Internet Explorer Toolbar (Deskband) Tutorial

By Erik Thompson | 21 Aug 2001 | Article

VC6IE6.0WindowsATLDevBeginner

A tutorial on Using RBDeskband and CWindowImpl ATL Object Wizards to create an Internet Explorer(IE) Toolbar.

. See Also

•More like this

•More by this author

Article Browse Code Stats Revisions (2) Alternatives

4.94 (166 votes)

1

2

3

4

5

4.94/5 - 166 votes6 removedμ 4.78, σa 1.21 [?]

Discuss this article

1,780 .

Download Toolbar Binary - 20Kb

Download Source - 21 Kb

Introduction

Having recieved a number of requests for a tutorial of sorts on developing Internet Explorer Toolbars with the RBDeskband and CWindowImpl wizards that I created, I have come up with a simple sample toolbar which can be used as a reference when developing your own toolbars or explorer bars. The tutorial will walk you through the stages of developing a toolbar for IE that is very similar to the Address bar that is already present in IE. I wanted to do a tutorial that would provide a realistic sample and would produce an end result that could be used by others after the tutorial was finish. So, the tutorial is going to show you how to develop an IE toolbar to get stock quote information from The Motley Fool website. So with that, let us get started.

Prequisites

This tutorial assumes that you already know how to program in C++ and know some information about ATL and COM. To work through this tutorial, you will need the following installed on your development machine:

•Visual C++6 installed

•RBDeskBand ATL Object Wizard (Version 2.0) [get it here]

•CWindowImpl ATL Object Wizard [get it here]

The Framework

The IE toolbar consists of a COM component supporting IDeskband and a few other necessary interfaces for which IE looks for when loading registered toolbars, explorer bars and deskbands. The RBDeskband ATL Object Wizard provides most of the framework for this article. What we will need to do is create our project, a new COM object to house our toolbar, and a few CWindowImpl classes using the CWindowImpl ATL Object Wizard. Then connecting these parts together we will produce the IE toolbar in the picture at the top of the article. Visually the toolbar consists of an editbox and a toolbar with one button on it. In actuality the toolbar consists of the fore mentioned and a non visible window that is used to reflect messages to the Toolbar window, which will process or forward messges to itself and the edit box.

Creating The Shell

We will not work through the steps in creating the shell for our toolbar.

Creating The Project

•If you have not done so already, start Visual C++6.

•Then, from the File menu select New menu item; the New Dialog pops up.

•In the New Dialog, select the Projects tab, if not already selected.

•Select ATL COM AppWizard from the list view, if not already selected.

•In the Project name, type "MotleyFool". See Figure 1.

•Click the OK Button.

Figure 1. New Dialog.

•The ATL COM AppWizard will kick in.

•Clicking the Finish Button, accepting the default AppWizard attributes. See Figure 2.

•The New Project Information Dialog will present itself requesting confirmation of your project settings.

•Click the OK Button.

Figure 2. ATL COM AppWizard.

Creating The DeskBand Object

Now that we have our project container we need to add our IDeskBand derived component so that the DLL actually exposes something.

•From the Insert menu, select New ATL Object menu item; the ATL Object Wizard dialog is invoked.

•In the ATL Object Wizard dialog, select the RadBytes Category. If this category is missing then make sure that the RBDeskband and CWindowImpl ATL Object Wizards are installed.

•Next select the DeskBand item from the Objects list.

•Click the Next button to invoke the ATL Object Wizard Properties dialog for the Deskband object. See Figure 3.

•On the Names property page, type "StockBar" into the Short Name field. See Figure 4.

•Select the DeskBand ATL Object Wizard property page

•Check the Internet Explorer Toolbar checkbox. See Figure 5.

•Click the OK button on the ATL Object Wizard Properties Dialog. The ATL Object Wizard will create the files necessary for our DeskBand's base implementation.

Figure 3. ATL Object Wizard.

Figure 4. ATL Object Wizard Properties - Names.

Figure 5. ATL Object Wizard Properties - DeskBand ATL Object Wizard

Now our project has the DeskBand implementation that we will modify to produce the toolbar pictured at the top of the article. First we will create the window classes we will need and then come back to the Desbkand object and update it to use our window classes.

Creating The Window Classes

So back in the Framework section we said that we would need three window classes. One for the Edit Box, one for the toolbar, and one for message reflection back to the toolbar. Let us now create these window classes.

The Edit Window

We need to create a derived class from the standard EDIT button window class because we are going to be adding methods to our class to help support functionality of the toolbar. This is one reason why we cannot use a CContainedWindow object directly.

•From the Insert menu, select New ATL Object menu item; the ATL Object Wizard dialog is invoked.

•In the ATL Object Wizard dialog, select the RadBytes Category. If this category is missing then make sure that the RBDeskband and CWindowImpl ATL Object Wizards are installed.

•Next select the CWindowImpl item from the Objects list.

•Click the Next button to invoke the ATL Object Wizard Properties dialog for the Deskband object. See Figure 3.

•On the Names property page, type "EditQuote" into the Short Name field.

•Select the CWindowImpl property page. See Figure 6.

•Select the SUPERCLASS radio button from the DECLAR\_WND\_\* group.

•In the Window Class Name field, type "EDITQUOTE".

•In the Original Class Name list, select the EDIT listbox item. See Figure 7.

•Click the OK button on the ATL Object Wizard Properties Dialog. The ATL Object Wizard will create the files necessary for our CWindowImpl derived class implementation.

Figure 6. ATL Object Wizard Properties - Names.

Figure 7. ATL Object Wizard Properties - CWindowImpl.

The Toolbar Window

We need to create a derived class from the standard TOOLBARCLASSNAME window class because we are going to be adding methods to our class to help support functionality of the toolbar. It will also be the parent for the edit box and the window which the IE host will request from our DeskBand.

•From the Insert menu, select New ATL Object menu item; the ATL Object Wizard dialog is invoked.

•In the ATL Object Wizard dialog, select the RadBytes Category. If this category is missing then make sure that the RBDeskband and CWindowImpl ATL Object Wizards are installed.

•Next select the CWindowImpl item from the Objects list.

•Click the Next button to invoke the ATL Object Wizard Properties dialog for the Deskband object. See Figure 3.

•On the Names property page, type "MFToolbar" into the Short Name field.

•Select the CWindowImpl property page. See Figure 8.

•Select the SUPERCLASS radio button from the DECLAR\_WND\_\* group.

•In the Window Class Name field, type "MOTLEYFOOLTOOLBAR".

•In the Original Class Name list, select the TOOLBARCLASSNAME listbox item. See Figure 9.

•Click the OK button on the ATL Object Wizard Properties Dialog. The ATL Object Wizard will create the files necessary for our CWindowImpl derived class implementation.

Figure 8. ATL Object Wizard Properties - Names.

Figure 9. ATL Object Wizard Properties - CWindowImpl.

The Reflection Window

We need to create a reflection window. It's just a CWindowImpl window implmented class. We are going to be adding a small bit of functionality just to create the toolbar object and be able to access the toolbar member from our deskband class.

•From the Insert menu, select New ATL Object menu item; the ATL Object Wizard dialog is invoked.

•In the ATL Object Wizard dialog, select the RadBytes Category. If this category is missing then make sure that the RBDeskband and CWindowImpl ATL Object Wizards are installed.

•Next select the CWindowImpl item from the Objects list.

•Click the Next button to invoke the ATL Object Wizard Properties dialog for the Deskband object. See Figure 3.

•On the Names property page, type "ReflectionWnd" into the Short Name field. See Figure 10.

•We will not change any of the CWindowImpl property page values this time.

•Click the OK button on the ATL Object Wizard Properties Dialog. The ATL Object Wizard will create the files necessary for our CWindowImpl derived class implementation.

Figure 10. ATL Object Wizard Properties - Names.

Adding The Details

Now that we have our window classes available we can add our functionality for our toolbar to the appropriate window classes. Let us start with the deepest window class and work our way back out.

The EditQuote Details

For the EditQuote implementation, we need to be able to process keystrokes from the user and let the host that created our deskband object know our edit box has focus. To accomplish the first part, we need to look ahead and see that our DeskBand object will be implementing the IInputObject interface. So the host will query for that interface and know that we want to recieve messages and be given the chance to recieve focus. When the host sends our band messages to process they come through the IInputObject::TranslateAccelerator method. Our DeskBand will implement this method and it is best if our edit box, which will process the message, copy the TranslateAcceleratorIO method definition so our deskband can forward the message easily through a logical method call.

Figure 11. FileView Pane.

In the FileView pane (See Figure 11), double click the EditQuote.h item under Header Files. This will open the header file in the editing area. We now need to define the method definition for TranslateAcceleratorIO. To do this, add below the virtual CEditQuote destructor the following line of code:

Collapse | Copy Code

STDMETHOD(TranslateAcceleratorIO)(LPMSG lpMsg);Now Open the EditQuote.cpp source file and add the implementation of TranslateAcceleratorIO to the file

Collapse | Copy Code

STDMETHODIMP CEditQuote::TranslateAcceleratorIO(LPMSG lpMsg)

{

TranslateMessage(lpMsg);

DispatchMessage(lpMsg);

return S\_OK;

}

Now our DeskBand implementation can call this message and the edit box will process the key strokes properly. But wait, our edit box should notify the toolbar to load the quote details entered if the key stroke is the enter key. For this, we will need to define a message id and send that message to the parent window to process. In the EditQuote.h header file below the include statement, add the definition of our message id as shown below in bold.

Collapse | Copy Code

#include <commctrl.h>

const int WM\_GETQUOTE = WM\_USER + 1024;

In our EditQuote.cpp file we will add code to our TranslateAcceleratorIO method to process the enter key. Add the code below in bold to the EditQuote.cpp file.

Collapse | Copy Code

STDMETHODIMP CEditQuote::TranslateAcceleratorIO(LPMSG lpMsg)

{

int nVirtKey = (int)(lpMsg->wParam);

if (VK\_RETURN == nVirtKey)

{

// remove system beep on enter key by setting key code to 0

lpMsg->wParam = 0;

::PostMessage(GetParent(), WM\_GETQUOTE, 0, 0);

return S\_OK;

}

TranslateMessage(lpMsg);

DispatchMessage(lpMsg);

return S\_OK;

}

Now our edit box will notify the parent when the user presses the enter key so that the parent can retrieve the requested ticker symbol details, this part will be implemented when we get to the toolbar details.

The first part of the Edit boxes implementation is finished. Now we need to be able for the edit box to have the deskband notify the host that we have focus or that we don't have focus any longer. To do this we will need to add a method for the deskband to pass us it's address so that we can call a method of the deskband class. These next steps will involve adding code to the CEditQuote class and to our Deskband class implementation.

Open the EditQuote.h file and add a forward reference to the CStockBar class so that we can defined our methods and members in our class header without knowing the implementation details of our deskband class, add the line in bold.

Collapse | Copy Code

#include <commctrl.h>

const int WM\_GETQUOTE = WM\_USER + 1024;

class CStockBar;

For our class to notify the host that our deskband has focus, we need to add a message handler for EN\_SETFOCUS. Add the command code handler code below in bold to your EditQuote.h file.

Collapse | Copy Code

BEGIN\_MSG\_MAP(CEditQuote)

COMMAND\_CODE\_HANDLER(EN\_SETFOCUS, OnSetFocus)

END\_MSG\_MAP()

Then add the method definition for OnSetFocus to the header file below the commented out handler prototypes as follows below in bold.

Collapse | Copy Code

// Handler prototypes:

// LRESULT MessageHandler(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);

// LRESULT CommandHandler(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);

// LRESULT NotifyHandler(int idCtrl, LPNMHDR pnmh, BOOL& bHandled);

LRESULT OnSetFocus(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);

Before we implement the OnSetFocus method, we need to define a method for our deskband to tells of it's address and to retain that address for later use. Add the following lines of code to your EditQuote.h file below the TranslateAcceleratorIO definition.

Collapse | Copy Code

void SetBand(CStockBar\* pBand);

private:

CStockBar\* m\_pBand;

Now we can move to your EditQuote.cpp source file and implement the message handler, the SetBand method, and update the TranslateAcceleratorIO method for focus change. At the top of the EditQuote.cpp file add the following includes to the include list as shown below in bold.

Collapse | Copy Code

#include "stdafx.h"

#include "EditQuote.h"

#include "MotleyFool.h"

#include "StockBar.h"

Now when we can use the methods of the CStockBar class in our code. Add to the end of the constructor, the initalization of m\_pBand. Don't forget the colon operator.

Collapse | Copy Code

CEditQuote::CEditQuote()

: m\_pBand(NULL)

{

}

Next we will add the SetBand implementation to our CEditQuote class. Notice that since it is not a com object we don't call AddRef or Release on it. It's just a pointer to the class and when it's destroyed our CEditQuote instance will also be destroyed. We could have also done this inline in our header file.

Collapse | Copy Code

void CEditQuote::SetBand(CStockBar\* pBand)

{

m\_pBand = pBand;

}

Next we need to add our message handler for our EN\_SETFOCUS message. Add the code below to the end of the EditQuote.cpp source file.

Collapse | Copy Code

LRESULT CEditQuote::OnSetFocus(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled)

{

//Notify host that our band has the focus so TranslateAcceleratorIO

//messages are directed towards our band.

if (m\_pBand) m\_pBand->FocusChange(TRUE);

return 0;

}

We have one more section of code to add to our CEditQuote implementation then we can move to our CStockBar class to define and implement the FocusChange method. Add the following code to the CEditQuote TranslateAcceleratorIO method as shown in bold. We add this code so the host knows that we are no longer needing messages.

Collapse | Copy Code

STDMETHODIMP CEditQuote::TranslateAcceleratorIO(LPMSG lpMsg)

{

int nVirtKey = (int)(lpMsg->wParam);

if (VK\_RETURN == nVirtKey)

{

// remove system beep on enter key by setting key code to 0

lpMsg->wParam = 0;

::PostMessage(GetParent(), WM\_GETQUOTE, 0, 0);

return S\_OK;

}

else if (WM\_KEYDOWN == lpMsg->message && nVirtKey == VK\_TAB)

{

// we no longer need messages forwarded to our band

if (m\_pBand) m\_pBand->FocusChange(FALSE);

return S\_FALSE;

}

TranslateMessage(lpMsg);

DispatchMessage(lpMsg);

return S\_OK;

}

Open the StockBar.h header file and add the definition of FocusChange to it as shown below in bold.

Collapse | Copy Code

// IStockBar

public:

void FocusChange(BOOL bHaveFocus);

Now open the StockBar.cpp source file and add the implementation of FocusChange to it at the bottom.

Collapse | Copy Code

void CStockBar::FocusChange(BOOL bHaveFocus)

{

if (m\_pSite)

{

IUnknown\* pUnk = NULL;

if (SUCCEEDED(QueryInterface(IID\_IUnknown, (LPVOID\*)&pUnk)) && pUnk != NULL)

{

m\_pSite->OnFocusChangeIS(pUnk, bHaveFocus);

pUnk->Release();

pUnk = NULL;

}

}

}

We have finished off the work needed for the edit box to work properly in our toolbar. Now we need to build our toolbar up so that it has a button and contains our edit box. Then we will add the nessecities to our reflection window and update our IDeskBand to provide the correct information to our host. We are almost there. If you were to compile the project and run it, it would except that the band would look like the following in figure X.

The MFToolbar Details

For the implementation of the MFToolbar window, we need to be able to have it do the following things. It must be able to process the WM\_GETQUOTE message from the EditQuote window, communicate with the web browser in which the toolbar is located, create the buttons and place the child windows on itself, forward messages to the EditQuote child window and size itself appropriately to the users actions.

So, the first thing we should do since our toolbar is going to contain an instance of CEditQuote is include the header file for the CEditQuote class. We will do this by opening the MFToolbar.h file and inserting the include statement for the CEditQuote class as shown in bold below.

Collapse | Copy Code

#include <commctrl.h>

#include "EditQuote.h"

Next we need to add a member to our toolbar class for the CEditQuote class. We will do this by adding a private section to the end of our class and defining a member variable as shown below in bold.

Collapse | Copy Code

CMFToolbar();

virtual ~CMFToolbar();

private:

CEditQuote m\_EditWnd;Now that we have our member defined for our EditQuote window, we need to forward window messages to it so that keyboard inputs are processed appropriately. We do this by updating the toolbar message map to chain messages to our member as shown below in bold.

Collapse | Copy Code

BEGIN\_MSG\_MAP(CMFToolbar)

CHAIN\_MSG\_MAP\_MEMBER(m\_EditWnd)

END\_MSG\_MAP()Looking forward, our deskband will need to get the EditQuote member to deterimine if it has focus and also to make it function. We could just expose the EditQuote member directly by having made it a public member instead of private, but by making it private we can expose a method that will expose our member giving us flexibility later to modify the class if the need should arise. So to expose the EditQuote member, we will add a fuction to our toolbar class to return the reference to the EditQuote member. In the toolbar header file, add the method definition and implementation below in bold to it.

Collapse | Copy Code

CMFToolbar();

virtual ~CMFToolbar();

inline CEditQuote& GetEditBox() {return m\_EditWnd;};Now we will create our toolbar window. Our toolbar consists of the EditQuote box and a button with an icon and text on it. To house the icon, our toolbar will need an image list handle to send to the toolbar window. So we need to add a few things to our toolbar header file before we go and implement the toolbar's creation. The first thing we will add is the member variable for our image list. Add the line in bold below to your toolbar header file.

Collapse | Copy Code

private:

CEditQuote m\_EditWnd;

HIMAGELIST m\_hImageList;Then we will add a message handler to our toolbar's message map and define the message handlers function definition to our header file and the follow lines of code in bold to your header file.

Collapse | Copy Code

BEGIN\_MSG\_MAP(CMFToolbar)

CHAIN\_MSG\_MAP\_MEMBER(m\_EditWnd)

MESSAGE\_HANDLER(WM\_CREATE, OnCreate)

END\_MSG\_MAP()

// Handler prototypes:

// LRESULT MessageHandler(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);

// LRESULT CommandHandler(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);

// LRESULT NotifyHandler(int idCtrl, LPNMHDR pnmh, BOOL& bHandled);

LRESULT OnCreate(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);

Before we can implement our toolbar's creation, we need to create a icon resource that our toolbar button will use next to its text. So go to the resource view and add a new icon to the project resources. You can do this by right clicking on "MotleyFool resources" and selecting "Insert..." from the context menu. In the Insert Resource dialog box, select Icon from the Resource type list and click the New button. This will insert a blank icon resource into your project. Rename the icon's resource ID by right clicking on the icon resource in the resource view and selecting the properties menu item from the context menu. Change the id to IDI\_MOTLEY. Then draw or graciously borrow an icon from The Motley Fool to use on the toolbar. I graciously borrowed the icon from their website and adapted it into the icon.

Now we can implement it our toolbars creation. Open the MFToolbar source file and implement the details of the toolbar creation as described below.

First we need to include the project resource file so we can use the icon ID in our code. Add the line in bold below to our toolbar's source file as shown.

Collapse | Copy Code

#include "stdafx.h"

#include "resource.h"

#include "MFToolbar.h"Next we need to update our constructor implementation. We need to initialize our handle to the image list by setting it to NULL. Don't forget the colon.

Collapse | Copy Code

CMFToolbar::CMFToolbar()

: m\_hImageList(NULL)

{

}

Next we need to update our destructor, it should destroy the image list and destroy the window if it has not yet been destroyed.

Collapse | Copy Code

CMFToolbar::~CMFToolbar()

{

ImageList\_Destroy(m\_hImageList);

if (IsWindow()) DestroyWindow();

}

Before we can implement our toolbar's creation, we need to add a resource symbol to our project for the toolbar button's ID. We could just use a #define statement at the top of the source file, but for cleanliness and since we are already including the resource.h file, we will add it to our resource file. Go to the "View" menu and select "Resource Symbols" menu item. Click the "New" button on the Resource Symbols dialog. Then enter a name of "IDM\_GETQUOTE" and click OK. Then close the Resource Symbols dialog.

Now we can create our toolbar, we defined the OnCreate method in our header file and need to now implement it. Add the following function and its implementation to the end of the toolbar source file.

Collapse | Copy Code

LRESULT CMFToolbar::OnCreate(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)

{

// buttons with images and text

SendMessage(m\_hWnd, TB\_SETEXTENDEDSTYLE, 0, (LPARAM)TBSTYLE\_EX\_MIXEDBUTTONS);

// Sets the size of the TBBUTTON structure.

SendMessage(m\_hWnd, TB\_BUTTONSTRUCTSIZE, sizeof(TBBUTTON), 0);

// Set the maximum number of text rows and bitmap size.

SendMessage(m\_hWnd, TB\_SETMAXTEXTROWS, 1, 0L);

// add our button's caption to the toolbar window

TCHAR\* pCaption = \_T("Get Quote");

int iIndex = ::SendMessage(m\_hWnd, TB\_ADDSTRING, 0,(LPARAM)pCaption);

// load our button's icon and create the image list to house it.

HICON hMotley = LoadIcon(\_Module.GetResourceInstance(), MAKEINTRESOURCE(IDI\_MOTLEY));

m\_hImageList = ImageList\_Create(16,16, ILC\_COLOR16, 1, 0);

int iImageIndex = ImageList\_AddIcon(m\_hImageList, hMotley);

DestroyIcon(hMotley);

// Set the toolbar's image

::SendMessage(m\_hWnd, TB\_SETIMAGELIST, 0, (LPARAM)m\_hImageList);

// add the button for the toolbar to the window

TBBUTTON Button;

ZeroMemory((void\*)&Button, sizeof(TBBUTTON));

Button.idCommand = IDM\_GETQUOTE;

Button.fsState = TBSTATE\_ENABLED;

Button.fsStyle = BTNS\_BUTTON | BTNS\_AUTOSIZE | BTNS\_SHOWTEXT;

Button.dwData = 0;

Button.iString = iIndex;

Button.iBitmap = 0;

::SendMessage(m\_hWnd, TB\_INSERTBUTTON, 0, (LPARAM)&Button);

// create our EditQuote window and set the font.

RECT rect = {0,0,0,0};

m\_EditWnd.Create(m\_hWnd, rect, NULL, WS\_CHILD|WS\_VISIBLE, WS\_EX\_CLIENTEDGE);

m\_EditWnd.SetFont(static\_cast<HFONT>(GetStockObject(DEFAULT\_GUI\_FONT)));

return 0;

}

If you try to compile at this point, you will see that there are unresolved externals for the image list method calls. We need to add a library to the project. To do this select the "Project|Settings" menu item. On the Project Settings dialog, Select All Configurations from the "Settings For" combo box. Then select the "Link" tab and append to the "Object/Library modules" edit box "comctl32.lib". Then click OK. If you compile the project now, it will compile successfully and the image list unresolved externals will disappear.

We still have a few things we need to do to the Toolbar window. It needs to process Command messages, resopnd to WM\_GETQUOTE messages, and resize itself. Let's conquer the latter first.

To orgainze the tooblar correctly, we should have the toolbar responsd to WM\_SIZE messages. To do this, we will add to our tooblar header file a message handler for the WM\_SIZE message and add a function definition for OnSize which WM\_SIZE messages will be sent to. Open our toolbar header file and add the lines in bold below to it as shown.

Collapse | Copy Code

BEGIN\_MSG\_MAP(CMFToolbar)

CHAIN\_MSG\_MAP\_MEMBER(m\_EditWnd)

MESSAGE\_HANDLER(WM\_CREATE, OnCreate)

MESSAGE\_HANDLER(WM\_SIZE, OnSize)

END\_MSG\_MAP()

// Handler prototypes:

// LRESULT MessageHandler(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);

// LRESULT CommandHandler(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);

// LRESULT NotifyHandler(int idCtrl, LPNMHDR pnmh, BOOL& bHandled);

LRESULT OnCreate(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);

LRESULT OnSize(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);

Now we need to implement our OnSize function. Open the toolbar source file and add the function implementation below to the end of the file.

Collapse | Copy Code

LRESULT CMFToolbar::OnSize(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)

{

// based on the size of the window area minus the size of the toolbar button,

// indent the toolbar so that we can place the edit box before the toolbar

// button. This will right justify the toolbar button in the toolbar and the

// edit box will use the vaction space to the left of the button but after the

// toolbar text as it's usable space.

RECT wndRect, btnRect;

GetClientRect(&wndRect);

::SendMessage(m\_hWnd, TB\_GETITEMRECT, 0, (LPARAM)&btnRect);

wndRect.right -= (btnRect.right - btnRect.left);

SendMessage(TB\_SETINDENT, wndRect.right - wndRect.left);

// put a small spacing gap between the edit box's right edge and the toolbar button's left edge

wndRect.right -= 3;

m\_EditWnd.MoveWindow(&wndRect, FALSE);

return 0;

}

We still need to respond to user input for the toolbar button and from the edit box when the user presses the enter key. What we want the toolbar to do is tell the web browser host to navigate to the motely fool website and retrieve the stock quotes requested. First we need for the Deskband object to tell our toolbar window what the web browser instance is so that the toolbar window can communicate with it. To do this, we will add a private member variable and a public method in which the deskband can set the web browser instance. Our window will then use the member variable set to tell the web browser where to navigate and what to retrieve.

To do this, open the toolbar header file and add the lines in bold to the file.

Collapse | Copy Code

CMFToolbar();

virtual ~CMFToolbar();

inline CEditQuote& GetEditBox() {return m\_EditWnd;};

void SetBrowser(IWebBrowser2\* pBrowser);

private:

CEditQuote m\_EditWnd;

HIMAGELIST m\_hImageList;

IWebBrowser2\* m\_pBrowser;Now, open the toolbar source file. We will update the constructor and initialize our member variable to null. Then we will update the toolbar destructor and release the member variable if it hasn't been. Then we will implement the SetBrowser method.

Initialize the web browser member variable.

Collapse | Copy Code

CMFToolbar::CMFToolbar()

: m\_hImageList(NULL)

, m\_pBrowser(NULL)

{

}

Release the web browser object if held.

Collapse | Copy Code

CMFToolbar::~CMFToolbar()

{

ImageList\_Destroy(m\_hImageList);

SetBrowser(NULL);

if (IsWindow()) DestroyWindow();

}

Implement SetBrowser()

Collapse | Copy Code

void CMFToolbar::SetBrowser(IWebBrowser2\* pBrowser)

{

if (m\_pBrowser) m\_pBrowser->Release();

m\_pBrowser = pBrowser;

if (m\_pBrowser) m\_pBrowser->AddRef();

}

If you try and compile the project, you will notice that IWebBrowser2 is undefine in our header file. This is because we need to update our stdafx.h header file to include system files that define IWebBrowser2. To do this, open stdafx.h and add the following lines in bold to the file, then recompile.

Collapse | Copy Code

extern CComModule \_Module;

#include <atlcom.h>

#include <atlwin.h>

//

// These are needed for IDeskBand

//

#include <shlguid.h>

#include <shlobj.h>

Now we can add message handlers for WM\_COMMAND and WM\_GETQUOTE to our toolbar class to handle the toolbar button being pressed and the enter key being pressed in the edit box by the user. To do this, we will need to add to our toolbar header file message handlers and function definitions for WM\_COMMAND and WM\_GETQUOTE. We will also need to add a private method which both will call if they need to to preform the same functionality (better than repeating code that does the same thing). So let's add the message handlers to the header file.

Collapse | Copy Code

BEGIN\_MSG\_MAP(CMFToolbar)

CHAIN\_MSG\_MAP\_MEMBER(m\_EditWnd)

MESSAGE\_HANDLER(WM\_CREATE, OnCreate)

MESSAGE\_HANDLER(WM\_SIZE, OnSize)

MESSAGE\_HANDLER(WM\_COMMAND, OnCommand)

MESSAGE\_HANDLER(WM\_GETQUOTE, OnGetQuote)

END\_MSG\_MAP()

// Handler prototypes:

// LRESULT MessageHandler(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);

// LRESULT CommandHandler(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);

// LRESULT NotifyHandler(int idCtrl, LPNMHDR pnmh, BOOL& bHandled);

LRESULT OnCreate(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);

LRESULT OnSize(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);

LRESULT OnCommand(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);

LRESULT OnGetQuote(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);Now we can add the function delcaration for our GetQuote privaet method.

Collapse | Copy Code

private:

CEditQuote m\_EditWnd;

HIMAGELIST m\_hImageList;

IWebBrowser2\* m\_pBrowser;

void GetQuote();

Now let's switch to our source file and implement our message handler functions and the GetQuote method. Add the code below to the end of the toolbar source file.

Collapse | Copy Code

LRESULT CMFToolbar::OnCommand(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)

{

if (!HIWORD(wParam))

{

long lSite = LOWORD(wParam);

if ( lSite == IDM\_GETQUOTE)

{

GetQuote();

return 0;

}

}

return -1;

}

LRESULT CMFToolbar::OnGetQuote(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)

{

GetQuote();

return 0;

}

void CMFToolbar::GetQuote()

{

// if we have a web browser pointer then try to navigate to The Motley Fool site to retrieve stock quotes.

if (m\_pBrowser)

{

VARIANT vEmpty;

VariantInit(&vEmpty);

m\_pBrowser->Stop();

\_bstr\_t bsSite;

// if the user has entered stock quotes then append them to the url

if (m\_EditWnd.GetWindowTextLength())

{

BSTR bstrTickers = NULL;

m\_EditWnd.GetWindowText(&bstrTickers);

bsSite = "http://quote.fool.com/news/symbolnews.asp?Symbols=";

bsSite += bstrTickers;

SysFreeString(bstrTickers);

}

// if the user has not entered any stock quotes then just take them to The Motley Fool website.

else

bsSite = "http://www.fool.com";

// have the webrowser navigate to the site URL requested depending on user input.

m\_pBrowser->Navigate(bsSite, &vEmpty, &vEmpty, &vEmpty, &vEmpty);

}

}If you try to compile, you will notice that \_bstr\_t is undefined. That is because the class is defined in comdef.h. We need to add this to our stdafx.h header file so that we can use it as well as any other class in our project (which we will need to for IInputObject). Open the stdafx.h header file and add the lines in bold to the file as indicated.

Collapse | Copy Code

#include <shlobj.h>

// needed for IInputObject and \_bstr\_t

#include <comdef.h>

Our implementation of the toolbar window is complete. Now we can move on to the Reflection window which creates our toolbars and forwards command messages to it.

The Reflection Window Details

For the Reflection Window, it's only purpose is to create the toolbar window (which it doesn't need to really do, but by doing so eases message forwarding) and forward messages to it. The reflection window is not visible, it's just a layer added so that message from the toolbar get to the toolbar. If we didn't have this window present, toolbar messages would get sent to the parent window (which we do not control) and we would never get them. This is not good since we need to respond to WM\_COMMAND messages from the toolbar. Thus the need for the reflection window. So let's create the toolbar window and the message forwarding for it.

Open the ReflectionWnd.h header file. We will need to include the toolbar header file and add a private member variable to our reflection window to create it and to pass it to the message chain. First things first, add the include statement below so we can use the CMFToolbar class.

Collapse | Copy Code

#include <commctrl.h>

#include "MFToolbar.h"Next add a private member variable to the end of the reflection window class for the toolbar as shown below.

Collapse | Copy Code

CReflectionWnd();

virtual ~CReflectionWnd();

private:

CMFToolbar m\_ToolbarWnd;

Next update the reflection window message map to forward messages to the toolbar window as shown below.

Collapse | Copy Code

BEGIN\_MSG\_MAP(CReflectionWnd)

CHAIN\_MSG\_MAP\_MEMBER(m\_ToolbarWnd)

END\_MSG\_MAP()

We will also need a public function for our deskband class to get at our toolbar window. We will do the same as we did with the EditQuote window by providing a function to get at the member variable indirectly. Add the line of code in bold below to the header file as indicated.

Collapse | Copy Code

CReflectionWnd();

virtual ~CReflectionWnd();

inline CMFToolbar& GetToolBar() { return m\_ToolbarWnd;};Lastly, we need to create the toolbar window and will do so in the WM\_CREATE message handler for our reflection window. Add the code below in bold to the reflection window header file. Then we will implement the OnCreate method in the source file.

Collapse | Copy Code

BEGIN\_MSG\_MAP(CReflectionWnd)

MESSAGE\_HANDLER(WM\_CREATE, OnCreate)

CHAIN\_MSG\_MAP\_MEMBER(m\_ToolbarWnd)

END\_MSG\_MAP()

// Handler prototypes:

// LRESULT MessageHandler(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);

// LRESULT CommandHandler(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled);

// LRESULT NotifyHandler(int idCtrl, LPNMHDR pnmh, BOOL& bHandled);

LRESULT OnCreate(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled);

Now open the ReflectionWnd.cpp source file and add the implementation of OnCreate to its end.

Collapse | Copy Code

const DWORD DEFAULT\_TOOLBAR\_STYLE =

/\*Window styles:\*/ WS\_CHILD | WS\_CLIPSIBLINGS | WS\_VISIBLE | WS\_TABSTOP |

/\*Toolbar styles:\*/ TBSTYLE\_TOOLTIPS | TBSTYLE\_FLAT | TBSTYLE\_TRANSPARENT | TBSTYLE\_LIST | TBSTYLE\_CUSTOMERASE |

TBSTYLE\_WRAPABLE |

/\*Common Control styles:\*/ CCS\_TOP | CCS\_NODIVIDER | CCS\_NOPARENTALIGN | CCS\_NORESIZE;

LRESULT CReflectionWnd::OnCreate(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)

{

RECT rect;

GetClientRect(&rect);

m\_ToolbarWnd.Create(m\_hWnd, rect, NULL, DEFAULT\_TOOLBAR\_STYLE);

return 0;

}

You will notice that we defined a constant for the toolbar style. This was done to make the code more readable.

The only thing left to do to the reflection window code is update the destructor to destory the window if it's still present.

Collapse | Copy Code

CReflectionWnd::~CReflectionWnd()

{

if (IsWindow()) DestroyWindow();

}

Finishing Off The Deskband Toolbar

All that's left is for our deskband to create the toolbar window, have the host use the toolbar window, remove the unused IPersistStream implementation, implement IInputObject (for focus control) and do some code tweaking. So lets wrap this toolbar up. Open the StockBar.h header file. Remove the following lines of code that are in bold since we moved them to our stdafx.h for our other classes to also use.

Collapse | Copy Code

#include "resource.h" // main symbols

//

// These are needed for IDeskBand

//

#include <shlguid.h>

#include <shlobj.h>

Next remove the following line from the class delcaration.

Collapse | Copy Code

public IPersistStream,The top of the class declaration should now look like this,

Collapse | Copy Code

class ATL\_NO\_VTABLE CStockBar :

public CComObjectRootEx<CComSingleThreadModel>,

public CComCoClass<CStockBar, &CLSID\_StockBar>,

public IDeskBand,

public IObjectWithSite,

public IDispatchImpl<IStockBar, &IID\_IStockBar, &LIBID\_MOTLEYFOOLLib>

{

Next scroll down to the COM Map and remove the following two lines of code,

Collapse | Copy Code

COM\_INTERFACE\_ENTRY(IPersist)

COM\_INTERFACE\_ENTRY(IPersistStream)

Your COM Map should now look like this,

Collapse | Copy Code

BEGIN\_COM\_MAP(CStockBar)

COM\_INTERFACE\_ENTRY(IStockBar)

COM\_INTERFACE\_ENTRY(IOleWindow)

COM\_INTERFACE\_ENTRY\_IID(IID\_IDockingWindow, IDockingWindow)

COM\_INTERFACE\_ENTRY(IObjectWithSite)

COM\_INTERFACE\_ENTRY\_IID(IID\_IDeskBand, IDeskBand)

COM\_INTERFACE\_ENTRY(IDispatch)

END\_COM\_MAP()

Scroll down the header file further and remove the sections of function declarations for IPersist and IPersistStream, which includes the following lines.

Collapse | Copy Code

// IPersist

public:

STDMETHOD(GetClassID)(CLSID \*pClassID);

// IPersistStream

public:

STDMETHOD(IsDirty)(void);

STDMETHOD(Load)(IStream \*pStm);

STDMETHOD(Save)(IStream \*pStm, BOOL fClearDirty);

STDMETHOD(GetSizeMax)(ULARGE\_INTEGER \*pcbSize);

There should now be nothing related to IPersist or IPersistStream between the IDockingWindow and IStockBar function declaration sections.

Now we need to remove the IPersist and IPersistStream function implementations from the StockBar.cpp source file. Find the GetClassID, IsDirty, Load, Save, and GetSizeMax function implementations and remove them from the file. They should be directly above the FocusChange Method we added earlier.

Now we can start adding code to our deskband. Open the stockbar.h header file and add the include for the reflection window class as shown below in bold.

Collapse | Copy Code

#include "resource.h" // main symbols

#include "ReflectionWnd.h"

Next, find the protected section at the end of the file and replace the line

Collapse | Copy Code

HWND m\_hWnd;with

Collapse | Copy Code

CReflectionWnd m\_ReflectWnd;If you try to compile, you will find that it will be unsuccessful since we have removed m\_hWnd and have not removed all occurances of m\_hWnd from the class source file. We will replace all occurances with more appropriate code for our Reflection window and toolbar window. Open the StockBar.cpp source file, Remove the following line from the class constructor

Collapse | Copy Code

m\_hWnd(NULL),your class constructor should now look as follows with a SetBand call added,

Collapse | Copy Code

CStockBar::CStockBar():

m\_dwBandID(0),

m\_dwViewMode(0),

m\_bShow(FALSE),

m\_bEnterHelpMode(FALSE),

m\_hWndParent(NULL),

m\_pSite(NULL)

{

m\_ReflectWnd.GetToolBar().GetEditBox().SetBand(this);

}

Next, update the RegisterAndCreateWindow function by replacing the temporary m\_hWnd construction with the Reflection Window construction. Your RegisterAndCreateWindow implementation should look as follows:

Collapse | Copy Code

BOOL CStockBar::RegisterAndCreateWindow()

{

RECT rect;

::GetClientRect(m\_hWndParent, &rect);

m\_ReflectWnd.Create(m\_hWndParent, rect, NULL, WS\_CHILD);

// The toolbar is the window that the host will be using so it is the window that is important.

return m\_ReflectWnd.GetToolBar().IsWindow();

}

As we scroll through the code, we should fix some other things. A toolbar has a default height of 22 so we need to update our GetBandInfo method so that all the y measurements are 22 not 20. We also want our Integral sizing to be non sizeable so we need to set the DBIM\_INTEGRAL x and y values to 0. While we are at it, we should fix the title of our toolbar so it says "The Motley Fool". You'll find this constant defined near the top of the source file, update it now.

We will now update the GetWindow call so that the toolbar window handle is returned and not the invalid m\_hWnd variable. Update your GetWindow method so it looks as follows,

Collapse | Copy Code

STDMETHODIMP CStockBar::GetWindow(HWND\* phwnd)

{

HRESULT hr = S\_OK;

if (NULL == phwnd)

{

hr = E\_INVALIDARG;

}

else

{

\*phwnd = m\_ReflectWnd.GetToolBar().m\_hWnd;

}

return hr;

}

Now we need to update teh CloseDW method so that it does not destory our window buy hide it. We do this much like the MFC CToolbar class does, the class destructor will destroy the window. Your CloseDW method should look as follows,

Collapse | Copy Code

STDMETHODIMP CStockBar::CloseDW(unsigned long dwReserved)

{

ShowDW(FALSE);

return S\_OK;

}

Working our way through our class implementation, we will update the next method that needs updating. Update the ShowDW class to show or hide the toolbar window which the host is using. Your ShowDW class should look as follows,

Collapse | Copy Code

STDMETHODIMP CStockBar::ShowDW(BOOL fShow)

{

m\_bShow = fShow;

m\_ReflectWnd.GetToolBar().ShowWindow(m\_bShow ? SW\_SHOW : SW\_HIDE);

return S\_OK;

}

We are almost done modifying the CStockBar class we need to do one last bit of updating. We need to modify the SetSite implementation to release the browser window that the toolbar may be using if we have a IInputObjectSite object and we need to query the OleCommandTarget's ServiceProvider for the IWebBrowser2 interface which we will then set to our toolbar. The modified parts of the SetSite method implementation are below in bold.

Collapse | Copy Code

STDMETHODIMP CStockBar::SetSite(IUnknown\* pUnkSite)

{

//If a site is being held, release it.

if(m\_pSite)

{

m\_ReflectWnd.GetToolBar().SetBrowser(NULL);

m\_pSite->Release();

m\_pSite = NULL;

}

//If punkSite is not NULL, a new site is being set.

if(pUnkSite)

{

//Get the parent window.

IOleWindow \*pOleWindow = NULL;

m\_hWndParent = NULL;

if(SUCCEEDED(pUnkSite->QueryInterface(IID\_IOleWindow, (LPVOID\*)&pOleWindow)))

{

pOleWindow->GetWindow(&m\_hWndParent);

pOleWindow->Release();

}

if(!::IsWindow(m\_hWndParent))

return E\_FAIL;

if(!RegisterAndCreateWindow())

return E\_FAIL;

//Get and keep the IInputObjectSite pointer.

if(FAILED(pUnkSite->QueryInterface(IID\_IInputObjectSite, (LPVOID\*)&m\_pSite)))

{

return E\_FAIL;

}

IWebBrowser2\* s\_pFrameWB = NULL;

IOleCommandTarget\* pCmdTarget = NULL;

HRESULT hr = pUnkSite->QueryInterface(IID\_IOleCommandTarget, (LPVOID\*)&pCmdTarget);

if (SUCCEEDED(hr))

{

IServiceProvider\* pSP;

hr = pCmdTarget->QueryInterface(IID\_IServiceProvider, (LPVOID\*)&pSP);

pCmdTarget->Release();

if (SUCCEEDED(hr))

{

hr = pSP->QueryService(SID\_SWebBrowserApp, IID\_IWebBrowser2, (LPVOID\*)&s\_pFrameWB);

pSP->Release();

\_ASSERT(s\_pFrameWB);

m\_ReflectWnd.GetToolBar().SetBrowser(s\_pFrameWB);

s\_pFrameWB->Release();

}

}

}

return S\_OK;

}

If you try to compile and use the toolbar right now, it will function partially. Tabbing and input control will not work correctly since we have yet to implement IInputObject for our deskband. Let's do that now since it's the last bit of code we will write for our simple deskband. You may also notice that the Toolbars context menu and View|Toolbars menus still say CStockBar Class. we will fix this problem in the Finishing Touches section below.

IInputObject Implementation

To get tabbing and input control to work correctly for any deskband is quite simple. You need but to implement IInputObject. The host will query our deskband to see if this interface is implemented and if it is will call the methods to see if we require input focus and let us also process messages from the user through the host. To do this, open the stockbar.h header file. To the stockbar class declaration add the line below in bold,

Collapse | Copy Code

class ATL\_NO\_VTABLE CStockBar :

public CComObjectRootEx<CComSingleThreadModel>,

public CComCoClass<CStockBar, &CLSID\_StockBar>,

public IDeskBand,

public IObjectWithSite,

public IInputObject,

public IDispatchImpl<IStockBar, &IID\_IStockBar, &LIBID\_MOTLEYFOOLLib>

{

Next scroll down to the COM Map and add an entry for IInputObject as shown below in bold,

Collapse | Copy Code

BEGIN\_COM\_MAP(CStockBar)

COM\_INTERFACE\_ENTRY(IStockBar)

COM\_INTERFACE\_ENTRY(IInputObject)

COM\_INTERFACE\_ENTRY(IOleWindow)

COM\_INTERFACE\_ENTRY\_IID(IID\_IDockingWindow, IDockingWindow)

COM\_INTERFACE\_ENTRY(IObjectWithSite)

COM\_INTERFACE\_ENTRY\_IID(IID\_IDeskBand, IDeskBand)

COM\_INTERFACE\_ENTRY(IDispatch)

END\_COM\_MAP()

Next add the following section of function declarations to your header file, I placed mine before the IStockBar section.

Collapse | Copy Code

// IInputObject

public:

STDMETHOD(HasFocusIO)(void);

STDMETHOD(TranslateAcceleratorIO)(LPMSG lpMsg);

STDMETHOD(UIActivateIO)(BOOL fActivate, LPMSG lpMsg);

All that remains is to implement these three functions. Add the function implementations below to the end of the stockbar.cpp source file.

Collapse | Copy Code

STDMETHODIMP CStockBar::HasFocusIO(void)

{

// if any of the windows in our toolbar have focus then return S\_OK else S\_FALSE.

if (m\_ReflectWnd.GetToolBar().m\_hWnd == ::GetFocus())

return S\_OK;

if (m\_ReflectWnd.GetToolBar().GetEditBox().m\_hWnd == ::GetFocus())

return S\_OK;

return S\_FALSE;

}

STDMETHODIMP CStockBar::TranslateAcceleratorIO(LPMSG lpMsg)

{

// the only window that needs to translate messages is our edit box so forward them.

return m\_ReflectWnd.GetToolBar().GetEditBox().TranslateAcceleratorIO(lpMsg);

}

STDMETHODIMP CStockBar::UIActivateIO(BOOL fActivate, LPMSG lpMsg)

{

// if our deskband is being activated then set focus to the edit box.

if(fActivate)

{

m\_ReflectWnd.GetToolBar().GetEditBox().SetFocus();

}

return S\_OK;

}

Our toolbar is functionaly done, compile, run it and see. It works as described and is fairly simple. Let's put some finishing UI touches on it for IE and our users to use.

Finishing Touches

The are only 2 finishing touches to make, One is to fix the context menu text. The other is to add button support to the main IE toolbar. Let's do them in order.

To fix the context menu text problem, open the StockBar.rgs project file and change all occurances of "StockBar Class" to "The Motley Fool Quotes". Compile it, run it, and see. While you only need to change one of them, it's nicer if they all match.

Now let's add button support for our toolbar. Update the stockbar.rgs file contents by appending the text below to it's contents.

Collapse | Copy Code

HKLM

{

Software

{

Microsoft

{

'Internet Explorer'

{

Extensions

{

ForceRemove {A26ABCF0-1C8F-46e7-A67C-0489DC21B9CC} = s 'The Motley Fool Quotes'

{

val BandClsid = s '{A6790AA5-C6C7-4BCF-A46D-0FDAC4EA90EB}'

val ButtonText = s 'The Motley Fool'

val Clsid = s '{E0DD6CAB-2D10-11D2-8F1A-0000F87ABD16}'

val 'Default Visible' = s 'Yes'

val 'Hot Icon' = s '%MODULE%,425'

val Icon = s '%MODULE%,425'

val MenuStatusBar = s 'The Motley Fool Stock Quote Toolbar'

val MenuText = s 'The Motley Fool'

}

}

}

}

}

}The replace the 425 with the id from resource.h of IDI\_MOTLEY. Also replace the BandClsid value with the GUID of our toolbar, the above values represent the source code for the article. Now compile the toolbar again. Then start IE and right click on the Standard Buttons toolbar, Select "Customize" from the context menu. Scroll down the Available toolbar buttons listbox and find "The Motley Fool" item, See Figure 12. Select it and click the Add button in the middle of the dialog. The item will move to the right as in Figure 13. Click the Close Button. You'll see the button added as shown in the before figure 14 and after figure 15.

Figure 12. CustomizeToolbar - Available Toolbar Buttons.

Figure 13. Customize Toolbar - Current Toolbar Buttons.

Figure 14. IE Standard Buttons - Before.

Figure 15. IE Standard Buttons - After.

Conclusion

While this tutorial is long hopefully the explaination was clear. From writing this tutorial it is easy to see that the RBDeskband ATL Object Wizard has some room for improvement but provided enough of a base for us to develop our simple example. In the end you can see that the toolbar we created is much like the Address bar. The differences lie in how MS implemented theirs versus how I implemented mine. As always feedback is welcome. Enjoy.

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

About the Author

Erik Thompson

Web Developer

United States

Member

Erik lives in Redmond, Washington. He works as a Senior Software Engineer specializing in C++, COM, ATL and the middle-tier and now .NET. When he isn't coding for work, he can be found trying to extend Internet Explorer with yet another Desk band or simplifying his development process with ATL Object Wizards.

He spends his free time snowboarding, mountain biking, and online gaming.