operation, actually the result of the called DoDragDrop().

MyDragDropText

Start dragging a test string. This function combines CreateMyDropSourceText and MyDragDropSource for easy of use when just dragging a simple text object. The handle used for the text object is allocated and released by the library itself.

Arguments:

Name	Type	Comments
pText	LPCTSTR	A pointer to the string to be dragged.

Returns:

DWORD - The result of the drag operation, actually the result of the called DoDragDrop().

FreeMyDropSource

Free a MYDROPSOURCE object.

Arguments:

Name	Type	Comments
pDropSrc	PMYDROPSOURCE	The source to freed.

Returns:

. PMYDROPSOURCE – Always returns NULL.

MyRegisterDragDrop

Register as a receiver of a Drag-and-Drop operation. When a drop occurs, a callback is always called unless *pDropProc* is NULL. Never is both a message and a callback used, and a callback has preference.

The formats in the *pFormat* array are in order of preference, the highest preference first. The format that has a match with a format in the dropped object will be used, and passed to *pDropProc*.

In a message is used instead of a callback, the message will receive a pointer to a MYDROPDATA struct as *wParam* and the *pUserData* pointer will be passed as *lParam*.

Arguments:

Name	Type	Comments
hWnd	HWND	The current window handle. That registers for drag and drop, and if a message is used to signal a drop, then the message is sent to this window.
pFormat	CLIPFORMAT *	An array of clipformats supported.
lFmt	ULONG	The number of elements in the *pFormat* array.
nMsg	UINT	The message to send unless a callback is used. Pass WM_NULL when using a callback.
pDropProc	MYDDCALLBACK	A pointer to a callback function. Pass NULL to send a message instead.

Name	Type	Comments
pUserData	void *	A pointer to some user defined data that will be passed to the callback or message.

Returns:
PMYDROPTARGET – Pointer to the drop target. Is used subsequently to free up stuff when the drop target stops being a drop target.

MyRevokeDragDrop

Revoke a target as being such a target for dropping objects.

Arguments:

Name	Type	Comments
pTarget	PMYDROPTARGET	The drop target that is going away.

Returns:
PMYDROPTARGET – Always NULL.

More advanced functions

The following functions are used internally by the library, but for convenience, I have exposed them, if you want to have more flexibility. The use of them should be obvious of you read the code. These may also be useful if you want to work directly with the OLE objects for some other purpose.

CreateDataObject

Create an *IDataObject*.

Arguments:

Name	Type	Comments
pFormat	CLIPFROMAT *	An array of clipboard formats supported by the object.
phData	HGLOBAL *	An array of handles that represents the formats in the *pFormat* array.
lFmt	ULONG	The number of elements in the two arrays.

Returns:
IDataObject * - A pointer to the created *IDataObject*.

CreateDropSource

Create an *IDropSource*.

Arguments:

Name	Type	Comments
bRightClick	BOOL	Flag if this object was created by right-clicking .

Returns:
IDropSource * - A pointer to the created *IDropSource*.

CreateDropTarget

Create an *IDropTarget*. This is probably the one of the advanced functions that you might want to use occasionally, as it provides some more flexibility. See the MyRegisterDragDrop function for some more information.

Arguments:

Name	Type	Comments
pFormat	CLIPFROMAT *	An array of clipboard formats supported by the object.
lFmt	ULONG	The number of supported formats.

Name	Type	Comments
hWnd	HWND	A handle to the window that is the target for this object.
nMsg	UINT	Message to send of messaging is used instead of callbacks. Pass WM_:NULL if callbacks are used instead.
pDropProc	MYDDCALLBACK	The callback to call. Pass NULL to cause a message to be used instead.
pUserData	void *	User defined data to be passed to the message or callback when something is dropped.

Returns:
IDropSource * - A pointer to the created *IDropSource*.