WTL for MFC Programmers

WTL

Windows
Template Library

Contents

About the Author

Michael Dunn

Software Developer (Senior)
VMware
🇺🇸United States

Michael lives in sunny Mountain View, California. He started programming with an Apple //e in 4th grade, graduated from UCLA with a math degree in 1994, and immediately landed a job as a QA engineer at Symantec, working on the Norton AntiVirus team. He pretty much taught himself Windows and **MFC** programming, and in 1999 he designed and coded a new interface for Norton AntiVirus 2000.
Mike has been a a developer at Napster and at his own lil' startup, Zabersoft, a development company he co-founded with offices in Los Angeles and Odense, Denmark. Mike is now a senior engineer at VMware.

He also enjoys his hobbies of playing pinball, bike riding, photography, and Domion on Friday nights (current favorite combo: Village + double Pirate Ship). He would get his own snooker table too if they weren't so darn big! He is also sad that he's forgotten the languages he's studied: French, Mandarin Chinese, and Japanese.

Mike was a VC MVP from 2005 to 2009.

Part I - ATL GUI Classes

This is the stuff I want you to read first, before proceeding on or posting messages to this article's discussion board.

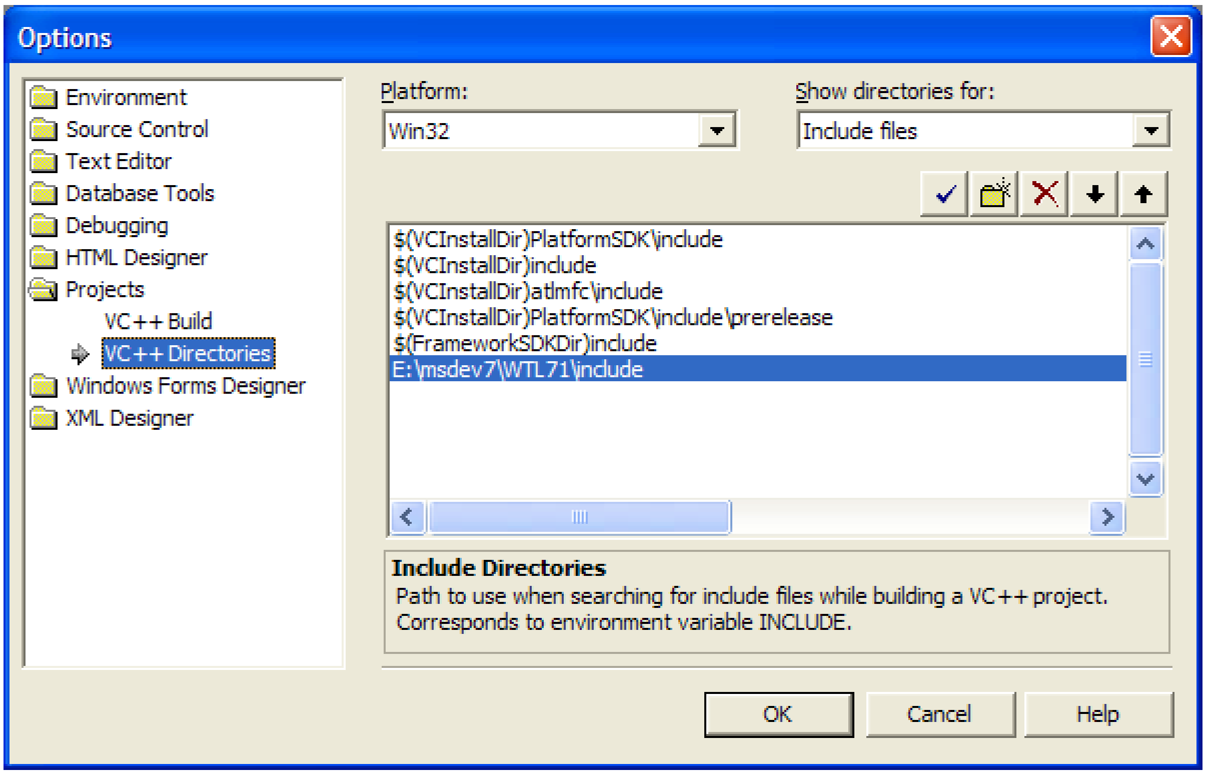
This series of articles originally covered WTL 7.0 and written for VC 6 users. Now that VC 8 is out, I felt it was about time to update the articles to cover VC 7.1. ;) (Also, the automatic 6-to-7 conversion done by VC 7.1 doesn't always go smoothly so VC 7.1 users could get stuck when trying to use the demo source code.) So as I go through and update this series, the articles will be updated to reflect new WTL 7.1 features, and I'll have VC 7.1 projects in the source downloads.

Important note for VC 2005 users: The Express edition of VC 2005 does **not** come with ATL (or MFC for that matter) so you can't build ATL or WTL projects with the Express version.

If you are using VC 6, then you need the Platform SDK. You can't use WTL without it. You can use the web install version, or download the CAB files or an ISO image and run the setup locally. Be sure you use the utility to add the SDK include and lib directories to the VC search path. You can find this in the *Visual Studio Registration* folder in the Platform SDK program group. It's a good idea to get the latest Platform SDK even if you're using VC 7 so you have the latest headers and libs.

You need WTL. Download version 7.1 from Microsoft. See the articles "Introduction to WTL - Part 1" and "Easy installation of WTL" for some tips on installing the files. Those articles are rather out-of-date now, but still contain some good info. The WTL distribution also has a readme file with installation instructions. One thing which I don't think is mentioned in those articles is how to add the WTL files to the VC include path. In VC 6, click *Tools|Options* and go to the *Directories* tab. In the *Show directories for* combo box, select *Include files*. Then add a new entry that points to the directory where you put the WTL header files. In VC 7, click *Tools|Options*, click *Projects* then *VC++ Directories*. In the *Show directories for* combo box, select *Include files*. Then add a new entry that points to the directory where you put the WTL header files.

Important: While we're on the subject of the VC 7 include path, you **must** make a change to the default directory list if you haven't updated your Platform SDK. Make sure that `$(VCInstallDir)PlatformSDK\include` is first in the list, above `($VCInstallDir)include`, as shown here:



Platform: Win32

Show directories for: Include files

```
$(VCInstallDir)PlatformSDK\include
$(VCInstallDir)include
$(VCInstallDir)atlmfc\include
$(VCInstallDir)PlatformSDK\include\prerelease
$(FrameworkSDKDir)include
E:\msdev7\WTL71\include
```

Include Directories
Path to use when searching for include files while building a VC++ project. Corresponds to environment variable INCLUDE.

You need to know MFC, and know it well enough that you understand what's behind the message map macros, and can edit the code marked "DO NOT EDIT" with no problems.

You need to know Win32 API programming, and know it well. If you learned Windows programming by going straight into MFC, without learning how messages work at the API level, you are unfortunately going to have trouble in WTL. If you don't know what a message's `WPARAM` and `LPARAM` mean, you should read other articles on API-level programming (there are lots of them here at CodeProject) so you understand.

You need to know C++ template syntax. See the VC Forum FAQ for links to C++ FAQs and template FAQs.

Since I haven't used VC 8 yet, I don't know if the sample code will compile on 8. Hopefully the 7-to-8 upgrade process will work better than the 6-to-7 process did. Please post on this article's forum if you have any trouble with VC 8.

Introduction to the Series

WTL rocks. It does. It has a lot of the power of MFC's GUI classes, yet produces substantially smaller executables. If you're like me, and learned GUI programming with MFC, you've become quite comfortable with all the control wrappers MFC provides, as well as with the flexible message handling built into MFC. If you're also like me and don't relish the idea of several hundred K of MFC framework getting added to your programs, WTL is for you. However, there are a few barriers we have to overcome:

- ATL-style templates look weird at first.
- No ClassWizard support, so writing message maps involves some manual labor.
- No documentation in MSDN, you need to find it somewhere else, or even look at the WTL source.
- No reference books you can buy and keep on your shelf.
- It has that "not officially supported by Microsoft" stigma
- ATL/WTL windowing is sufficiently different from MFC windowing that not all your knowledge transfers over.

On the other hand, the benefits of WTL are:

- No complex doc/view framework to learn or work around.
- Has some essential UI features from MFC, like DDX/DDV and "update command UI" functionality.
- Actually improves on some MFC features (e.g., more flexible splitter windows)
- Much smaller executables compared to a statically-linked MFC app.
- You can apply bug fixes to WTL yourself and not affect existing apps (compare this with replacing the MFC/CRT DLLs to fix a bug in one app, and having other apps break)
- If you still need MFC, MFC and ATL/WTL windows can co-exist peacefully (for a prototype at work, I ended up creating an MFC `CFrameWnd` that contained a WTL `CSplitterWindow`, which contained MFC `CDialog`s -- that wasn't me showing off, it was modifying existing MFC code but using the nicer WTL splitter).

In this series, I will first introduce the ATL windowing classes. WTL is, after all, a set of add-on classes to ATL, so a good understanding of ATL windowing is essential. Once I've covered that, I'll introduce WTL features and show how they make GUI programming a lot easier.

Introduction to Part I

WTL rocks. But before we get to why, we need to cover ATL first. WTL is a set of add-on classes to ATL, and if you've been strictly an MFC programmer in the past, you may never have encountered the ATL GUI classes. So please bear with me if I don't get to WTL right off the bat; the detour into ATL is necessary.

In this first part, I will give a little background on ATL, cover some essential points you need to know before writing ATL code, quickly explain those funky ATL templates, and cover the basic ATL windowing classes.

ATL Background

ATL and WTL history

The Active Template Library... kind of an odd name, isn't it? Old-timers might remember that it was originally the Active**X** Template Library, which is a more accurate name, since ATL's goal is to make writing COM objects and ActiveX controls easier. (ATL was also developed during the time that Microsoft was naming new products ActiveX-something, just as new products nowadays are called something.NET.) Since ATL is really about writing COM objects, it only has the most basic of GUI classes, the equivalent of MFC's `CWnd` and `CDialog`. Fortunately, the GUI classes are flexible enough to allow for something like WTL to be built on top of them.

WTL had two major revisions as a Microsoft-owned project, numbered 3 and 7. (The version numbers were chosen to match the ATL version numbers, that's why they're not 1 and 2.) Version 3.1 is obsolete now so it will not be covered in this series. Version 7.0 was a major update to version 3, and version 7.1 added some bug fixes and minor features.

Following version 7.1, Microsoft made WTL an open-source project, hosted at Sourceforge. The latest release from that site is version 7.5. I have not looked at 7.5 yet, so this series will not cover 7.5 at this time. (I'm already behind by two versions! I'll catch up eventually.)

ATL-style templates

Even if you can read C++ templates without getting a headache, there are two things ATL does that might trip you up at first. Take this class for example:

```
class CMyWnd : public CWindowImpl<CMyWnd>
{
...
};
```

That actually is legal, because the C++ spec says that immediately after the `class CMyWnd` part, the name `CMyWnd` is defined and can be used in the inheritance list.

The reason for having the class name as a template parameter is so ATL can do the second tricky thing, compile-time virtual function calls.

To see this in action, look at this set of classes:

```
template <class T>
class B1
{
public:
    void SayHi()
    {
    T* pT = static_cast<T*>(this);   // HUH?? I'll explain this below


        pT->PrintClassName();
    }

    void PrintClassName() { cout << "This is B1"; }
};

class D1 : public B1<D1>
{
    // No overridden functions at all

};

class D2 : public B1<D2>
{
    void PrintClassName() { cout << "This is D2"; }
};

int main()
{
D1 d1;
D2 d2;

    d1.SayHi();    // prints "This is B1"

    d2.SayHi();    // prints "This is D2"


    return 0;
}
```

The static_cast<T*>(this) is the trick here. It casts this, which is of type B1*, to either D1* or D2* depending on which specialization is being invoked. Because template code is generated at compile-time, this cast is guaranteed to be safe, as long as the inheritance list is written correctly. (If you wrote class D3 : public B1<D2> you'd be in trouble.) It's safe because the this object can *only* be of type D1* or D2* (as appropriate), and nothing else. Notice that this is almost exactly like normal C++ polymorphism, except that the SayHi() method isn't virtual.

To explain how this works, let's look at each call to SayHi(). In the first call, the specialization B1<D1> is being used, so the SayHi() code expands to:

```
void B1<D1>::SayHi()
{
D1* pT = static_cast<D1*>(this);

    pT->PrintClassName();
}
```

Since D1 does not override PrintClassName(), D1's base classes are searched. B1 has a PrintClassName() method, so that is the one called.

Now, take the second call to SayHi(). This time, it's using the specialization B1<D2>, and SayHi() expands to:

```
void B1<D2>::SayHi()
{
D2* pT = static_cast<D2*>(this);

    pT->PrintClassName();
}
```

This time, D2 *does* contain a PrintClassName() method, so that is the one that gets called.

The benefits of this technique are:

- It doesn't require using pointers to objects.
- It saves memory because there is no need for vtbls.
- It's impossible to call a virtual function through a null pointer at runtime because of an uninitialized vtbl.
- All function calls are resolved at compile time, so they can be optimized.

While the saving of a vtbl doesn't seem significant in this example (it would only be 4 bytes), think of the case where there are 15 base classes, some of those containing 20 methods, and the savings adds up.

ATL Windowing Classes

OK, enough background! Time to dive into ATL. ATL is designed with a strict interface/implementation division, and that's evident in the windowing classes. This is similar to COM, where interface definitions are completely separate from an implementation (or possibly several implementations).

ATL has one class that defines the "interface" for a window, that is, what can be done with a window. This class is called `CWindow`. It is nothing more than a wrapper around an `HWND`, and it provides wrappers for almost all of the User32 APIs that take an `HWND` as the first parameter, such as `SetWindowText()` and `DestroyWindow()`. `CWindow` has a public member `m_hWnd` that you can access if you need the raw `HWND`. `CWindow` also has a `operator HWND` method, so you can pass a `CWindow` object to a function that takes an `HWND`. There is no equivalent to `CWnd::GetSafeHwnd()`.

`CWindow` is very different from MFC's `CWnd`. `CWindow` objects are inexpensive to create, since there is only one data member, and there is no equivalent to the object maps that MFC keeps internally to map `HWND`s to `CWnd` objects. Also unlike `CWnd`, when a `CWindow` object goes out of scope, *the associated window is not destroyed*. This means you don't have to remember to detach any temp `CWindow` objects you might create.

The ATL class that has the implementation of a window is `CWindowImpl`. `CWindowImpl` contains the code for such things as window class registration, window subclassing, message maps, and a basic `WindowProc()`. This is unlike MFC where everything is in one class, `CWnd`.

There are also two separate classes that contain the implementation of a dialog box, `CDialogImpl` and `CAxDialogImpl`. `CDialogImpl` is used for plain dialogs, while `CAxDialogImpl` is used for dialogs that host ActiveX controls.

Defining a Window Implementation

Any non-dialog window you create will derive from `CWindowImpl`. Your new class needs to contain three things:

1. A window class definition
2. A message map

3. The default styles to use for the window, called the *window traits*

The window class definition is done using the DECLARE_WND_CLASS or DECLARE_WND_CLASS_EX macro. Both of these define an ATL struct CWndClassInfo that wraps the WNDCLASSEX struct. DECLARE_WND_CLASS lets you specify the new window class name and uses default values for the other members, while DECLARE_WND_CLASS_EX lets you also specify a class style and window background color. You can also use NULL for the class name, and ATL will generate a name for you.

Let's start out a new class definition, and I'll keep adding to it as we go through this section.

```
class CMyWindow : public CWindowImpl<CMyWindow>
{
public:
    DECLARE_WND_CLASS(_T("My Window Class"))
};
```

Next comes the message map. ATL message maps are much simpler than MFC maps. An ATL map expands into a big switch statement; the switch looks for the right handler and calls the corresponding function. The macros for the message map are BEGIN_MSG_MAP and END_MSG_MAP. Let's add an empty map to our window.

```
class CMyWindow : public CWindowImpl<CMyWindow>
{
public:
    DECLARE_WND_CLASS(_T("My Window Class"))

    BEGIN_MSG_MAP(CMyWindow)
    END_MSG_MAP()
};
```

I'll cover how to add handlers to the map in the next section. Finally, we need to define the *window traits* for our class. Window traits are a combination of window styles and extended window styles that are used when creating the window. The styles are specified as template parameters so the caller doesn't have to be bothered with getting the styles right when it creates our window. Here's a sample traits definition using the ATL class CWinTraits:

```
typedef CWinTraits<WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                   WS_EX_APPWINDOW> CMyWindowTraits;
```

```
class CMyWindow : public CWindowImpl<CMyWindow, CWindow, CMyWindowTraits>
{
public:
    DECLARE_WND_CLASS(_T("My Window Class"))

    BEGIN_MSG_MAP(CMyWindow)
    END_MSG_MAP()
};
```

The caller *can* override the styles in the `CMyWindowTraits` definition, but generally this is not necessary. ATL also has a few predefined `CWinTraits` specializations, one of which is perfect for top-level windows like ours, `CFrameWinTraits`:

```
typedef CWinTraits<WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN |
                     WS_CLIPSIBLINGS,
                   WS_EX_APPWINDOW | WS_EX_WINDOWEDGE>
        CFrameWinTraits;
```

Filling in the message map

The ATL message map is one area that is lacking in developer-friendliness, and something that WTL greatly improves on. ClassView does at least give you the ability to add message handers, however ATL doesn't have message-specific macros and automatic parameter unpacking like MFC does. In ATL, there are just three types of message handlers, one for `WM_NOTIFY`, one for `WM_COMMAND`, and one for everything else. Let's start by adding handlers for `WM_CLOSE` and `WM_DESTROY` to our window.

```
class CMyWindow : public CWindowImpl<CMyWindow, CWindow, CFrameWinTraits>
{
public:
    DECLARE_WND_CLASS(_T("My Window Class"))

    BEGIN_MSG_MAP(CMyWindow)
        MESSAGE_HANDLER(WM_CLOSE, OnClose)
        MESSAGE_HANDLER(WM_DESTROY, OnDestroy)
    END_MSG_MAP()

    LRESULT OnClose(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
    {
        DestroyWindow();
        return 0;
    }

    LRESULT OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
    {
        PostQuitMessage(0);
        return 0;
    }
};
```

You'll notice that the handlers get the raw WPARAM and LPARAM values; you have to unpack them yourself when a message uses those parameters. There is also a fourth parameter, bHandled. This parameter is set to TRUE by ATL before the handler is called. If you want ATL's default WindowProc() to handle the message as well after your handler returns, you set bHandled to FALSE. This is different than MFC, where you have to explicitly call the base-class implementation of a message handler.

Let's add a WM_COMMAND handler as well. Assuming our window's menu has an About item with ID IDC_ABOUT:

```
class CMyWindow : public CWindowImpl<CMyWindow, CWindow, CFrameWinTraits>
{
public:
    DECLARE_WND_CLASS(_T("My Window Class"))

    BEGIN_MSG_MAP(CMyWindow)
        MESSAGE_HANDLER(WM_CLOSE, OnClose)
        MESSAGE_HANDLER(WM_DESTROY, OnDestroy)
        COMMAND_ID_HANDLER(IDC_ABOUT, OnAbout)
    END_MSG_MAP()

    LRESULT OnClose(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
    {
        DestroyWindow();
        return 0;
    }

    LRESULT OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
    {
        PostQuitMessage(0);
        return 0;
    }

    LRESULT OnAbout(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled)
    {
        MessageBox ( _T("Sample ATL window"), _T("About MyWindow") );
        return 0;
    }
};
```

Notice that the COMMAND_HANDLER macro does unpack the message parameters for you. The NOTIFY_HANDLER macro is similar, and unpacks the WM_NOTIFY message parameters.

Advanced Message Maps and Mix-in Classes

One major difference in ATL is that *any* C++ class can handle messages, unlike MFC where message-handling duties are split between CWnd and CCmdTarget, plus several classes that have a PreTranslateMessage() method. This ability lets us write what are commonly called "mix-in" classes, so that we can add features to our window simply by adding classes to the inheritance list.

A base class with a message map is usually a template that takes the derived class name as a template parameter, so it can access members of the derived class like m_hWnd (the HWND member in CWindow). Let's look at a mix-in class that paints the window background by handling WM_ERASEBKGND.

```
template <class T, COLORREF t_crBrushColor>
class CPaintBkgnd
{
public:
    CPaintBkgnd() { m_hbrBkgnd = CreateSolidBrush(t_crBrushColor); }
    ~CPaintBkgnd() { DeleteObject ( m_hbrBkgnd ); }

    BEGIN_MSG_MAP(CPaintBkgnd)
        MESSAGE_HANDLER(WM_ERASEBKGND, OnEraseBkgnd)
    END_MSG_MAP()

    LRESULT OnEraseBkgnd(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
    {
    T*   pT = static_cast<T*>(this);
    HDC  dc = (HDC) wParam;
    RECT rcClient;

        pT->GetClientRect ( &rcClient );
        FillRect ( dc, &rcClient, m_hbrBkgnd );
        return 1;    // we painted the background
    }

protected:
    HBRUSH m_hbrBkgnd;
};
```

Let's go through this new class. First, `CPaintBkgnd` has two template parameters: the name of the derived class that is using `CPaintBkgnd`, and a color to use for the background. (The `t_` prefix is often used for template parameters that are plain values.)

The constructor and destructor are pretty simple, they create and destroy a brush of the color passed as `t_crBrushColor`. Next comes the message map, which handles `WM_ERASEBKGND`. Finally, there's the `OnEraseBkgnd()` handler which fills in the window with the brush created in the constructor. There are two things of note going on in `OnEraseBkgnd()`. First, it uses the derived class's window functions (namely `GetClientRect()`). How do we know that there even *is* a `GetClientRect()` in the derived class? The code wouldn't compile if there weren't! The compiler checks that the derived class `T` contains the methods that we call through the `pT` variable. Second, `OnEraseBkgnd()` has to unpack the device context from `wParam`.

To use this mix-in class with our window, we do two things. First, we add it to the inheritance list:

```
class CMyWindow : public CWindowImpl<CMyWindow, CWindow, CFrameWinTraits>,
                  public CPaintBkgnd<CMyWindow, RGB(0,0,255)>
```

Second, we need to get `CMyWindow` to pass messages along to `CPaintBkgnd`. This is called *chaining* message maps. In the `CMyWindow` message map, we add the `CHAIN_MSG_MAP` macro:

```
class CMyWindow : public CWindowImpl<CMyWindow, CWindow, CFrameWinTraits>,
                  public CPaintBkgnd<CMyWindow, RGB(0,0,255)>
{
...
typedef CPaintBkgnd<CMyWindow, RGB(0,0,255)> CPaintBkgndBase;

    BEGIN_MSG_MAP(CMyWindow)
        MESSAGE_HANDLER(WM_CLOSE, OnClose)
        MESSAGE_HANDLER(WM_DESTROY, OnDestroy)
        COMMAND_HANDLER(IDC_ABOUT, OnAbout)
        CHAIN_MSG_MAP(CPaintBkgndBase)
    END_MSG_MAP()
...
};
```

Any messages that reach the CHAIN_MSG_MAP line without being handled are passed on to the map in CPaintBkgnd. Note that WM_CLOSE, WM_DESTROY, and IDC_ABOUT will **not** be chained, because once those are handled, the message map search ends. The typedef is necessary because CHAIN_MSG_MAP is a preprocessor macro that takes one parameter; if we wrote CPaintBkgnd<CMyWindow, RGB(0,0,255)> as the parameter, the commas would make the preprocessor think that we're calling it with more than one parameter.

You could conceivably have several mix-in classes in the inheritance list, each with a CHAIN_MSG_MAP macro so that messages are passed to it. This is different from MFC, where each CWnd-derived class can have only one base class, and MFC automatically passes unhandled messages to the base class.

Structure of an ATL EXE

Now that we have a complete (if not entirely useful) main window, let's see how to use it in a program. ATL EXEs have one or more global variables that roughly correspond to the global CWinApp variable in an MFC program (normally called theApp). This area of ATL changed radically between VC6 and VC7, so I will cover the two versions separately.

In VC 6

An ATL executable contains a global CComModule variable that *must* be called _Module. Here's how we start our *stdafx.h*:

```
// stdafx.h:

#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <atlbase.h>       // Base ATL classes

extern CComModule _Module;  // Global _Module

#include <atlwin.h>        // ATL windowing classes
```

atlbase.h will include the basic Windows headers, so there's no need to include *windows.h*, *tchar.h*, etc. In our CPP file, we declare the `_Module` variable:

```
// main.cpp:

CComModule _Module;
```

`CComModule` has explicit initialization and shutdown functions that we need to call in `WinMain()`, so let's start with those:

```
// main.cpp:

CComModule _Module;

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hInstPrev,
                LPSTR szCmdLine, int nCmdShow)
{
    _Module.Init(NULL, hInst);
    _Module.Term();
}
```

The first parameter to `Init()` is only used in COM servers. Since our EXE is not a server, we can pass `NULL`. ATL doesn't provide its own `WinMain()` or message pump like MFC, so to get our program running, we create a `CMyWindow` object and add a message pump.

```
// main.cpp:

#include "MyWindow.h"

CComModule _Module;

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hInstPrev,
                LPSTR szCmdLine, int nCmdShow)
{
    _Module.Init(NULL, hInst);

CMyWindow wndMain;
MSG msg;

    // Create & show our main window

    if ( NULL == wndMain.Create ( NULL, CWindow::rcDefault,
                                  _T("My First ATL Window") ))
        {
        // Bad news, window creation failed

        return 1;
        }

    wndMain.ShowWindow(nCmdShow);
    wndMain.UpdateWindow();

    // Run the message loop

    while ( GetMessage(&msg, NULL, 0, 0) > 0 )
        {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
        }

    _Module.Term();
    return msg.wParam;
}
```

The only unusual thing in the above code is `CWindow::rcDefault`, which is a `RECT` member of `CWindow`. Using that as the window's initial `RECT` is like using `CW_USEDEFAULT` for the width and height with the `CreateWindow()` API.

Under the hood, ATL uses some assembly-language black magic to connect the main window's handle to its corresponding `CMyWindow` object. The upside of this is that there is no problem passing `CWindow` objects between threads, something that fails miserably with `CWnd`s in MFC.

In VC 7

ATL 7 split up the module-management code into several classes. `CComModule` is still present for backwards-compatibility, but code written in VC 6 that is converted by VC 7 doesn't always compile cleanly (if at all) so I'll cover the new class here.

In VC 7, the ATL headers automatically declare global instances of all the module classes, and the `Init()` and `Term()` methods are called for you, so those manual steps aren't necessary. Our stdafx.h therefore looks like:

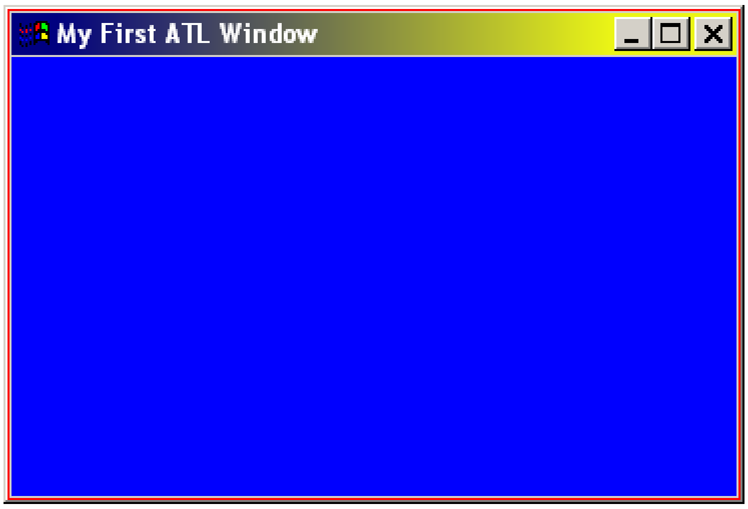
```
// stdafx.h:
```

```
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <atlbase.h>       // Base ATL classes

#include <atlwin.h>        // ATL windowing classes
```

The `WinMain()` function doesn't call any `_Module` methods, and looks like:

```
// main.cpp:

#include "MyWindow.h"


int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hInstPrev,
                LPSTR szCmdLine, int nCmdShow)
{
CMyWindow wndMain;
MSG msg;

    // Create & show our main window

    if ( NULL == wndMain.Create ( NULL, CWindow::rcDefault,
                           _T("My First ATL Window") ))
        {
        // Bad news, window creation failed

        return 1;
        }

    wndMain.ShowWindow(nCmdShow);
    wndMain.UpdateWindow();

    // Run the message loop

    while ( GetMessage(&msg, NULL, 0, 0) > 0 )
        {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
        }

    return msg.wParam;
}
```

Here's what our window looks like:



Nothing particularly exciting, I'll admit. To spice it up, we'll add an *About* menu item that shows a dialog.

Dialogs in ATL

As mentioned earlier, ATL has two dialog classes. We'll use `CDialogImpl` for our about dialog. Making a new dialog class is almost like making a new frame window class; there are just two differences:

1. The base class is `CDialogImpl` instead of `CWindowImpl`.
2. You need to define a public member called `IDD` that holds the resource ID of the dialog.

Here is the start of a new class definition for an about dialog:

```
class CAboutDlg : public CDialogImpl<CAboutDlg>
{
public:
    enum { IDD = IDD_ABOUT };

    BEGIN_MSG_MAP(CAboutDlg)
    END_MSG_MAP()
};
```

ATL has no built-in handlers for the *OK* and *Cancel* buttons, so we need to code them ourselves, along with a `WM_CLOSE` handler that is called if the user clicks the close button in the title bar. We also need to handle `WM_INITDIALOG` so that the keyboard focus is set properly when the dialog appears. Here is the complete class definition with message handlers.

```
class CAboutDlg : public CDialogImpl<CAboutDlg>
{
public:
    enum { IDD = IDD_ABOUT };

    BEGIN_MSG_MAP(CAboutDlg)
        MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
        MESSAGE_HANDLER(WM_CLOSE, OnClose)
        COMMAND_ID_HANDLER(IDOK, OnOKCancel)
        COMMAND_ID_HANDLER(IDCANCEL, OnOKCancel)
    END_MSG_MAP()

    LRESULT OnInitDialog(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
    {
        CenterWindow();
        return TRUE;    // let the system set the focus

    }

    LRESULT OnClose(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
    {
        EndDialog(IDCANCEL);
        return 0;
    }

    LRESULT OnOKCancel(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled)
    {
        EndDialog(wID);
        return 0;
    }
};
```

I used one handler for both *OK* and *Cancel* to demonstrate the `wID` parameter, which is set to either `IDOK` or `IDCANCEL`, depending on which button is clicked.

Showing the dialog is similar to MFC, you create an object of the new class and call `DoModal()`. Let's go back to our main window and add a menu with an *About* item that shows our new about dialog. We'll need to add two message handlers, one for `WM_CREATE` and one for the new menu item `IDC_ABOUT`.

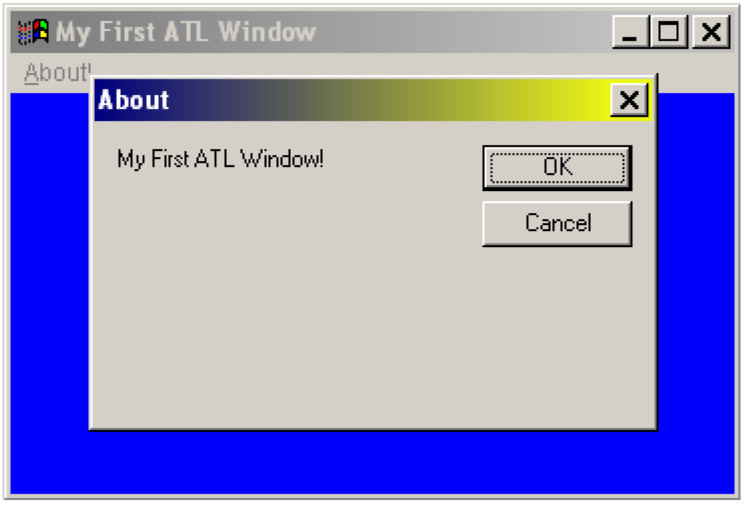
```
class CMyWindow : public CWindowImpl<CMyWindow, CWindow, CFrameWinTraits>,
                 public CPaintBkgnd<CMyWindow,RGB(0,0,255)>
{
public:
    BEGIN_MSG_MAP(CMyWindow)
        MESSAGE_HANDLER(WM_CREATE, OnCreate)
        COMMAND_ID_HANDLER(IDC_ABOUT, OnAbout)
        // ...

        CHAIN_MSG_MAP(CPaintBkgndBase)
    END_MSG_MAP()

    LRESULT OnCreate(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
    {
    HMENU hmenu = LoadMenu ( _Module.GetResourceInstance(),  // _AtlBaseModule
in VC7

                         MAKEINTRESOURCE(IDR_MENU1) );

        SetMenu ( hmenu );
        return 0;
    }

    LRESULT OnAbout(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL& bHandled)
    {
    CAboutDlg dlg;

        dlg.DoModal();
        return 0;
    }
    // ...

};
```

One small difference in modal dialogs is where you specify the dialog's parent window. In MFC you pass the parent to the CDialog constructor, but in ATL you pass it as the first parameter to DoModal(). If you don't specify a window, as in the code above, ATL uses the result of GetActiveWindow() (which will be our frame window) as the parent.

The LoadMenu() call also demonstrates one of the CComModule methods, GetResourceInstance(). This returns the HINSTANCE of a module that holds resources, similar to AfxGetResourceHandle(). The default behavior is to return the HINSTANCE of the EXE. (There is also CComModule::GetModuleInstance(), which functions like AfxGetInstanceHandle().)

Note that `OnCreate()` is different between VC6 and 7, which is due to the different module-management classes. `GetResourceInstance()` is now in `CAtlBaseModule`, and we call the global `_AtlBaseModule` object that ATL sets up for us.

Here is our revised main window and the about dialog:



I'll Get to WTL, I Promise!

But it will be in Part 2. Since I'm writing these articles for MFC developers, I felt that it was best to do an intro to ATL first, before diving into WTL. If this has been your first exposure to ATL, now might be a good time to write some simple apps on your own, so you get the hang of message maps and using mix-in classes. You can also experiment with ClassView's support for ATL message maps, as it can add message handlers for you. To get started in VC 6, right-click the *CMyWindow* item and pick *Add Windows Message Handler* on the context menu. In VC 7, right-click the *CMyWindow* item and pick *Properties* on the context menu. In the properties window, click the *Messages* button on the toolbar to see a list of window messages. You can add a handler for a message by going to its row, clicking the right column to turn it into a combo box, clicking the combo box arrow, then clicking the `<Add>` item in the drop-down list.

In Part II, I will cover the base WTL windowing classes, the WTL AppWizard, and the much nicer message map macros.

Part II - WTL GUI Base Classes

Introduction to Part II

OK, time to actually start talking about WTL! In this part I'll cover the basics of writing a main frame window and cover some of the welcome improvements WTL gives, such as UI updating and better message maps. To get the most out of this part, you should have WTL installed so the header files are in the VC search path, and the AppWizard is in the appropriate directory. The WTL distribution comes with instructions on how to install the AppWizard, so consult the docs.

Remember, if you encounter any problems installing WTL or compiling the demo code, read the readme section of Part I before posting questions here.

WTL Overview

The WTL classes can be divided into a few major categories:

1. Frame window implementation - `CFrameWindowImpl`, `CMDIFrameWindowImpl`
2. Control wrappers - `CButton`, `CListViewCtrl`
3. GDI wrappers - `CDC`, `CMenu`
4. Special UI features - `CSplitterWindow`, `CUpdateUI`, `CDialogResize`, `CCustomDraw`
5. Utility classes and macros - `CString`, `CRect`, `BEGIN_MSG_MAP_EX`

This article will cover frame windows in depth and touch on some of the UI features and utility classes. Most of the classes are stand-alone classes, although a few such as `CDialogResize` are mix-ins.

Beginning a WTL EXE

If you don't use the WTL AppWizard (we'll get to that later), a WTL EXE begins much like an ATL EXE. The sample code in this article will be another frame window as in Part I, but will be a bit less trivial in order to show some WTL features.

For this section, we'll start a new EXE from scratch. The main window will show the current time in its client area. Here is a basic *stdafx.h*:

```
#define STRICT
#define WIN32_LEAN_AND_MEAN
```

```
#define _WTL_USE_CSTRING

#include <atlbase.h>      // base ATL classes
#include <atlapp.h>       // base WTL classes
extern CAppModule _Module; // WTL version of CComModule
#include <atlwin.h>       // ATL GUI classes
#include <atlframe.h>     // WTL frame window classes
#include <atlmisc.h>      // WTL utility classes like CString
#include <atlcrack.h>     // WTL enhanced msg map macros
```

atlapp.h is the first WTL header you include. It contains classes for message handling and `CAppModule`, a class derived from `CComModule`. You should also define `_WTL_USE_CSTRING` if you plan on using `CString`, because `CString` is defined in *atlmisc.h* yet there are other headers that come before *atlmisc.h* which have features that use `CString`. Defining `_WTL_USE_CSTRING` makes atlapp.h forward-declare the `CString` class so those other headers know what a `CString` is.

(Note that we need a global `CAppModule` variable, even though in Part I this was not necessary. `CAppModule` has some features relating to idle processing and UI updating that we need, so we need that `CAppModule` to be present.)

Next let's define our frame window. SDI windows like ours are derived from `CFrameWindowImpl`. The window class is defined with `DECLARE_FRAME_WND_CLASS` instead of `DECLARE_WND_CLASS`. Here's the beginning of our window definition in *MyWindow.h*:

```
// MyWindow.h:
class CMyWindow : public CFrameWindowImpl<CMyWindow>
{
public:
    DECLARE_FRAME_WND_CLASS(_T("First WTL window"), IDR_MAINFRAME);

    BEGIN_MSG_MAP(CMyWindow)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMyWindow>)
    END_MSG_MAP()
};
```

`DECLARE_FRAME_WND_CLASS` takes two parameters, the window class name (which can be NULL to have ATL generate a name for you), and a resource ID. WTL will look for an icon, menu, and accelerator table with that ID, and load them when the window is created. It will also look for a string with that ID and use it as the window title. We also chain messages to CFrameWindowImpl as it has some message handlers of its own (most notably WM_SIZE and WM_DESTROY).

Now let's look at WinMain(). It is pretty similar to the WinMain() we had in Part I, the difference is in the call to create the main window.

```
// main.cpp:
#include "stdafx.h"
#include "MyWindow.h"

CAppModule _Module;

int APIENTRY WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPSTR lpCmdLine, int nCmdShow )
{
   _Module.Init ( NULL, hInstance );

CMyWindow wndMain;
MSG msg;

   // Create the main window
   if ( NULL == wndMain.CreateEx() )
      return 1;      // Window creation failed

   // Show the window
   wndMain.ShowWindow ( nCmdShow );
   wndMain.UpdateWindow();

   // Standard Win32 message loop
   while ( GetMessage ( &msg, NULL, 0, 0 ) > 0 )
      {
      TranslateMessage ( &msg );
      DispatchMessage ( &msg );
      }

   _Module.Term();
   return msg.wParam;
}
```

CFrameWindowImpl has a CreateEx() method that has the most common default values, so we don't need to specify any parameters. CFrameWindowImpl will also handle loading resources as explained earlier, so you should make some dummy resources now with IDs of IDR_MAINFRAME, or check out the sample code accompanying the article.

If you run this now, you'll see the main frame window, but of course it doesn't actually *do* anything yet. We'll need to add some message handlers to do stuff, so now is a good time to cover the WTL message map macros.

WTL Message Map Enhancements

One of the cumbersome and error-prone things you do when using the Win32 API is unpacking parameters from the `WPARAM` and `LPARAM` data sent with a message. Unfortunately, ATL doesn't help much, and we still have to unpack data from all messages aside from `WM_COMMAND` and `WM_NOTIFY`. But WTL comes to the rescue here!

WTL's enhanced message map macros are in *atlcrack.h*. (The name comes from "message cracker", a term used for similar macros in *windowsx.h*.) The initial step in using these macros differs on VC 6 and VC 7, this note in *atlcrack.h* explains it:

For ATL 3.0, a message map using cracked handlers **must** use `BEGIN_MSG_MAP_EX`.

For ATL 7.0/7.1, you can use `BEGIN_MSG_MAP` for `CWindowImpl`/`CDialogImpl` derived classes, but must use `BEGIN_MSG_MAP_EX` for classes that don't derive from `CWindowImpl`/`CDialogImpl`.

So if you are using VC 6, you would make the change in *MyWindow.h*:

```
// MyWindow.h, VC6 only:
class CMyWindow : public CFrameWindowImpl<CMyWindow>
{
public:
    BEGIN_MSG_MAP_EX(CMyWindow)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMyWindow>)
    END_MSG_MAP()
};
```

(The `_EX` macro is needed in VC 6 because it contains some code that the message handler macros use. For the sake of readability, I won't show the VC 6 and VC 7 versions of the header file here, since they only differ in that one macro. Just keep in mind that the `_EX` macro is not needed in VC 7.)

For our clock program, we'll need to handle `WM_CREATE` and set a timer. The WTL message handler for a message is called `MSG_` followed by the message name, for example `MSG_WM_CREATE`. These macros take just the name of the handler, so let's add one for `WM_CREATE`:

```
class CMyWindow : public CFrameWindowImpl<CMyWindow>
{
public:
    BEGIN_MSG_MAP_EX(CMyWindow)
        MSG_WM_CREATE(OnCreate)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMyWindow>)
    END_MSG_MAP()

    // OnCreate(...) ?
};
```

WTL message handlers look a lot like MFC's, where each handler has a different prototype depending on what parameters are passed with the message. But since we don't have a wizard to write the handler, we'll have to find the prototype ourselves. Fortunately, VC can help out. Put the cursor on the "MSG_WM_CREATE" text and press F12 to go to the definition of that macro. On VC 6, VC will rebuild the project first to build its browse info database. Once that's done, VC will open *atlcrack.h* at the definition of `MSG_WM_CREATE`:

```
#define MSG_WM_CREATE(func) \
    if (uMsg == WM_CREATE) \
    { \
        SetMsgHandled(TRUE); \
        lResult = (LRESULT)func((LPCREATESTRUCT)lParam); \
        if(IsMsgHandled()) \
            return TRUE; \
    }
```

The underlined code is the important part, it's the actual call to the handler, and it tells us the handler returns an `LRESULT` and takes one parameter, a `LPCREATESTRUCT`. Notice that there is no `bHandled` parameter like the ATL macros use. The `SetMsgHandled()` function replaces that parameter; I'll explain this more shortly.

Now we can add an `OnCreate()` handler to our window class:

```
class CMyWindow : public CFrameWindowImpl<CMyWindow>
{
public:
    BEGIN_MSG_MAP_EX(CMyWindow)
        MSG_WM_CREATE(OnCreate)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMyWindow>)
    END_MSG_MAP()
    LRESULT OnCreate(LPCREATESTRUCT lpcs)
    {
        SetTimer ( 1, 1000 );
        SetMsgHandled(false);
        return 0;
    }
};
```

CFrameWindowImpl inherits indirectly from CWindow, so it has all the CWindow functions like SetTimer(). This makes windowing API calls look a lot like MFC code, where you use the various CWnd methods that wrap APIs.

We call SetTimer() to create a timer that fires every second (1000 ms). Since we want to let CFrameWindowImpl handle WM_CREATE as well, we call SetMsgHandled(false) so that the message gets chained to base classes via the CHAIN_MSG_MAP macro. This call replaces the bHandled parameter that the ATL macros use. (Even though CFrameWindowImpl doesn't handle WM_CREATE, calling SetMsgHandled(false) is a good habit to get into when using base classes, so you don't have to remember which messages the base classes handle. This is similar to the code that ClassWizard generates; most handlers start or end with a call to the base class handler.)

We'll also need a WM_DESTROY handler so we can stop the timer. Going through the same process as before, we find the MSG_WM_DESTROY macro looks like this:

```
#define MSG_WM_DESTROY(func) \
    if (uMsg == WM_DESTROY) \
    { \
        SetMsgHandled(TRUE); \
        func(); \
        lResult = 0; \
        if(IsMsgHandled()) \
            return TRUE; \
    }
```

So our OnDestroy() handler takes no parameters and returns nothing. CFrameWindowImpl *does* handle WM_DESTROY as well, so in this case we definitely need to call SetMsgHandled(false):

```
class CMyWindow : public CFrameWindowImpl<CMyWindow>
{
public:
    BEGIN_MSG_MAP_EX(CMyWindow)
        MSG_WM_CREATE(OnCreate)
        MSG_WM_DESTROY(OnDestroy)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMyWindow>)
    END_MSG_MAP()

    void OnDestroy()
    {
        KillTimer(1);
        SetMsgHandled(false);
    }
};
```

Next up is our `WM_TIMER` handler, which is called every second. You should have the hang of the F12 trick by now, so I'll just show the handler:

```
class CMyWindow : public CFrameWindowImpl<CMyWindow>
{
public:
    BEGIN_MSG_MAP_EX(CMyWindow)
        MSG_WM_CREATE(OnCreate)
        MSG_WM_DESTROY(OnDestroy)
        MSG_WM_TIMER(OnTimer)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMyWindow>)
    END_MSG_MAP()

    void OnTimer ( UINT uTimerID, TIMERPROC pTimerProc )
    {
        if ( 1 != uTimerID )
            SetMsgHandled(false);
        else
            RedrawWindow();
    }
};
```

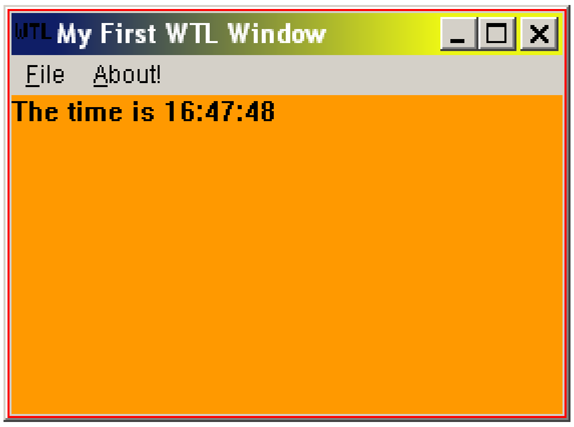
This handler just redraws the window so the new time appears in the client area. Finally, we handle `WM_ERASEBKGND` and in that handler, we draw the current time in the upper-left corner of the client area.

```
class CMyWindow : public CFrameWindowImpl<CMyWindow>
{
public:
    BEGIN_MSG_MAP_EX(CMyWindow)
```

```
        MSG_WM_CREATE(OnCreate)
        MSG_WM_DESTROY(OnDestroy)
        MSG_WM_TIMER(OnTimer)
        MSG_WM_ERASEBKGND(OnEraseBkgnd)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMyWindow>)
    END_MSG_MAP()

    LRESULT OnEraseBkgnd ( HDC hdc )
    {
    CDCHandle  dc(hdc);
    CRect      rc;
    SYSTEMTIME st;
    CString    sTime;

        // Get our window's client area.
        GetClientRect ( rc );

        // Build the string to show in the window.
        GetLocalTime ( &st );
        sTime.Format ( _T("The time is %d:%02d:%02d"),
                       st.wHour, st.wMinute, st.wSecond );

        // Set up the DC and draw the text.
        dc.SaveDC();

        dc.SetBkColor ( RGB(255,153,0) );
        dc.SetTextColor ( RGB(0,0,0) );
        dc.ExtTextOut ( 0, 0, ETO_OPAQUE, rc, sTime,
                        sTime.GetLength(), NULL );

        // Restore the DC.
        dc.RestoreDC(-1);
        return 1;    // We erased the background (ExtTextOut did it)
    }
};
```

This handler demonstrates one of the GDI wrappers, `CDCHandle`, as well as `CRect` and `CString`. All I need to say about `CString` is, it's identical to MFC's `CString`. I'll cover the wrapper classes later on, but for now you can think of `CDCHandle` as a simple wrapper around an `HDC`, similar to MFC's `CDC`, although when the `CDCHandle` goes out of scope, it doesn't destroy the underlying device context.

So after all that, here's what our window looks like:



The time is 16:47:48

The sample code also has `WM_COMMAND` handlers for the menu items; I won't cover those here, but you can check out the sample project and see the WTL macro `COMMAND_ID_HANDLER_EX` in action.

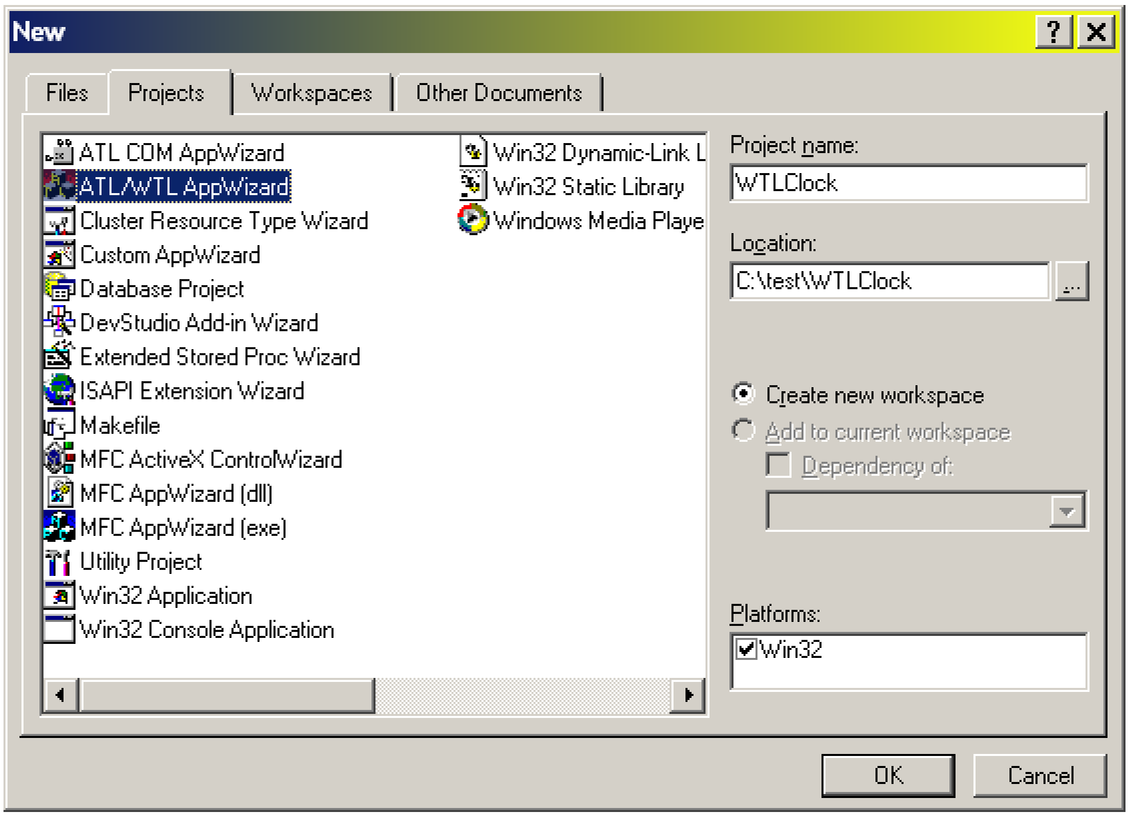
If you're using VC 7.1, check out the WTL Helper plug-in by Sergey Solozhentsev, which will do the grunt work of adding message map macros for you.

What You Get with the WTL AppWizard

The WTL distribution comes with a very nice AppWizard, so let's see what features it puts in an SDI app.

Going through the wizard (VC 6)

Click *File|New* in VC and select *ATL/WTL AppWizard* from the list. We'll be rewriting the clock app, so enter *WTLClock* for the project name:



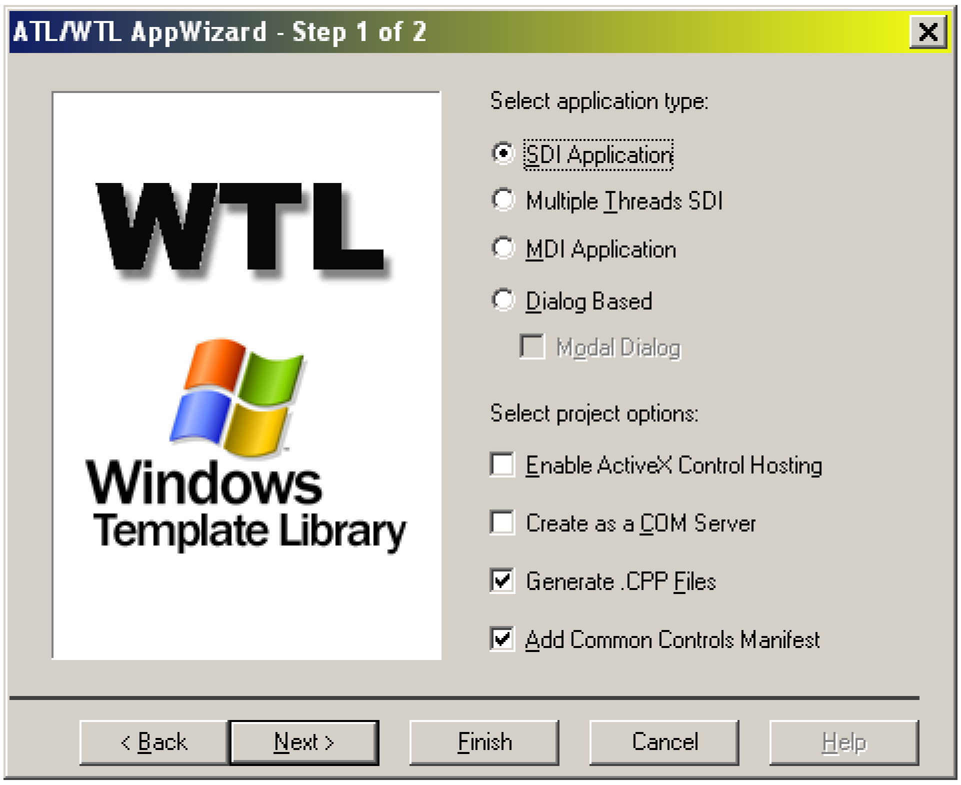
Project name: WTLClock

Location: C:\test\WTLClock

(●) Create new workspace
( ) Add to current workspace
[ ] Dependency of:

Platforms:
[✓] Win32

The next page is where you select an SDI, MDI, or dialog-based app, along with some other options. Select the options as shown here and click *Next*:



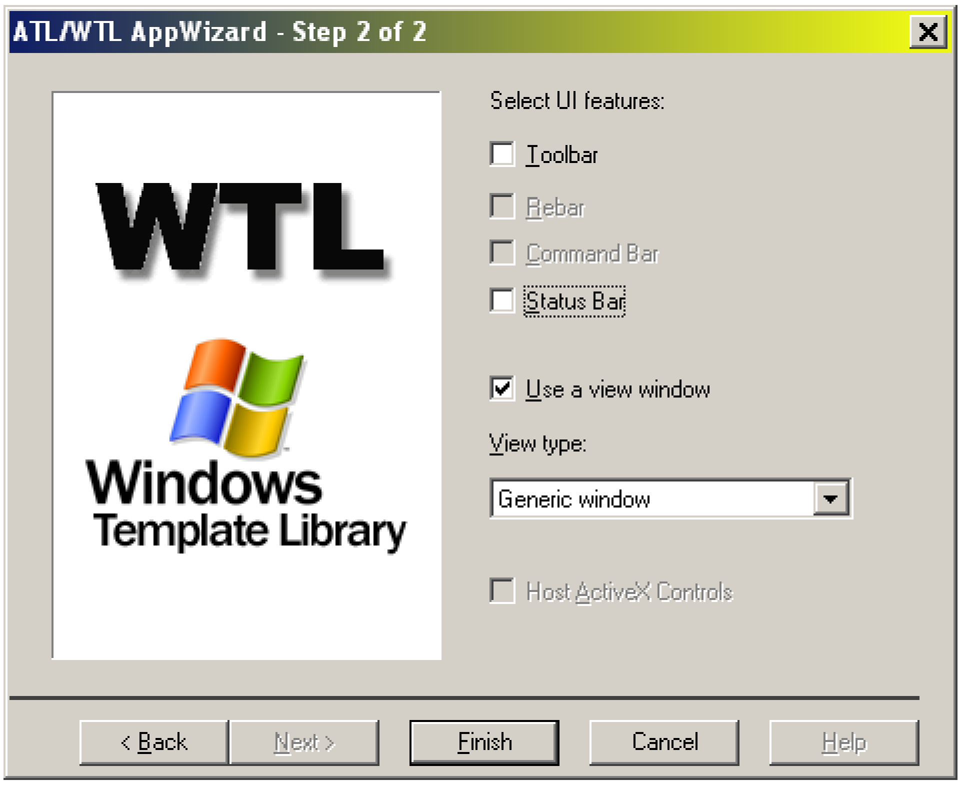
Select application type:

(●) SDI Application
( ) Multiple Threads SDI
( ) MDI Application
( ) Dialog Based
[ ] Modal Dialog

Select project options:

[ ] Enable ActiveX Control Hosting
[ ] Create as a COM Server
[✓] Generate .CPP Files
[✓] Add Common Controls Manifest

The last page is where we can choose to have a toolbar, rebar, and status bar. To keep this app simple, uncheck all those and click *Finish*.



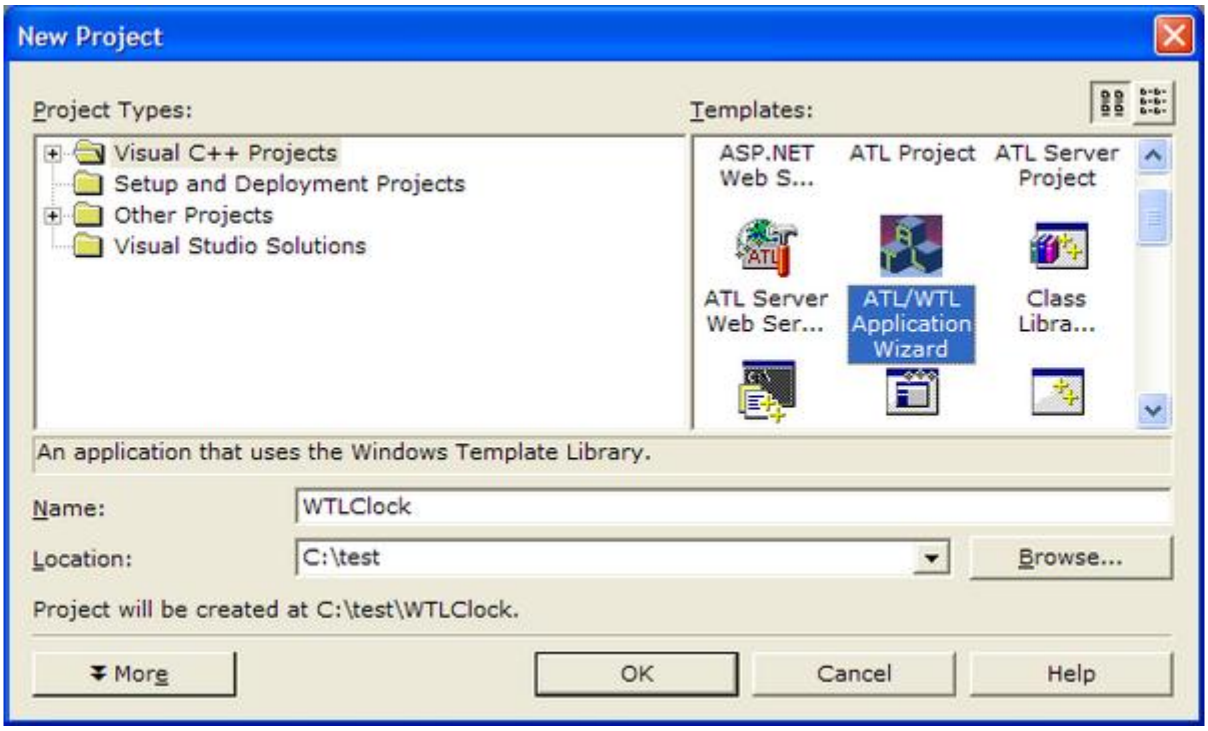
Select UI features:
- [ ] Toolbar
- [ ] Rebar
- [ ] Command Bar
- [ ] Status Bar
- [✓] Use a view window

View type: Generic window

- [ ] Host ActiveX Controls

Going through the wizard (VC 7)

Click *File|New|Project* in VC and select *ATL/WTL AppWizard* from the list. We'll be rewriting the clock app, so enter *WTLClock* for the project name and click *OK*:


New Project

Project Types:
- Visual C++ Projects
- Setup and Deployment Projects
- Other Projects
- Visual Studio Solutions

Templates: ASP.NET Web S..., ATL Project, ATL Server Project, ATL Server Web Ser..., ATL/WTL Application Wizard, Class Libra...

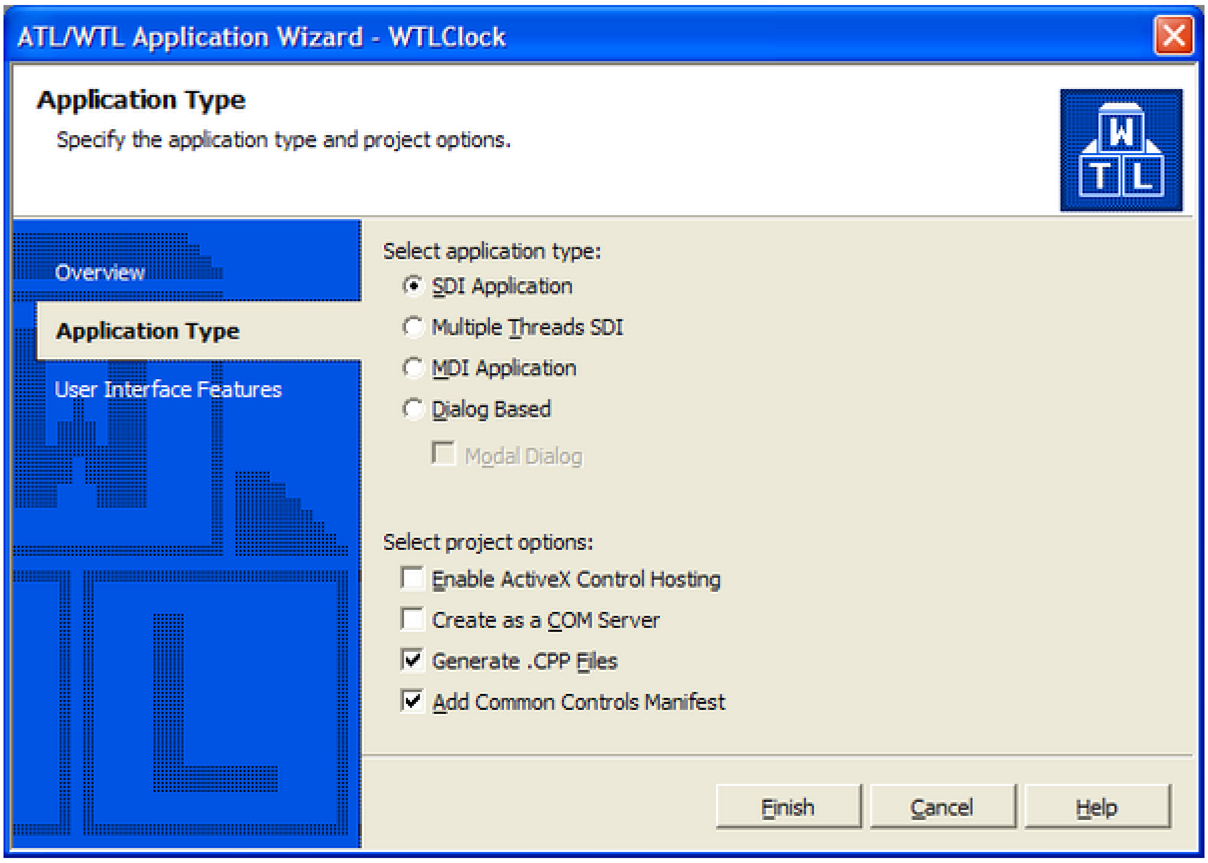
An application that uses the Windows Template Library.

Name: WTLClock

Location: C:\test

Project will be created at C:\test\WTLClock.

When the AppWizard screen comes up, click *Application Type*. This page is where you select an SDI, MDI, or dialog-based app, along with some other options. Select the options as shown here and click *User Interface Features*:


ATL/WTL Application Wizard - WTLClock

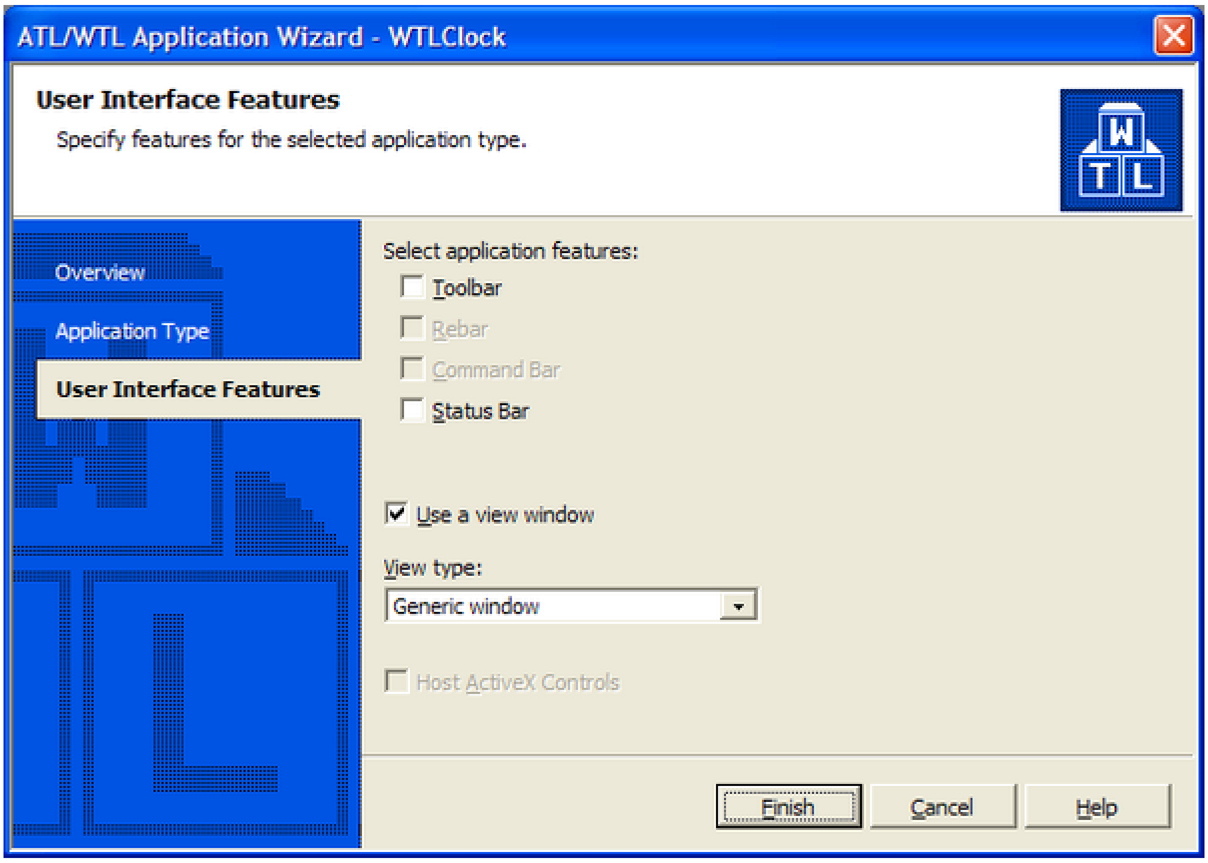
Application Type

Specify the application type and project options.

Select application type:
- (●) SDI Application
- ( ) Multiple Threads SDI
- ( ) MDI Application
- ( ) Dialog Based
  - [ ] Modal Dialog

Select project options:
- [ ] Enable ActiveX Control Hosting
- [ ] Create as a COM Server
- [✓] Generate .CPP Files
- [✓] Add Common Controls Manifest

The last page is where we can choose to have a toolbar, rebar, and status bar. To keep this app simple, uncheck all those and click *Finish*.


ATL/WTL Application Wizard - WTLClock
User Interface Features
Specify features for the selected application type.
Overview
Application Type
User Interface Features
Finish
Cancel
Help

Select application features:
- [ ] Toolbar
- [ ] Rebar
- [ ] Command Bar
- [ ] Status Bar

- [✓] Use a view window

View type: Generic window

- [ ] Host ActiveX Controls

Examining the generated code

After finishing the wizard, you'll see three classes in the generated code: `CMainFrame`, `CAboutDlg`, and `CWTLClockView`. From the names, you can guess what the purpose of each class is. While there is a "view" class, it is strictly a "plain" window derived from `CWindowImpl`; there is no framework at all like MFC's doc/view architecture.

There is also a `_tWinMain()`, which initializes COM, the common controls, and `_Module`, and then calls a global `Run()` function. `Run()` handles creating the main window and starting the message pump. It also uses a new class, `CMessageLoop`. `Run()` calls `CMessageLoop::Run()` which actually contains the message pump. I'll cover `CMessageLoop` in more detail in the next section.

`CAboutDlg` is a simple `CDialogImpl`-derived class that is associated with a dialog with ID `IDD_ABOUTBOX`. I covered dialogs in Part I, so you should be able to understand the code in `CAboutDlg`.

CWTLClockView is our app's "view" class. This works like an MFC view, in that it is a captionless window that occupies the client area of the main frame. CWTLClockView has a PreTranslateMessage() function, which works just like the MFC function of the same name, and a WM_PAINT handler. Neither function does anything significant yet, but we'll fill in the OnPaint() method to show the time.

Finally, we have CMainFrame, which has lots of interesting new stuff. Here is an abbreviated version of the class definition:

```
class CMainFrame : public CFrameWindowImpl<CMainFrame>,
                   public CUpdateUI<CMainFrame>,
                   public CMessageFilter,
                   public CIdleHandler
{
public:
    DECLARE_FRAME_WND_CLASS(NULL, IDR_MAINFRAME)

    BEGIN_UPDATE_UI_MAP(CMainFrame)
    END_UPDATE_UI_MAP()

    BEGIN_MSG_MAP(CMainFrame)
        // ...
        CHAIN_MSG_MAP(CUpdateUI<CMainFrame>)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMainFrame>)
    END_MSG_MAP()

    BOOL PreTranslateMessage(MSG* pMsg);
    BOOL OnIdle();

protected:
    CWTLClockView m_view;
};
```

CMessageFilter is a mix-in class that provides PreTranslateMessage(), and CIdleHandler is another mix-in that provides OnIdle(). CMessageLoop, CIdleHandler, and CUpdateUI work together to provide UI updating that works like ON_UPDATE_COMMAND_UI in MFC.

CMainFrame::OnCreate() creates the view window and saves its window handle, so that the view will be resized when the frame window is resized. OnCreate() also adds the CMainFrame object to lists of message filters and idle handlers kept by the CAppModule; I will cover those later.

```
LRESULT CMainFrame::OnCreate(UINT /*uMsg*/, WPARAM /*wParam*/,
                        LPARAM /*lParam*/, BOOL& /*bHandled*/)
{
   m_hWndClient = m_view.Create(m_hWnd, rcDefault, NULL, |
                        WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS |
                         WS_CLIPCHILDREN, WS_EX_CLIENTEDGE);

   // register object for message filtering and idle updates
   CMessageLoop* pLoop = _Module.GetMessageLoop();
   pLoop->AddMessageFilter(this);
   pLoop->AddIdleHandler(this);

   return 0;
}
```

m_hWndClient is a member of CFrameWindowImpl; this is the window that will be resized when the frame is resized.

The generated CMainFrame also has handlers for *File|New*, *File|Exit*, and *Help|About*. We won't need most of the default menu items for our clock, but it won't do any harm to leave them in for now. You can build and run the wizard-generated code now, although the app won't be very useful yet. You might be interested in stepping through the CMainFrame::CreateEx() call in the global Run() to see exactly how the frame window and its resources are loaded and created.

Our next stop on the WTL tour is CMessageLoop, which handles the message pump and idle processing.

CMessageLoop Internals

CMessageLoop provides a message pump for our app. Aside from a standard TranslateMessage/DispatchMessage loop, it also provides message filtering via PreTranslateMessage() and idle processing via OnIdle(). Here is pseudo code for the logic in Run():

```
int Run()
{
MSG msg;

   for(;;)
      {
      while ( !PeekMessage(&msg) )
         CallIdleHandlers();
```

```
        if ( 0 == GetMessage(&msg) )
            break;    // WM_QUIT retrieved from the queue

        if ( !CallPreTranslateMessageFilters(&msg) )
            {
            // if we get here, message was not filtered out
            TranslateMessage(&msg);
            DispatchMessage(&msg);
            }
        }

    return msg.wParam;
}
```

CMessageLoop knows which PreTranslateMessage() functions to call because objects that need to filter messages call CMessageLoop::AddMessageFilter(), just like CMainFrame::OnCreate() does. And similarly, objects that need to do idle processing call CMessageLoop::AddIdleHandler().

Notice that there are no calls to TranslateAccelerator() or IsDialogMessage() in the message loop. CFrameWindowImpl handles the former, but if you add any modeless dialogs to your app, you'll need to add a call to IsDialogMessage() in CMainFrame::PreTranslateMessage().

CFrameWindowImpl Internals

CFrameWindowImpl and its base class CFrameWindowImplBase provide a lot of the features you're used to having in MFC's CFrameWnd: toolbars, rebars, status bars, tooltips for toolbar buttons, and flyby help for menu items. I'll be covering all those features gradually, since discussing the entire CFrameWindowImpl class could take up two whole articles! For now, it will be sufficient to see how CFrameWindowImpl handles WM_SIZE and its client area. For this discussion, remember that m_hWndClient is a member of CFrameWindowImplBase that holds the HWND of the "view" window inside the frame.

CFrameWindowImpl has a handler for WM_SIZE:

```
LRESULT OnSize(UINT /*uMsg*/, WPARAM wParam, LPARAM /*lParam*/, BOOL& bHandled)
{
    if(wParam != SIZE_MINIMIZED)
    {
        T* pT = static_cast<T*>(this);
        pT->UpdateLayout();
    }

    bHandled = FALSE;
    return 1;
}
```

This checks that the window is not being minimized. If not, it delegates to `UpdateLayout()`. Here is `UpdateLayout()`:

```
void UpdateLayout(BOOL bResizeBars = TRUE)
{
RECT rect;

    GetClientRect(&rect);

    // position bars and offset their dimensions
    UpdateBarsPosition(rect, bResizeBars);

    // resize client window
    if(m_hWndClient != NULL)
        ::SetWindowPos(m_hWndClient, NULL, rect.left, rect.top,
            rect.right - rect.left, rect.bottom - rect.top,
            SWP_NOZORDER | SWP_NOACTIVATE);
}
```

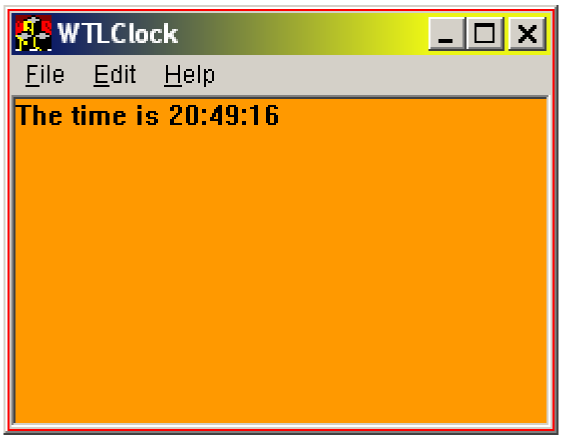
Notice how the code is referencing `m_hWndClient`. Since `m_hWndClient` is a plain `HWND`, it can be any window at all. There is no restriction on what kind of window it can be, unlike MFC where some features (like splitter windows) require a `CView`-derived class. If you go back to `CMainFrame::OnCreate()`, you'll see that it creates a view window and stores its handle in `m_hWndClient`, which ensures that the view will be resized properly.

Back to the Clock Program

Now that we've seen some of the frame window class details, let's get back to our clock app. The view window can handle the timer and drawing, just as `CMyWindow` did in the previous sample. Here's a partial class definition:

```
class CWTLClockView : public CWindowImpl<CWTLClockView>
{
public:
    DECLARE_WND_CLASS(NULL)

    BOOL PreTranslateMessage(MSG* pMsg);

    BEGIN_MSG_MAP_EX(CWTLClockView)
        MESSAGE_HANDLER(WM_PAINT, OnPaint)
        MSG_WM_CREATE(OnCreate)
        MSG_WM_DESTROY(OnDestroy)
        MSG_WM_TIMER(OnTimer)
        MSG_WM_ERASEBKGND(OnEraseBkgnd)
    END_MSG_MAP()
};
```

Note that it's fine to mix the ATL message map macros with the WTL versions, as long as you change `BEGIN_MSG_MAP` to `BEGIN_MSG_MAP_EX` when necessary. `OnPaint()` has all the drawing code that was in `OnEraseBkgnd()` in the previous sample. Here's what the new window looks like:



The time is 20:49:16

The last thing we'll add to this app is UI updating. To demonstrate this, we'll add a new top-level menu item called *Clock* that has *Start* and *Stop* commands to stop and start the clock. The *Start* and *Stop* menu items will be enabled and disabled as appropriate.

UI Updating

Several things work together to provide idle-time UI updating: a `CMessageLoop` object, the mix-in classes `CIdleHandler` and `CUpdateUI` that `CMainFrame` inherits from, and the `UPDATE_UI_MAP` in `CMainFrame`. `CUpdateUI` can operate on five different types of elements: top-level menu items (in the menu bar itself), menu items in popup menus, toolbar buttons, status bar panes, and child windows (such as dialog controls). Each type of element has a corresponding constant in `CUpdateUIBase`:

- menu bar items: `UPDUI_MENUBAR`
- popup menu items: `UPDUI_MENUPOPUP`
- toolbar buttons: `UPDUI_TOOLBAR`
- status bar panes: `UPDUI_STATUSBAR`
- child windows: `UPDUI_CHILDWINDOW`

`CUpdateUI` can set the enabled state, checked state, and text of items (but of course not all items support all states; you can't put a check on an edit box). It can also set a popup menu item to the default state so the text appears in bold.

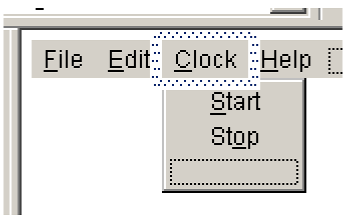
To hook up UI updating, we need to do four things:

1. Derive the frame window class from `CUpdateUI` and `CIdleHandler`
2. Chain messages from `CMainFrame` to `CUpdateUI`
3. Add the frame window to the module's list of idle handlers
4. Fill in the frame window's `UPDATE_UI_MAP`

The AppWizard-generated code takes care of the first three parts for us, so all that's left is to decide which menu items we'll update, and when they will be enabled or disabled.

New menu items to control the clock

Let's add a new *Clock* menu to the menu bar, with two items: `IDC_START` and `IDC_STOP`:



Then we add an entry in the `UPDATE_UI_MAP` for each menu item:

```
class CMainFrame : public ...
{
public:
    // ...
    BEGIN_UPDATE_UI_MAP(CMainFrame)
        UPDATE_ELEMENT(IDC_START, UPDUI_MENUPOPUP)
        UPDATE_ELEMENT(IDC_STOP, UPDUI_MENUPOPUP)
    END_UPDATE_UI_MAP()
    // ...
};
```

Then whenever we want to change the enabled state of either item, we call `CUpdateUI::UIEnable()`. `UIEnable()` takes the ID of the item, and a `bool` that indicates the enabled state (`true` for enabled, `false` for disabled).

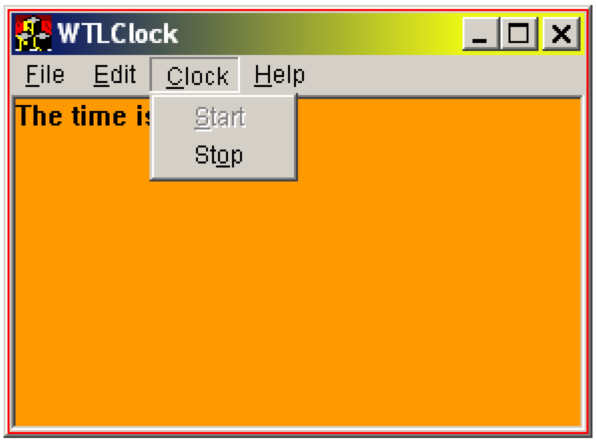
This system works differently from MFC's `ON_UPDATE_COMMAND_UI` system. In MFC, we write update UI handlers for any UI elements that need to have their state updated. MFC then calls the handlers either at idle time, or when it is about to show a menu. In WTL, we call `CUpdateUI` methods when the state of an item changes. `CUpdateUI` keeps track of the UI elements and their states, and updates the elements at idle time, or before a menu is shown.

Calling UIEnable()

Let's go back to the `OnCreate()` function and see how we set up the initial state of the *Clock* menu items.

```
LRESULT CMainFrame::OnCreate(UINT /*uMsg*/, WPARAM /*wParam*/,
                             LPARAM /*lParam*/, BOOL& /*bHandled*/)
{
    m_hWndClient = m_view.Create(...);

    // register object for message filtering and idle updates
    // [omitted for clarity]

    // Set the initial state of the Clock menu items:
    UIEnable ( IDC_START, false );
    UIEnable ( IDC_STOP, true );

    return 0;
}
```

And here's what the *Clock* menu looks like when the app is started:



CMainFrame now needs handlers for our two new items. The handlers will toggle the states of the menu items, then call methods in the view class to start and stop the clock. This is one area where MFC's built-in message routing is sorely missed; if this were an MFC app, all the UI updating and command handling could be put entirely in the view class. However, in WTL, the frame and view classes have to communicate in some way; the menu is owned by the frame window, so the frame gets menu-related messages and is responsible for either acting on them or sending them to the view class.

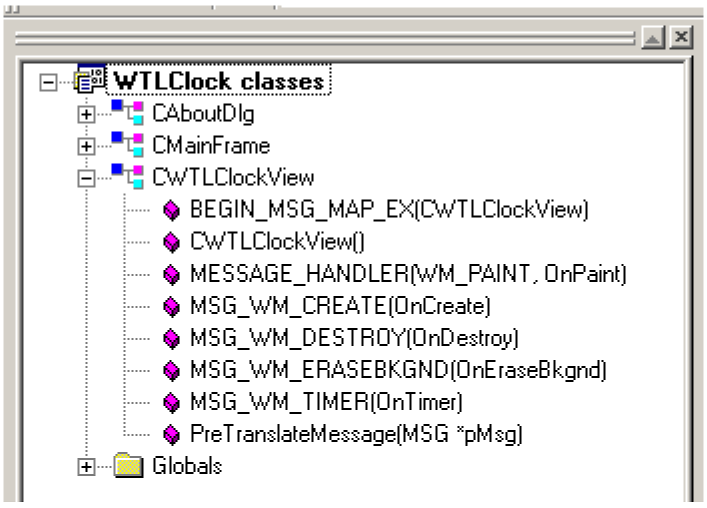
The communication *could* be done through PreTranslateMessage(), however the UIEnable() calls still have to be done from CMainFrame. CMainFrame could get around this by passing its this pointer to the view class, so the view could call UIEnable() through that pointer. For this sample, I have chosen the solution that results in a tightly-coupled frame and view, since I find it easier to understand (and explain!).

```
class CMainFrame : public ...
{
public:
    BEGIN_MSG_MAP_EX(CMainFrame)
        // ...
        COMMAND_ID_HANDLER_EX(IDC_START, OnStart)
        COMMAND_ID_HANDLER_EX(IDC_STOP, OnStop)
    END_MSG_MAP()

    // ...
    void OnStart(UINT uCode, int nID, HWND hwndCtrl);
    void OnStop(UINT uCode, int nID, HWND hwndCtrl);
};

void CMainFrame::OnStart(UINT uCode, int nID, HWND hwndCtrl)
{
    // Enable Stop and disable Start
    UIEnable ( IDC_START, false );
    UIEnable ( IDC_STOP, true );

    // Tell the view to start its clock.
    m_view.StartClock();
}

void CMainFrame::OnStop(UINT uCode, int nID, HWND hwndCtrl)
{
    // Enable Start and disable Stop
    UIEnable ( IDC_START, true );
    UIEnable ( IDC_STOP, false );

    // Tell the view to stop its clock.
    m_view.StopClock();
}
```

Each handler updates the *Clock* menu, then calls a method in the view, since the view is the class that controls the clock. The `StartClock()` and `StopClock()` methods are not shown here, but you can find them in the sample project.

One Last Note on Message Maps

If you're using VC 6, you might have noticed that when you change `BEGIN_MSG_MAP` to `BEGIN_MSG_MAP_EX`, ClassView gets a bit confused:



```
WTLClock classes
  CAboutDlg
  CMainFrame
  CWTLClockView
    BEGIN_MSG_MAP_EX(CWTLClockView)
    CWTLClockView()
    MESSAGE_HANDLER(WM_PAINT, OnPaint)
    MSG_WM_CREATE(OnCreate)
    MSG_WM_DESTROY(OnDestroy)
    MSG_WM_ERASEBKGND(OnEraseBkgnd)
    MSG_WM_TIMER(OnTimer)
    PreTranslateMessage(MSG *pMsg)
  Globals
```

This happens because ClassView doesn't recognize `BEGIN_MSG_MAP_EX` as something it should parse specially, and it thinks that all the WTL message map macros are actually functions. You can fix this by changing the macros back to `BEGIN_MSG_MAP` and adding these lines at the **end** of *stdafx.h*:

```
#if (ATL_VER < 0x0700)
#undef BEGIN_MSG_MAP
#define BEGIN_MSG_MAP(x) BEGIN_MSG_MAP_EX(x)
#endif
```

Next Stop, 1995

We've only just begun to scratch the surface of WTL. In the next article, I'll bring our sample clock app up to 1995 UI standards and introduce toolbars and status bars. In the meantime, experiment a bit with the `CUpdateUI` methods; for example, try calling `UISetCheck()` instead of `UIEnable()` to see the different ways the menu items can be changed.

Part III - Toolbars and Status Bars

Introduction to Part III

Ever since they were made into common controls in Windows 95, toolbars and status bars have become commonplace. MFC's support for multiple floating toolbars also helped their popularity. In later common controls updates, rebars (or coolbars as they were originally called) added another way to present toolbars. In Part III, I will cover how WTL supports these kinds of control bars and how to use them in your own apps.

Remember, if you encounter any problems installing WTL or compiling the demo code, read the readme section of Part I before posting questions here.

Toolbars and Status Bars in a Frame

`CFrameWindowImpl` has three `HWND` members that are set when the frame window is created. We've already seen `m_hWndClient`, which is the handle of the "view" window in the frame's client area. Now we'll encounter two others:

- `m_hWndToolBar`: `HWND` of the toolbar or rebar
- `m_hWndStatusBar`: `HWND` of the status bar

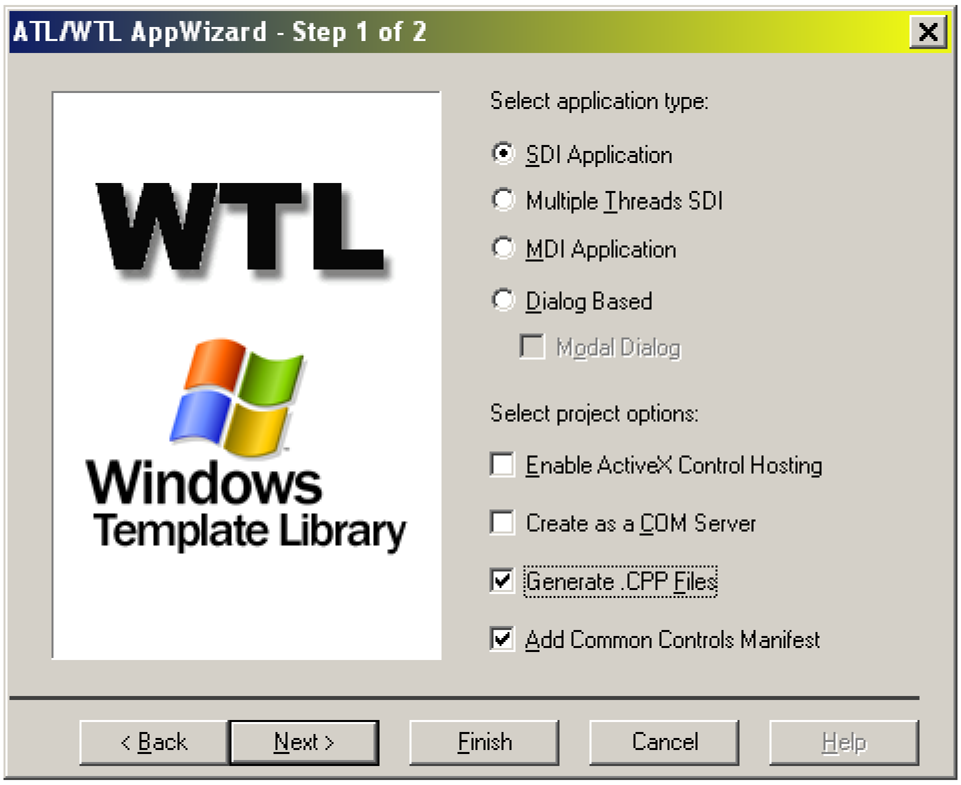
`CFrameWindowImpl` only supports one toolbar, and there is no equivalent to the MFC system of multiple dockable toolbars. If you need more than one toolbar, and don't want to hack around in `CFrameWindowImpl` internals, then you will need to use a rebar. I will cover both of these options and show how to select from them in the AppWizard.

The `CFrameWindowImpl::OnSize()` handler calls `UpdateLayout()`, which does two things: position the bars, and resize the view window to fill the client area. `UpdateLayout()` calls `UpdateBarsPosition()` which does the actual work. The code is quite simple, it just sends a `WM_SIZE` message to the toolbar and status bar, if those bars have been created. The bars' default window procedures take care of moving the bars to the top or bottom of the frame window.

When you tell the AppWizard to give your frame a toolbar and status bar, the wizard puts code to create the bars in `CMainFrame::OnCreate()`. Let's take a closer look at that code now, as we write yet another clock app.

AppWizard Code for Toolbars and Status Bars

We'll start a new project and have the wizard create a toolbar and status bar for our frame. Start a new WTL project called WTLClock2. On the first AppWizard page, choose an SDI app and check *Generate CPP files*:

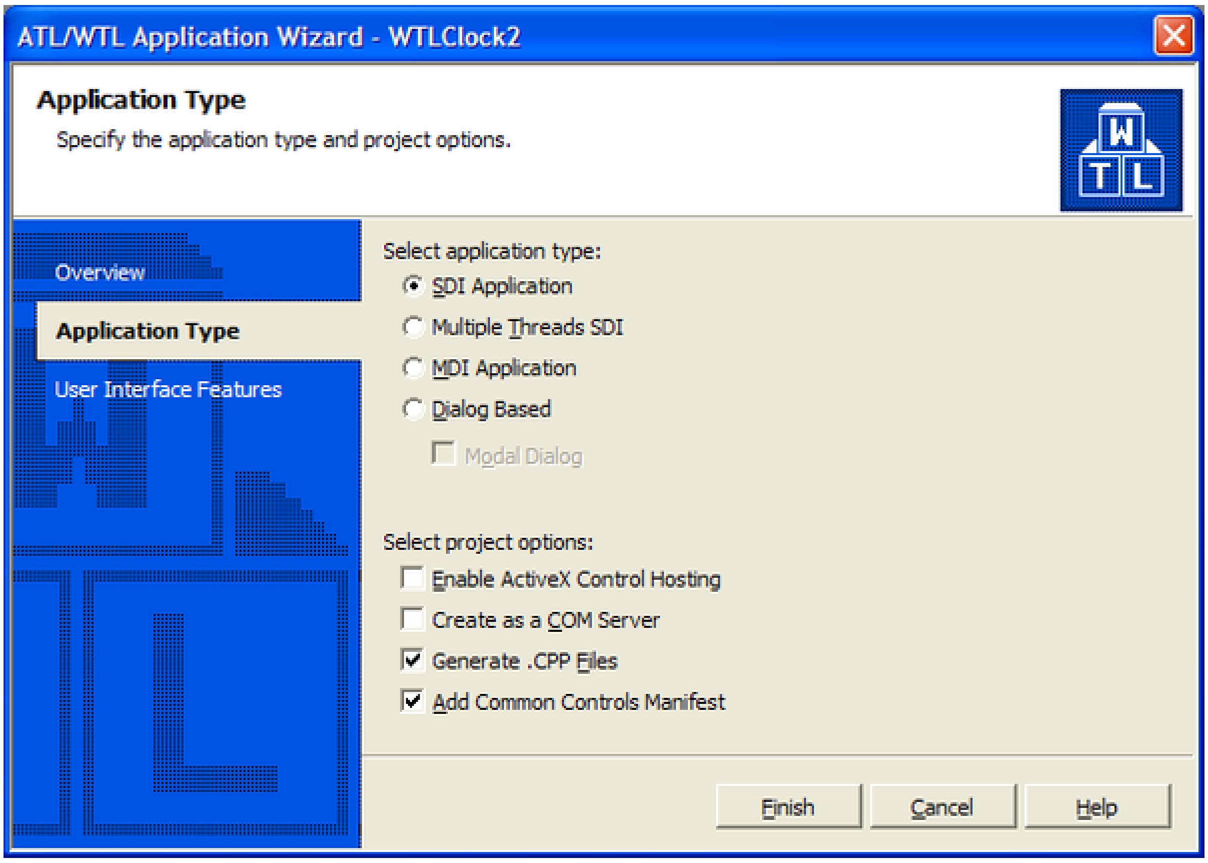

ATL/WTL AppWizard - Step 1 of 2

Select application type:

- (●) SDI Application
- ( ) Multiple Threads SDI
- ( ) MDI Application
- ( ) Dialog Based
  - [ ] Modal Dialog

Select project options:

- [ ] Enable ActiveX Control Hosting
- [ ] Create as a COM Server
- [✓] Generate .CPP Files
- [✓] Add Common Controls Manifest



Application Type

Specify the application type and project options.

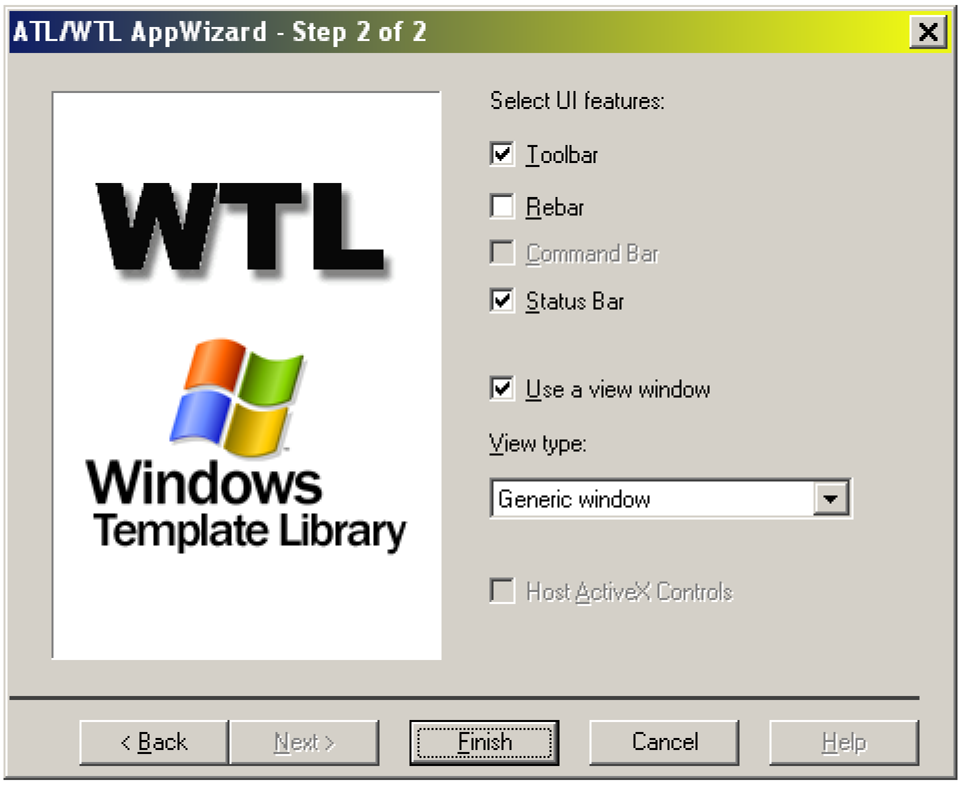
Select application type:

- [✓ SDI Application]
- Multiple Threads SDI
- MDI Application
- Dialog Based
- Modal Dialog

Select project options:

- [ ] Enable ActiveX Control Hosting
- [ ] Create as a COM Server
- [✓] Generate .CPP Files
- [✓] Add Common Controls Manifest

On the next page, uncheck *Rebar* so the wizard creates a normal toolbar:



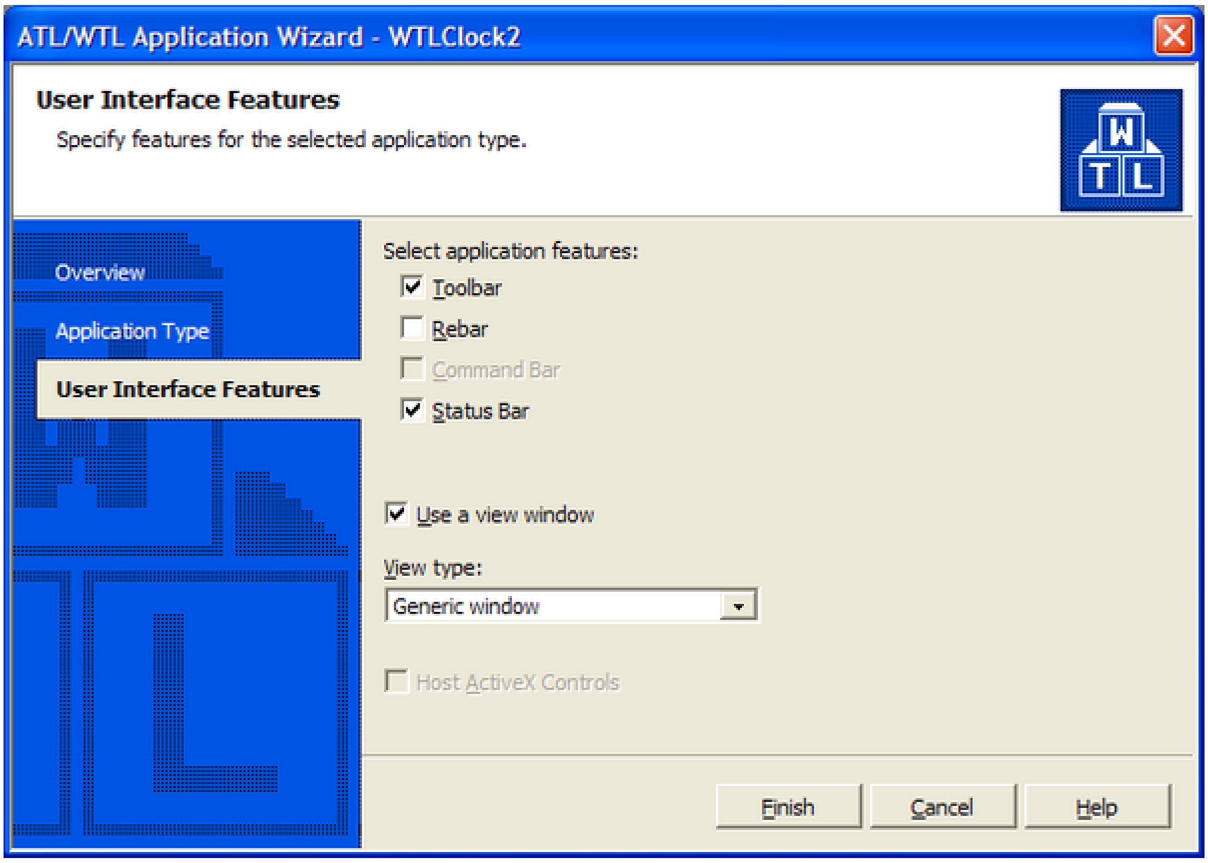
Select UI features:

- [✓] Toolbar
- [ ] Rebar
- [ ] Command Bar
- [✓] Status Bar

- [✓] Use a view window

View type: Generic window

- [ ] Host ActiveX Controls



ATL/WTL Application Wizard - WTLClock2

User Interface Features

Specify features for the selected application type.

Select application features:

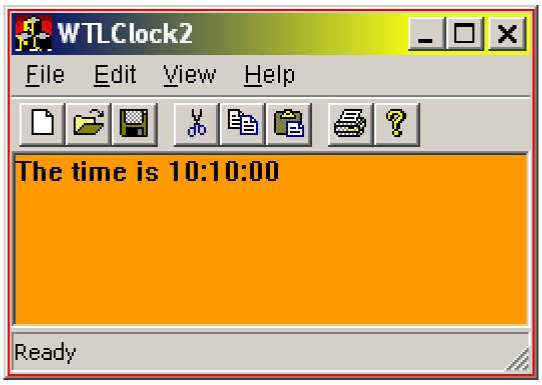
- [✓] Toolbar
- [ ] Rebar
- [ ] Command Bar
- [✓] Status Bar

- [✓] Use a view window

View type: Generic window

- [ ] Host ActiveX Controls

After copying the clock code from the app in Part II, the new app looks like this:



The time is 10:10:00

Ready

How CMainFrame creates the bars

The AppWizard puts more code in `CMainFrame::OnCreate()` in this project. Its job is to create the bars and tell `CUpdateUI` to update the toolbar buttons.

```
LRESULT CMainFrame::OnCreate(UINT /*uMsg*/, WPARAM /*wParam*/,
                             LPARAM /*lParam*/, BOOL& /*bHandled*/)
{
    CreateSimpleToolBar();
    CreateSimpleStatusBar();
    m_hWndClient = m_view.Create(...);

// ...


    // register object for message filtering and idle updates

    CMessageLoop* pLoop = _Module.GetMessageLoop();
    ATLASSERT(pLoop != NULL);
    pLoop->AddMessageFilter(this);
    pLoop->AddIdleHandler(this);

    return 0;
}
```

The new code here is at the beginning. `CFrameWindowImpl::CreateSimpleToolBar()` creates a new toolbar using the toolbar resource `IDR_MAINFRAME` and stores its handle in `m_hWndToolBar`. Here is the code for `CreateSimpleToolBar()`:

```
BOOL CFrameWindowImpl::CreateSimpleToolBar(
    UINT nResourceID = 0,
    DWORD dwStyle = ATL_SIMPLE_TOOLBAR_STYLE,
    UINT nID = ATL_IDW_TOOLBAR)
{
    ATLASSERT(!::IsWindow(m_hWndToolBar));

    if(nResourceID == 0)
        nResourceID = T::GetWndClassInfo().m_uCommonResourceID;

    m_hWndToolBar = T::CreateSimpleToolBarCtrl(m_hWnd,
                      nResourceID, TRUE, dwStyle, nID);
    return (m_hWndToolBar != NULL);
}
```

The parameters are:

`nResourceID`

ID of the toolbar resource to use. The default of 0 means to use the ID specified in the `DECLARE_FRAME_WND_CLASS` macro. This is `IDR_MAINFRAME` in the wizard-generated code.

`dwStyle`

Styles for the toolbar. The default value `ATL_SIMPLE_TOOLBAR_STYLE` is defined as `TBSTYLE_TOOLTIPS` plus the usual child and visible styles. This makes the toolbar create a tooltip control for use when the cursor hovers over a button.

`nID`

Window ID for the toolbar, you will usually use the default value.

`CreateSimpleToolBar()` checks that a toolbar hasn't been created yet, then calls `CreateSimpleToolBarCtrl()` to actually create the control. The handle returned by `CreateSimpleToolBarCtrl()` is saved in `m_hWndToolBar`. `CreateSimpleToolBarCtrl()` reads the resource and creates toolbar buttons accordingly, then returns the toolbar's window handle. The code to do that is rather long, so I won't cover it here. You can find it in *atlframe.h* if you're interested.

The next call in `OnCreate()` is `CFrameWindowImpl::CreateSimpleStatusBar()`. This creates a status bar and stores its handle in `m_hWndStatusBar`. Here is the code:

```
BOOL CFrameWindowImpl::CreateSimpleStatusBar(
    UINT nTextID = ATL_IDS_IDLEMESSAGE,
    DWORD dwStyle = ... SBARS_SIZEGRIP,
    UINT nID = ATL_IDW_STATUS_BAR)
{
    TCHAR szText[128];    // max text lentgth is 127 for status bars

    szText[0] = 0;
    ::LoadString(_Module.GetResourceInstance(), nTextID, szText, 128);
    return CreateSimpleStatusBar(szText, dwStyle, nID);
}
```

This loads a string from the string table, which will be shown in the status bar. The parameters are:

`nTextID`

Resource ID of the string to be initially shown in the status bar. The AppWizard generates the string "Ready" with ID `ATL_IDS_IDLEMESSAGE`.

`dwStyle`

Styles for the status bar. The default includes `SBARS_SIZEGRIP` to have a resizing gripper added to the bottom-right corner.

`nID`

Window ID for the status bar, you will usually use the default value.

CreateSimpleStatusBar() calls the other overload to do the work:

```
BOOL CFrameWindowImpl::CreateSimpleStatusBar(
    LPCTSTR lpstrText,
    DWORD dwStyle = ... SBARS_SIZEGRIP,
    UINT nID = ATL_IDW_STATUS_BAR)
{
    ATLASSERT(!::IsWindow(m_hWndStatusBar));
    m_hWndStatusBar = ::CreateStatusWindow(dwStyle, lpstrText, m_hWnd, nID);
    return (m_hWndStatusBar != NULL);
}
```

This version checks that a status bar has not been created yet, then calls CreateStatusWindow() to create the status bar. The status bar's handle is then stored in m_hWndStatusBar.

Showing and hiding the bars

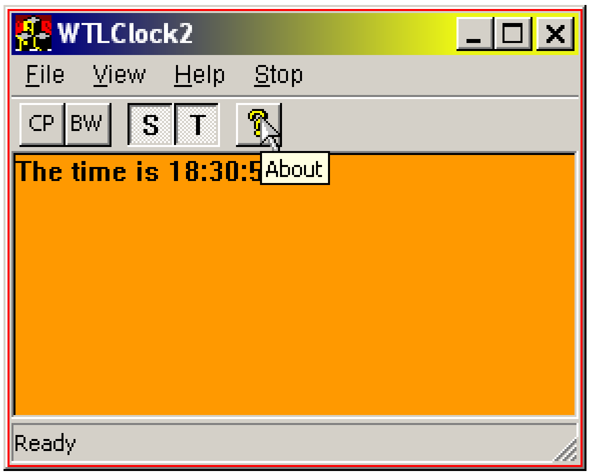
CMainFrame also has a *View* menu with two commands to show/hide the toolbar and status bar. These commands have IDs ID_VIEW_TOOLBAR and ID_VIEW_STATUS_BAR. CMainFrame has handlers for both commands that show or hide the corresponding bar. Here is the OnViewToolBar() handler:

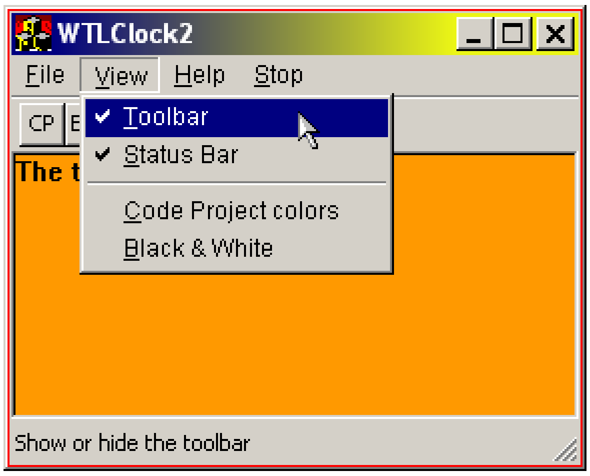
```
LRESULT CMainFrame::OnViewToolBar(WORD /*wNotifyCode*/, WORD /*wID*/,
                                  HWND /*hWndCtl*/, BOOL& /*bHandled*/)
{
    BOOL bVisible = !::IsWindowVisible(m_hWndToolBar);
    ::ShowWindow(m_hWndToolBar, bVisible ? SW_SHOWNOACTIVATE : SW_HIDE);
    UISetCheck(ID_VIEW_TOOLBAR, bVisible);
    UpdateLayout();
    return 0;
}
```

This toggles the visible state of the bar, toggles the check mark next to the *View|Toolbar* menu item, then calls UpdateLayout() to position the bar (if it is becoming visible) and resize the view window.

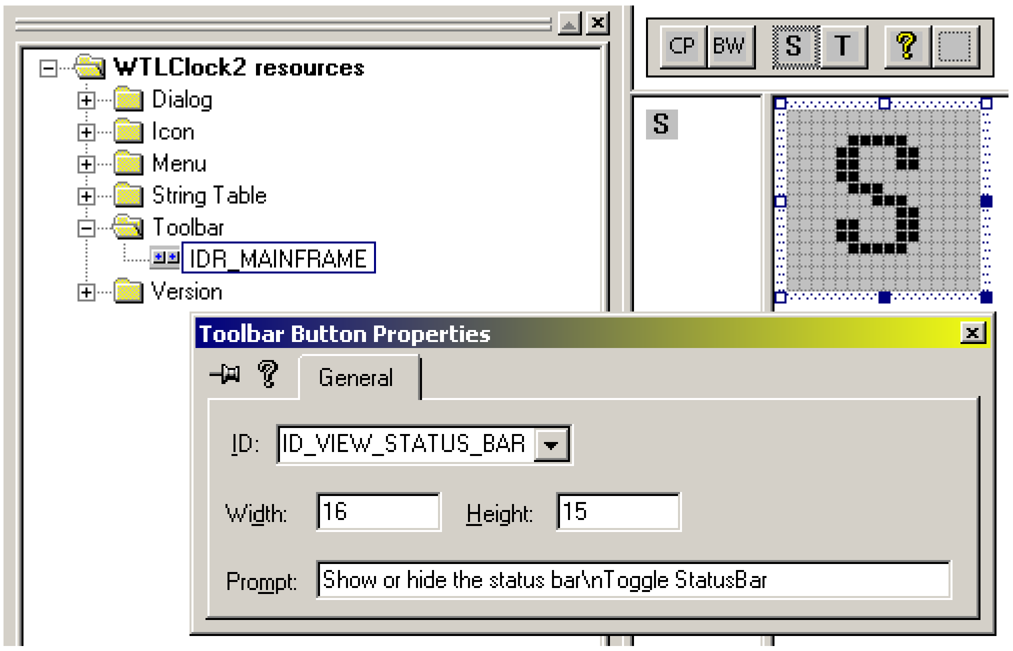
Built-in features of the bars

MFC's framework provides some nice features in its toolbars and status bars, such as tooltips for toolbar buttons and flyby help for menu items. WTL has comparable features implemented in `CFrameWindowImpl`. Below are screen shots showing both the tooltip and flyby help.





`CFrameWindowImplBase` has two message handlers that are used for these features. `OnMenuSelect()` handles `WM_MENUSELECT`, and it finds the flyby help string just like MFC does - it loads a string resource with the same ID as the currently-selected menu item, looks for a `\n` character in the string, and uses the text preceding the `\n` as the flyby help. `OnToolTipTextA()` and `OnToolTipTextW()` handle `TTN_GETDISPINFOA` and `TTN_GETDISPINFOW` respectively to provide tooltip text for toolbar buttons. Those handlers load the same string as `OnMenuSelect()`, but use the text *following* the `\n`. (As a side note, in WTL 7.0 and 7.1, `OnMenuSelect()` and `OnToolTipTextA()` are not DBCS-safe, as they do not check for DBCS lead/trail bytes when looking for the `\n`.) Here's an example of a toolbar button and its associated help string:



ID: ID_VIEW_STATUS_BAR

Width: 16 Height: 15

Prompt: Show or hide the status bar\nToggle StatusBar

Creating a toolbar with a different style

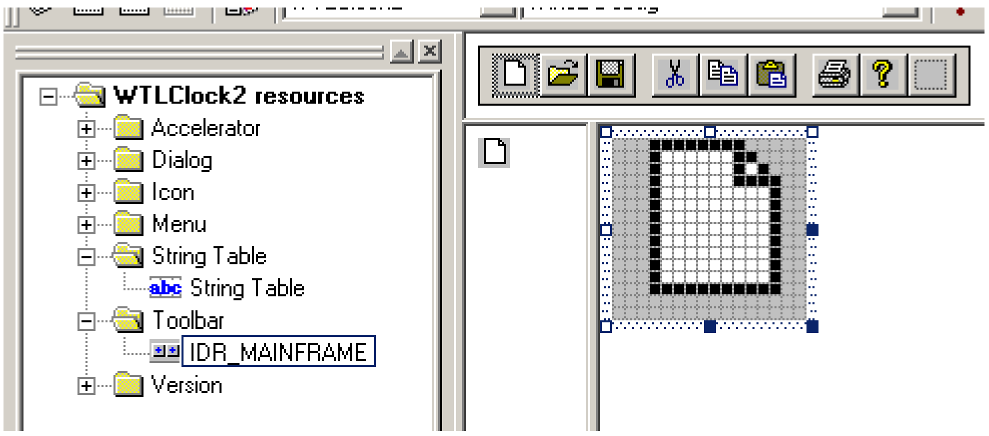
You can change the style for the toolbar by passing the style bits as the second parameter to `CreateSimpleToolBar()`. For example, to use 3D buttons and make the toolbar resemble the ones in Internet Explorer, use this code in `CMainFrame::OnCreate()`:

```
    CreateSimpleToolBar ( 0, ATL_SIMPLE_TOOLBAR_STYLE |
                             TBSTYLE_FLAT | TBSTYLE_LIST );
```

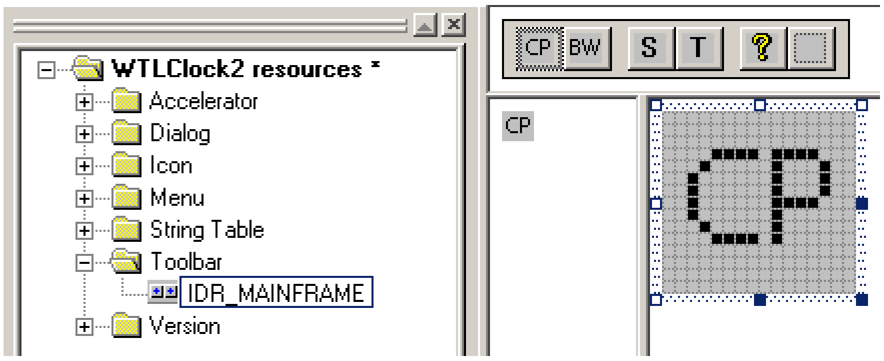
Note that if you use a common controls v6 manifest in your app, you don't have a choice with the buttons when running on Windows XP or later - toolbars always use flat buttons even if you don't specify the `TBSTYLE_FLAT` style when the toolbar is created.

The Toolbar Editor

The AppWizard creates several default buttons as we saw earlier. Only the *About* button is hooked up, however. You can edit the toolbar just as you do with an MFC project; the editor modifies the toolbar resource that `CreateSimpleToolBarCtrl()` uses to build the bar. Here's how the AppWizard-generated toolbar looks in the editor:



For our clock app, we'll add two buttons to change the colors in the view window, and two buttons that show/hide the toolbar and status bar. Here's our new toolbar:



The buttons are:

- `IDC_CP_COLORS`: Change the view to CodeProject colors
- `IDC_BW_COLORS`: Change the view to black and white
- `ID_VIEW_STATUS_BAR`: Show or hide the status bar
- `ID_VIEW_TOOLBAR`: Show or hide the toolbar

The first two buttons also have items on the *View* menu. They call a new function in the view class, `SetColors()`. Each one passes a foreground and background color that the view will use for its clock display. Handling these two buttons is no different from handling menu items with the `COMMAND_ID_HANDLER_EX` macro; check out the sample project if you want to see the details of the message handlers. In the next section, I'll cover UI updating the View Status Bar and View Toolbar buttons so they reflect the current state of the bars.

UI Updating Toolbar Buttons

The AppWizard-generated `CMainFrame` already has UI update handlers that check and uncheck the *View|Toolbar* and *View|Status Bar* menu items. This is done just as the app in Part II - there are UI update macros for each command in `CMainFrame`:

```
BEGIN_UPDATE_UI_MAP(CMainFrame)
    UPDATE_ELEMENT(ID_VIEW_TOOLBAR, UPDUI_MENUPOPUP)
    UPDATE_ELEMENT(ID_VIEW_STATUS_BAR, UPDUI_MENUPOPUP)
END_UPDATE_UI_MAP()
```

Our clock app has toolbar buttons with those same IDs, so the first step is to add the `UPDUI_TOOLBAR` flag to each macro:

```
BEGIN_UPDATE_UI_MAP(CMainFrame)
    UPDATE_ELEMENT(ID_VIEW_TOOLBAR, UPDUI_MENUPOPUP | UPDUI_TOOLBAR)
    UPDATE_ELEMENT(ID_VIEW_STATUS_BAR, UPDUI_MENUPOPUP | UPDUI_TOOLBAR)
END_UPDATE_UI_MAP()
```

There are two other functions to call to hook up toolbar button updating, but the wizard-generated code calls them, so if you build the project at this point, the menu items and toolbar buttons will both be updated.

Enabling toolbar UI updating

If you look in `CMainFrame::OnCreate()`, you'll see a new block of code that sets up the initial state of the two *View* menu items:

```
LRESULT CMainFrame::OnCreate( ... )
{
// ...

    m_hWndClient = m_view.Create(...);

    UIAddToolBar(m_hWndToolBar);
    UISetCheck(ID_VIEW_TOOLBAR, 1);
    UISetCheck(ID_VIEW_STATUS_BAR, 1);
// ...

}
```

UIAddToolBar() tells CUpdateUI the HWND of our toolbar, so it knows what window to send messages to when it needs to update the button states. The other important call is in OnIdle():

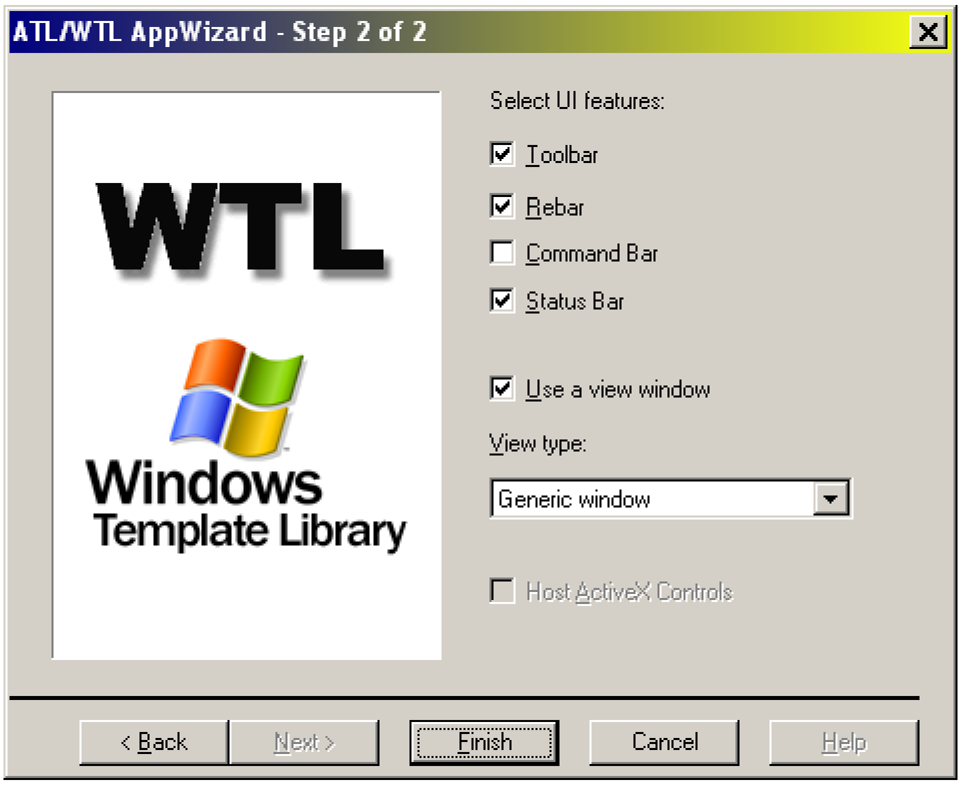
```
BOOL CMainFrame::OnIdle()
{
    UIUpdateToolBar();
    return FALSE;
}
```

Recall that OnIdle() is called by CMessageLoop::Run() when it has no messages waiting in the message queue. UIUpdateToolBar() goes through the update UI map, looks for elements with the UPDUI_TOOLBAR flag that have been changed with calls such as UISetCheck(), and changes the state of the buttons accordingly. Note that we didn't need these two steps when we were just updating popup menu items, because CUpdateUI handles WM_INITMENUPOPUP and updates the menu when that message is sent.

If you check out the sample project, it also shows how to UI update top-level menu items in the frame's menu bar. There is an item that executes the *Start* and *Stop* commands to start and stop the clock. While this is not a common (or even recommended) thing to do -- items in the menu bar should always be popups -- I included it for the sake of completeness in covering CUpdateUI. Look for the calls to UIAddMenuBar() and UIUpdateMenuBar().

Using a Rebar Instead of a Plain Toolbar

CFrameWindowImpl also supports using a rebar to give your app a look similar to Internet Explorer. Using a rebar is also the way to go if you need multiple toolbars. To use a rebar, check the Rebar box in the second page of the AppWizard, as shown here:



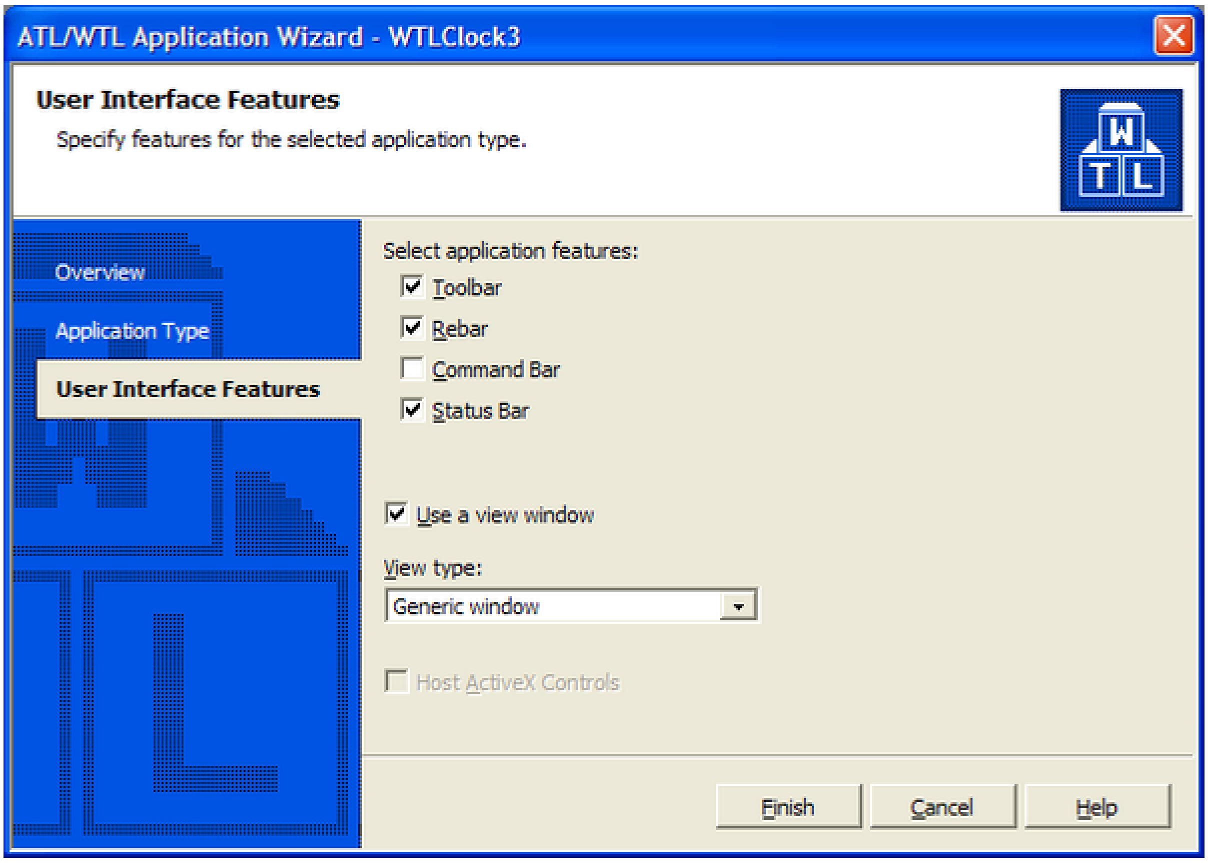
Select UI features:

- [✓] Toolbar
- [✓] Rebar
- [ ] Command Bar
- [✓] Status Bar

[✓] Use a view window

View type: Generic window

[ ] Host ActiveX Controls



User Interface Features

Specify features for the selected application type.

Select application features:

- [✓] Toolbar
- [✓] Rebar
- [ ] Command Bar
- [✓] Status Bar

[✓] Use a view window

View type: Generic window

[ ] Host ActiveX Controls

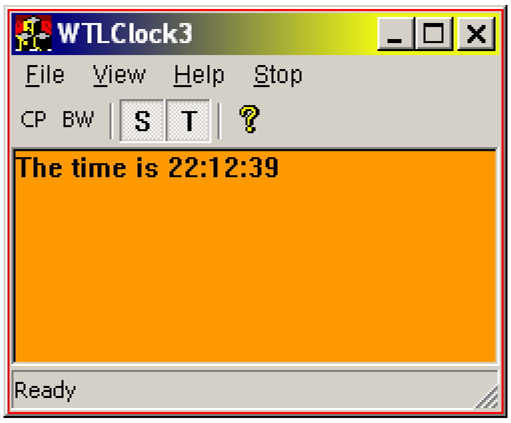
The second sample project, WTLClock3, was made with this option checked. If you're following along in the sample code, open WTLClock3 now.

The first thing you'll notice is that the code to create the toolbar is different. This makes sense since we're using a rebar in this app. Here is the relevant code:

```
LRESULT CMainFrame::OnCreate(...)
{
    HWND hWndToolBar = CreateSimpleToolBarCtrl ( m_hWnd,
                           IDR_MAINFRAME, FALSE,
                           ATL_SIMPLE_TOOLBAR_PANE_STYLE );

    CreateSimpleReBar(ATL_SIMPLE_REBAR_NOBORDER_STYLE);
    AddSimpleReBarBand(hWndToolBar);
// ...

}
```

It begins by creating a toolbar, but using a different style, ATL_SIMPLE_TOOLBAR_PANE_STYLE. This is a #define in atlframe.h that is similar to ATL_SIMPLE_TOOLBAR_STYLE, but has additional styles such as CCS_NOPARENTALIGN that are necessary for the toolbar to work properly as the child of the rebar.

The next line calls CreateSimpleReBar(), which creates a rebar and stores its HWND in m_hWndToolBar. Next, AddSimpleReBarBand() adds a band to the rebar, and tells the rebar that the toolbar will be contained in that band.



The time is 22:12:39

Ready

CMainFrame::OnViewToolBar() is also different. Instead of hiding m_hWndToolBar (that would hide the entire rebar, not just the one toolbar), it hides the band that contains the toolbar.

If you want to have multiple toolbars, you can create them and call AddSimpleReBarBand() in OnCreate() just like the wizard-generated code does for the first toolbar. Since CFrameWindowImpl uses the standard rebar control, there is no support for dockable toolbars like in MFC; all the user can do is rearrange the positions of the toolbars within the rebar.

Multi-Pane Status Bars

WTL has another status bar class that implements a bar with multiple panes, similar to the default MFC status bar that has CAPS LOCK and NUM LOCK indicators. The class is `CMultiPaneStatusBarCtrl`, and is demonstrated in the WTLClock3 sample project. This class supports limited UI updating, as well as a "default" pane that stretches to the full width of the bar to show flyby help when a popup menu is displayed.

The first step is to declare a `CMultiPaneStatusBarCtrl` member variable in `CMainFrame`:

```
class CMainFrame : public ...
{
//...

protected:
    CMultiPaneStatusBarCtrl m_wndStatusBar;
};
```

Then in `OnCreate()`, we create the bar and set it up for UI updating:

```
    m_hWndStatusBar = m_wndStatusBar.Create ( *this );
    UIAddStatusBar ( m_hWndStatusBar );
```

Notice that we store the status bar handle in `m_hWndStatusBar`, just like `CreateSimpleStatusBar()` would.

The next step is to set up the panes by calling `CMultiPaneStatusBarCtrl::SetPanes()`:

```
    BOOL SetPanes(int* pPanes, int nPanes, bool bSetText = true);
```

The parameters are:

pPanes

An array of pane IDs

nPanes

The number of elements in `pPanes`

bSetText

If true, all panes have their text set immediately. This is explained below.

The pane IDs can either be `ID_DEFAULT_PANE` to create the flyby help pane, or IDs of strings in the string table. For the non-default panes, WTL loads the string with the matching ID and calculates its width, then sets the corresponding pane to the same width. This is the same logic that MFC uses.

`bSetText` controls whether the panes show the strings immediately. If it is set to true, `SetPanes()` shows the strings in each pane, otherwise the panes are left blank.

Here's our call to `SetPanes()`:

```
    // Create the status bar panes.

int anPanes[] = { ID_DEFAULT_PANE, IDPANE_STATUS,
                  IDPANE_CAPS_INDICATOR };

    m_wndStatusBar.SetPanes ( anPanes, 3, false );
```

The string `IDPANE_STATUS` is "@@@@", which should (hopefully) be enough space to show the two clock status strings "Running" and "Stopped". Just as with MFC, you have to approximate how much space you'll need for the pane. The string `IDPANE_CAPS_INDICATOR` is "CAPS".

UI updating the panes

In order for us to update the pane text, we'll need entries in the update UI map:

```
    BEGIN_UPDATE_UI_MAP(CMainFrame)
        //...

        UPDATE_ELEMENT(1, UPDUI_STATUSBAR)  // clock status

        UPDATE_ELEMENT(2, UPDUI_STATUSBAR)  // CAPS indicator

    END_UPDATE_UI_MAP()
```

The first parameter in the macro is the *index* of the pane, not its ID. Watch out for this if you rearrange the panes - if not every pane has an entry in this map, you will need to update the numbers in the UPDATE_ELEMENT macros to match the new order.

Since we pass false as the third parameter to SetPanes(), the panes are initially empty. Our next step is to set the initial text of the clock status pane to "Running".

```
    // Set the initial text for the clock status pane.

    UISetText ( 1, _T("Running") );
```

Again, the first parameter is the index of the pane. UISetText() is the only UI update call that works on status bars.

Finally, we need to add a call to UIUpdateStatusBar() in CMainFrame::OnIdle() so that the status bar panes are updated at idle time:

```
BOOL CMainFrame::OnIdle()
{
    UIUpdateToolBar();
    UIUpdateStatusBar();
    return FALSE;
}
```

There is a problem in CUpdateUI that appears when you use UIUpdateStatusBar() - text in menu items is not updated after you use UISetText()! If you look at the WTLClock3 project, the clock start/stop menu item has been moved to a *Clock* menu, and the handler for that command sets the menu item's text. However, if the call to UIUpdateStatusBar() is present, the UISetText() call does not take effect. This bug is still present in WTL 7.1; some folks have investigated and come up with fixes - see the discussion forum at the end of the article for more details.

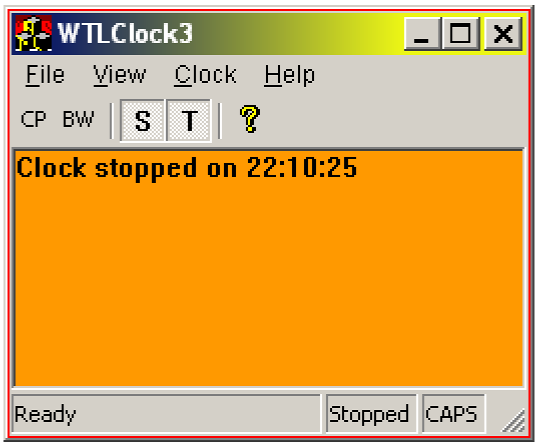
Finally, we need to check the state of the CAPS LOCK key and update pane 2 accordingly. This code can go right in OnIdle(), so the state will get checked every time the app goes idle.

```
BOOL CMainFrame::OnIdle()
{
    // Check the current Caps Lock state, and if it is on, show the

    // CAPS indicator in pane 2 of the status bar.

    if ( GetKeyState(VK_CAPITAL) & 1 )
```

```
        UISetText ( 2, CString(LPCTSTR(IDPANE_CAPS_INDICATOR)) );
    else
        UISetText ( 2, _T("") );

    UIUpdateToolBar();
    UIUpdateStatusBar();
    return FALSE;
}
```

The first `UISetText()` call loads the "CAPS" string from the string table using a neat (but fully documented) trick in the `CString` constructor.

So after all that code, here's what the status bar looks like:



Up Next: All About Dialogs

In Part IV, I'll cover dialogs (both the ATL classes and WTL enhancements), control wrappers, and more WTL message handling improvements that relate to dialogs and controls.

References

"How to use the WTL multipane status bar control" by Ed Gadziemski goes into more detail on the `CMultiPaneStatusBarCtrl` class.

Part IV - Dialogs and Controls

Introduction to Part IV

Dialogs and controls are one area where MFC really saves you time and effort. Without MFC's control classes, you'd be stuck filling in structs and writing tons of `SendMessage` calls to manage controls. MFC also provides dialog data exchange (DDX), which transfers data between controls and variables. WTL supports all those features as well, and adds some more improvements in its common control wrappers. In this article we'll look at a dialog-based app that demonstrates the MFC features that you're used to, as well as some more WTL message-handling enhancements. Advanced UI features and new controls in WTL will be covered in Part V.

Refresher on ATL Dialogs

Recall from Part I that ATL has two dialog classes, `CDialogImpl` and `CAxDialogImpl`. `CAxDialogImpl` is used for dialogs that host ActiveX controls. We won't cover ActiveX controls in this article, so the sample code uses `CDialogImpl`.

To create a new dialog class, you do three things:

1. Create the dialog resource
2. Write a new class derived from `CDialogImpl`
3. Create a public member variable called `IDD`, and set it to the dialog's resource ID.

Then you can add message handlers just like in a frame window. WTL doesn't change that process, but it does add other features that you can use in dialogs.

Control Wrapper Classes

WTL has lots of control wrappers which should be familiar to you because the WTL classes are usually named the same (or almost the same) as their MFC counterparts. The methods are usually named the same as well, so you can use the MFC documentation while using the WTL wrappers. Failing that, the F12 key comes in handy when you need to jump to the definition of one of the classes.

Here are the wrapper classes for built-in controls:

- User controls: `CStatic`, `CButton`, `CListBox`, `CComboBox`, `CEdit`, `CScrollBar`, `CDragListBox`
- Common controls: `CImageList`, `CListViewCtrl` (`CListCtrl` in MFC), `CTreeViewCtrl` (`CTreeCtrl` in MFC), `CHeaderCtrl`, `CToolBarCtrl`, `CStatusBarCtrl`, `CTabCtrl`, `CToolTipCtrl`, `CTrackBarCtrl` (`CSliderCtrl` in MFC), `CUpDownCtrl` (`CSpinButtonCtrl` in MFC), `CProgressBarCtrl`, `CHotKeyCtrl`, `CAnimateCtrl`, `CRichEditCtrl`, `CReBarCtrl`, `CComboBoxEx`, `CDateTimePickerCtrl`, `CMonthCalendarCtrl`, `CIPAddressCtrl`
- Common control wrappers not in MFC: `CPagerCtrl`, `CFlatScrollBar`, `CLinkCtrl` (clickable hyperlink, available in XP and later)

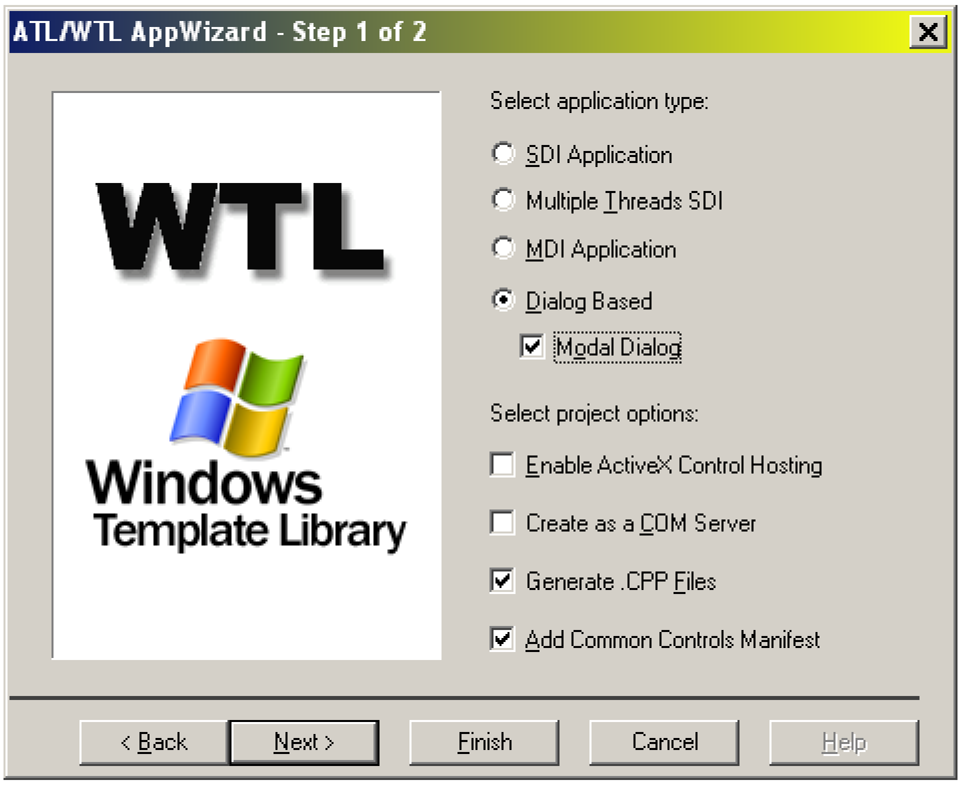
There are also a few WTL-specific classes: `CBitmapButton`, `CCheckListViewCtrl` (list view control with check boxes), `CTreeViewCtrlEx` and `CTreeItem` (used together, `CTreeItem` wraps an `HTREEITEM`), `CHyperLink` (clickable hyperlink, available on all OSes)

One thing to note is that most of the wrapper classes are window interface classes, like `CWindow`. They wrap an `HWND` and provide wrappers around messages (for instance, `CListBox::GetCurSel()` wraps `LB_GETCURSEL`). So like `CWindow`, it is cheap to create a temporary control wrapper and attach it to an existing control. Also like `CWindow`, the control is not destroyed when the control wrapper is destructed. The exceptions are `CBitmapButton`, `CCheckListViewCtrl`, and `CHyperLink`.

Since these articles are aimed at experienced MFC programmers, I won't be spending much time on the details of the wrapper classes that are similar to their MFC counterparts. I will, however, be covering the new classes in WTL; `CBitmapButton` is quite different from the MFC class of the same name, and `CHyperLink` is completely new.

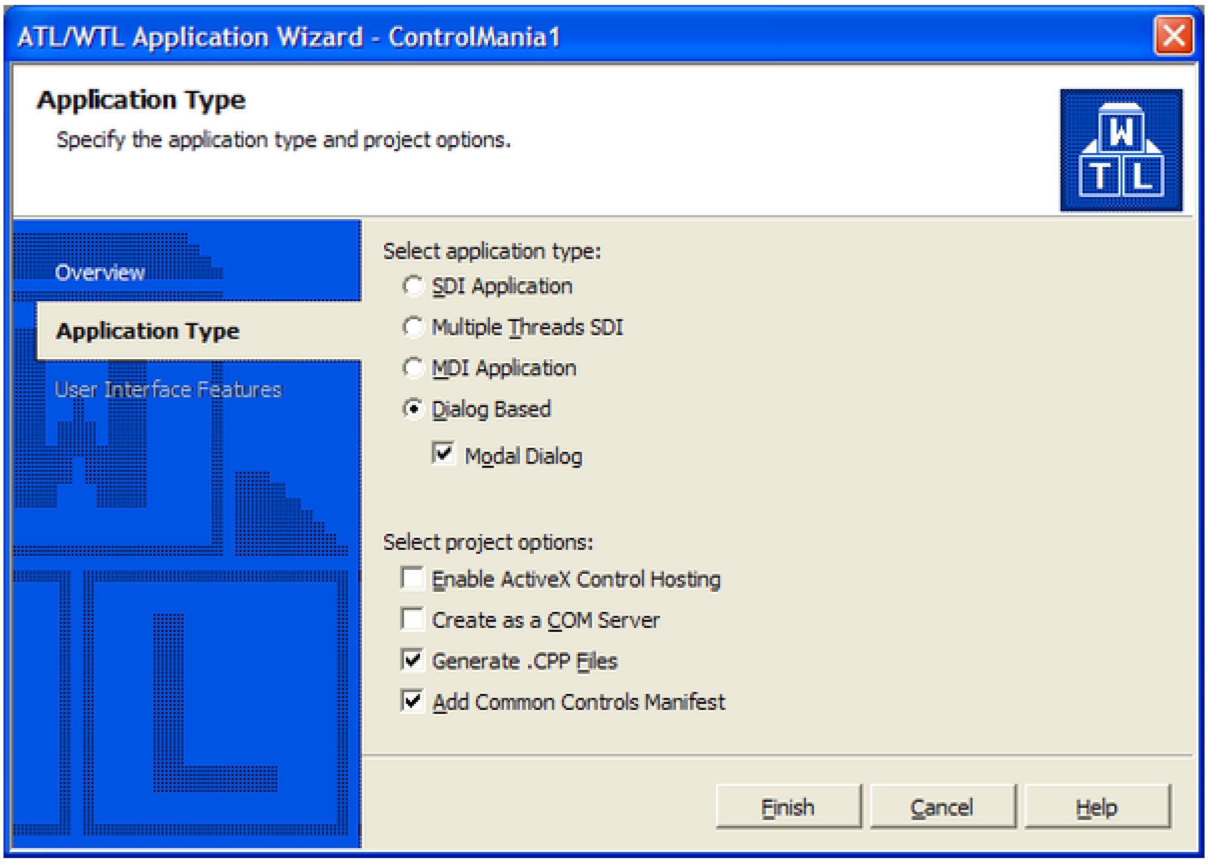
Creating a Dialog-Based App with the AppWizard

Fire up VC and start the WTL AppWizard. I'm sure you're as tired as I am with clock apps, so let's call our next app *ControlMania1*. On the first page of the AppWizard, click *Dialog Based*. We also have a choice between making a modal or modeless dialog. The difference is important and I will cover it in Part V, but for now let's go with the simpler one, modal. Check *Modal Dialog* and *Generate .CPP Files* as shown here:


ATL/WTL AppWizard - Step 1 of 2

Select application type:
- ○ SDI Application
- ○ Multiple Threads SDI
- ○ MDI Application
- ● Dialog Based
  - ☑ Modal Dialog

Select project options:
- ☐ Enable ActiveX Control Hosting
- ☐ Create as a COM Server
- ☑ Generate .CPP Files
- ☑ Add Common Controls Manifest


ATL/WTL Application Wizard - ControlMania1

Application Type

Specify the application type and project options.

Select application type:
- ○ SDI Application
- ○ Multiple Threads SDI
- ○ MDI Application
- ● Dialog Based
  - ☑ Modal Dialog

Select project options:
- ☐ Enable ActiveX Control Hosting
- ☐ Create as a COM Server
- ☑ Generate .CPP Files
- ☑ Add Common Controls Manifest

All the options on the second page (for the VC6 wizard) or the *User Interface Features* tab (in VC7) are only meaningful when the main window is a frame window, so they are all disabled. Click *Finish* to complete the wizard.

As you might expect, the AppWizard-generated code is much simpler for a dialog-based app. *ControlMania1.cpp* has the `_tWinMain()` function, here are the important parts:

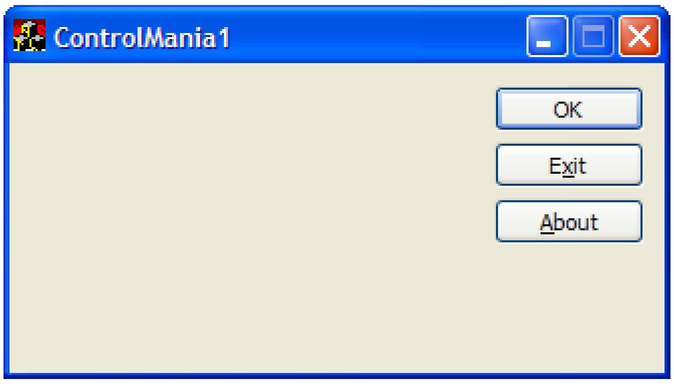
```
int WINAPI _tWinMain (
    HINSTANCE hInstance, HINSTANCE /*hPrevInstance*/,
    LPTSTR lpstrCmdLine, int nCmdShow )
{
    HRESULT hRes = ::CoInitialize(NULL);

    AtlInitCommonControls(ICC_COOL_CLASSES | ICC_BAR_CLASSES);

    hRes = _Module.Init(NULL, hInstance);

    int nRet = 0;
    // BLOCK: Run application

    {
        CMainDlg dlgMain;
        nRet = dlgMain.DoModal();
    }

    _Module.Term();
    ::CoUninitialize();
    return nRet;
}
```

The code first initializes COM and creates a single-threaded apartment. This is necessary for dialogs that host ActiveX controls; if your app isn't using COM, you can safely remove the `CoInitialize()` and `CoUninitialize()` calls. Next, the code calls the WTL utility function `AtlInitCommonControls()`, which is a wrapper for `InitCommonControlsEx()`. The global `_Module` is initialized, and the main dialog is displayed. (Note that ATL dialogs created with `DoModal()` actually are modal, unlike MFC where all dialogs are modeless and MFC simulates modality by manually disabling the dialog's parent.) Finally, `_Module` and COM are uninitialized, and the value returned by `DoModal()` is used as the app's exit code.

The block around the `CMainDlg` variable is important because `CMainDlg` may have members that use ATL and WTL features. Those members may also use ATL/WTL features in their destructors. If the block were not present, the `CMainDlg` destructor (and the destructors of the members) would run after the call to `_Module.Term()` (which uninitializes ATL/WTL) and try to use ATL/WTL features, which could cause a hard-to-diagnose crash. (As a historical note, the WTL 3

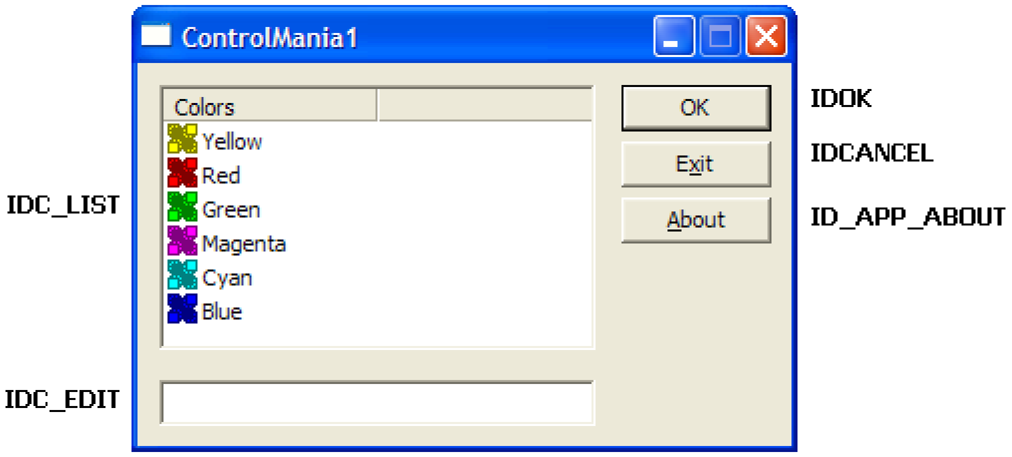
AppWizard-generated code did not have a block there, and some of my apps did crash.)

You can build and run the app right away, although the dialog is pretty bare:



The code in `CMainDlg` handles `WM_INITDIALOG`, `WM_CLOSE`, and all three buttons. Take a quick glance through the code now if you like; you should be able to follow the `CMainDlg` declaration, its message map, and its message handlers.

This sample project will demonstrate how to hook up variables to the controls. Here's the app with a couple more controls; you can refer back to this diagram for the following discussions.



Since the app uses a list view control, the call to `AtlInitCommonControls()` will need to be changed. Change it to:

```
AtlInitCommonControls ( ICC_WIN95_CLASSES );
```

That registers more classes than necessary, but it saves us having to remember to add `ICC_*` constants when we add different types of controls to the dialog.

Using the Control Wrapper Classes

There are several ways to associate a member variable with a control. Some use plain `CWindow`s (or another window interface class, like `CListViewCtrl`), while others use a `CWindowImpl`-derived class. Using a `CWindow` is fine if you just need a temporary variable, while a `CWindowImpl` is required if you need to subclass a control and handle messages sent to it.

ATL Way 1 - Attaching a CWindow

The simplest method is to declare a `CWindow` or other window interface class, and call its `Attach()` method. You can also use the `CWindow` constructor or assignment operator to associate the variable with a control's `HWND`.

This code demonstrates all three methods of associating variables with the list control:

```
HWND hwndList = GetDlgItem(IDC_LIST);
CListViewCtrl wndList1 (hwndList);  // use constructor

CListViewCtrl wndList2, wndList3;

  wndList2.Attach ( hwndList );     // use Attach method

  wndList3 = hwndList;             // use assignment operator
```

Remember that the `CWindow` destructor does not destroy the window, so there is no need to detach the variables before they go out of scope. You can also use this technique with member variables if you wish - you can attach the variables in the `OnInitDialog()` handler.

ATL Way 2 - CContainedWindow

`CContainedWindow` is sort of halfway between using a `CWindow` and a `CWindowImpl`. It lets you subclass a control, then handle that control's messages *in the control's parent window*. This lets you put all the message handlers in the dialog class, and you don't have to write separate `CWindowImpl` classes for each control. Note that you don't use `CContainedWindow` to handle `WM_COMMAND`, `WM_NOTIFY`, or other notification messages, because those messages are always sent to the control's parent.

The actual class, `CContainedWindowT`, is a template class that takes a window interface class name as its template parameter. There is a specialization `CContainedWindowT<CWindow>` that works like a plain `CWindow`; this is typedef'ed to the shorter name `CContainedWindow`. To use a different window interface class, specify its name as the template parameter, for example `CContainedWindowT<CListViewCtrl>`.

To hook up a `CContainedWindow`, you do four things:

1. Create a `CContainedWindowT` member variable in the dialog.
2. Put handlers in an `ALT_MSG_MAP` section of the dialog's message map
3. In the dialog's constructor, call the `CContainedWindowT` constructor and tell it which `ALT_MSG_MAP` section it should route messages to.
4. In `OnInitDialog()`, call the `CContainedWindowT::SubclassWindow()` method to associate a variable with a control.

In ControlMania1, we'll use a `CContainedWindow` for the *OK* and *Exit* buttons. The dialog will handle `WM_SETCURSOR` messages sent to each button, and change the cursor.

Let's go through the steps. First, we add `CContainedWindow` members in `CMainDlg`.

```
class CMainDlg : public CDialogImpl<CMainDlg>
{
// ...

protected:
    CContainedWindow m_wndOKBtn, m_wndExitBtn;
};
```

Second, we add `ALT_MSG_MAP` sections. The *OK* button will use section 1, while the *Exit* button will use section 2. This means that all messages sent to the *OK* button will be routed to the `ALT_MSG_MAP(1)` section, and all messages sent to the *Exit* button will be routed to the `ALT_MSG_MAP(2)` section.

```
class CMainDlg : public CDialogImpl<CMainDlg>
{
public:
    BEGIN_MSG_MAP_EX(CMainDlg)
        MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
        COMMAND_ID_HANDLER(ID_APP_ABOUT, OnAppAbout)
        COMMAND_ID_HANDLER(IDOK, OnOK)
        COMMAND_ID_HANDLER(IDCANCEL, OnCancel)
    ALT_MSG_MAP(1)
        MSG_WM_SETCURSOR(OnSetCursor_OK)
    ALT_MSG_MAP(2)
        MSG_WM_SETCURSOR(OnSetCursor_Exit)
    END_MSG_MAP()

    LRESULT OnSetCursor_OK(HWND hwndCtrl, UINT uHitTest, UINT uMouseMsg);
    LRESULT OnSetCursor_Exit(HWND hwndCtrl, UINT uHitTest, UINT uMouseMsg);
};
```

Third, we call the CContainedWindow constructor for each member and tell it which ALT_MSG_MAP section to use.

```
CMainDlg::CMainDlg() : m_wndOKBtn(this, 1),
                       m_wndExitBtn(this, 2)
{
}
```

The constructor parameters are a CMessageMap* and an ALT_MSG_MAP section number. The first parameter will usually be this, meaning that the dialog's own message map will be used, and the second parameter tells the object which ALT_MSG_MAP section it should send its messages to.

Important note: If you are using VC 7.0 or 7.1 and WTL 7.0 or 7.1, you will run into an assert failure if a CWindowImpl- or CDialogImpl-derived class does all of these things together:

- The message map uses BEGIN_MSG_MAP instead of BEGIN_MSG_MAP_EX.
- The map contains an ALT_MSG_MAP section.
- A CContainedWindowT variable routes messages to that ALT_MSG_MAP section.
- That ALT_MSG_MAP section uses the new WTL message handler macros.

See this thread in the article's discussion forum for more details. The solution is to use BEGIN_MSG_MAP_EX instead of BEGIN_MSG_MAP.

Finally, we associate each CContainedWindow with a control.

```
LRESULT CMainDlg::OnInitDialog(...)
{
// ...
```

```
    // Attach CContainedWindows to OK and Exit buttons

    m_wndOKBtn.SubclassWindow ( GetDlgItem(IDOK) );
    m_wndExitBtn.SubclassWindow ( GetDlgItem(IDCANCEL) );

    return TRUE;
}
```

Here are the new `WM_SETCURSOR` handlers:

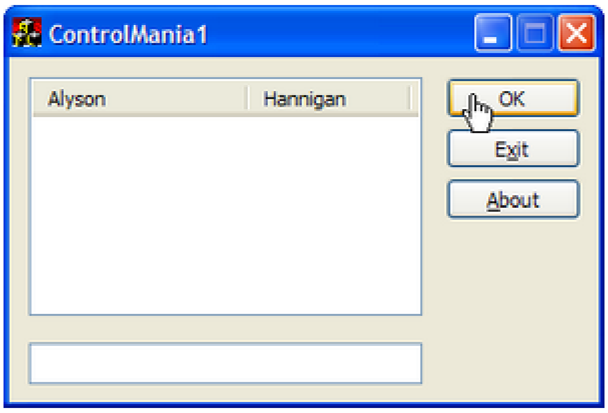
```
LRESULT CMainDlg::OnSetCursor_OK (
    HWND hwndCtrl, UINT uHitTest, UINT uMouseMsg )
{
static HCURSOR hcur = LoadCursor ( NULL, IDC_HAND );

    if ( NULL != hcur )
        {
        SetCursor ( hcur );
        return TRUE;
        }
    else
        {
        SetMsgHandled(false);
        return FALSE;
        }
}

LRESULT CMainDlg::OnSetCursor_Exit (
    HWND hwndCtrl, UINT uHitTest, UINT uMouseMsg )
{
static HCURSOR hcur = LoadCursor ( NULL, IDC_NO );

    if ( NULL != hcur )
        {
        SetCursor ( hcur );
        return TRUE;
        }
    else
        {
        SetMsgHandled(false);
        return FALSE;
        }
}
```

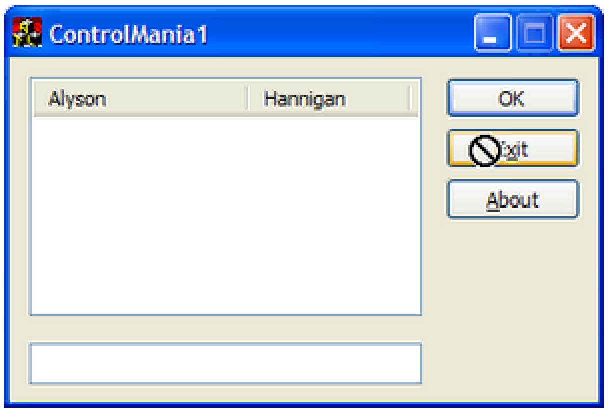
If you wanted to use CButton features, you could declare the variables like this:

```
    CContainedWindowT<CButton> m_wndOKBtn;
```

and then the CButton methods would be available.

You can see the WM_SETCURSOR handlers in action when you move the cursor over the buttons:





ATL Way 3 - Subclassing

Method 3 involves creating a CWindowImpl-derived class and using it to subclass a control. This is similar to method 2, however the message handlers go in the CWindowImpl class instead of the dialog class.

ControlMania1 uses this method to subclass the *About* button in the main dialog. Here is the CButtonImpl class, which is derived from CWindowImpl and handles WM_SETCURSOR:

```
class CButtonImpl : public CWindowImpl<CButtonImpl, CButton>
{
    BEGIN_MSG_MAP_EX(CButtonImpl)
        MSG_WM_SETCURSOR(OnSetCursor)
    END_MSG_MAP()

    LRESULT OnSetCursor(HWND hwndCtrl, UINT uHitTest, UINT uMouseMsg)
    {
    static HCURSOR hcur = LoadCursor ( NULL, IDC_SIZEALL );

        if ( NULL != hcur )
            {
            SetCursor ( hcur );
            return TRUE;
            }
        else
            {
            SetMsgHandled(false);
            return FALSE;
            }
    }
};
```

Then in the main dialog, we declare a `CButtonImpl` member variable:

```
class CMainDlg : public CDialogImpl<CMainDlg>
{
// ...

protected:
    CContainedWindow m_wndOKBtn, m_wndExitBtn;
    CButtonImpl m_wndAboutBtn;
};
```

And finally, in `OnInitDialog()`, we subclass the button.

```
LRESULT CMainDlg::OnInitDialog(...)
{
// ...

    // Attach CContainedWindows to OK and Exit buttons

    m_wndOKBtn.SubclassWindow ( GetDlgItem(IDOK) );
    m_wndExitBtn.SubclassWindow ( GetDlgItem(IDCANCEL) );

    // CButtonImpl: subclass the About button

    m_wndAboutBtn.SubclassWindow ( GetDlgItem(ID_APP_ABOUT) );

    return TRUE;
}
```

WTL Way 1 - DDX_CONTROL

WTL's DDX (dialog data exchange) support works a lot like MFC's, and it can rather painlessly connect a variable to a control. To begin, you need a `CWindowImpl`-derived class as in the previous example. We'll be using a new class this time, `CEditImpl`, since this example will subclass the edit control. You also need to #include atlddx.h in stdafx.h to bring in the DDX code.

To add DDX support to `CMainDlg`, add `CWinDataExchange` to the inheritance list:

```
class CMainDlg : public CDialogImpl<CMainDlg>,
                 public CWinDataExchange<CMainDlg>
{
//...

};
```

Next you create a DDX map in the class, which is similar to the `DoDataExchange()` function that the ClassWizard generates in MFC apps. There are several `DDX_*` macros for varying types of data; the one we'll use here is `DDX_CONTROL` to connect a variable with a control. This time, we'll use `CEditImpl` that handles `WM_CONTEXTMENU` to do something when you right-click in the control.

```
class CEditImpl : public CWindowImpl<CEditImpl, CEdit>
{
    BEGIN_MSG_MAP_EX(CEditImpl)
        MSG_WM_CONTEXTMENU(OnContextMenu)
    END_MSG_MAP()

    void OnContextMenu ( HWND hwndCtrl, CPoint ptClick )
    {
        MessageBox("Edit control handled WM_CONTEXTMENU");
    }
};

class CMainDlg : public CDialogImpl<CMainDlg>,
                 public CWinDataExchange<CMainDlg>
{
//...


    BEGIN_DDX_MAP(CMainDlg)
        DDX_CONTROL(IDC_EDIT, m_wndEdit)
    END_DDX_MAP()

protected:
    CContainedWindow m_wndOKBtn, m_wndExitBtn;
    CButtonImpl m_wndAboutBtn;
    CEditImpl   m_wndEdit;
};
```

Finally, in `OnInitDialog()`, we call the `DoDataExchange()` function that is inherited from `CWinDataExchange`. The first time `DoDataExchange()` is called, it subclasses controls as necessary. So in this example, `DoDataExchange()` will subclass the control with ID `IDC_EDIT`, and connect it to the variable `m_wndEdit`.

```
LRESULT CMainDlg::OnInitDialog(...)
{
// ...

    // Attach CContainedWindows to OK and Exit buttons

    m_wndOKBtn.SubclassWindow ( GetDlgItem(IDOK) );
    m_wndExitBtn.SubclassWindow ( GetDlgItem(IDCANCEL) );

    // CButtonImpl: subclass the About button

    m_wndAboutBtn.SubclassWindow ( GetDlgItem(ID_APP_ABOUT) );

    // First DDX call, hooks up variables to controls.

    DoDataExchange(false);

    return TRUE;
}
```

The parameter to DoDataExchange() has the same meaning as the parameter to MFC's UpdateData() function. We'll cover that in more detail in the next section.

If you run the ControlMania1 project, you can see all this subclassing in action. Right-clicking in the edit box will pop up the message box, and the cursor will change shape over the buttons as shown earlier.

WTL Way 2 - DDX_CONTROL_HANDLE

A new feature that was added in WTL 7.1 is the DDX_CONTROL_HANDLE macro. In WTL 7.0, if you wanted to hook up a plain window interface class (such as CWindow, CListViewCtrl, etc.) with DDX, you couldn't use DDX_CONTROL because DDX_CONTROL only works with CWindowImpl-derived classes. With the exception of the different base class requirement, DDX_CONTROL_HANDLE works the same as DDX_CONTROL.

If you're still using WTL 7.0, you can use this macro to define CWindowImpl-derived classes that will work with DDX_CONTROL:

```
#define DDX_CONTROL_IMPL(x) \
    class x##_ddx : public CWindowImpl<x##_ddx, x> \
        { public: DECLARE_EMPTY_MSG_MAP() };
```

You can then write:

```
DDX_CONTROL_IMPL(CListViewCtrl)
```

and you will have a class called CListViewCtrl_ddx that works like a CListViewCtrl but will be accepted by DDX_CONTROL.

More on DDX

DDX can, of course, actually do data exchange too. WTL supports exchanging string data between an edit box and a string variable. It can also parse a string as a number, and transfer that data between an integer or floating-point variable. And it also supports transferring the state of a check box or group of radio buttons to/from an int.

DDX macros

Each DDX macro expands to a `CWinDataExchange` method call that does the work. The macros all have the general form: `DDX_FOO(controlID, variable)`. Each macro takes a different type of variable, and some like `DDX_TEXT` are overloaded to accept many types.

`DDX_TEXT`

Transfers text data to/from an edit box. The variable can be a `CString`, `BSTR`, `CComBSTR`, or statically-allocated character array. Using an array allocated with `new` will not work.

`DDX_INT`

Transfers numerical data between an edit box and an `int`.

`DDX_UINT`

Transfers numerical data between an edit box and an `unsigned int`.

`DDX_FLOAT`

Transfers numerical data between an edit box and a `float` or `double`.

`DDX_CHECK`

Transfers the state of a check box to/from an `int` or `bool`.

`DDX_RADIO`

Transfers the state of a group of radio buttons to/from an `int`.

`DDX_CHECK` can take either an `int` or `bool` variable. The `int` version accepts/returns the values 0, 1, and 2 (or equivalently, `BST_UNCHECKED`, `BST_CHECKED`, and `BST_INDETERMINATE`). The `bool` version (added in WTL 7.1) can be used when a check box will never be in the indeterminate state; this version accepts/returns `true` if the check box is checked, or `false` if it's unchecked. If the check box happens to be in the indeterminate state, `DDX_CHECK` returns `false`.

There is also an additional floating-point macro that was added in WTL 7.1:

`DDX_FLOAT_P(controlID, variable, precision)`

Similar to `DDX_FLOAT`, but when setting the text in an edit box, the number is formatted to show at most `precision` significant digits.

A note about using `DDX_FLOAT` and `DDX_FLOAT_P`: When you use this in your app, you need to add a #define to stdafx.h, before any WTL headers are included:

```
#define _ATL_USE_DDX_FLOAT
```

This is necessary because by default, floating-point support is disabled as a size optimization.

More about DoDataExchange()

You call the DoDataExchange() method just as you call UpdateData() in MFC. The prototype for DoDataExchange() is:

```
BOOL DoDataExchange ( BOOL bSaveAndValidate = FALSE,
                      UINT nCtlID = (UINT)-1 );
```

The parameters are:

bSaveAndValidate

Flag indicating which direction the data will be transferred. Passing TRUE transfers from the controls to the variables. Passing FALSE transfers from the variables to the controls. Note that the default for this parameter is FALSE, while the default for MFC's UpdateData() is TRUE. You can also use the symbols DDX_SAVE and DDX_LOAD (defined as TRUE and FALSE respectively) as the parameter, if you find that easier to remember.

nCtlID

Pass -1 to update all controls. Otherwise, if you want to use DDX on only one control, pass the control's ID.

DoDataExchange() returns TRUE if the controls are updated successfully, or FALSE if not. There are two functions you can override in your dialog to handle errors. The first, OnDataExchangeError() is called if the data exchange fails for any reason. The default implementation in CWinDataExchange sounds a beep and sets focus to the control that caused the error. The other function is OnDataValidateError(), but we'll get to that in Part V when we cover DDV.

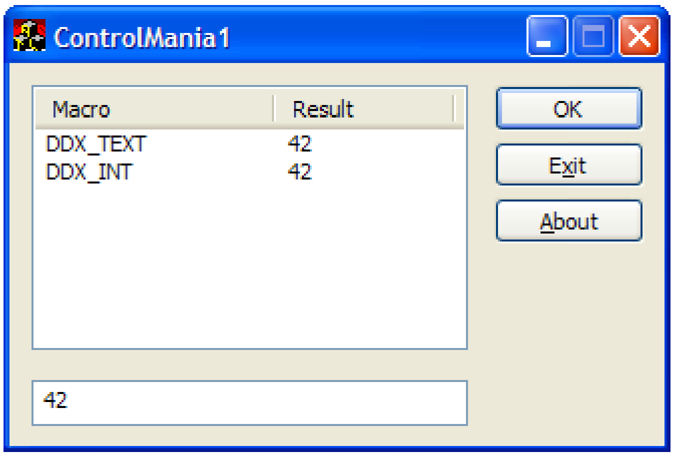
Using DDX

Let's add a couple of variables to CMainDlg for use with DDX.

```
class CMainDlg : public ...
{
//...

    BEGIN_DDX_MAP(CMainDlg)
        DDX_CONTROL(IDC_EDIT, m_wndEdit)
        DDX_TEXT(IDC_EDIT, m_sEditContents)
        DDX_INT(IDC_EDIT, m_nEditNumber)
    END_DDX_MAP()

protected:
    // DDX variables

    CString m_sEditContents;
    int     m_nEditNumber;
};
```

In the *OK* button handler, we first call `DoDataExchange()` to transfer the data from the edit control to the two variables we just added. We then show the results in the list control.

```
LRESULT CMainDlg::OnOK ( UINT uCode, int nID, HWND hWndCtl )
{
CString str;

    // Transfer data from the controls to member variables.

    if ( !DoDataExchange(true) )
        return;

    m_wndList.DeleteAllItems();

    m_wndList.InsertItem ( 0, _T("DDX_TEXT") );
    m_wndList.SetItemText ( 0, 1, m_sEditContents );

    str.Format ( _T("%d"), m_nEditNumber );
    m_wndList.InsertItem ( 1, _T("DDX_INT") );
    m_wndList.SetItemText ( 1, 1, str );
}
```

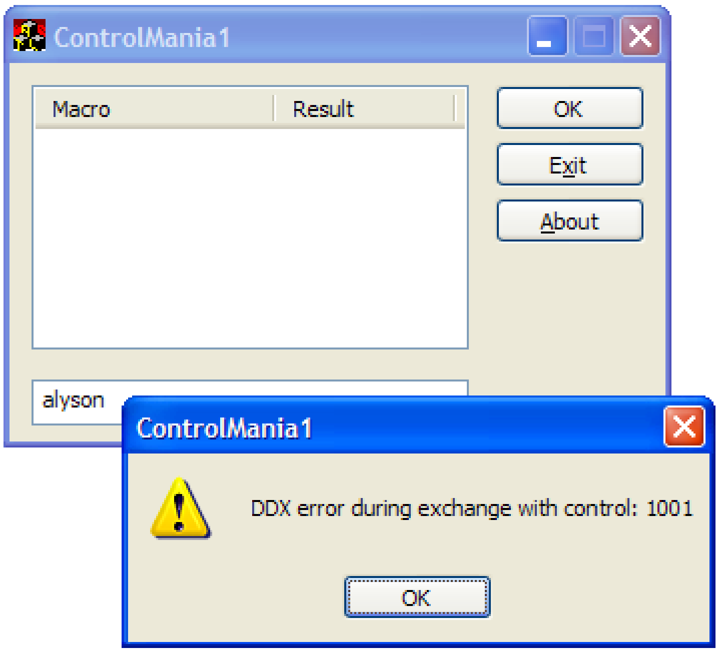


Macro	Result
DDX_TEXT	42
DDX_INT	42

42

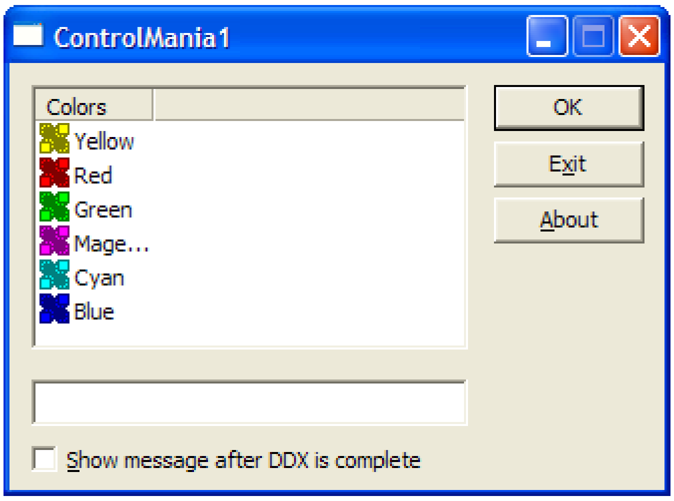
If you enter non-numerical text in the edit box, DDX_INT will fail and call OnDataExchangeError(). CMainDlg overrides OnDataExchangeError() to show a message box:

```
void CMainDlg::OnDataExchangeError ( UINT nCtrlID, BOOL bSave )
{
CString str;

    str.Format ( _T("DDX error during exchange with control: %u"), nCtrlID );
    MessageBox ( str, _T("ControlMania1"), MB_ICONWARNING );

    ::SetFocus ( GetDlgItem(nCtrlID) );
}
```



DDX error during exchange with control: 1001

As our last DDX example, let's add a check box to show the usage of DDX_CHECK:



IDC_SHOW_MSG

This check box will never be in the indeterminate state, so we can use a `bool` variable with `DDX_CHECK`. Here are the changes to make hook up the check box to DDX:

```
class CMainDlg : public ...
{
//...

    BEGIN_DDX_MAP(CMainDlg)
        DDX_CONTROL(IDC_EDIT, m_wndEdit)
        DDX_TEXT(IDC_EDIT, m_sEditContents)
        DDX_INT(IDC_EDIT, m_nEditNumber)
        DDX_CHECK(IDC_SHOW_MSG, m_bShowMsg)
    END_DDX_MAP()

protected:
    // DDX variables

    CString m_sEditContents;
    int     m_nEditNumber;
    bool    m_bShowMsg;
};
```

At the end of `OnOK()`, we test `m_bShowMsg` to see if the check box was checked.

```
void CMainDlg::OnOK ( UINT uCode, int nID, HWND hWndCtl )
{
    // Transfer data from the controls to member variables.

    if ( !DoDataExchange(true) )
```

```
        return;
//...

    if ( m_bShowMsg )
        MessageBox ( _T("DDX complete!"), _T("ControlMania1"),
                     MB_ICONINFORMATION );
}
```

The sample project has examples of using the other DDX_* macros as well.

Handling Notifications from Controls

Handling notifications in WTL is similar to API-level programming. A control sends its parent a notification in the form of a WM_COMMAND or WM_NOTIFY message, and it's the parent's responsibility to handle it. A few other messages can be considered notifications, such as WM_DRAWITEM, which is sent when an owner-drawn control needs to be painted. The parent window can act on the message itself, or it can *reflect* the message back to the control. Reflection works as in MFC - the control is able to handle notifications itself, making the code self-contained and easier to move to other projects.

Handling notifications in the parent

Notifications sent as WM_NOTIFY and WM_COMMAND contain various information. The parameters in a WM_COMMAND message contain the ID of the control sending the message, the HWND of the control, and the notification code. WM_NOTIFY messages have all those as well as a pointer to an NMHDR data structure. ATL and WTL have various message map macros for handling these notifications. I'll only be covering the WTL macros here, since this is a WTL article after all. Note that for all of these macros, you need to use BEGIN_MSG_MAP_EX in your message map, and #include atlcrack.h in stdafx.h.

Message map macros

To handle a WM_COMMAND notification, use one of the COMMAND_HANDLER_EX macros:

COMMAND_HANDLER_EX(id, code, func)

Handles a notification from a particular control with a particular code.

COMMAND_CODE_HANDLER_EX(id, func)

Handles all notifications with a particular code, regardless of which control sends them.

COMMAND_ID_HANDLER_EX(code, func)

Handles all notifications from a particular control, regardless of the code.

COMMAND_RANGE_HANDLER_EX(idFirst, idLast, func)

Handles all notifications from controls whose IDs are in the range idFirst to idLast inclusive, regardless of the code.

COMMAND_RANGE_CODE_HANDLER_EX(idFirst, idLast, code, func)

Handles all notifications from controls whose IDs are in the range idFirst to idLast inclusive, with a particular code.

Examples:

- `COMMAND_HANDLER_EX(IDC_USERNAME, EN_CHANGE, OnUsernameChange)`: Handles `EN_CHANGE` when sent from the edit box with ID IDC_USERNAME.
- `COMMAND_ID_HANDLER_EX(IDOK, OnOK)`: Handles all notifications sent from the control with ID IDOK.
- `COMMAND_RANGE_CODE_HANDLER_EX(IDC_MONDAY, IDC_FRIDAY, BN_CLICKED, OnDayClicked)`: Handles all `BN_CLICKED` notifications from the controls with IDs in the range IDC_MONDAY to IDC_FRIDAY

There are also macros for handling `WM_NOTIFY` messages. They work just like the macros above, but their names start with "`NOTIFY_`" instead of "`COMMAND_`".

The prototype for a `WM_COMMAND` handler is:

```
void func ( UINT uCode, int nCtrlID, HWND hwndCtrl );
```

`WM_COMMAND` notifications do not use a return value, so the handlers return `void`. The prototype for a `WM_NOTIFY` handler is:

```
LRESULT func ( NMHDR* phdr );
```

The return value of the handler is used as the message result. This is different from MFC, where the handler receives an `LRESULT*` and sets the message result through that variable. The notification code and the `HWND` of the control that sent the notification are available in the `NMHDR` struct, as the `code` and `hwndFrom` members. Just as in MFC, if the notification sends a struct that is not a plain `NMHDR`, your handler should cast the `phdr` parameter to the correct type.

We'll add a notification handler to `CMainDlg` that handles `LVN_ITEMCHANGED` sent from the list control, and shows the currently-selected item in the dialog. We start by adding the message map macro and message handler:

```
class CMainDlg : public ...
{
    BEGIN_MSG_MAP_EX(CMainDlg)
        NOTIFY_HANDLER_EX(IDC_LIST, LVN_ITEMCHANGED, OnListItemchanged)
    END_MSG_MAP()

    LRESULT OnListItemchanged(NMHDR* phdr);
//...

};
```

Here's the message handler:

```
LRESULT CMainDlg::OnListItemchanged ( NMHDR* phdr )
{
NMLISTVIEW* pnmlv = (NMLISTVIEW*) phdr;
int nSelItem = m_wndList.GetSelectedIndex();
CString sMsg;

    // If no item is selected, show "none". Otherwise, show its index.

    if ( -1 == nSelItem )
        sMsg = _T("(none)");
    else
        sMsg.Format ( _T("%d"), nSelItem );

    SetDlgItemText ( IDC_SEL_ITEM, sMsg );
    return 0;   // retval ignored

}
```

This handler doesn't use the `phdr` parameter, but I included the cast to `NMLISTVIEW*` as a demonstration.

Reflecting Notifications

If you have a `CWindowImpl`-derived class that implements a control, like our `CEditImpl` from before, you can handle notifications in that class, instead of the parent dialog. This is called *reflecting* the notifications, and works similarly to MFC's message reflection. The difference is that both the parent and the control participate in the reflection, whereas in MFC only the control does.

When you want to reflect notifications back to the control classes, you just add one macro to the dialog's message map, `REFLECT_NOTIFICATIONS()`:

```
class CMainDlg : public ...
{
public:
    BEGIN_MSG_MAP_EX(CMainDlg)
        //...

        NOTIFY_HANDLER_EX(IDC_LIST, LVN_ITEMCHANGED, OnListItemchanged)
        REFLECT_NOTIFICATIONS()
    END_MSG_MAP()
};
```

That macro adds some code to the message map that handles any notification messages that are not handled by any earlier macros. The code examines the `HWND` of the message and sends the message to that window. The value of the message is changed, however, to a value used by OLE controls which have a similar message reflection system. The new value is called `OCM_xxx` instead of `WM_xxx`, but otherwise the message is handled just like non-reflected messages.

There are 18 messages which are reflected:

- Control notifications: `WM_COMMAND`, `WM_NOTIFY`, `WM_PARENTNOTIFY`
- Owner drawing: `WM_DRAWITEM`, `WM_MEASUREITEM`, `WM_COMPAREITEM`, `WM_DELETEITEM`
- List box keyboard messages: `WM_VKEYTOITEM`, `WM_CHARTOITEM`
- Others: `WM_HSCROLL`, `WM_VSCROLL`, `WM_CTLCOLOR*`

In the control class, you add handlers for the reflected messages you are interested in, then at the end, add `DEFAULT_REFLECTION_HANDLER()`. `DEFAULT_REFLECTION_HANDLER()` ensures that unhanded messages are properly routed on to `DefWindowProc()`. Here is a simple owner-drawn button class that handles the reflected `WM_DRAWITEM`.

```
class CODButtonImpl : public CWindowImpl<CODButtonImpl, CButton>
{
public:
    BEGIN_MSG_MAP_EX(CODButtonImpl)
        MSG_OCM_DRAWITEM(OnDrawItem)
        DEFAULT_REFLECTION_HANDLER()
    END_MSG_MAP()

    void OnDrawItem ( UINT idCtrl, LPDRAWITEMSTRUCT lpdis )
    {
        // do drawing here...

    }
};
```

WTL macros for handling reflected messages

We just saw one of the WTL macros for reflected messages, MSG_OCM_DRAWITEM. There are MSG_OCM_* macros for the other 17 messages that can be reflected as well. Since WM_NOTIFY and WM_COMMAND have parameters that need to be unpacked, WTL provides special macros for them in addition to MSG_OCM_COMMAND and MSG_OCM_NOTIFY. These macros work like COMMAND_HANDLER_EX and NOTIFY_HANDLER_EX, but have "REFLECTED_" prepended. For example, a tree control class could have this message map:

```
class CMyTreeCtrl : public CWindowImpl<CMyTreeCtrl, CTreeViewCtrl>
{
public:
 BEGIN_MSG_MAP_EX(CMyTreeCtrl)
  REFLECTED_NOTIFY_CODE_HANDLER_EX(TVN_ITEMEXPANDING, OnItemExpanding)
  DEFAULT_REFLECTION_HANDLER()
 END_MSG_MAP()

 LRESULT OnItemExpanding ( NMHDR* phdr );
};
```

If you check out the ControlMania1 dialog in the sample code, there is a tree control that handles TVN_ITEMEXPANDING as shown above. The CMainDlg member m_wndTree is connected to the tree control using DDX, and CMainDlg reflects notifications. The tree's OnItemExpanding() handler looks like this:

```
LRESULT CBuffyTreeCtrl::OnItemExpanding ( NMHDR* phdr )
{
NMTREEVIEW* pnmtv = (NMTREEVIEW*) phdr;

    if ( pnmtv->action & TVE_COLLAPSE )
        return TRUE;    // don't allow it

    else
        return FALSE;   // allow it

}
```

If you run ControlMania1 and click the +/- buttons in the tree, you'll see this handler in action - once you expand a node, it will not collapse again.

Odds & Ends

Dialog Fonts

If you're picky about UI like me, and you're using Win 2000 or XP, you might be wondering why the dialogs are using MS Sans Serif instead of Tahoma. Since VC 6 is so old, the resource files it generates work fine for NT 4, but not later versions of NT. You can fix this, but it requires hand-editing the resource file.

There are three things you need to change in each dialog's entry in the resource file

1. The dialog type: Change `DIALOG` to `DIALOGEX`.
2. The window styles: Add `DS_SHELLFONT`
3. The dialog font: Change *MS Sans Serif* to *MS Shell Dlg*

Unfortunately, the first two changes get lost whenever you modify and save the resources, so you'll need to make the changes repeatedly. Here's a "before" picture of a dialog:

```
IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 187, 102
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About"
FONT 8, "MS Sans Serif"
BEGIN
  ...
END
```

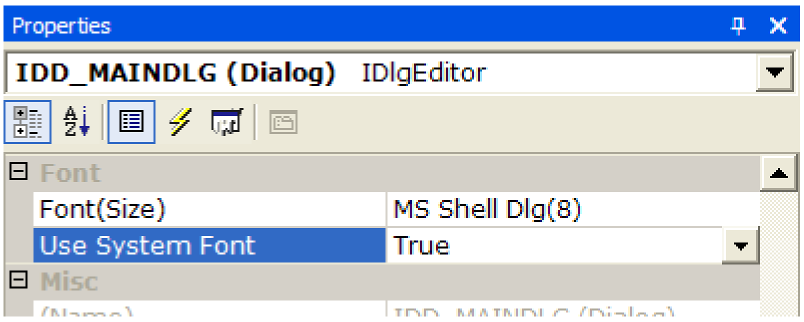
And the "after" picture:

```
IDD_ABOUTBOX DIALOGEX DISCARDABLE  0, 0, 187, 102
STYLE DS_SHELLFONT | DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About"
FONT 8, "MS Shell Dlg"
BEGIN
```

```
  ...
END
```

After making these changes, the dialog will use Tahoma on newer OSes, but still use MS Sans Serif when necessary on older OSes.

In VC 7, you just need to change one setting in the dialog editor to use the right font:


Properties

IDD_MAINDLG (Dialog) IDlgEditor

Font	
Font(Size)	MS Shell Dlg(8)
Use System Font	True
Misc	

When you change *Use System Font* to *True*, the editor will change the font to *MS Shell Dlg* for you.

_ATL_MIN_CRT

As explained in this VC Forum FAQ, ATL contains an optimization that lets you create an app that does not link to the C runtime library (CRT). This optimization is enabled by adding the `_ATL_MIN_CRT` symbol to the preprocessor settings. An AppWizard-generated app contains this symbol in the Release configurations. Since I've never written a non-trivial app that didn't need *something* from the CRT, I always remove that symbol. You must remove it, in any case, if you use floating-point features in `CString` or DDX.

Up Next

In Part V, we'll cover dialog data validation (DDV), the new controls in WTL, and some advanced UI features like owner draw and custom draw.

Part V - Advanced Dialog UI

Introduction to Part V

In the last article, we saw some of WTL's features relating to dialogs and controls that worked like the corresponding MFC classes. In this one, we'll cover some new WTL classes that implement some more advanced UI features: Owner draw and custom draw, new WTL controls, UI updating, and dialog data validation (DDV).

Specialized Owner Draw and Custom Draw Classes

Since owner drawn and custom drawn controls are so common in GUI work, WTL provides mix-in classes that handle some of the grunt work. We'll cover each of them next, as we start on the sequel to the last sample project, ControlMania2. If you are following along by creating a project with the AppWizard, be sure to make this dialog modeless. It has to be modeless in order for UI updating to work properly. I'll give more details on this in the section on UI updating.

COwnerDraw

Owner drawing involves handling up to four messages: `WM_MEASUREITEM`, `WM_DRAWITEM`, `WM_COMPAREITEM`, and `WM_DELETEITEM`. The `COwnerDraw` class, defined in atlframe.h, simplifies your code since you don't need messages handlers for all those messages. Instead, you chain messages to `COwnerDraw`, and it calls overridable functions that you implement in your class.

How you chain messages depends on whether you are reflecting messages to the control or not. Here is the `COwnerDraw` message map, which will make the distinction clear:

```
template <class T> class COwnerDraw
{
public:
  BEGIN_MSG_MAP(COwnerDraw<T>)
    MESSAGE_HANDLER(WM_DRAWITEM, OnDrawItem)
    MESSAGE_HANDLER(WM_MEASUREITEM, OnMeasureItem)
    MESSAGE_HANDLER(WM_COMPAREITEM, OnCompareItem)
    MESSAGE_HANDLER(WM_DELETEITEM, OnDeleteItem)
  ALT_MSG_MAP(1)
    MESSAGE_HANDLER(OCM_DRAWITEM, OnDrawItem)
    MESSAGE_HANDLER(OCM_MEASUREITEM, OnMeasureItem)
    MESSAGE_HANDLER(OCM_COMPAREITEM, OnCompareItem)
    MESSAGE_HANDLER(OCM_DELETEITEM, OnDeleteItem)
  END_MSG_MAP()
};
```

Notice that the main section of the map handles the `WM_*` messages, while the `ALT_MSG_MAP(1)` section handles the reflected versions, `OCM_*`. The owner draw notifications are like `WM_NOTIFY`, in that you can handle them in the control's parent, or reflect them back to the control. If you choose the former, you chain messages directly to `COwnerDraw`:

```
// C++ class for a dialog that contains owner-drawn controls

class CSomeDlg : public CDialogImpl<CSomeDlg>,
                 public COwnerDraw<CSomeDlg>, ...
{
  BEGIN_MSG_MAP(CSomeDlg)
    //...

    CHAIN_MSG_MAP(COwnerDraw<CSomeDlg>)
  END_MSG_MAP()

  void DrawItem ( LPDRAWITEMSTRUCT lpdis );
};
```

However, if you want the control to handle the messages, you need to chain to the `ALT_MSG_MAP(1)` section, using the `CHAIN_MSG_MAP_ALT` macro:

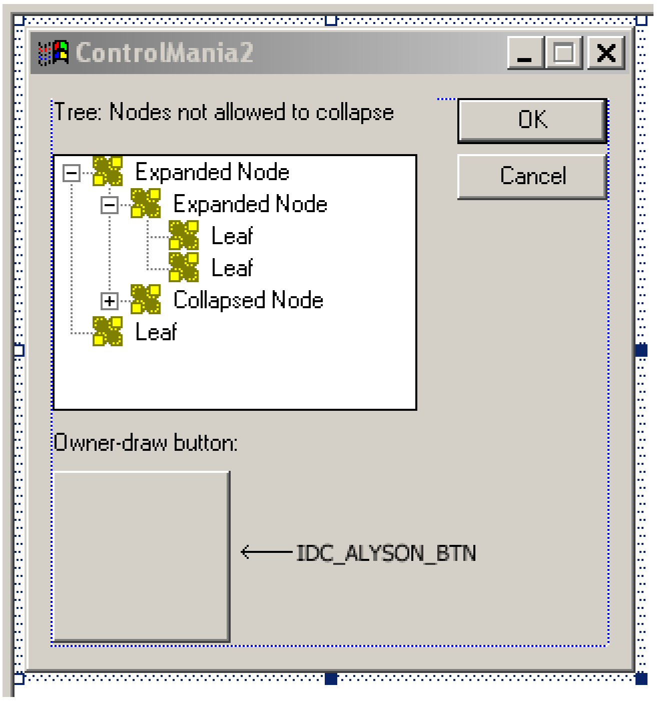
```
// C++ class that implements an owner-drawn button

class CMyButton : public CWindowImpl<CMyButton, CButton>,
                  public COwnerDraw<CMyButton>, ...
{
  BEGIN_MSG_MAP(CMyButton)
    //...

    CHAIN_MSG_MAP_ALT(COwnerDraw<CMyButton>, 1)
    DEFAULT_REFLECTION_HANDLER()
  END_MSG_MAP()

  void DrawItem ( LPDRAWITEMSTRUCT lpdis );
};
```

COwnerDraw unpacks the parameters sent with the message, then calls an implementation function in your class. In the example above, the classes implement DrawItem(), which is called when WM_DRAWITEM or OCM_DRAWITEM is chained to COwnerDraw. The methods you can override are:

```
void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
void MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct);
int  CompareItem(LPCOMPAREITEMSTRUCT lpCompareItemStruct);
void DeleteItem(LPDELETEITEMSTRUCT lpDeleteItemStruct);
```

If for some reason you don't want to handle a message in an override, you can call SetMsgHandled(false) and the message will be passed along to any other handlers that might be later in the message map.

For ControlMania2, we'll start with the tree control from ControlMania1, and add an owner-drawn button and handle the reflected WM_DRAWITEM in the button class. Here's the new button in the resource editor:



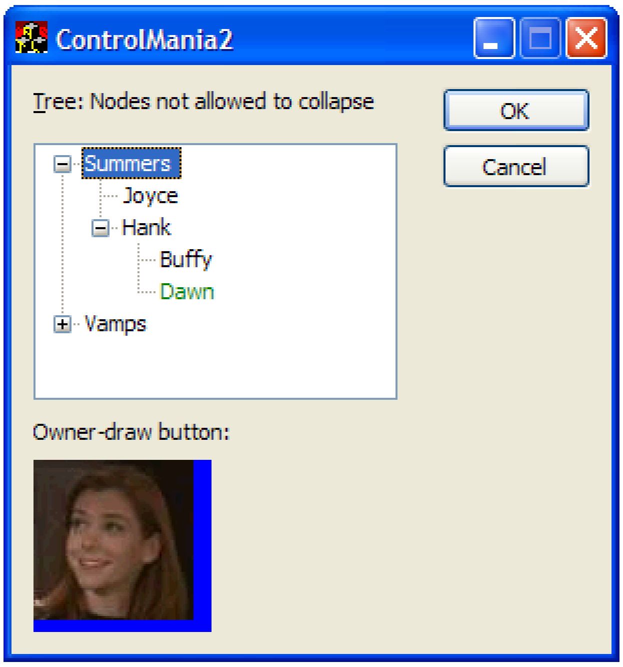
Now we need a class that implements this button:

```
class CODButtonImpl : public CWindowImpl<CODButtonImpl, CButton>,
                      public COwnerDraw<CODButtonImpl>
{
public:
    BEGIN_MSG_MAP_EX(CODButtonImpl)
        CHAIN_MSG_MAP_ALT(COwnerDraw<CODButtonImpl>, 1)
        DEFAULT_REFLECTION_HANDLER()
    END_MSG_MAP()

    void DrawItem ( LPDRAWITEMSTRUCT lpdis );
};
```

`DrawItem()` uses GDI calls like `BitBlt()` to draw a picture on the button face. This code should be easy to follow, since once again the WTL class names and methods are similar to MFC.

```
void CODButtonImpl::DrawItem ( LPDRAWITEMSTRUCT lpdis )
{
// NOTE: m_bmp is a CBitmap init'ed in the constructor.

CDCHandle dc = lpdis->hDC;
CDC dcMem;

    dcMem.CreateCompatibleDC ( dc );
    dc.SaveDC();
    dcMem.SaveDC();

    // Draw the button's background, red if it has the focus, blue if not.

    if ( lpdis->itemState & ODS_FOCUS )
        dc.FillSolidRect ( &lpdis->rcItem, RGB(255,0,0) );
    else
        dc.FillSolidRect ( &lpdis->rcItem, RGB(0,0,255) );

    // Draw the bitmap in the top-left, or offset by 1 pixel if the button

    // is clicked.

    dcMem.SelectBitmap ( m_bmp );

    if ( lpdis->itemState & ODS_SELECTED )
        dc.BitBlt ( 1, 1, 80, 80, dcMem, 0, 0, SRCCOPY );
    else
        dc.BitBlt ( 0, 0, 80, 80, dcMem, 0, 0, SRCCOPY );

    dcMem.RestoreDC(-1);
    dc.RestoreDC(-1);
}
```

Here's what the button looks like:



CCustomDraw

`CCustomDraw` works similarly to `COwnerDraw` in the way you handle `NM_CUSTOMDRAW` messages and chain them. `CCustomDraw` has an overridable method for each of the custom draw stages:

```
DWORD OnPrePaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD);
DWORD OnPostPaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD);
DWORD OnPreErase(int idCtrl, LPNMCUSTOMDRAW lpNMCD);
DWORD OnPostErase(int idCtrl, LPNMCUSTOMDRAW lpNMCD);

DWORD OnItemPrePaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD);
DWORD OnItemPostPaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD);
DWORD OnItemPreErase(int idCtrl, LPNMCUSTOMDRAW lpNMCD);
DWORD OnItemPostEraset(int idCtrl, LPNMCUSTOMDRAW lpNMCD);

DWORD OnSubItemPrePaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD);
```

The default handlers all return `CDRF_DODEFAULT`, so you only need to override a method if you need to do your own drawing or return a different value.

You might have noticed that in the last screen shot, "Dawn" was shown in green. That is achieved by using a new class (`CBuffyTreeCtrl` in the source) derived from `CTreeCtrl`, which chains messages to `CCustomDraw` and overrides

OnPrePaint() and OnItemPrePaint(). When the tree is filled, that item data for the "Dawn" node is set to 1, and OnItemPrePaint() checks for that value and changes the text color if it's found.

```
DWORD CBuffyTreeCtrl::OnPrePaint(
    int idCtrl, LPNMCUSTOMDRAW lpNMCD)
{
    return CDRF_NOTIFYITEMDRAW;
}

DWORD CBuffyTreeCtrl::OnItemPrePaint(
    int idCtrl, LPNMCUSTOMDRAW lpNMCD)
{
    if ( 1 == lpNMCD->lItemlParam )
        pnmtv->clrText = RGB(0,128,0);

    return CDRF_DODEFAULT;
}
```

Just as with COwnerDraw, you can call SetMsgHandled(false) in a custom draw message handler to have the message passed on to any other handlers in the message map.

New WTL Controls

WTL has a few new controls of its own that either improve on other wrappers (like CTreeViewCtrlEx), or provide new functionality that isn't in the built-in controls (like CHyperLink).

CBitmapButton

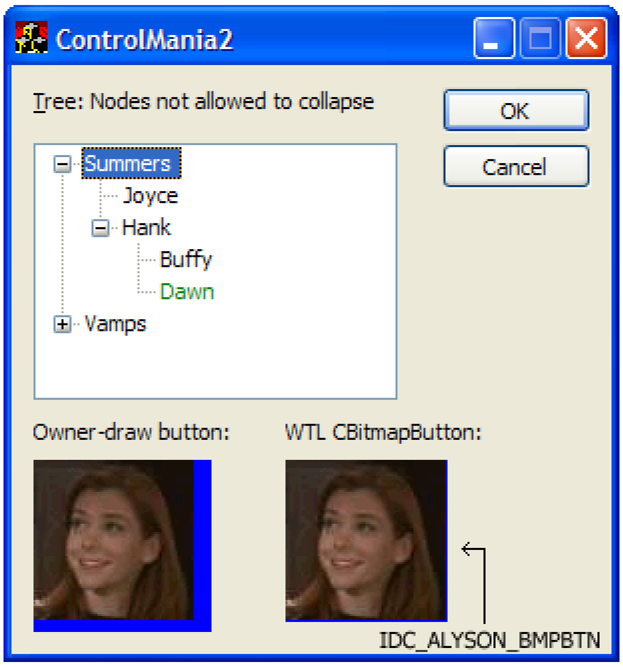
WTL's CBitmapButton, declared in atlctrlx.h, is rather easier to use than the MFC version. The WTL class uses an image list instead of four separate bitmap resources, which means you can keep multiple button images in one bitmap and reduce your GDI usage a bit. This is especially nice if you have a lot of graphics and your app runs on Windows 9x, because using lots of separate graphics can quickly exhaust GDI resources and bring down the system.

CBitmapButton is a CWindowImpl-derived class that has many features including: automatic sizing of the control, automatic generation of a 3D border, hot-tracking support, and several images per button for the various states the control can be in.

In ControlMania2, we'll use a CBitmapButton along side the owner-draw button we created earlier. We start by adding a CBitmapButton member called m_wndBmpBtn to CMainDlg. We then connect it to the new button in the usual way, either by calling SubclassWindow() or using DDX. We load a bitmap into an image list, then tell the button to use that image list. We also tell the button which image in the image list corresponds to which control state. Here's the section from OnInitDialog() that sets up the button:

```
    // Set up the bitmap button

CImageList iml;

    iml.CreateFromImage ( IDB_ALYSON_IMGLIST, 81, 1, CLR_NONE,
                          IMAGE_BITMAP, LR_CREATEDIBSECTION );

    m_wndBmpBtn.SubclassWindow ( GetDlgItem(IDC_ALYSON_BMPBTN) );
    m_wndBmpBtn.SetToolTipText ( _T("Alyson") );
    m_wndBmpBtn.SetImageList ( iml );
    m_wndBmpBtn.SetImages ( 0, 1, 2, 3 );
```

By default, the button assumes ownership of the image list, so OnInitDialog() must not delete the image list it creates. Here is the new button in its default state. Notice how the control is resized to exactly fit the size of the image.



Since CBitmapButton is a very useful class, I'll cover its public methods here.

CBitmapButton methods

The class CBitmapButtonImpl contains all the code to implement a button, but unless you need to override a method or message handler, you can use CBitmapButton for your controls.

```
CBitmapButtonImpl(DWORD dwExtendedStyle = BMPBTN_AUTOSIZE,
               HIMAGELIST hImageList = NULL)
```

The constructor sets up the button extended style (not to be confused with its window styles) and can assign an image list. Usually the defaults are sufficient, since you can set both attributes with other methods.

```
BOOL SubclassWindow(HWND hWnd)
```

SubclassWindow() is overridden to perform the subclassing and initialize internal data that the class keeps.

```
DWORD GetBitmapButtonExtendedStyle()
DWORD SetBitmapButtonExtendedStyle(DWORD dwExtendedStyle,
                             DWORD dwMask = 0)
```

CBitmapButton supports some extended styles that affect the appearance or operation of the button:

BMPBTN_HOVER

Enables hot-tracking. When the cursor is over the button, it will be drawn in the focused state.

BMPBTN_AUTO3D_SINGLE, BMPBTN_AUTO3D_DOUBLE

Automatically generates a 3D border around the image, as well as a focus rectangle when the button has the focus. In addition, if you do not provide an image for the pressed state, one is generated for you. BMPBTN_AUTO3D_DOUBLE produces a slightly thicker border.

BMPBTN_AUTOSIZE

Makes the button resize itself to match the size of the image. This style is the default.

BMPBTN_SHAREIMAGELISTS

If set, the button object does not destroy the image list used to hold the button images. If not set, the image list is destroyed by the CBitmapButton destructor.

BMPBTN_AUTOFIRE

If set, clicking the button and holding down the mouse button generates repeated WM_COMMAND messages.

When calling SetBitmapButtonExtendedStyle(), the dwMask parameter controls which styles are affected. Use the default of 0 to have the new styles completely replace the old ones.

```
HIMAGELIST GetImageList()
HIMAGELIST SetImageList(HIMAGELIST hImageList)
```

Use GetImageList() and SetImageList() to associate an image list with the button, or get the image list currently associated with the button.

```
int  GetToolTipTextLength()
bool GetToolTipText(LPTSTR lpstrText, int nLength)
bool SetToolTipText(LPCTSTR lpstrText)
```

CBitmapButton supports showing a tooltip when the mouse hovers over the button. Call GetToolTipText() and SetToolTipText() to get or set the text to show in the tooltip.

```
void SetImages(int nNormal, int nPushed = -1,
               int nFocusOrHover = -1, int nDisabled = -1)
```

Call SetImages() to tell the button which image in the image list to use for which button state. The parameters are all 0-based indexes into the image list. nNormal is required, but the others are optional. Passing -1 indicates that there is no image for the corresponding state.

CCheckListViewCtrl

CCheckListViewCtrl, defined in atlctrlx.h, is a CWindowImpl-derived class that implements a list view control containing check boxes. This is different from MFC's CCheckListBox, which uses a list box, not a list view. CCheckListViewCtrl is quite simple, since the class adds minimal functionality on its own. However, it does introduce a new helper class, CCheckListViewCtrlImplTraits, that is like CWinTraits but with a third template parameter that is the extended list view styles to use for the control. If you don't define your own set of CCheckListViewCtrlImplTraits, the class uses these styles by default: LVS_EX_CHECKBOXES | LVS_EX_FULLROWSELECT.

Here is a sample traits definition that uses different extended list view styles, plus a new class that uses those traits. (Note that you must include LVS_EX_CHECKBOXES in the extended list view styles, or else you will get an assert failed message.)

```
typedef CCheckListViewCtrlImplTraits<
    WS_CHILD | WS_VISIBLE | LVS_REPORT,
    WS_EX_CLIENTEDGE,
    LVS_EX_CHECKBOXES | LVS_EX_GRIDLINES | LVS_EX_UNDERLINEHOT |
      LVS_EX_ONECLICKACTIVATE> CMyCheckListTraits;

class CMyCheckListCtrl :
    public CCheckListViewCtrlImpl<CMyCheckListCtrl, CListViewCtrl,
                                  CMyCheckListTraits>
{
private:
    typedef CCheckListViewCtrlImpl<CMyCheckListCtrl, CListViewCtrl,
                                   CMyCheckListTraits> baseClass;
public:
    BEGIN_MSG_MAP(CMyCheckListCtrl)
        CHAIN_MSG_MAP(baseClass)
    END_MSG_MAP()
};
```

CCheckListViewCtrl methods

```
BOOL SubclassWindow(HWND hWnd)
```

When you subclass an existing list view control, SubclassWindow() looks at the extended list view styles in the associated CCheckListViewCtrlImplTraits class and applies them to the control. The first two template parameters of the traits class (windows styles and extended window styles) are not used.

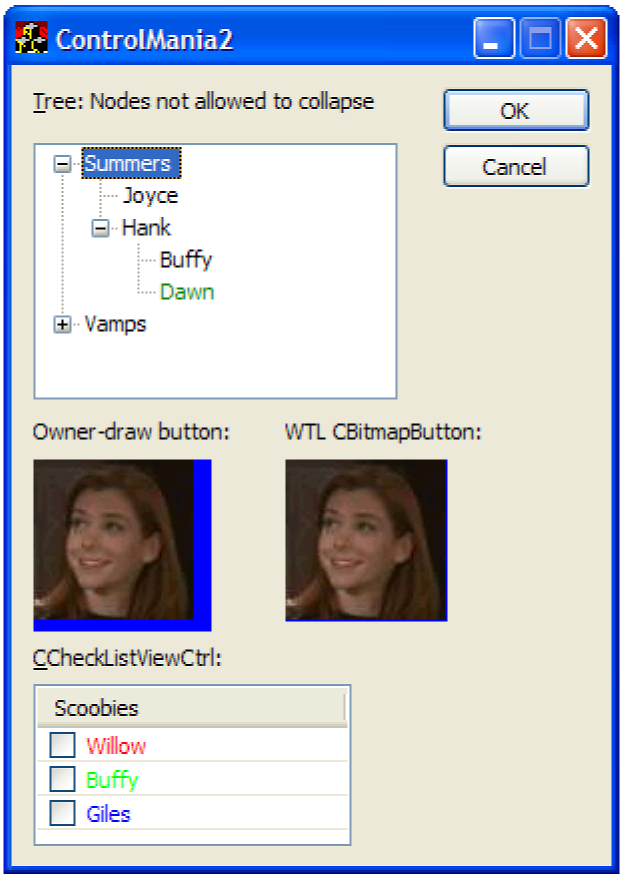
```
BOOL GetCheckState(int nIndex)
BOOL SetCheckState(int nItem, BOOL bCheck)
```

These methods are actually in CListViewCtrl. SetCheckState() takes an item index and a boolean indicating whether to check or uncheck that item. GetCheckState() takes just an index and returns the current checked state of that item.

```
void CheckSelectedItems(int nCurrItem)
```

This method takes an item index. It toggles the check state of that item, which must be selected, and changes the check state of all other selected items to match. You probably won't use this method yourself, since `CCheckListViewCtrl` handles checking items when the check box is clicked or the user presses the space bar.

Here's how a `CCheckListViewCtrl` looks in ControlMania2:



CTreeViewCtrlEx and CTreeItem

These two classes make it easier to use tree control features by wrapping an `HTREEITEM`. A `CTreeItem` object keeps an `HTREEITEM` and a pointer to the tree control that contains the item. You can then perform operations on that item using just `CTreeItem`; you don't have to refer to the tree control in every call. `CTreeViewCtrlEx` is like `CTreeViewCtrl`, however its methods deal with `CTreeItem`s instead of `HTREEITEM`s. So for example, when you call `InsertItem()`, it returns a `CTreeItem` instead of an `HTREEITEM`. You can then operate on the newly-inserted item using the `CTreeItem`. Here's an example:

```
// Using plain HTREEITEMs:

HTREEITEM hti, hti2;
```

```
    hti = m_wndTree.InsertItem ( "foo", TVI_ROOT, TVI_LAST );
    hti2 = m_wndTree.InsertItem ( "bar", hti, TVI_LAST );
    m_wndTree.SetItemData ( hti2, 37 );

// Using CTreeItems:

CTreeItem ti, ti2;

    ti = m_wndTreeEx.InsertItem ( "baz", TVI_ROOT, TVI_LAST );
    ti2 = ti.AddTail ( "yen", 0 );
    ti2.SetData ( 42 );
```

`CTreeItem` has a method corresponding to every `CTreeViewCtrl` method that takes an `HTREEITEM`, just like `CWindow` contains methods corresponding to APIs that take an `HWND`. Check out the ControlMania2 code, which demonstrates more methods of `CTreeViewCtrlEx` and `CTreeItem`.

CHyperLink

`CHyperLink` is a `CWindowImpl`-derived class that can subclass a static text control and make it into a clickable hyperlink. `CHyperLink` automatically handles drawing the link (following the user's IE color preferences) and also supports keyboard navigation. The class `CHyperLinkImpl` is the base class for `CHyperLink` and contains all the code to implement a link, but unless you need to override a method or message handler, you can stick with `CHyperLink` for your controls.

The default behavior of a `CHyperLink` control is to launch a URL in your default browser when the link is clicked. If the subclassed static control has the `WS_TABSTOP` style, you can also tab to the control and press the spacebar or Enter key to click the link. `CHyperLink` will also show a tooltip when the cursor hovers over the link. By default, `CHyperLink` uses the static control's text as the default for both the URL and the tooltip text, but you can change those properties using method calls as explained below.

In WTL 7.1, many features were added to `CHyperLink`; these new features are enabled using extended styles. The styles and their usage is explained after the method list.

CHyperLink methods

These are the `CHyperLink` methods that you'll commonly use. There are others for calculating the control size, parsing the link text, and so on; you can check out the class in atlctrlx.h to see the full list.

```
CHyperLinkImpl ( DWORD dwExtendedStyle = HLINK_UNDERLINED )
CHyperLink()
```

The CHyperLinkImpl constructor takes the extended styles to apply to the control. CHyperLink is missing a matching constructor, but you can use SetHyperLinkExtendedStyle() to set those styles.

```
BOOL SubclassWindow(HWND hWnd)
```

SubclassWindow() is overridden to perform the subclassing, then initialize internal data that the class keeps. This is called for you automatically if you associate a hyperlink variable with a static control via DDX_CONTROL, or you can call it yourself to subclass a control manually.

```
DWORD GetHyperLinkExtendedStyle()
DWORD SetHyperLinkExtendedStyle(DWORD dwExtendedStyle, DWORD dwMask = 0)
```

Gets or sets the extended styles for the control. You must set the extended style before calling SubclassWindow() or Create() so the control knows how to paint the text.

```
bool GetLabel(LPTSTR lpstrBuffer, int nLength)
bool SetLabel(LPCTSTR lpstrLabel)
```

Gets or sets the text to use in the control. If you do not set the label text, the label is set to the static control's window text.

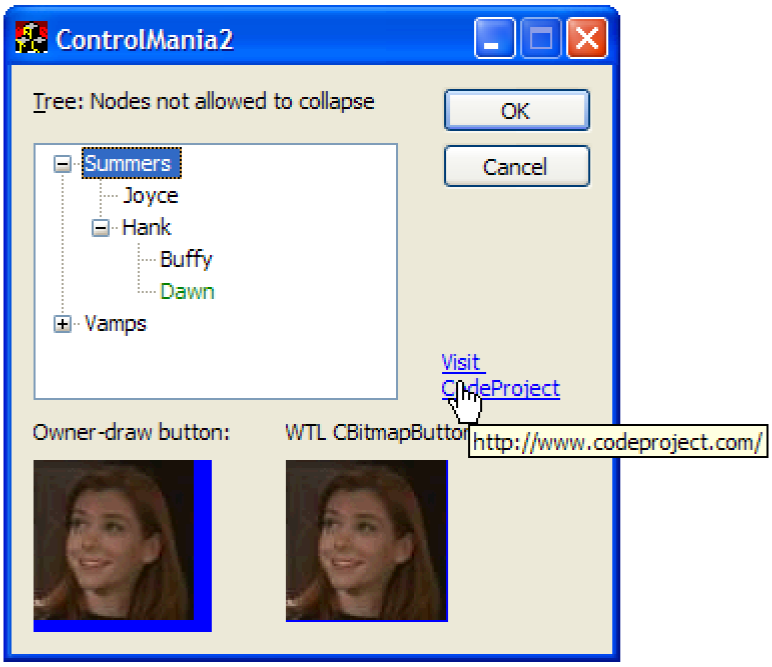
```
bool GetHyperLink(LPTSTR lpstrBuffer, int nLength)
bool SetHyperLink(LPCTSTR lpstrLink)
```

Gets or sets the URL associated with the control. If you do not set the hyperlink, the hyperlink is set to the static control's window text.

```
bool GetToolTipText(LPTSTR lpstrBuffer, int nLength)
bool SetToolTipText(LPCTSTR lpstrToolTipText)
```

Gets or sets the text displayed in a tooltip when the cursor hovers over the link. However, these methods can only be used with links that have the HLINK_COMMANDBUTTON or HLINK_NOTIFYBUTTON extended styles. See below for more information on the tooltip.

Here is how a "plain" hyperlink control looks in the ControlMania2 dialog:



The URL is set with this call in `OnInitDialog()`:

```
m_wndLink.SetHyperLink ( _T("http://www.codeproject.com/") );
```

CHyperLink extended styles

The new WTL 7.1 features are enabled by setting the appropriate extended style bits. The styles are:

HLINK_UNDERLINED

The link text will be underlined. This is the default behavior.

HLINK_NOTUNDERLINED

The link text will never be underlined.

HLINK_UNDERLINEHOVER

The link text will be underlined only when the cursor is over the link.

HLINK_COMMANDBUTTON

When the link is clicked, the control sends a `WM_COMMAND` message (with the notification code set to `BN_CLICKED`) to the control's parent window.

HLINK_NOTIFYBUTTON

When the link is clicked, the control sends a `WM_NOTIFY` message (with the notification code set to `NM_CLICK`) to the control's parent window.

HLINK_USETAGS

The control considers only text inside an <a> tag to be the link, other text is drawn normally.

HLINK_USETAGSBOLD

Same as HLINK_USETAGS, but the text inside the <a> tag is drawn bold. When this style is set, the underlining extended styles are ignored, and the link text is never underlined.

HLINK_NOTOOLTIP

The control will not display a tooltip.

If neither the HLINK_COMMANDBUTTON nor HLINK_NOTIFYBUTTON style is set, then the CHyperLink object calls its Navigate() method when clicked. Navigate() calls ShellExecuteEx() to launch a URL in the default browser. If you want to perform some other action when the link is clicked, set HLINK_COMMANDBUTTON or HLINK_NOTIFYBUTTON and then handle the notification message that the control sends.

Other CHyperLink details

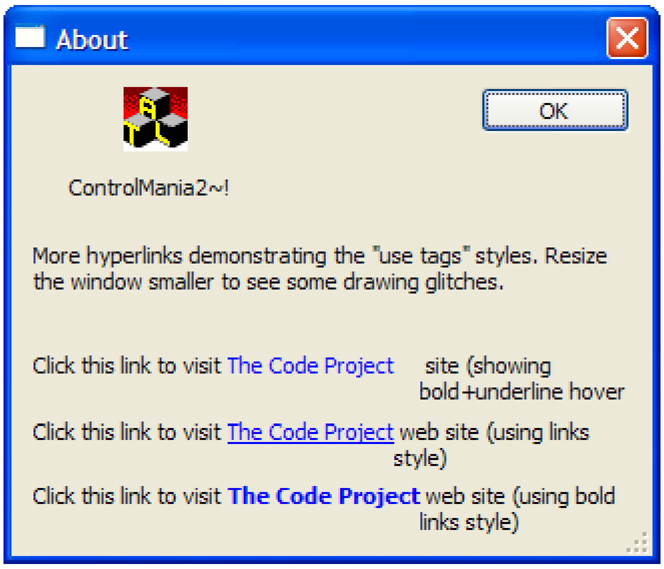
You can set the SS_CENTER or SS_RIGHT style on a static control to have the hyperlink text center- or right-aligned. However, if the control has the HLINK_USETAGS or HLINK_USETAGSBOLD style, those bits are ignored and the text is always left-aligned.

If you're using a CHyperLink to open a URL (that is, you haven't set HLINK_COMMANDBUTTON or HLINK_NOTIFYBUTTON), you cannot change the tooltip text with SetToolTipText(). However, you can access the tooltip control directly through the CHyperLink member m_tip and set the text using AddTool():

```
  m_wndLink.m_tip.AddTool ( m_wndLink, _T("Clickety!"), &m_wndLink.m_rcLink,
1 );
```

Note that there is a breaking change here from WTL 7.0: CHyperLink uses a tool ID of 1 in WTL 7.1. In WTL 7.0, the ID was the same as the window handle, and you could change the text using m_tip.UpdateTipText(). I didn't have any luck using UpdateTipText() in WTL 7.1; the above code duplicates what CHyperLink::Init() does to set up the tooltip initially.

Due to some painting problems, the HLINK_USETAGS and HLINK_USETAGSBOLD styles are best used when the link text is always going to be on one line. The painting code finds the text within <a> tags and splits the text into three parts: before the tag, within the tag, after the tag. However, if the text for one part requires word-breaking, it will wrap incorrectly. I have illustrated this in ControlMania2 in a separate dialog:



ControlMania2~!

More hyperlinks demonstrating the "use tags" styles. Resize the window smaller to see some drawing glitches.

Click this link to visit The Code Project site (showing bold+underline hover

Click this link to visit The Code Project web site (using links style)

Click this link to visit **The Code Project** web site (using bold links style)

You should also make sure that `HLINK_UNDERLINEHOVER` is not set along with `HLINK_USETAGSBOLD`, since that will cause some empty space to appear after the link text, as shown in the first hyperlink above.

UI Updating Dialog Controls

UI updating controls in a dialog is much easier than in MFC. In MFC, you have to know about the undocumented `WM_KICKIDLE` message and how to handle it and trigger control updating. In WTL, there are no such tricks, although there is a bug in the AppWizard that requires you to add one line of code.

The first thing to remember is that the dialog **must** be modeless. This is necessary because for `CUpdateUI` to do its job, your app needs to be in control of the message loop. If you make the dialog modal, the system handles the message loop, so idle handlers won't get called. Since `CUpdateUI` does its work at idle time, no idle processing means no UI updating.

ControlMania2's dialog is modeless, and the first part of the class definition resembles a frame window class:

```
class CMainDlg : public CDialogImpl<CMainDlg>, public CUpdateUI<CMainDlg>,
                 public CMessageFilter, public CIdleHandler
{
public:
    enum { IDD = IDD_MAINDLG };
    BOOL PreTranslateMessage(MSG* pMsg);
    BOOL OnIdle();

    BEGIN_MSG_MAP_EX(CMainDlg)
        MSG_WM_INITDIALOG(OnInitDialog)
        COMMAND_ID_HANDLER_EX(IDOK, OnOK)
        COMMAND_ID_HANDLER_EX(IDCANCEL, OnCancel)
        COMMAND_ID_HANDLER_EX(IDC_ALYSON_BTN, OnAlysonODBtn)
    END_MSG_MAP()

    BEGIN_UPDATE_UI_MAP(CMainDlg)
    END_UPDATE_UI_MAP()
//...

};
```

Notice that CMainDlg derives from CUpdateUI and has an update UI map. OnInitDialog() has this code, which should be familiar from the earlier frame window examples:

```
    // register object for message filtering and idle updates

    CMessageLoop* pLoop = _Module.GetMessageLoop();
    ATLASSERT(pLoop != NULL);
    pLoop->AddMessageFilter(this);
    pLoop->AddIdleHandler(this);

    UIAddChildWindowContainer(m_hWnd);
```

This time, instead of UIAddToolbar() or UIAddStatusBar(), we call UIAddChildWindowContainer(). This tells CUpdateUI that our dialog contains child windows that will need updating. If you look at OnIdle(), you might suspect that something is missing:

```
BOOL CMainDlg::OnIdle()
{
    return FALSE;
}
```

You might expect there to be another CUpdateUI method call here to do the actual updating, and you're right, there should be; the AppWizard left out a line of code. You need to add this line to OnIdle():

```
BOOL CMainDlg::OnIdle()
{
    UIUpdateChildWindows();
    return FALSE;
}
```

To demonstrate UI updating, when you click the left-hand bitmap button, the right-hand button is enabled or disabled. So first, we add an entry to the update UI map, using the flag `UPDUI_CHILDWINDOW` to indicate that the entry is for a child window:

```
    BEGIN_UPDATE_UI_MAP(CMainDlg)
        UPDATE_ELEMENT(IDC_ALYSON_BMPBTN, UPDUI_CHILDWINDOW)
    END_UPDATE_UI_MAP()
```

Then in the handler for the left button, we call `UIEnable()` to toggle the enabled state of the other button:

```
void CMainDlg::OnAlysonODBtn ( UINT uCode, int nID, HWND hwndCtrl )
{
    UIEnable ( IDC_ALYSON_BMPBTN, !m_wndBmpBtn.IsWindowEnabled() );
}
```

DDV

WTL's dialog data validation (DDV) support is a bit simpler than MFC's. In MFC, you need create separate macros for DDX (to transfer the data to variables) and DDV (to validate the data). In WTL, one macro does both at the same time. WTL contains basic DDV support using the following macros in the DDX map:

DDX_TEXT_LEN

Does DDX like `DDX_TEXT`, and verifies that the string's length (not counting the null terminator) is less than or equal to a specified limit.

DDX_INT_RANGE and DDX_UINT_RANGE

These do DDX like `DDX_INT` and `DDX_UINT`, plus they verify that the number is between a given minimum and maximum.

DDX_FLOAT_RANGE

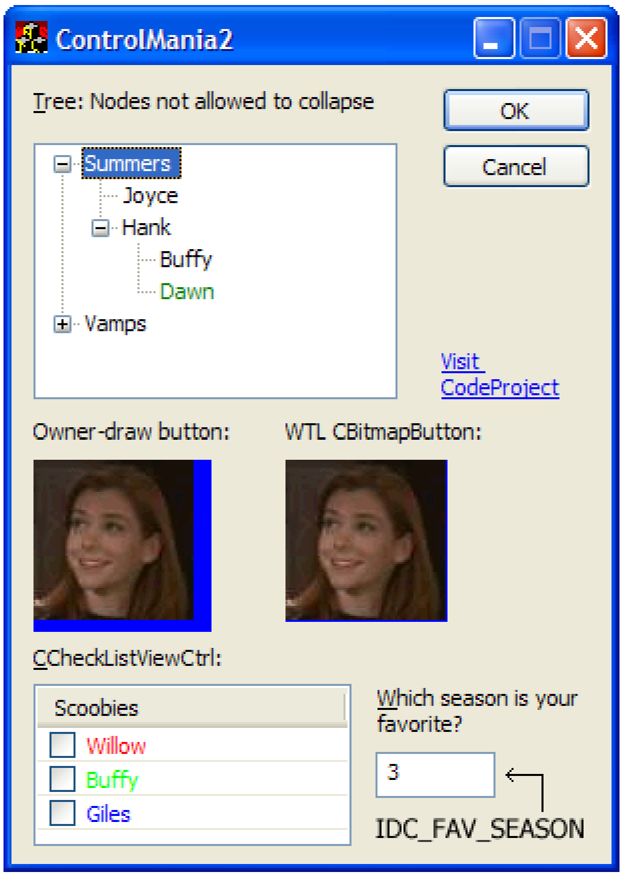
Does DDX like `DDX_FLOAT` and verifies that the number is between a given minimum and maximum.

DDX_FLOAT_P_RANGE (new in WTL 7.1)

Does DDX like DDX_FLOAT_P and verifies that the number is between a given minimum and maximum.

The parameters for these macros are like the corresponding non-validating macros, with an additional one or two parameters indicating the acceptable range. DDX_TEXT_LEN takes one parameter, the maximum allowed length. The others take two additional parameters, indicating the minimum and maximum allowable values.

ControlMania2 has an edit box with ID IDC_FAV_SEASON that is tied to the member variable m_nSeason.



There were seven seasons of *Buffy*, so the legal values for the season are 1 to 7, and the DDV macro looks like:

```
BEGIN_DDX_MAP(CMainDlg)
//...

    DDX_INT_RANGE(IDC_FAV_SEASON, m_nSeason, 1, 7)
END_DDX_MAP()
```

OnOK() calls DoDataExchange() to validate the season number. m_nSeason is filled in as part of the work done in DoDataExchange().

Handling DDV failures

If a control's data fails validation, CWinDataExchange calls the overridable function OnDataValidateError() and DoDataExchange() returns false. The default implementation of OnDataValidateError() just beeps the speaker, so you'll probably want to provide a friendlier indication of the error. The prototype of OnDataValidateError() is:

```
void OnDataValidateError ( UINT nCtrlID, BOOL bSave, _XData& data );
```

_XData is a struct that CWinDataExchange fills in with details about the data that was entered and the allowable range. Here's the definition of the struct:

```
struct _XData
{
    _XDataType nDataType;
    union
    {
        _XTextData textData;
        _XIntData intData;
        _XFloatData floatData;
    };
};
```

nDataType indicates which of the three members of the union is meaningful. Its possible values are:

```
enum _XDataType
{
    ddxDataNull = 0,
    ddxDataText = 1,
    ddxDataInt = 2,
    ddxDataFloat = 3,
    ddxDataDouble = 4
};
```

In our case, nDataType will be ddxDataInt, which means that the _XIntData member in _XData is filled in. _XIntData is a simple struct:

```
struct _XIntData
{
    long nVal;
    long nMin;
    long nMax;
};
```

Our OnDataValidateError() override shows an error message telling the user what the allowable range is:

```
void CMainDlg::OnDataValidateError ( UINT nCtrlID, BOOL bSave, _XData& data )
{
CString sMsg;

    sMsg.Format ( _T("Enter a number between %d and %d"),
                  data.intData.nMin, data.intData.nMax );

    MessageBox ( sMsg, _T("ControlMania2"), MB_ICONEXCLAMATION );

    GotoDlgCtrl ( GetDlgItem(nCtrlID) );
}
```

Check out atlddx.h to see the other types of data in an _XData struct - _XTextData and _XFloatData.

Resizing Dialogs

One of the first things about WTL that got my attention was its built-in support for resizable dialogs. Some time ago, I wrote an article on this subject, so please refer to that article for more details. To summarize, you add the CDialogResize class to the dialog's inheritance list, call DlgResize_Init() in OnInitDialog(), then chain messages to CDialogResize.

Up Next

In the next article, we'll look at hosting ActiveX controls in dialogs, and how to handle events fired by the controls.

References

Using WTL's Built-in Dialog Resizing Class - Michael Dunn

Using DDX and DDV with WTL - Less Wright

Part VI - Hosting ActiveX Controls

Introduction

Here in Part VI, I'll cover ATL's support for hosting ActiveX controls in dialogs. Since ActiveX controls are ATL's specialty, there are no additional WTL classes involved. However, the ATL way of hosting is rather different from the MFC way, so this is an important topic to cover. I will cover how to host controls and sink events, and develop an application that loses no functionality compared to an MFC app written with ClassWizard. You can, naturally, use the ATL hosting support in the WTL apps that you write.

The sample project for this article demonstrates how to host the IE WebBrowser control. I chose the browser control for two good reasons:

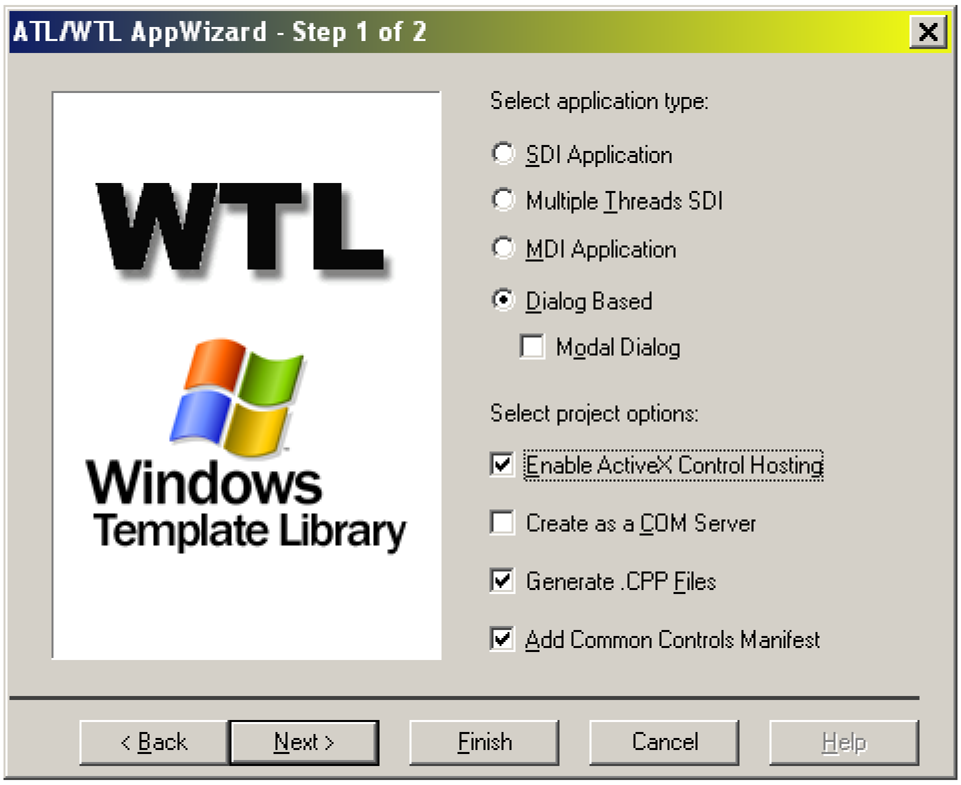
1. Everybody has it on their computers, and
2. It has many methods and fires many events, so it's a good control to use for demonstration purposes.

I certainly can't compete with folks who have spent tons of time writing custom browsers around the WebBrowser control. However, after reading through this article, you'll know enough to start working on a custom browser of your own.

Starting with the AppWizard

Creating the project

The WTL AppWizard can create an app for us that is ready to host ActiveX controls. We'll start with a new project, this one will be called *IEHost*. We'll use a modeless dialog as in the last article, only this time check the *Enable ActiveX Control Hosting* checkbox as shown here:



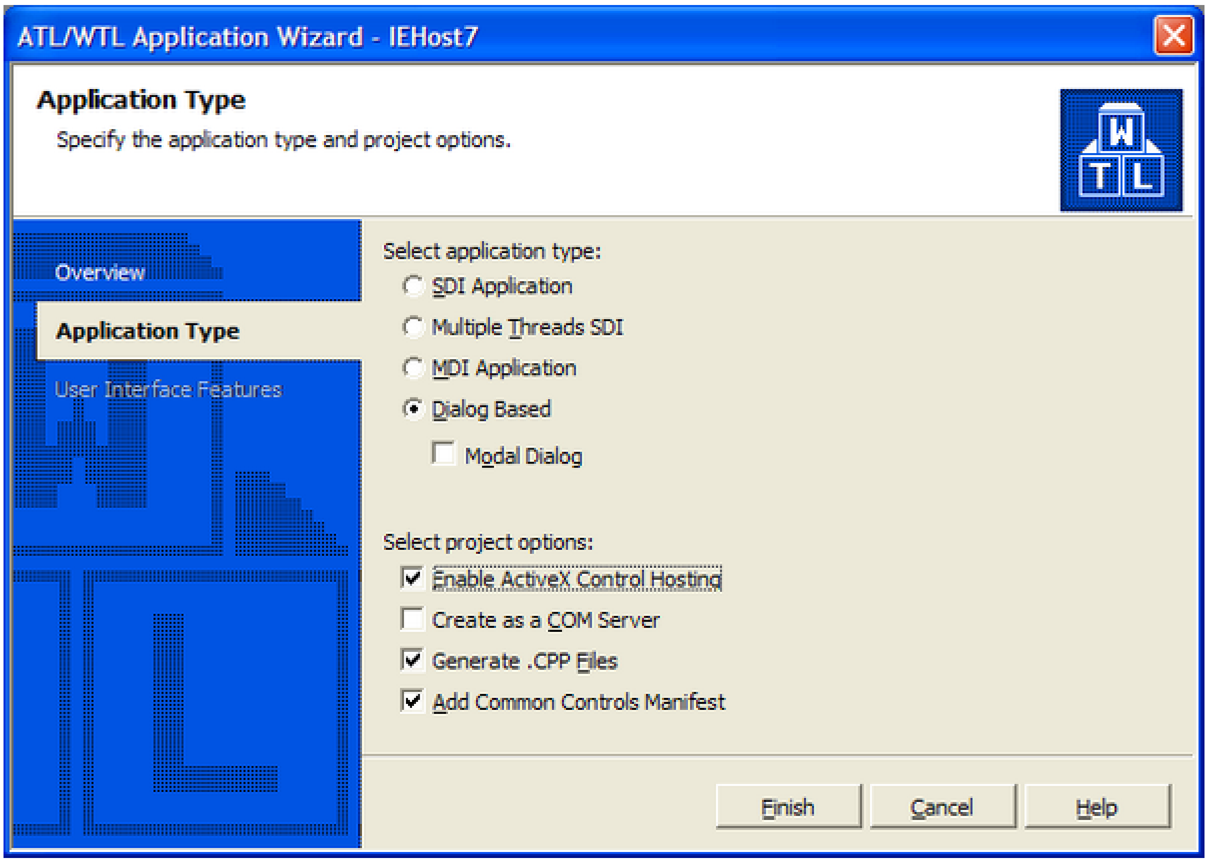
ATL/WTL AppWizard - Step 1 of 2

Select application type:

( ) SDI Application
( ) Multiple Threads SDI
( ) MDI Application
(●) Dialog Based
[ ] Modal Dialog

Select project options:

[✓] Enable ActiveX Control Hosting
[ ] Create as a COM Server
[✓] Generate .CPP Files
[✓] Add Common Controls Manifest



ATL/WTL Application Wizard - IEHost7

Application Type

Specify the application type and project options.

Select application type:

( ) SDI Application
( ) Multiple Threads SDI
( ) MDI Application
(●) Dialog Based
[ ] Modal Dialog

Select project options:

[✓] Enable ActiveX Control Hosting
[ ] Create as a COM Server
[✓] Generate .CPP Files
[✓] Add Common Controls Manifest

Checking that box makes our main dialog derive from `CAxDialogImpl` so it can host ActiveX controls. In the VC 6 wizard, there is another checkbox on page 2 labeled *Host ActiveX Controls*, however checking that seems to have no effect on the resulting code, so you can click Finish from page 1.

The generated code

In this section, I'll cover some new pieces of code that we haven't seen before from the AppWizard. In the next section, I'll cover the ActiveX hosting classes in detail.

The first file to check out is *stdafx.h*, which has these includes:

```
#include <atlbase.h>

#include <atlapp.h>


extern CAppModule _Module;

#include <atlcom.h>

#include <atlhost.h>

#include <atlwin.h>

#include <atlctl.h>

// .. other WTL headers ...
```

atlcom.h and *atlhost.h* are the important ones. They contain the definitions of some COM-related classes (like the smart pointer `CComPtr`), and the window class used to host controls.

Next, look at the declaration of `CMainDlg` in *maindlg.h*:

```
class CMainDlg : public CAxDialogImpl<CMainDlg>,
              public CUpdateUI<CMainDlg>,
              public CMessageFilter, public CIdleHandler
```

`CMainDlg` is now derived from `CAxDialogImpl`, which is the first step in enabling the dialog to host ActiveX controls.

Finally, there's one new line in `WinMain()`:

```
int WINAPI _tWinMain(...)
```

```
{
//...

   _Module.Init(NULL, hInstance);

   AtlAxWinInit();

   int nRet = Run(lpstrCmdLine, nCmdShow);

   _Module.Term();
   return nRet;
}
```

`AtlAxWinInit()` registers a window class called `AtlAxWin`. This is used by ATL when it creates the container window for an ActiveX control.

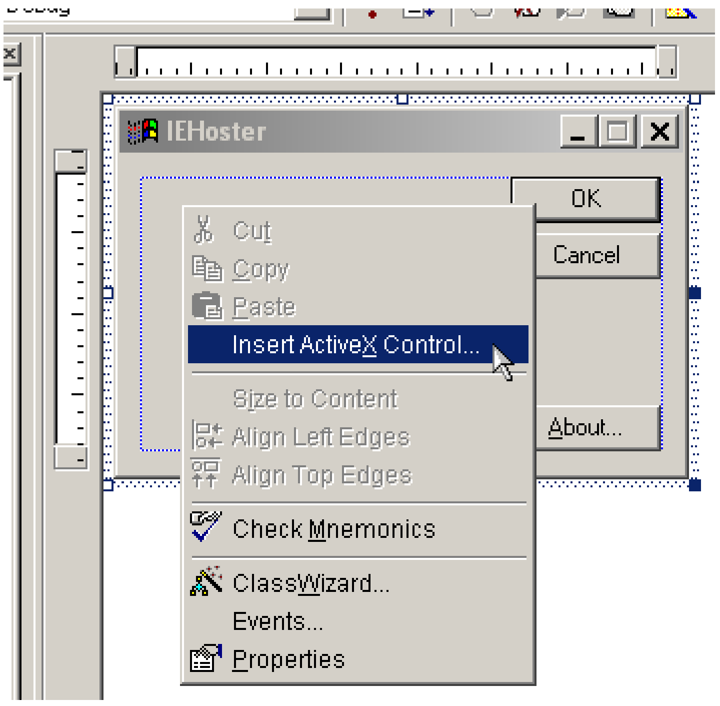
Due to a change in ATL 7, you have to pass a LIBID to `_Module.Init()`. Some folks on the forum suggested using this code on VC 7:

```
   _Module.Init(NULL, hInstance, &LIBID_ATLLib);
```

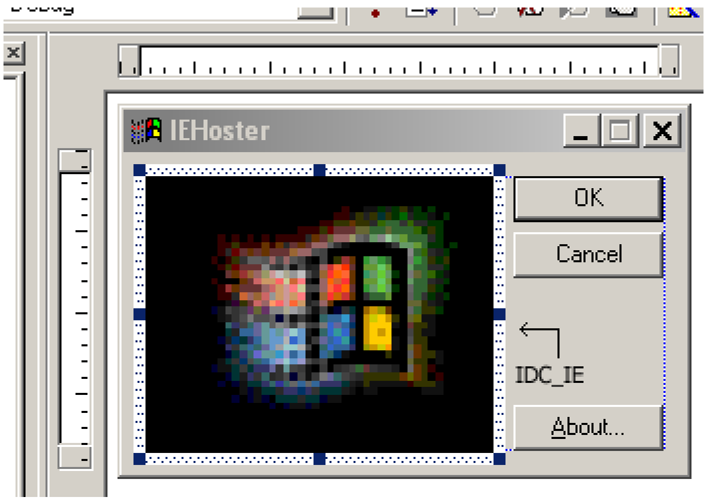
This change has worked fine for me.

Adding Controls with the Resource Editor

ATL lets you add ActiveX controls to a dialog using the resource editor, just as you can in an MFC app. First, right-click in the dialog editor and select *Insert ActiveX control*:



VC will show a list of the controls installed on your system. Scroll down to *Microsoft Web Browser* and click *OK* to insert the control into the dialog. View the properties of the new control and set its ID to `IDC_IE`. The dialog should now look like this, with the control visible in the editor:



If you compile and run the app now, you'll see the WebBrowser control in the dialog. However, it will be blank, since we haven't told it to navigate anywhere.

In the next section, I'll cover the ATL classes involved in creating and hosting ActiveX controls, and then we'll see how to use those classes to communicate with the browser.

ATL Classes for Control Hosting

When hosting ActiveX controls in a dialog, there are two classes that work together: CAxDialogImpl and CAxWindow. They handle all the interfaces that control containers have to implement, and provide some utility functions for common actions such as querying the control for a particular COM interface.

CAxDialogImpl

The first class is CAxDialogImpl. When you write your dialog class, you derive it from CAxDialogImpl, instead of CDialogImpl, to enable control hosting. CAxDialogImpl has overrides for Create() and DoModal(), which call through to the global functions AtlAxCreateDialog() and AtlAxDialogBox() respectively. Since the IEHost dialog is created with Create(), we'll take a closer look at AtlAxCreateDialog().

AtlAxCreateDialog() loads the dialog resource and uses the helper class _DialogSplitHelper to iterate through the controls and look for special entries created by the resource editor that indicate an ActiveX control needs to be created. For example, here is the entry in the *IEHost.rc* file for the WebBrowser control:

```
CONTROL "",IDC_IE,"{8856F961-340A-11D0-A96B-00C04FD705A2}",
        WS_TABSTOP,7,7,116,85
```

The first parameter is the window text (an empty string), the second is the control ID, and the third is the window class name. _DialogSplitHelper::SplitDialogTemplate() sees that the window class begins with '{' and knows it's an ActiveX control entry. It creates a new dialog template in memory that has those special CONTROL entries replaced by ones that create AtlAxWin windows instead. The new entry is the in-memory equivalent of:

```
CONTROL "{8856F961-340A-11D0-A96B-00C04FD705A2}",
         IDC_IE, "AtlAxWin", WS_TABSTOP, 7, 7, 116, 85
```

The result is that an AtlAxWin window will be created with the same ID, and its window text will be the GUID of the ActiveX control. So if you call GetDlgItem(IDC_IE), the return value is the HWND of the AtlAxWin window, not the ActiveX control itself.

Once SplitDialogTemplate() returns, AtlAxCreateDialog() calls CreateDialogIndirectParam() to create the dialog using the modified template.

AtlAxWin and CAxWindow

As mentioned above, an `AtlAxWin` is used as the container window for an ActiveX control. There is a special window interface class that you use with an `AtlAxWin` called `CAxWindow`. When an `AtlAxWin` is created from a dialog template, the `AtlAxWin` window procedure, `AtlAxWindowProc()`, handles `WM_CREATE` and creates the ActiveX control in response to that message. An ActiveX control can also be created at runtime, without being in the dialog template, but we'll cover that case later.

The `WM_CREATE` handler calls the global `AtlAxCreateControl()`, passing it the `AtlAxWin`'s window text. Recall that this was set to the GUID of the WebBrowser control. `AtlAxCreateControl()` calls a couple more functions, but eventually the code reaches `CreateNormalizedObject()`, which converts the window text to a GUID, and finally calls `CoCreateInstance()` to create the ActiveX control.

Since the ActiveX control is a child of the `AtlAxWin`, the dialog can't directly access the control. However, `CAxWindow` has methods for communicating with the control. The one you'll use most often is `QueryControl()`, which calls `QueryInterface()` on the control. For example, you can use `QueryControl()` to get an `IWebBrowser2` interface from the WebBrowser control, and use that interface to navigate the browser to a URL.

Calling the Methods of a Control

Now that our dialog has a WebBrowser in it, we can use its COM interfaces to interact with it. The first thing we'll do is make it navigate to a new URL using its `IWebBrowser2` interface. In the `OnInitDialog()` handler, we can attach a `CAxWindow` variable to the `AtlAxWin` that is hosting the browser.

```
BOOL CMainDlg::OnInitDialog()
{
CAxWindow wndIE = GetDlgItem(IDC_IE);
```

Next, we declare an `IWebBrowser2` interface pointer and query the browser control for that interface, using `CAxWindow::QueryControl()`:

```
CComPtr<IWebBrowser2> pWB2;
HRESULT hr;

  hr = wndIE.QueryControl ( &pWB2 );
```

`QueryControl()` calls `QueryInterface()` on the WebBrowser, and if that succeeds, the `IWebBrowser2` interface is returned to us. We can then call `Navigate()`:

```
if ( pWB2 )
  {
  CComVariant v;  // empty variant


  pWB2->Navigate ( CComBSTR("http://www.codeproject.com/"),
                   &v, &v, &v, &v );
  }
```

Sinking Events Fired by a Control

Getting an interface from the WebBrowser is pretty simple, and it lets us communicate in one direction - *to* the control. There is also a lot of communication *from* the control, in the form of **events**. ATL has classes that encapsulate connection points and event sinking, so that we can receive the events fired by the browser. To use this support, we need to do four things:

1. Add `IDispEventImpl` to `CMainDlg`'s inheritance list
2. Write an *event sink map* that indicates which events we want to handle
3. Write handlers for those events
4. Connect the control to the sink map (a process called *advising*)

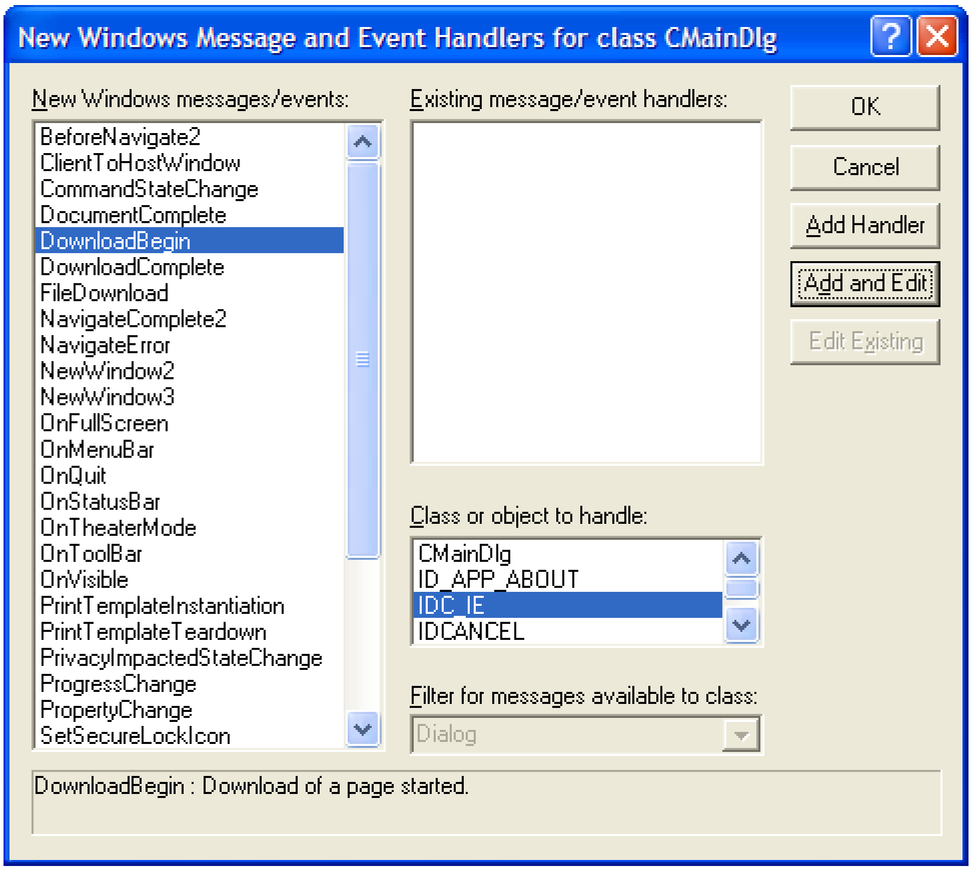
The VC IDE helps out greatly in this process - it will make the changes to `CMainDlg` for you, as well as query the ActiveX control's type library to show a list of the events that the control fires. Since the UI for adding handlers is different in VC 6 and VC 7, I'll cover each separately.

Adding handlers in VC 6

There are two ways to get to the UI for adding handlers:

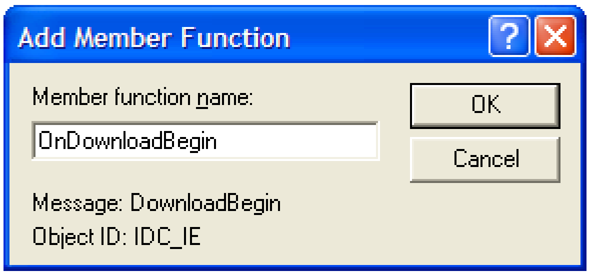
1. In the ClassView pane, right-click `CMainDlg` and choose *Add Windows Message Handler* on the menu.
2. When viewing the `CMainDlg` code or the associated dialog in the resource editor, click the dropdown arrow on the WizardBar action button, then choose *Add Windows Message Handler* on the menu.

After choosing that command, VC shows a dialog with a list of controls, labeled *Class or object to handle*. Select *IDC_IE* in that list, and VC will fill in the *New Windows messages/events* list with the events that the WebBrowser control fires.



DownloadBegin : Download of a page started.

We'll add a handler for the DownloadBegin event, so select that event and click the *Add and Edit* button. VC will prompt you for the method name:



Member function name: OnDownloadBegin

Message: DownloadBegin

Object ID: IDC_IE

The first time you add an event handler, VC will make a few changes to `CMainDlg`, enabling it to be an event sink. The changes are a bit spread out over the header file; they are summarized in the snippet below:

```
#import "C:\WINNT\System32\shdocvw.dll"

class CMainDlg : public CAxDialogImpl<CMainDlg>,
                 public CUpdateUI<CMainDlg>,
                public CMessageFilter, public CIdleHandler,
                public IDispEventImpl<IDC_IE, CMainDlg>
{
// ...

public:
  BEGIN_SINK_MAP(CMainDlg)
    SINK_ENTRY(IDC_IE, 0x6a, OnDownloadBegin)
  END_SINK_MAP()

  void __stdcall OnDownloadBegin()
    {
    // TODO : Add Code for event handler.

    }
};
```

The #import statement is a compiler directive to read the type library in *shdocvw.dll* (the file that has the implementation of the WebBrowser ActiveX control) and create wrapper classes for using the coclasses and interfaces in that control. You would normally move this directive to *stdafx.h*, however in this case we don't need it at all since the Platform SDK already has header files that contain the WebBrowser interfaces and methods.

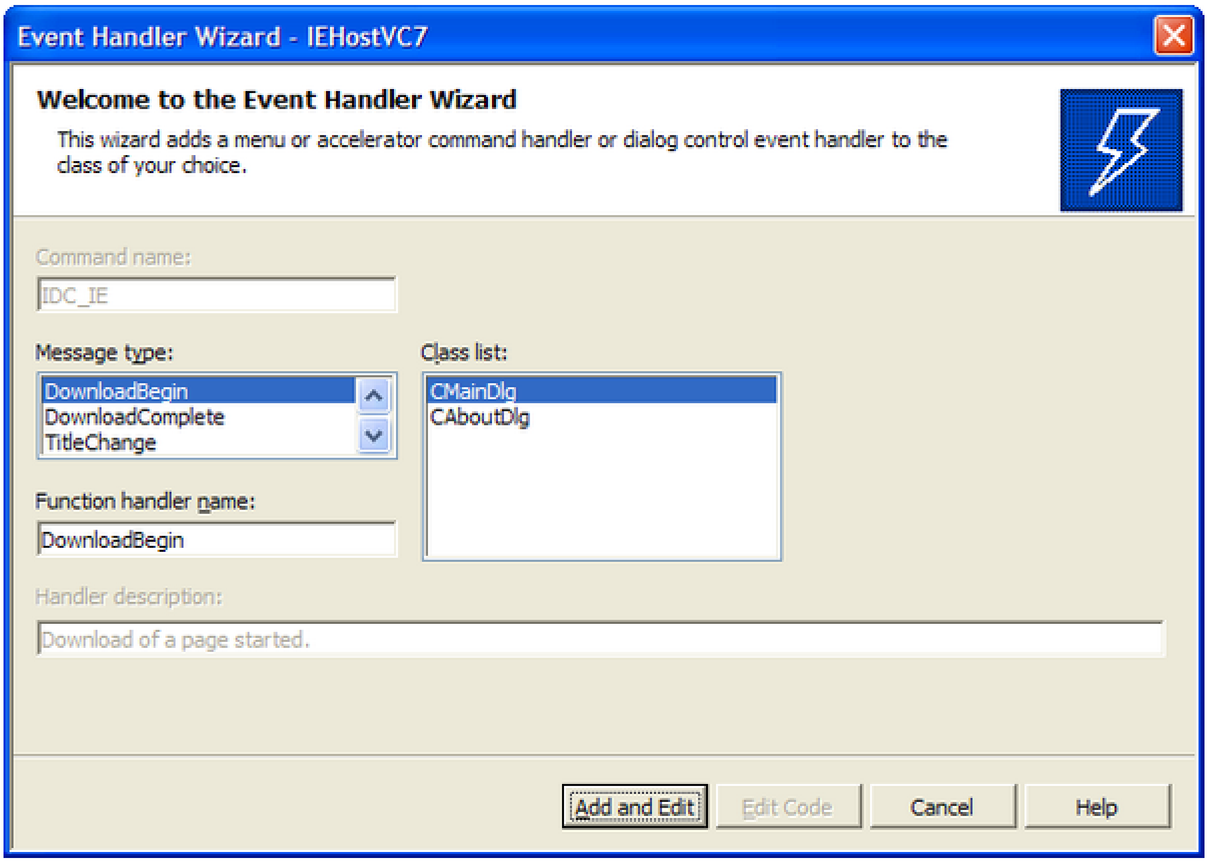
The inheritance list now has IDispEventImpl, which has two template parameters. The first is the ID that we assigned to the ActiveX control, IDC_IE, and the second is the name of the class deriving from IDispEventImpl.

The sink map is delimited by the BEGIN_SINK_MAP and END_SINK_MAP macros. Each SINK_ENTRY macro lists one event that CMainDlg will handle. The parameters to the macro are the control ID (IDC_IE again), the dispatch ID of the event, and the name of the function to call when the event is received. VC reads the dispatch ID from the ActiveX control's type library, so you don't have to worry about figuring out what the numbers are. (If you look in the *exdispid.h* header file, which lists IDs for various events sent by IE and Explorer, you'll see that 0x6A corresponds to the constant DISPID_DOWNLOADBEGIN.)

Finally, there is the new method OnDownloadBegin(). Some events have parameters; for those that do, VC will set up the method to have the proper prototype. All event handlers have the __stdcall calling convention since they are COM methods.

Adding handlers in VC 7

There are again two ways to add an event handler. You can right-click the ActiveX control in the dialog editor and pick *Add Event Handler* on the menu. This brings up a dialog where you select the event name and set the name of the handler.


Event Handler Wizard - IEHostVC7

Welcome to the Event Handler Wizard

This wizard adds a menu or accelerator command handler or dialog control event handler to the class of your choice.

Command name: IDC_IE

Message type: DownloadBegin, DownloadComplete, TitleChange

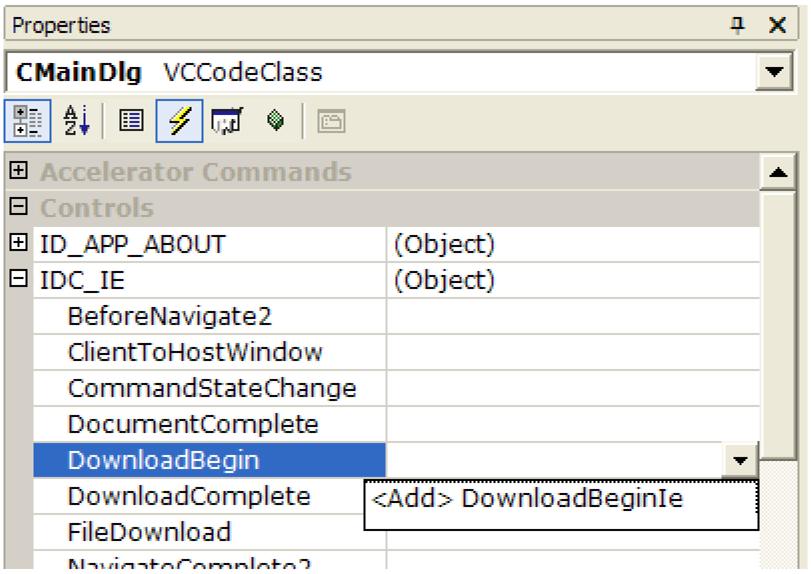
Class list: CMainDlg, CAboutDlg

Function handler name: DownloadBegin

Handler description: Download of a page started.

Clicking the *Add and Edit* button will add the handler, make necessary changes to `CMainDlg`, and open the *maindlg.cpp* file with the new handler highlighted.

The other way is to view the property page for `CMainDlg`, and expand the *Controls* node, then the *IDC_IE* node. Under *IDC_IE*, you'll find the events that control fires.



You can click the arrow next to an event name and pick *<Add> [MethodName]* on the menu to add the handler. You can modify the handler's name later by changing it right in the property page.

VC 7 makes almost the same changes to `CMainDlg` as VC 6, the exception being that no `#import` directive is added.

Advising for events

The last step is to *advise* the control that `CMainDlg` wants to sink (that is, receive) events fired by the WebBrowser control. Again, this process differs in VC 6 and VC 7, so I will cover each separately. In both cases, advising occurs in `OnInitDialog()`, and unadvising occurs in `OnDestroy()`.

Advising in VC 6

ATL in VC 6 has a global function `AtlAdviseSinkMap()`. The function takes a pointer to a C++ object that has a sink map (which will usually be `this`), and a boolean. If the boolean is `true`, the object wants to start receiving events. If it's `false`, the object wants to stop receiving them. `AtlAdviseSinkMap()` advises all controls in the dialog to start or stop sending events to the C++ object.

To use this function, add handlers for `WM_INITDIALOG` and `WM_DESTROY`, then call `AtlAdviseSinkMap()` like this:

```
BOOL CMainDlg::OnInitDialog(...)
{
  // Begin sinking events
  AtlAdviseSinkMap ( this, true );
}

void CMainDlg::OnDestroy()
{
  // Stop sinking events

  AtlAdviseSinkMap ( this, false );
}
```

`AtlAdviseSinkMap()` returns an `HRESULT` that indicates whether the advising succeeded. If `AtlAdviseSinkMap()` fails in `OnInitDialog()`, then you won't get events from some (or all) of the ActiveX controls.

Advising in VC 7

In VC 7, `CAxDialogImpl` has a method called `AdviseSinkMap()` that is a wrapper for `AtlAdviseSinkMap()`. `AdviseSinkMap()` takes one boolean parameter, which has the same meaning as the second parameter to `AtlAdviseSinkMap()`. `AdviseSinkMap()` checks that the class has a sink map, then calls `AtlAdviseSinkMap()`.

The big difference compared to VC 6 is that `CAxDialogImpl` has handlers for `WM_INITDIALOG` and `WM_DESTROY` that call `AdviseSinkMap()` for you. To take advantage of this feature, add a `CHAIN_MSG_MAP` macro at the beginning of the `CMainDlg` message map, like this:

```
  BEGIN_MSG_MAP(CMainDlg)
    CHAIN_MSG_MAP(CAxDialogImpl<CMainDlg>)
    // rest of the message map...

  END_MSG_MAP()
```

Overview of the Sample Project

Now that we've seen how event sinking works, let's check out the complete IEHost project. It hosts the WebBrowser control, as we've been discussing, and handles six events. It also shows a list of the events, so you can get a feel for how custom browsers can use them to provide progress UI. The program handles the following events:

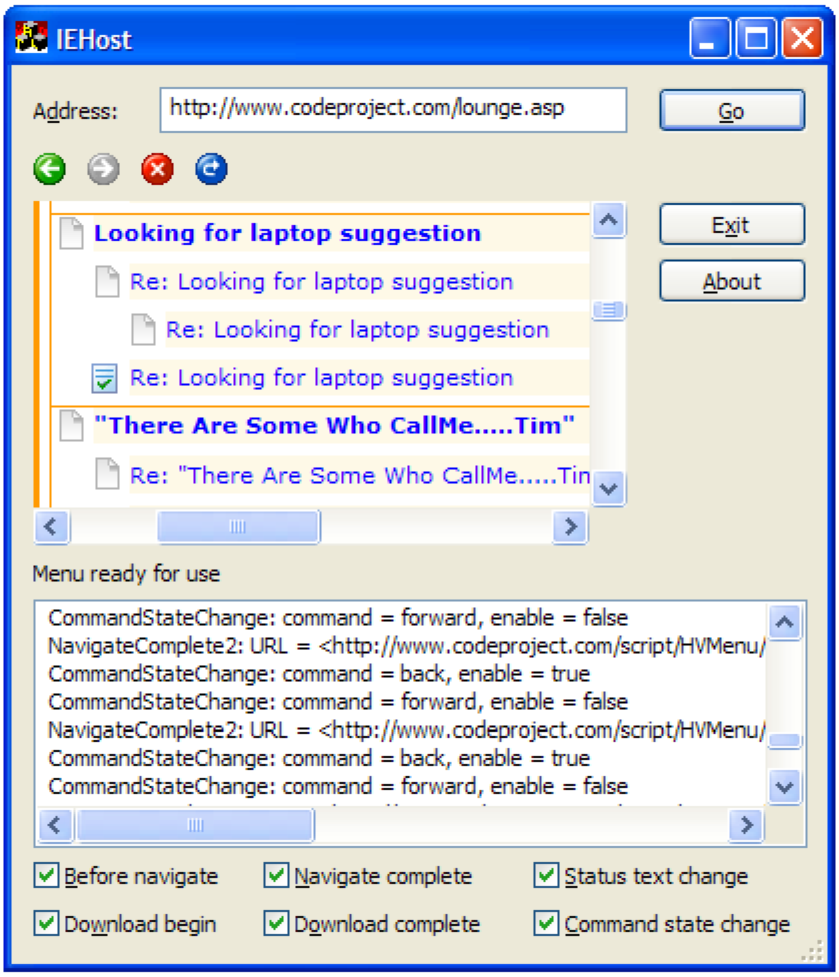
- BeforeNavigate2 and NavigateComplete2: These events let the app watch as the WebBrowser navigates to URLs. You can cancel navigation if you want in response to BeforeNavigate2.
- DownloadBegin and DownloadComplete: The app uses these events to control the "wait" message, which indicates that the browser is working. A more polished app could use an animation, similar to what IE itself uses.
- CommandStateChange: This event tells the app when the Back and Forward navigation commands are available. The app enables or disables the back and forward buttons accordingly.
- StatusTextChange: This event is fired in several cases, for example when the cursor moves over a hyperlink. This event sends a string, and the app responds by showing that string in a static control under the browser window.

The app also has four buttons for controlling the browser: back, forward, stop, and reload. These call the corresponding IWebBrowser2 methods.

The events and the data accompanying the events are all logged to a list control, so you can see the events as they are fired. You can also turn off logging of any events so you can watch just one or two. To demonstrate some non-trivial event handling, the BeforeNavigate2 handler checks the URL, and if it contains "doubleclick.net", the navigation is cancelled. Ad and popup blockers that run as IE plug-ins, instead of HTTP proxies, use this method. Here is the code that does this check.

```
void __stdcall CMainDlg::OnBeforeNavigate2 (
    IDispatch* pDisp, VARIANT* URL, VARIANT* Flags,
    VARIANT* TargetFrameName, VARIANT* PostData,
    VARIANT* Headers, VARIANT_BOOL* Cancel )
{
CString sURL = URL->bstrVal;

    // You can set *Cancel to VARIANT_TRUE to stop the

    // navigation from happening. For example, to stop

    // navigates to evil tracking companies like doubleclick.net:

    if ( sURL.Find ( _T("doubleclick.net") ) > 0 )
        *Cancel = VARIANT_TRUE;
}
```

Here's what the app looks like while viewing the Lounge:


IEHost
Address:
http://www.codeproject.com/lounge.asp
Go
Exit
About
Looking for laptop suggestion
Re: Looking for laptop suggestion
Re: Looking for laptop suggestion
Re: Looking for laptop suggestion
"There Are Some Who CallMe.....Tim"
Re: "There Are Some Who CallMe.....Tin
Menu ready for use

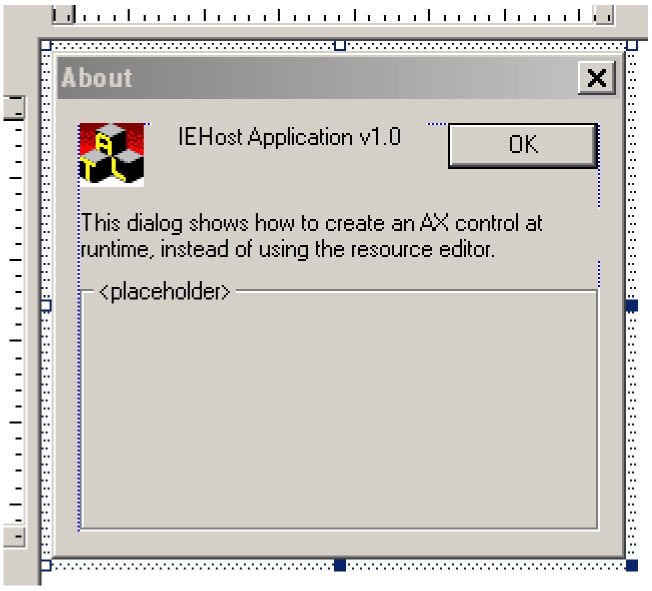
```
CommandStateChange: command = forward, enable = false
NavigateComplete2: URL = <http://www.codeproject.com/script/HVMenu/
CommandStateChange: command = back, enable = true
CommandStateChange: command = forward, enable = false
NavigateComplete2: URL = <http://www.codeproject.com/script/HVMenu/
CommandStateChange: command = back, enable = true
CommandStateChange: command = forward, enable = false
```

[✓] Before navigate [✓] Navigate complete [✓] Status text change
[✓] Download begin [✓] Download complete [✓] Command state change

IEHost demonstrates several other WTL features that have been covered in earlier articles: `CBitmapButton` (for the browser control buttons), `CListViewCtrl` (for the event logging), DDX (to keep track of the checkbox states), and `CDialogResize`.

Creating an ActiveX Control at Run Time

It is also possible to create an ActiveX control at run time, instead of in the resource editor. The About box demonstrates this technique. The dialog resource contains a placeholder group box that marks where the WebBrowser control will go:



IEHost Application v1.0

This dialog shows how to create an AX control at runtime, instead of using the resource editor.

<placeholder>

In OnInitDialog(), we use CAxWindow to create a new AtlAxWin in the same RECT as the placeholder (which is then destroyed):

```
LRESULT CAboutDlg::OnInitDialog(...)
{
CWindow wndPlaceholder = GetDlgItem ( IDC_IE_PLACEHOLDER );
CRect rc;
CAxWindow wndIE;

    // Get the rect of the placeholder group box, then destroy

    // that window because we don't need it anymore.

    wndPlaceholder.GetWindowRect ( rc );
    ScreenToClient ( rc );
    wndPlaceholder.DestroyWindow();

    // Create the AX host window.

    wndIE.Create ( *this, rc, _T(""),
                   WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN );
```

Next, we use a CAxWindow method to create the ActiveX control. The two methods we can choose from are CreateControl() and CreateControlEx(). CreateControlEx() has additional parameters that can return interface pointers, so you don't have to subsequently call QueryControl(). The two parameters we're interested in are the first, which is the string version of the WebBrowser control's GUID; and the fourth, which is a pointer to an IUnknown*. This pointer will be filled in with the IUnknown of the ActiveX control. After creating the control, we query for an IWebBrowser2 interface, just as before, and navigate the control to a URL.

```
CComPtr<IUnknown> punkCtrl;
CComQIPtr<IWebBrowser2> pWB2;
CComVariant v;    // empty VARIANT


    // Create the browser control using its GUID.

    wndIE.CreateControlEx ( L"{8856F961-340A-11D0-A96B-00C04FD705A2}",
                            NULL, NULL, &punkCtrl );

    // Get an IWebBrowser2 interface on the control and navigate to a page.

    pWB2 = punkCtrl;
    pWB2->Navigate ( CComBSTR("about:mozilla"), &v, &v, &v, &v );
}
```

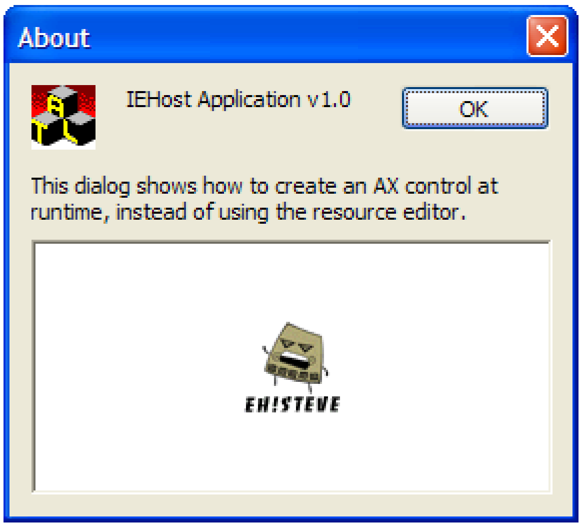
For ActiveX controls that have ProgIDs, you can pass the ProgID to CreateControlEx() instead of the GUID. For example, we could create the WebBrowser control with this call:

```
    // Use the control's ProgID, Shell.Explorer:

    wndIE.CreateControlEx ( L"Shell.Explorer", NULL,
                            NULL, &punkCtrl );
```

CreateControl() and CreateControlEx() also have overloads that are used specifically with WebBrowser controls. If your app contains a web page as an HTML resource, you can pass its resource ID as the first parameter. ATL will create a WebBrowser control and navigate it to the resource. IEHost contains a page whose ID is IDR_ABOUTPAGE, so we can show it in the about box with this code:

```
    wndIE.CreateControl ( IDR_ABOUTPAGE );
```

Here's the result:



This dialog shows how to create an AX control at runtime, instead of using the resource editor.

The sample project contains code for all three of the techniques described above. Check out `CAboutDlg::OnInitDialog()` and comment/uncomment code to see each one in action.

Keyboard Handling

One final, but very important, detail is keyboard messages. Keyboard handling with ActiveX controls is rather complicated, since the host and the control have to work together to make sure the control sees the messages it's interested in. For example, the WebBrowser control lets you move through links with the TAB key. MFC handles all this itself, so you may never have realized the amount of work required to get keyboard support perfectly right.

Unfortunately, the AppWizard doesn't generate keyboard handling code for a dialog-based app. However, if you make an SDI app that uses a form view as the view window, you'll see the necessary code in `PreTranslateMessage()`. When a mouse or keyboard message is read from the message queue, the code gets the control with the focus and forwards the message to the control using the special ATL message `WM_FORWARDMSG`. Normally, when a window receives `WM_FORWARDMSG`, it does nothing, since it doesn't know about that message. However, when an ActiveX control has the focus, the `WM_FORWARDMSG` ends up being sent to the `AtlAxWin` that hosts the control. `AtlAxWin` recognizes `WM_FORWARDMSG` and takes the necessary steps to see if the control wants to handle the message itself.

If the window with the focus does not recognize `WM_FORWARDMSG`, then `PreTranslateMessage()` calls `IsDialogMessage()` so that the standard dialog navigation keys like TAB work properly.

The sample project contains the necessary code in `CMainDlg::PreTranslateMessage()`. Since `PreTranslateMessage()` works only in modeless dialogs, your dialog-based app *must* use a modeless dialog if you want proper keyboard handling.

Up Next

In the next article, we'll return to frame windows and cover the topic of using splitter windows.

Part VII - Splitter Windows

Introduction

Splitter windows have been a popular UI element since Explorer debuted in Windows 95, with its two-pane view of the file system. MFC has a complex and powerful splitter window class, however it is somewhat difficult to learn how to use, and coupled to the doc/view framework. In Part VII, I will discuss the WTL splitter window, which is much less complicated than MFC's. While WTL's splitter implementation does have fewer features than MFC's, it is far easier to use and extend.

The sample project for this part will be a rewrite of ClipSpy, using WTL of course instead of MFC. If you're not familiar with that program, please check out the article now, as I will be duplicating the functionality of ClipSpy here without providing in-depth explanations of how it works. This article's focus is the splitter window, not the clipboard.

WTL Splitter Windows

The header file *atlsplit.h* contains all of the WTL splitter window classes. There are three classes: `CSplitterImpl`, `CSplitterWindowImpl`, and `CSplitterWindowT`. The classes and their basic methods are explained below.

Classes

`CSplitterImpl` is a template class that takes two template parameters, a window interface class name and a boolean that indicates the splitter orientation: `true` for vertical, `false` for horizontal. `CSplitterImpl` has almost all of the implementation for a splitter, and many methods are overridable so you can provide custom drawing of the split bar or other effects. `CSplitterWindowImpl` derives from `CWindowImpl` and `CSplitterImpl`, but doesn't have much code. It has an empty `WM_ERASEBKGND` handler, and a `WM_SIZE` handler that resizes the splitter window.

Finally, `CSplitterWindowT` derives from `CSplitterImpl` and provides a window class name. If you don't need to do any customization, there are two convenient typedefs that you can use: `CSplitterWindow` for a vertical splitter, and `CHorSplitterWindow` for a horizontal splitter.

Creating a splitter

Since CSplitterWindow derives from CWindowImpl, you create a splitter just like any other child window. When a splitter will exist for the lifetime of the main frame, as it will in ClipSpy, you can add a CSplitterWindow member variable to CMainFrame. In CMainFrame::OnCreate(), you create the splitter as a child of the frame, then set the splitter as the main frame's client window:

```
LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )
{
// ...

  m_wndSplit.Create ( *this, rcDefault );
  m_hWndClient = m_wndSplit;
}
```

After creating the splitter, you can assign windows to its panes, and do any other necessary initialization.

Basic methods

```
bool SetSplitterPos(int xyPos = -1, bool bUpdate = true)
int GetSplitterPos()
```

Call SetSplitterPos() to set the position of the splitter bar. The position is expressed in pixels relative to the top edge (for horizontal splitters) or left edge (for vertical splitters) of the splitter window. You can use the default of -1 to position the splitter bar in the middle, making both panes the same size. You will usually pass true for bUpdate, to have the splitter immediately resize the panes accordingly. GetSplitterPos() returns the current position of the splitter bar, relative to the top or left edge of the splitter window. (If the splitter is in single-pane mode, GetSplitterPos() returns the position the bar will return to when both panes are shown.)

```
bool SetSinglePaneMode(int nPane = SPLIT_PANE_NONE)
int GetSinglePaneMode()
```

Call SetSinglePaneMode() to change the splitter between one-pane and two-pane mode. In one-pane mode, only one pane is visible and the splitter bar is hidden, similar to how MFC dynamic splitters work (although there is no little gripper handle to re-split the splitter). The allowable values for nPane are:

SPLIT_PANE_LEFT, SPLIT_PANE_RIGHT, SPLIT_PANE_TOP, SPLIT_PANE_BOTTOM, and SPLIT_PANE_NONE. The first four indicate which pane to show (for example, passing SPLIT_PANE_LEFT shows the left-side pane and hides the right-side pane). Passing SPLIT_PANE_NONE shows both panes. GetSinglePaneMode() returns one of those five SPLIT_PANE_* values indicating the current mode.

```
DWORD SetSplitterExtendedStyle(DWORD dwExtendedStyle, DWORD dwMask = 0)
DWORD GetSplitterExtendedStyle()
```

Splitter windows have extended styles that control how the splitter bar moves when the entire splitter window is resized. The available styles are:

- SPLIT_PROPORTIONAL: Both panes in the splitter resize together
- SPLIT_RIGHTALIGNED: The right pane stays the same size when the entire splitter is resized, and the left pane resizes
- SPLIT_BOTTOMALIGNED: The bottom pane stays the same size when the entire splitter is resized, and the top pane resizes

If none of those three styles are specified, the splitter defaults to being left- or top-aligned. If you pass SPLIT_PROPORTIONAL and SPLIT_RIGHTALIGNED/SPLIT_BOTTOMALIGNED together, SPLIT_PROPORTIONAL takes precedence.

There is one additional style that controls whether the user can move the splitter bar:

- SPLIT_NONINTERACTIVE: The splitter bar cannot be moved and does not respond to the mouse

The default value of the extended styles is SPLIT_PROPORTIONAL.

```
bool SetSplitterPane(int nPane, HWND hWnd, bool bUpdate = true)
void SetSplitterPanes(HWND hWndLeftTop, HWND hWndRightBottom, bool bUpdate =
true)
HWND GetSplitterPane(int nPane)
```

Call SetSplitterPane() to assign a child window to one pane of the splitter. nPane is one of the SPLIT_PANE_* values indicating which pane you are setting. hWnd is the window handle of the child window. You can assign child windows to both panes at once with SetSplitterPanes(). You will usually use the default value of bUpdate, which tells the splitter to immediately resize the child windows to fit in the panes. SetSplitterPane() returns a bool, however it will only return false if you pass an invalid value for nPane.

You can get the HWND of the window in a pane with GetSplitterPane(). If no window has been assigned to a pane, GetSplitterPane() returns NULL.

```
bool SetActivePane(int nPane)
int GetActivePane()
```

SetActivePane() sets the focus to one of the windows in the splitter. nPane is one of the SPLIT_PANE_* values indicating which pane you are setting as the active one. It also sets the default active pane (explained below). GetActivePane() checks the window with the focus, and if that window is a pane window or a child of a pane window, returns a SPLIT_PANE_* value indicating which pane. If the window with the focus is not a child of a pane, GetActivePane() returns SPLIT_PANE_NONE.

```
bool ActivateNextPane(bool bNext = true)
```

If the splitter is in single-pane mode, the focus is set to the visible pane. Otherwise, ActivateNextPane() checks the window with the focus using GetActivePane(). If a pane (or child of a pane) has the focus, the splitter sets the focus to the other pane. Otherwise, ActivateNextPane() activates the left/top pane if bNext is true, or the right/bottom pane if bNext is false.

```
bool SetDefaultActivePane(int nPane)
bool SetDefaultActivePane(HWND hWnd)
int GetDefaultActivePane()
```

Call SetDefaultActivePane() with either a SPLIT_PANE_* value or window handle to set that pane as the default active pane. If the splitter window itself gets the focus, via a SetFocus() call, it in turn sets the focus to the default active pane. GetDefaultActivePane() returns a SPLIT_PANE_* value indicating the current default active pane.

```
void GetSystemSettings(bool bUpdate)
```

GetSystemSettings() reads various system settings and sets data members accordingly. Pass true for bUpdate to have the splitter immediately redraw itself using the new settings.

The splitter calls this method when it is created, so you don't have to call it yourself. However, your main frame should handle the WM_SETTINGCHANGE message and pass it along to the splitter; CSplitterWindow calls GetSystemSettings() in its WM_SETTINGCHANGE handler.

Data members

Some other splitter features are controlled by setting public members of CSplitterWindow. These are all reset when GetSystemSettings() is called.

m_cxySplitBar

For vertical splitters: Controls the width of the splitter bar. The default is the value returned by GetSystemMetrics(SM_CXSIZEFRAME).
For horizontal splitters: Controls the height of the splitter bar. The default is the value returned by GetSystemMetrics(SM_CYSIZEFRAME).

m_cxyMin

For vertical splitters: Controls the minimum width of each pane. The splitter will not allow you to drag the bar if it would make either pane smaller than this number of pixels. The default is 0 if the splitter window has the WS_EX_CLIENTEDGE extended window style. Otherwise, the default is 2*GetSystemMetrics(SM_CXEDGE).
For horizontal splitters: Controls the minimum height of each pane. The default is 0 if the splitter window has the WS_EX_CLIENTEDGE extended window style. Otherwise, the default is 2*GetSystemMetrics(SM_CYEDGE).

m_cxyBarEdge

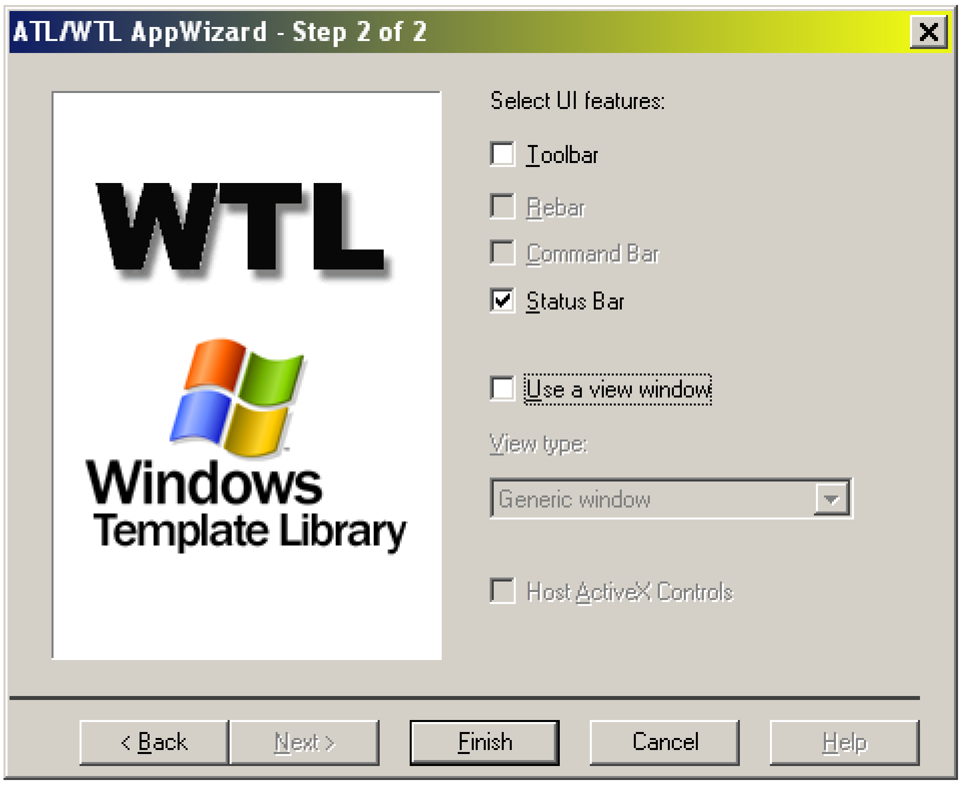
For vertical splitters: Controls the width of the 3D edge drawn on the sides of the splitter bar. The default value is 2*GetSystemMetrics(SM_CXEDGE) if the splitter window has the WS_EX_CLIENTEDGE extended window style, otherwise the default is 0.
For horizontal splitters: Controls the height of the 3D edge drawn on the sides of the splitter bar. The default value is 2*GetSystemMetrics(SM_CYEDGE) if the splitter window has the WS_EX_CLIENTEDGE extended window style, otherwise the default is 0.

m_bFullDrag

If this member is set to true, the panes resize as the splitter bar is dragged. If it is false, only a ghost image of the splitter bar is drawn, and the panes don't resize until the user releases the splitter bar. The default is the value returned by SystemParametersInfo(SPI_GETDRAGFULLWINDOWS).

Starting the Sample Project

Now that we have the basics out of the way, let's see how to set up a frame window that contains a splitter. Start a new project with the WTL AppWizard. On the first page, leave *SDI Application* selected and click Next. On the second page, uncheck *Toolbar*, then uncheck *Use a view window* as shown here:



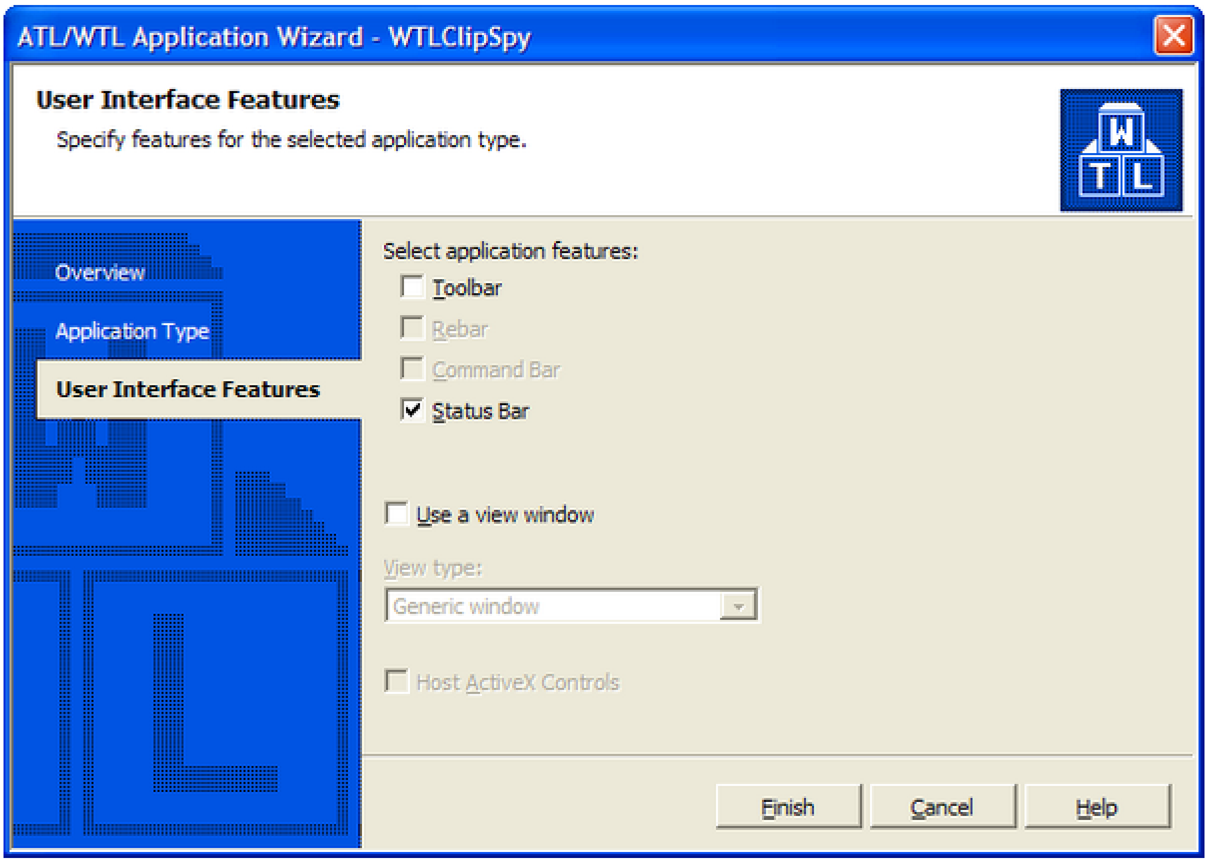
Select UI features:

- [ ] Toolbar
- [ ] Rebar
- [ ] Command Bar
- [x] Status Bar

- [ ] Use a view window

View type: Generic window

- [ ] Host ActiveX Controls



User Interface Features

Specify features for the selected application type.

Select application features:

- [ ] Toolbar
- [ ] Rebar
- [ ] Command Bar
- [x] Status Bar

- [ ] Use a view window

View type: Generic window

- [ ] Host ActiveX Controls

We don't need a view window because the splitter and its panes will become the "view." In `CMainFrame`, add a `CSplitterWindow` member:

```
class CMainFrame : public ...
{
//...

protected:
  CSplitterWindow  m_wndVertSplit;
};
```

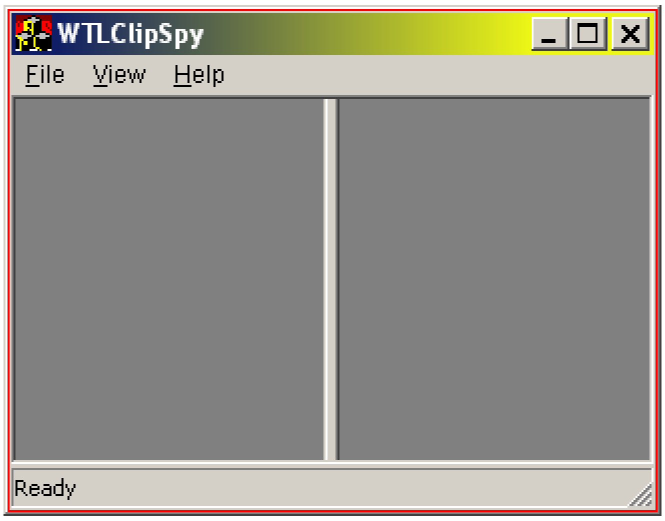
Then in `OnCreate()`, create the splitter and set it as the view window:

```
LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )
{
//...

  // Create the splitter window

  m_wndVertSplit.Create ( *this, rcDefault, NULL,
                          0, WS_EX_CLIENTEDGE );

  // Set the splitter as the client area window, and resize

  // the splitter to match the frame size.

  m_hWndClient = m_wndVertSplit;
  UpdateLayout();

  // Position the splitter bar.

  m_wndVertSplit.SetSplitterPos ( 200 );

  return 0;
}
```

Note that you need to set `m_hWndClient` and call `CFrameWindowImpl::UpdateLayout()` before setting the splitter position. `UpdateLayout()` resizes the splitter window to its initial size. If you skip that step, the splitter's size isn't under your control and it might be smaller than 200 pixels wide. The end result would be that `SetSplitterPos()` wouldn't have the effect you wanted.

An alternative to calling `UpdateLayout()` is to get the client `RECT` of the frame window, and use that `RECT` when creating the splitter, instead of `rcDefault`. This way, you create the splitter in its initial position, and all subsequent methods dealing with position (like `SetSplitterPos()`) will work correctly.

If you run the app now, you'll see the splitter in action. Even without creating anything for the panes, the basic behavior is there. You can drag the bar, and double-clicking it moves the bar to the center.



To demonstrate different ways of managing the pane windows, I'll use one `CListViewCtrl`-derived class, and a plain `CRichEditCtrl`. Here's a snippet from the `CClipSpyListCtrl` class, which we'll use in the left pane:

```
typedef CWinTraitsOR<LVS_REPORT | LVS_SINGLESEL | LVS_NOSORTHEADER>
        CListTraits;

class CClipSpyListCtrl :
  public CWindowImpl<CClipSpyListCtrl, CListViewCtrl, CListTraits>,
  public CCustomDraw<CClipSpyListCtrl>
{
public:
  DECLARE_WND_SUPERCLASS(NULL, WC_LISTVIEW)

  BEGIN_MSG_MAP(CClipSpyListCtrl)
    MSG_WM_CHANGECBCHAIN(OnChangeCBChain)
    MSG_WM_DRAWCLIPBOARD(OnDrawClipboard)
    MSG_WM_DESTROY(OnDestroy)
    CHAIN_MSG_MAP_ALT(CCustomDraw<CClipSpyListCtrl>, 1)
    DEFAULT_REFLECTION_HANDLER()
  END_MSG_MAP()
//...
};
```

If you've been following the previous articles, you should have no trouble reading this class. It handles `WM_CHANGECBCHAIN` to know when other clipboard viewers come and go, and `WM_DRAWCLIPBOARD` to know when the contents of the clipboard change.

Since the pane windows will exist for the life of the app, we can use member variables in `CMainFrame` for them as well:

```
class CMainFrame : public ...
{
//...

protected:
  CSplitterWindow  m_wndVertSplit;
  CClipSpyListCtrl m_wndFormatList;
  CRichEditCtrl    m_wndDataViewer;
};
```

Creating windows in the panes

Now that we have member variables for the splitter and the panes, filling in the splitter is a simple matter. After creating the splitter window, we create both child windows, using the splitter as their parent:

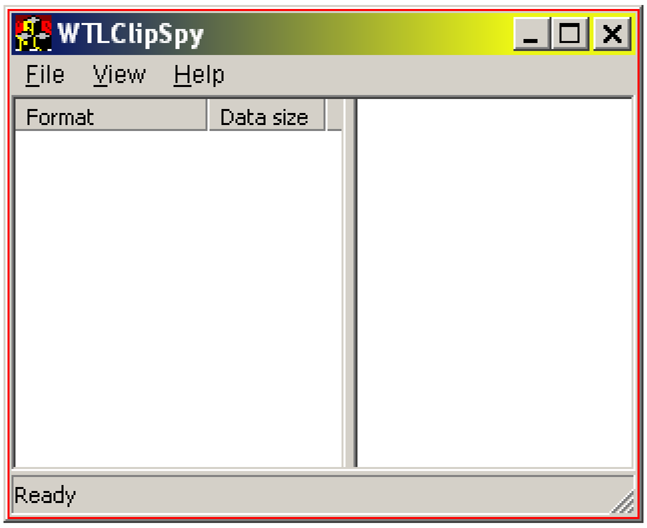
```
LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )
{
//...

  // Create the splitter window

  m_wndVertSplit.Create ( *this, rcDefault, NULL,
                          0, WS_EX_CLIENTEDGE );

  // Create the left pane (list of clip formats)

  m_wndFormatList.Create ( m_wndVertSplit, rcDefault );

  // Create the right pane (rich edit ctrl)

  DWORD dwRichEditStyle =
       WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
       ES_READONLY | ES_AUTOHSCROLL |
       ES_AUTOVSCROLL | ES_MULTILINE;

  m_wndDataViewer.Create ( m_wndVertSplit, rcDefault,
                           NULL, dwRichEditStyle );
  m_wndDataViewer.SetFont ( AtlGetStockFont(ANSI_FIXED_FONT) );

  // Set the splitter as the client area window, and resize

  // the splitter to match the frame size.

  m_hWndClient = m_wndVertSplit;
  UpdateLayout();

  m_wndVertSplit.SetSplitterPos ( 200 );

  return 0;
}
```

Notice that both `Create()` calls use `m_wndVertSplit` as the parent window. The `RECT` parameter is not important, since the splitter will resize both pane windows as necessary, so we can use `CWindow::rcDefault`.

The last step is to pass the `HWND`s of the panes to the splitter. This also has to come before `UpdateLayout()` so all the windows end up the correct size.

```
LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )
{
//...

  m_wndDataViewer.SetFont ( AtlGetStockFont(ANSI_FIXED_FONT) );

  // Set up the splitter panes

  m_wndVertSplit.SetSplitterPanes ( m_wndFormatList, m_wndDataViewer );

  // Set the splitter as the client area window, and resize

  // the splitter to match the frame size.

  m_hWndClient = m_wndVertSplit;
  UpdateLayout();
  m_wndVertSplit.SetSplitterPos ( 200 );

  return 0;
}
```

And here's what the result looks like, after the list control has had some columns added:



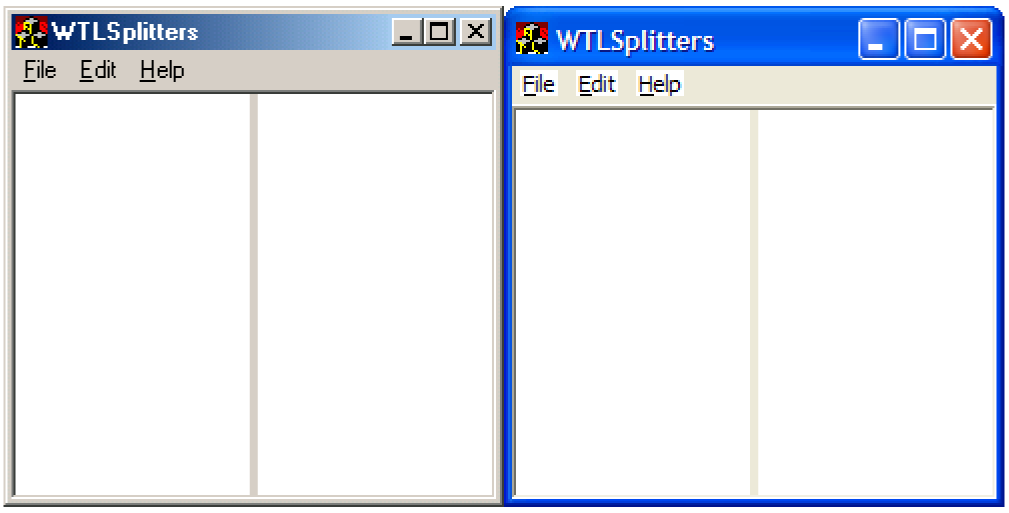
Note that the splitter puts no restrictions on what windows can go in the panes, unlike MFC where you are supposed to use `CView`s. The pane windows should have at least the `WS_CHILD` style, but beyond that you're pretty much free to use anything.

Effects of WS_EX_CLIENTEDGE

A little sidebar is in order about the effect that the `WS_EX_CLIENTEDGE` style has on the splitter and the windows in the panes. There are three places where we can apply this style: the main frame, the splitter window, or the window in a splitter pane. `WS_EX_CLIENTEDGE` creates a different look in each case, so I will illustrate them here.

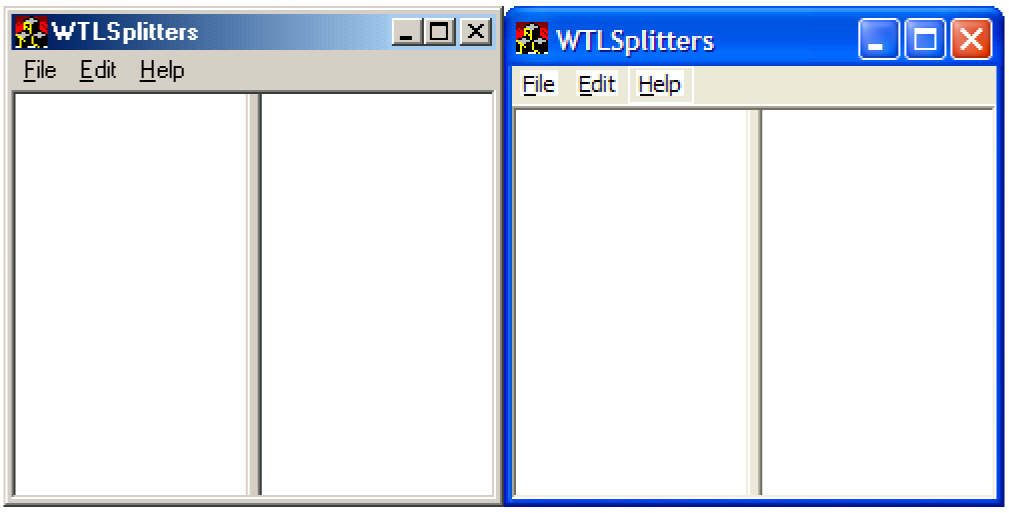
`WS_EX_CLIENTEDGE` on the frame window:

This is the least appealing choice, since the border of the splitter has an edge, but the bar has no edge.



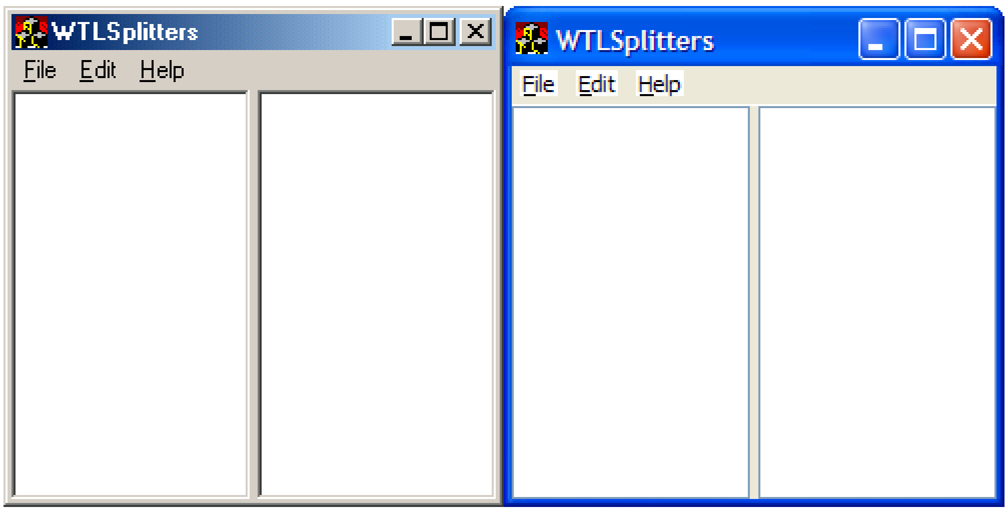
`WS_EX_CLIENTEDGE` on the splitter window:

When a `CSplitterWindow` has the `WS_EX_CLIENTEDGE` style, the drawing code takes the extra step of drawing a border along each side of the bar, so that there is an edge around each pane as well as around the entire splitter window.



`WS_EX_CLIENTEDGE` on the pane windows:

Each pane window has a border, and the splitter bar merges into the frame window's menu and border without any breaks. This is more noticeable on pre-XP Windows (or XP with themes turned off). On XP with themes turned on, it's hard tell that there's a splitter bar there unless you go hunting with the mouse.



Message Routing

Since we now have another window sitting between the main frame and the pane windows, you might have wondered how notification messages work. Specifically, how can the main frame receive NM_CUSTOMDRAW notifications so it can reflect them to the list? The answer can be found in the CSplitterWindowImpl message map:

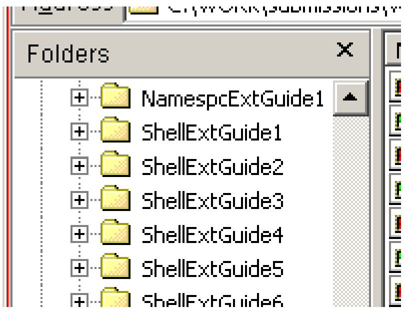
```
BEGIN_MSG_MAP()
  MESSAGE_HANDLER(WM_ERASEBKGND, OnEraseBackground)
  MESSAGE_HANDLER(WM_SIZE, OnSize)
  CHAIN_MSG_MAP(baseClass)
  FORWARD_NOTIFICATIONS()
END_MSG_MAP()
```

The FORWARD_NOTIFICATIONS() macro at the end is the important one. Recall from Part IV that there are several notification messages which are always sent to the parent of a child window. What FORWARD_NOTIFICATIONS() does is re-send the message to *the splitter's* parent window. So when the list sends a WM_NOTIFY message to the splitter (the list's parent), the splitter in turn sends the WM_NOTIFY to the main frame (the splitter's parent). When the main frame reflects the message, it is sent back to the window that generated the WM_NOTIFY in the first place, so the splitter doesn't get involved in reflection.

The result of all this is that notification messages sent between the main frame and the list don't get affected by the presence of the splitter window. This makes it rather easy to add or remove splitters, because the child window classes won't have to be changed at all for their message processing to continue working.

Pane Containers

WTL also supports a widget like the one in the left pane of Explorer, called a *pane container*. This control provides a header area with text, and optionally a Close button:



The pane container manages a child window, just as the splitter manages two pane windows. When the container is resized, the child is automatically resized to match the space inside the container.

Classes

There are two classes in the implementation of pane containers, both in *atlctrlx.h*: `CPaneContainerImpl` and `CPaneContainer`. `CPaneContainerImpl` is a `CWindowImpl`-derived class that contains the complete implementation; `CPaneContainer` provides just a window class name. Unless you want to override any methods to change how the container is drawn, you will always use `CPaneContainer`.

Basic methods

```
HWND Create(
    HWND hWndParent, LPCTSTR lpstrTitle = NULL,
    DWORD dwStyle = WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS | WS_CLIPCHILDREN,
    DWORD dwExStyle = 0, UINT nID = 0, LPVOID lpCreateParam = NULL)
HWND Create(
    HWND hWndParent, UINT uTitleID,
    DWORD dwStyle = WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS | WS_CLIPCHILDREN,
    DWORD dwExStyle = 0, UINT nID = 0, LPVOID lpCreateParam = NULL)
```

Creating a `CPaneContainer` is similar to creating other child windows. There are two `Create()` methods that differ in just the second parameter. In the first version, you pass a string that will be used for the title text drawn in the header. In the second method, you pass the ID of a string table entry. The defaults for the remaining parameters are usually sufficient.

```
DWORD SetPaneContainerExtendedStyle(DWORD dwExtendedStyle, DWORD dwMask = 0)
DWORD GetPaneContainerExtendedStyle()
```

`CPaneContainer` has additional extended styles that control the close button and the layout of the container:

- `PANECNT_NOCLOSEBUTTON`: Set this style to remove the Close button from the header.
- `PANECNT_VERTICAL`: Set this style to make the header area vertical, along the left side of the container window.

The default value of the extended styles is 0, which results in a horizontal container with a close button.

```
HWND SetClient(HWND hWndClient)
HWND GetClient()
```

Call `SetClient()` to assign a child window to the pane container. This works similarly to the `SetSplitterPane()` method in `CSplitterWindow`. `SetClient()` returns the `HWND` of the old client window. Call `GetClient()` to get the `HWND` of the current client window.

```
BOOL SetTitle(LPCTSTR lpstrTitle)
BOOL GetTitle(LPTSTR lpstrTitle, int cchLength)
int GetTitleLength()
```

Call `SetTitle()` to change the text shown in the header area of the container. Call `GetTitle()` to retrieve the current header text, and call `GetTitleLength()` to get the length in characters of the current header text (not including the null terminator).

```
BOOL EnableCloseButton(BOOL bEnable)
```

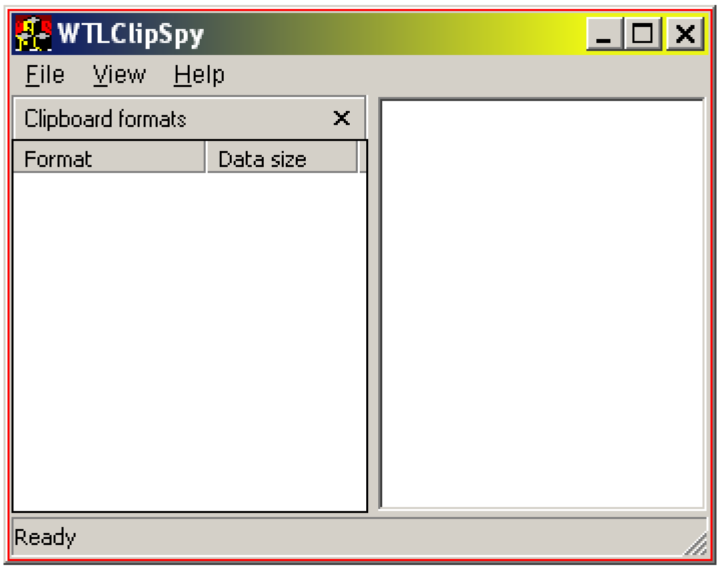
If the pane container has a Close button, you can use `EnableCloseButton()` to enable and disable it.

Using a pane container in a splitter window

To demonstrate how to add a pane container to an existing splitter, we'll add a container to the left pane of the ClipSpy splitter. Instead of assigning the list control to the left pane, we assign the pane container. The list is then assigned to the pane container. Here are the lines in `CMainFrame::OnCreate()` to change to set up the pane container.

```
LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )
{
//...

  m_wndVertSplit.Create ( *this, rcDefault );

  // Create the pane container.

  m_wndPaneContainer.Create ( m_wndVertSplit, IDS_PANE_CONTAINER_TEXT );

  // Create the left pane (list of clip formats)

  m_wndFormatList.Create ( m_wndPaneContainer, rcDefault );
//...

  // Set up the splitter panes

  m_wndPaneContainer.SetClient ( m_wndFormatList );
  m_wndVertSplit.SetSplitterPanes ( m_wndPaneContainer, m_wndDataViewer );
```

Notice that the parent of the list control is `m_wndPaneContainer`. Also, `m_wndPaneContainer` is set as the left pane of the splitter. Here's what the modified left pane looks like.



The Close button and message handling

When the user clicks the Close button, the pane container sends a `WM_COMMAND` message to its parent, with a command ID of `ID_PANE_CLOSE` (a constant defined in *atlres.h*). When you use the pane container in a splitter, the usual course of action is to call `SetSinglePaneMode()` to hide the splitter pane that has the pane container. (But remember to provide a way for the user to show the pane again!)

The `CPaneContainer` message map also has the `FORWARD_NOTIFICATIONS()` macro, just like `CSplitterWindow`, so the container passes notification messages from its client window to its parent. In the case of ClipSpy, there are two windows between the list control and the main frame (the pane container and the splitter), but the `FORWARD_NOTIFICATIONS()` macros ensure that all notifications from the list arrive at the main frame.

Advanced Splitter Features

In this section, I'll describe how to do some common advanced UI tricks with WTL splitters.

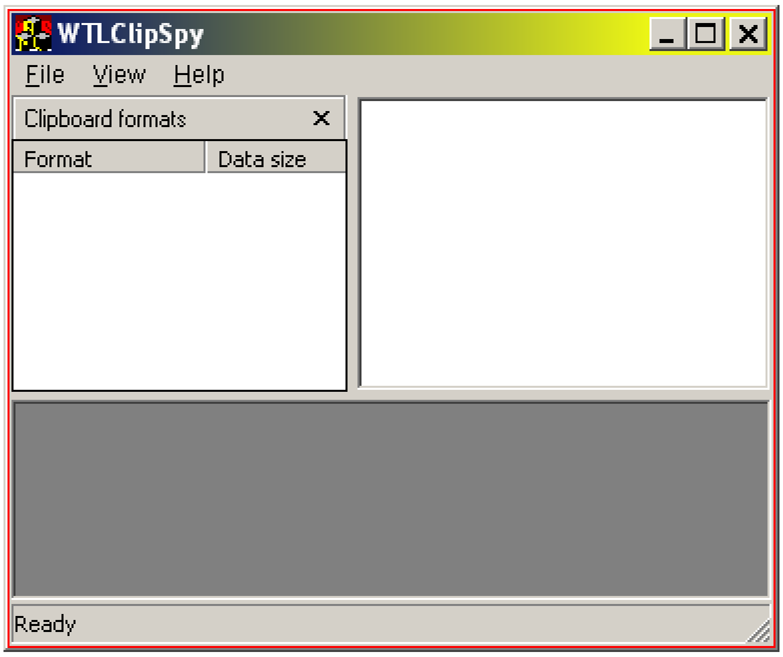
Nested splitters

If you plan on writing an app such as an email client or RSS reader, you'll probably end up using nested splitters - one horizontal and one vertical. This is easy to do with WTL splitters - you create one splitter as the child of the other.

To show this in action, we'll add a horizontal splitter to ClipSpy. The horizontal splitter will be the topmost one, and the vertical splitter will be nested in it. After adding a `CHorSplitterWindow` member called `m_wndHorzSplitter`, we create that splitter the same way as we create `m_wndVertSplitter`. To make `m_wndHorzSplitter` the topmost splitter. `m_wndVertSplitter` is now created as a child of `m_wndHorzSplitter`. Finally, `m_hWndClient` is set to `m_wndHorzSplitter`, since that's the window that now occupies the main frame's client area.

```
LRESULT CMainFrame::OnCreate()
{
//...

  // Create the splitter windows.

  m_wndHorzSplit.Create ( *this, rcDefault );
  m_wndVertSplit.Create ( m_wndHorzSplit, rcDefault );
//...

  // Set the horizontal splitter as the client area window.

  m_hWndClient = m_wndHorzSplit;

  // Set up the splitter panes

  m_wndPaneContainer.SetClient ( m_wndFormatList );
  m_wndHorzSplit.SetSplitterPane ( SPLIT_PANE_TOP, m_wndVertSplit );
  m_wndVertSplit.SetSplitterPanes ( m_wndPaneContainer, m_wndDataViewer );
//...

}
```

And here's what the result looks like:



Using ActiveX controls in a pane

Hosting an ActiveX control in a splitter pane is similar to hosting a control in a dialog. You create the control at runtime using CAxWindow methods, then assign the CAxWindow to a pane in the splitter. Here's how you would add a browser control to the bottom pane of the horizontal splitter:

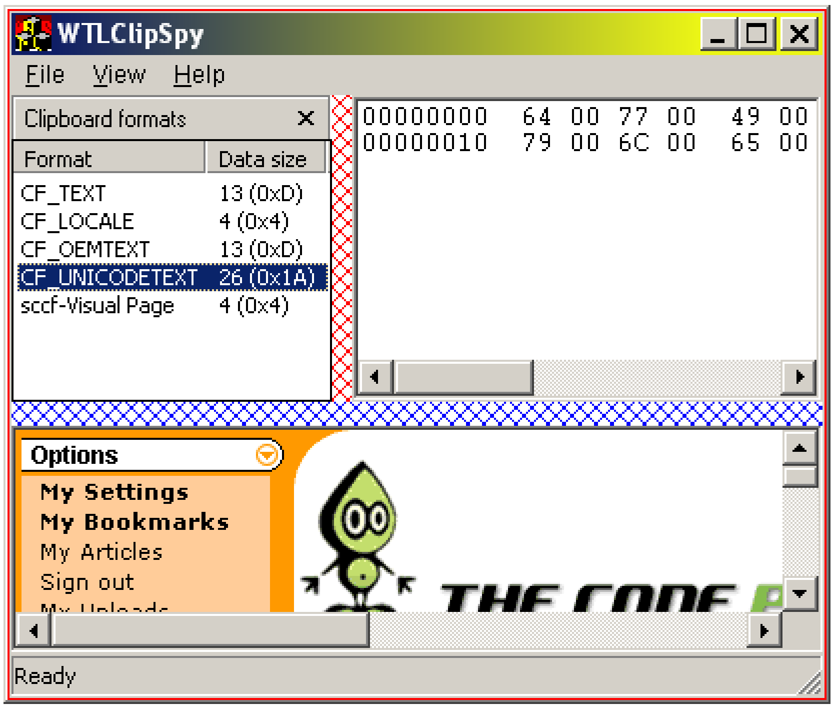
```
  // Create the bottom pane (browser)

CAxWindow wndIE;
DWORD dwIEStyle = WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN |
               WS_HSCROLL | WS_VSCROLL;

wndIE.Create ( m_wndHorzSplit, rcDefault,
            _T("http://www.codeproject.com"), dwIEStyle );

// Set the horizontal splitter as the client area window.

m_hWndClient = m_wndHorzSplit;

// Set up the splitter panes

m_wndPaneContainer.SetClient ( m_wndFormatList );
m_wndHorzSplit.SetSplitterPanes ( m_wndVertSplit, wndIE );
m_wndVertSplit.SetSplitterPanes ( m_wndPaneContainer, m_wndDataViewer );
```

Special drawing

If you want to provide a different appearance for the splitter bar, for example to draw a texture on it, you can derive a class from `CSplitterWindowImpl` and override `DrawSplitterBar()`. If you just want to tweak the appearance, you can copy the existing function in `CSplitterWindowImpl` and make any little changes you want. Here's an example that paints a diagonal hatch pattern in the bar.

```
template <bool t_bVertical = true>
class CMySplitterWindowT :
    public CSplitterWindowImpl<CMySplitterWindowT<t_bVertical>, t_bVertical>
{
public:
  DECLARE_WND_CLASS_EX(_T("My_SplitterWindow"),
                       CS_DBLCLKS, COLOR_WINDOW)

  // Overrideables

  void DrawSplitterBar(CDCHandle dc)
  {
  RECT rect;

    if ( m_br.IsNull() )
      m_br.CreateHatchBrush ( HS_DIAGCROSS,
                              t_bVertical ? RGB(255,0,0)
                                          : RGB(0,0,255) );

    if ( GetSplitterBarRect ( &rect ) )
      {
      dc.FillRect ( &rect, m_br );

      // draw 3D edge if needed

      if ( (GetExStyle() & WS_EX_CLIENTEDGE) != 0)
        {
        dc.DrawEdge(&rect, EDGE_RAISED,
                    t_bVertical ? (BF_LEFT | BF_RIGHT)
                                : (BF_TOP | BF_BOTTOM));
        }
      }
  }

protected:
  CBrush m_br;
};

typedef CMySplitterWindowT<true>  CMySplitterWindow;
typedef CMySplitterWindowT<false> CMyHorSplitterWindow;
```

Here's the result (with the bars made wider so the effect is easier to see):



```
Format          Data size
CF_TEXT         13 (0xD)
CF_LOCALE       4 (0x4)
CF_OEMTEXT      13 (0xD)
CF_UNICODETEXT  26 (0x1A)
sccf-Visual Page  4 (0x4)
```

```
00000000   64 00 77 00   49 00
00000010   79 00 6C 00   65 00
```

Special Drawing in Pane Containers

CPaneContainer has a few methods that you can override to change the appearance of a pane container. You can derive a new class from CPaneContainerImpl and override the methods you want, for example:

```
class CMyPaneContainer :
  public CPaneContainerImpl<CMyPaneContainer>
{
public:
  DECLARE_WND_CLASS_EX(_T("My_PaneContainer"), 0, -1)
//... overrides here ...

};
```

Some of the more interesting methods are:

```
void CalcSize()
```

The purpose of `CalcSize()` is simply to set `m_cxyHeader`, which controls the width or height of the container's header area. However, there is a bug in `SetPaneContainerExtendedStyle()` that results in a derived class's `CalcSize()` not being called when the pane is switched between horizontal and vertical modes. You can fix this by changing line 2215 in *atlctrlx.h* to call `pT->CalcSize()` instead of `CalcSize()`.

```
HFONT GetTitleFont()
```

This method returns an `HFONT`, which will be used to draw the header text. The default is the value returned by `GetStockObject(DEFAULT_GUI_FONT)`, which is MS Sans Serif. If you want to use the more modern-looking Tahoma, you can override `GetTitleFont()` and return a handle to a Tahoma font that you create.

```
BOOL GetToolTipText(LPNMHDR lpnmh)
```

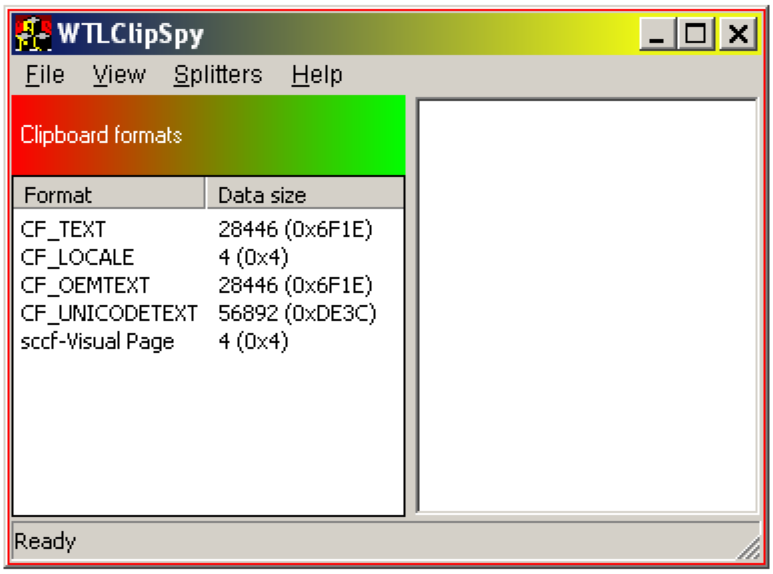
Override this method to provide tooltip text when the cursor hovers over the Close button. This method is actually a handler for `TTN_GETDISPINFO`, so you cast `lpnmh` to a `NMTTDISPINFO*` and set the members of that struct accordingly. Keep in mind that you have to check the notification code - it may be `TTN_GETDISPINFO` or `TTN_GETDISPINFOW` - and access the struct accordingly.

```
void DrawPaneTitle(CDCHandle dc)
```

You can override this method to provide your own drawing for the header area. You can use `GetClientRect()` and `m_cxyHeader` to calculate the `RECT` of the header area. Here is sample code to draw a gradient fill in the header area of a horizontal container:

```
void CMyPaneContainer::DrawPaneTitle ( CDCHandle dc )
{
RECT rect;

  GetClientRect(&rect);

TRIVERTEX tv[] =
  {
    { rect.left, rect.top, 0xff00 },
    { rect.right, rect.top + m_cxyHeader, 0, 0xff00 }
  };
GRADIENT_RECT gr = { 0, 1 };

  dc.GradientFill ( tv, 2, &gr, 1, GRADIENT_FILL_RECT_H );
}
```

The sample project demonstrates overriding some of these methods, and the result is shown here:



Format	Data size
CF_TEXT	28446 (0x6F1E)
CF_LOCALE	4 (0x4)
CF_OEMTEXT	28446 (0x6F1E)
CF_UNICODETEXT	56892 (0xDE3C)
sccf-Visual Page	4 (0x4)

The demo project has a *Splitters* menu, shown above, that lets you toggle various special drawing features of the splitters and pane containers, so you can see the differences. You can also lock the splitters, which is done by toggling on the `SPLIT_NONINTERACTIVE` extended style.

Bonus: Progress Bar in the Status Bar

As I promised a couple of articles ago, this new ClipSpy demonstrates how to create a progress bar in the status bar. It works just like the MFC version - the steps involved are:

1. Get the `RECT` of the first status bar pane
2. Create a progress bar control as a child of the status bar, with is `RECT` set to the `RECT` of the pane
3. Update the progress bar position as the edit control is being filled

You can check out the code in `CMainFrame::CreateProgressCtrlInStatusBar()`.

Up Next

In Part 8, I'll tackle the topic of property sheets and wizards.

References

WTL Splitters and Pane Containers by Ed Gadziemski

Part VIII - Property Sheets and Wizards

Introduction

Property sheets have been a popular way of presenting options, even before Windows 95 introduced the sheet as a common control. Wizards are often used to guide users through installing software or other complex tasks. WTL provides good support for creating both of these types of property sheets, and lets you use all of the dialog-related features that we've covered earlier, like DDX and DDV. In this article, I'll demonstrate creating a basic property sheet and a wizard, and how to handle events and notification messages sent by the sheet.

WTL Property Sheet Classes

There are two classes that combine to implement a property sheet, `CPropertySheetWindow` and `CPropertySheetImpl`. Both are defined in the *atldlgs.h* header file. `CPropertySheetWindow` is a window interface class (that is, it derives from `CWindow`), and `CPropertySheetImpl` has a message map and actually implements the window. This is similar to the basic ATL windowing classes, where `CWindow` and `CWindowImpl`are used together.

`CPropertySheetWindow` contains wrappers for the various `PSM_*` messages, such as `SetActivePageByID()` which wraps `PSM_SETCURSELID`. `CPropertySheetImpl` manages a `PROPSHEETHEADER`struct and an array of `HPROPSHEETPAGE`s. `CPropertySheetImpl` also has methods for setting some `PROPSHEETHEADER` fields, and adding and removing pages. You can access the `PROPSHEETHEADER`directly by accessing the `m_psh` member variable.

Finally, `CPropertySheet` is a specialization of `CPropertySheetImpl` that you can use if you don't need to customize the sheet at all.

CPropertySheetImpl methods

Here are some of the important methods of `CPropertySheetImpl`. Since many methods are just wrappers around window messages, I won't present an exhaustive list here, but you can check out *atldlgs.h* to see the complete list of methods.

```
CPropertySheetImpl(_U_STRINGorID title = (LPCTSTR) NULL,
                   UINT uStartPage = 0, HWND hWndParent = NULL)
```

The CPropertySheetImpl constructor lets you specify some common properties right away, so you don't have to call other methods later to set them. title specifies the text to use in the property sheet's caption. _U_STRINGorID is a WTL utility class that lets you pass either an LPCTSTR or string resource ID. For example, both of these lines will work:

```
  CPropertySheet mySheet ( IDS_SHEET_TITLE );
  CPropertySheet mySheet ( _T("My prop sheet") );
```

if IDS_SHEET_TITLE is the ID of a string in the string table. uStartPage is the zero-based index of the page that should be active when the sheet is first made visible. hWndParent sets the sheet's parent window.

```
BOOL AddPage(HPROPSHEETPAGE hPage)
BOOL AddPage(LPCPROPSHEETPAGE pPage)
```

Adds a property page to the sheet. If the page is already created, you can pass its handle (an HPROPSHEETPAGE) to the first overload. The more common way is to use the second overload. With that version, you set up a PROPSHEETPAGEstruct (which you can do with CPropertyPageImpl, covered later) and CPropertySheetImplwill create and manage the page for you.

```
BOOL RemovePage(HPROPSHEETPAGE hPage)
BOOL RemovePage(int nPageIndex)
```

Removes a page from the sheet. You can pass either the page's handle or its zero-based index.

```
BOOL SetActivePage(HPROPSHEETPAGE hPage)
BOOL SetActivePage(int nPageIndex)
```

Sets the active page in the sheet. You can pass either the handle or zero-based index of the page to be made active. You can call this method before showing the property sheet to set which page will be active when the sheet is first made visible.

```
void SetTitle(LPCTSTR lpszText, UINT nStyle = 0)
```

Sets the text to be used in the caption of the property sheet. nStyle can be either 0 or PSH_PROPTITLE. If it is PSH_PROPTITLE, then that style bit is added to the sheet, which causes the words "Properties for" to be prepended to the text you pass in lpszText.

```
void SetWizardMode()
```

Sets the PSH_WIZARD style, which changes the sheet into a wizard. You must call this method before showing the sheet.

```
void EnableHelp()
```

Sets the PSH_HASHELP style, which adds a Help button to the sheet. Note that you also need to enable help in each page that provides help for this to take effect.

```
INT_PTR DoModal(HWND hWndParent = ::GetActiveWindow())
```

Creates and shows a modal property sheet. The return value is positive to indicate success, see the docs on the PropertySheet() API for a full description of the return value. If an error occurs and the sheet can't be created, DoModal() returns -1.

```
HWND Create(HWND hWndParent = NULL)
```

Creates and shows a modeless property sheet, and returns its window handle. If an error occurs and the sheet can't be created, Create() returns NULL.

WTL Property Page Classes

The WTL classes that encapsulate property pages work similarly to the property sheet classes. There is a window interface class, CPropertyPageWindow, and an implementation class, CPropertyPageImpl.CPropertyPageWindow is very small, and contains mostly utility functions that call methods in the parent sheet.

CPropertyPageImpl derives from the ATL class CDialogImplBaseT, since a page is built from a dialog resource. That means that all the WTL features we've used in dialogs are also available in property sheets, like DDX and DDV. CPropertyPageImpl has two main purposes: it manages a PROPSHEETPAGEstruct (kept in the member variable m_psp), and handles PSN_* notification messages. For very simple property pages, you can use the CPropertyPage class. This is only suitable for pages that do not interact with the user at all, for example an *About* page, or the introduction page in a wizard.

You can also create pages that host ActiveX controls. You first include *atlhost.h* in *stdafx.h*. For the page, you use CAxPropertyPageImpl instead of

CPropertyPageImpl. For simple pages that host ActiveX controls, you can use CAxPropertyPage instead of CPropertyPage.

CPropertyPageWindow methods

CPropertyPageWindow's most important method is GetPropertySheet():

```
CPropertySheetWindow GetPropertySheet()
```

This method gets the HWND of the page's parent window (the sheet) and attaches a CPropertySheetWindowto that HWND. The new CPropertySheetWindow is then returned to the caller. Note that this only creates a temporary object; it does **not** return a pointer or reference to the actual CPropertySheetor CPropertySheetImpl object used to create the sheet. This is important if you are using your ownCPropertySheetImpl-derived class and need to access data members in the sheet object.

The remaining members just call through to CPropertySheetWindow functions that wrap PSM_*messages:

```
BOOL Apply()
void CancelToClose()
void SetModified(BOOL bChanged = TRUE)
LRESULT QuerySiblings(WPARAM wParam, LPARAM lParam)
void RebootSystem()
void RestartWindows()
void SetWizardButtons(DWORD dwFlags)
```

For example, in a CPropertyPageImpl-derived class, you can call:

```
  SetWizardButtons ( PSWIZB_BACK | PSWIZB_FINISH );
```

instead of:

```
CPropertySheetWindow wndSheet;

  wndSheet = GetPropertySheet();
  wndSheet.SetWizardButtons ( PSWIZB_BACK | PSWIZB_FINISH );
```

CPropertyPageImpl methods

CPropertyPageImpl manages a PROPSHEETPAGE struct, the public member m_psp.CPropertyPageImpl also has an operator PROPSHEETPAGE* converter, so you can pass a CPropertyPageImplto a method that takes an LPPROPSHEETPAGE or LPCPROPSHEETPAGE, such as CPropertySheetImpl::AddPage().

The CPropertyPageImpl constructor lets you set the page title, which is the text that appears on the page's tab:

```
CPropertyPageImpl(_U_STRINGorID title = (LPCTSTR) NULL)
```

If you ever need to create a page manually, instead of letting the sheet do it, you can call Create():

```
HPROPSHEETPAGE Create()
```

Create() just calls the Win32 API CreatePropertySheetPage() with m_pspas the parameter. You would only need to call Create() if you are adding pages to a sheet after the sheet is created, or if you are creating a page to be passed to some other sheet not under your control (for example, a property sheet handler shell extension).

There are three methods for setting various title text on the page:

```
void SetTitle(_U_STRINGorID title)
void SetHeaderTitle(LPCTSTR lpstrHeaderTitle)
void SetHeaderSubTitle(LPCTSTR lpstrHeaderSubTitle)
```

The first changes the text on the page's tab. The other two are used in Wizard97-style wizards to set the text in the header above the property page area.

```
void EnableHelp()
```

Sets the PSP_HASHELP flag in m_psp to enable the Help button when the page is active.

Handling notification messages

CPropertyPageImpl has a message map that handles WM_NOTIFY. If the notification code is a PSN_* value, OnNotify() calls a handler for that particular notification. This is done using the compile-time virtual function technique, so the handlers can be easily overridden in derived classes.

There are two sets of notification handlers, due to a design change between WTL 3 and 7. In WTL 3, the notification handlers had return values that differ from the return values for the PSN_* messages. For example, the WTL 3 handler for PSN_WIZFINISH is:

```
case PSN_WIZFINISH:
  lResult = !pT->OnWizardFinish();
break;
```

OnWizardFinish() was expected to return TRUE to let the wizard finish, or FALSEto prevent the wizard from closing. This code broke when the IE 5 common controls added the ability to return a window handle from the PSN_WIZFINISH handler to give that window the focus. WTL 3 apps could not use this feature because all non-zero values were treated the same.

In WTL 7, OnNotify() does not change any value returned from a PSN_* handler. The handlers can return any documented legal value and the behavior will be correct. However, for backwards compatibility, the WTL 3 handlers are still present and are used by default. To use the WTL 7 handlers, you must add this line to *stdafx.h* before including *atldlgs.h*:

```
#define _WTL_NEW_PAGE_NOTIFY_HANDLERS
```

When writing new code, there is no reason not to use the WTL 7 handlers, so the WTL 3 handlers will not be covered here.

CPropertyPageImpl has default handlers for all notifications, so you can override only the handlers that are relevant to your program. The default handlers and their actions are:

int OnSetActive() - allows the page to become active

BOOL OnKillActive() - allows the page to become inactive

int OnApply() - returns PSNRET_NOERROR to indicate the apply operation succeeded `void OnReset()` - no action

`BOOL OnQueryCancel()` - allows the cancel operation

`int OnWizardBack()` - goes to the previous page

`int OnWizardNext()` - goes to the next page

`INT_PTR OnWizardFinish()` - allows the wizard to finish

`void OnHelp()` - no action

`BOOL OnGetObject(LPNMOBJECTNOTIFY lpObjectNotify)` - no action

`int OnTranslateAccelerator(LPMSG lpMsg)` - returns `PSNRET_NOERROR` to indicate that the message was not handled

`HWND OnQueryInitialFocus(HWND hWndFocus)` - returns NULL to set the focus to the first control in the tab order

Creating a Property Sheet

Now that our tour of the classes is complete, we need a program to illustrate how to use them. The sample project for this article is a simple SDI app that shows a picture in the client area, and fills the background with a color. The picture and color can be changed via an options dialog (a property sheet), and a wizard (which I'll describe later).

The simplest property sheet, ever

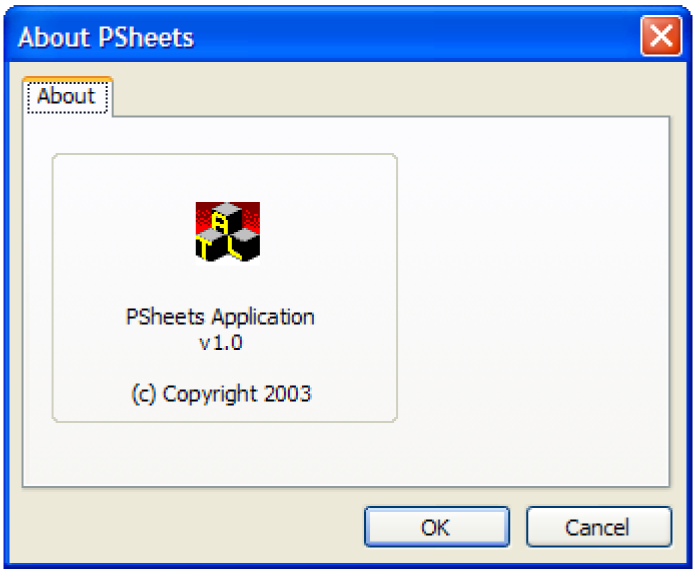
After making an SDI project with the WTL AppWizard, we can start by creating a sheet to use for our about box. Let's start with the *About* dialog the wizard creates for us, and change the styles so it will work as a property page.

The first step is to remove the OK button, since that doesn't make sense in a sheet. In the dialog's properties, change the *Style* to *Child*, the *Border* to *Thin*, and set *Disabled* to checked.

The second (and final) step is to create a property sheet in the `OnAppAbout()` handler. We can do this with `CPropertySheet` and `CPropertyPage`:

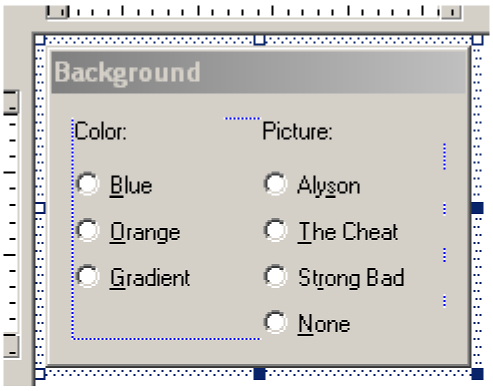
```
void CMainFrame::OnAppAbout(...)
{
CPropertySheet sheet ( _T("About PSheets") );
CPropertyPage<IDD_ABOUTBOX> pgAbout;

  sheet.AddPage ( pgAbout );
  sheet.DoModal ( *this );
}
```

The result looks like this:



Creating a useful property page

Since not every page in every sheet is as simple as an about box, most pages will require a `CPropertyPageImpl`-derived class, so we'll take a look at such a class now. We'll make a new property page that contains the settings for the graphics shown in the background of the client area. Here's the dialog:



This dialog has the same styles as the *About* page. We'll need a new class to go along with the page, called `CBackgroundOptsPage`. This class derives from

CPropertyPageImpl since it's a property page, and CWinDataExchange to enable DDX.

```
class CBackgroundOptsPage :
  public CPropertyPageImpl<CBackgroundOptsPage>,
  public CWinDataExchange<CBackgroundOptsPage>
{
public:
  enum { IDD = IDD_BACKGROUND_OPTS };

  // Construction

  CBackgroundOptsPage();
  ~CBackgroundOptsPage();

  // Maps

  BEGIN_MSG_MAP(CBackgroundOptsPage)
    MSG_WM_INITDIALOG(OnInitDialog)
    CHAIN_MSG_MAP(CPropertyPageImpl<CBackgroundOptsPage>)
  END_MSG_MAP()

  BEGIN_DDX_MAP(CBackgroundOptsPage)
    DDX_RADIO(IDC_BLUE, m_nColor)
    DDX_RADIO(IDC_ALYSON, m_nPicture)
  END_DDX_MAP()

  // Message handlers

  BOOL OnInitDialog ( HWND hwndFocus, LPARAM lParam );

  // Property page notification handlers

  int OnApply();

  // DDX variables

  int m_nColor, m_nPicture;
};
```

Things to note in this class:

- There is a public member named IDD that holds the associated dialog resource ID.

- The message map resembles that of a `CDialogImpl` class.
- The message map chains messages to `CPropertyPageImpl` so that property sheet-related notification messages are handled.
- There is an `OnApply()` handler to save the user's choices when he clicks OK in the sheet.

`OnApply()` is pretty simple, it calls `DoDataExchange()` to update the DDX variables, then returns a code indicating whether the sheet should close or not:

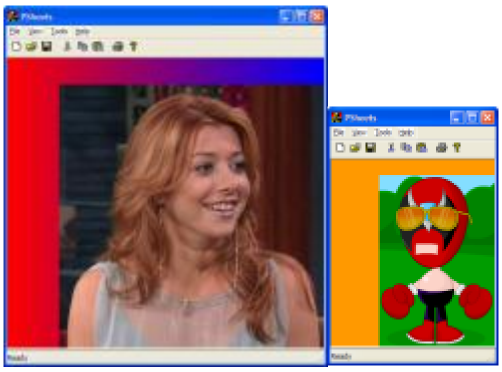
```
int CBackgroundOptsPage::OnApply()
{
  return DoDataExchange(true) ? PSNRET_NOERROR : PSNRET_INVALID;
}
```

We'll add a *Tools|Options* menu item that brings up the property sheet, and put add handler for this command to the view class. The handler creates the property sheet as before, but with the new `CBackgroundOptsPage`added to the sheet.

```
void CPSheetsView::OnOptions ( UINT uCode, int nID, HWND hwndCtrl )
{
CPropertySheet sheet ( _T("PSheets Options"), 0 );
CBackgroundOptsPage pgBackground;
CPropertyPage<IDD_ABOUTBOX> pgAbout;

  pgBackground.m_nColor = m_nColor;
  pgBackground.m_nPicture = m_nPicture;

  sheet.m_psh.dwFlags |= PSH_NOAPPLYNOW|PSH_NOCONTEXTHELP;

  sheet.AddPage ( pgBackground );
  sheet.AddPage ( pgAbout );

  if ( IDOK == sheet.DoModal() )
    SetBackgroundOptions ( pgBackground.m_nColor,
                           pgBackground.m_nPicture );
}
```

The `sheet` constructor now has a second parameter of 0, meaning that the page at index 0 should be visible initially. You could change that to 1 to have the *About* page be visible when the sheet first appears. Since this is just demo code, I'm going to be lazy and make the `CBackgroundOptsPage` variables connected to the radio buttons public. The view stores the current options in those variables, and saves the new values if the user clicks OK in the sheet.

If the user clicks OK, DoModal() returns IDOK, and the view redraws itself using the new picture and color. Here are some screen shots to show the different views:



Creating a better property sheet class

The OnOptions() handler creates the sheet just fine, but there's an awful lot of setup and initialization code there, which really shouldn't be CMainFrame's responsibility. A better way is to make a class derived from CPropertySheetImpl that handles those tasks.

```
#include "BackgroundOptsPage.h"


class COptionsSheet : public CPropertySheetImpl<COptionsSheet>
{
public:
  // Construction

  COptionsSheet(_U_STRINGorID title = (LPCTSTR) NULL,
                UINT uStartPage = 0, HWND hWndParent = NULL);

  // Maps

  BEGIN_MSG_MAP(COptionsSheet)
    CHAIN_MSG_MAP(CPropertySheetImpl<COptionsSheet>)
  END_MSG_MAP()

  // Property pages

  CBackgroundOptsPage         m_pgBackground;
  CPropertyPage<IDD_ABOUTBOX> m_pgAbout;
};
```

With this class, we've encapsulated the details of what pages are in the sheet, and moved them into the sheet class itself. The constructor handles adding the pages to the sheet, and setting any other necessary flags:

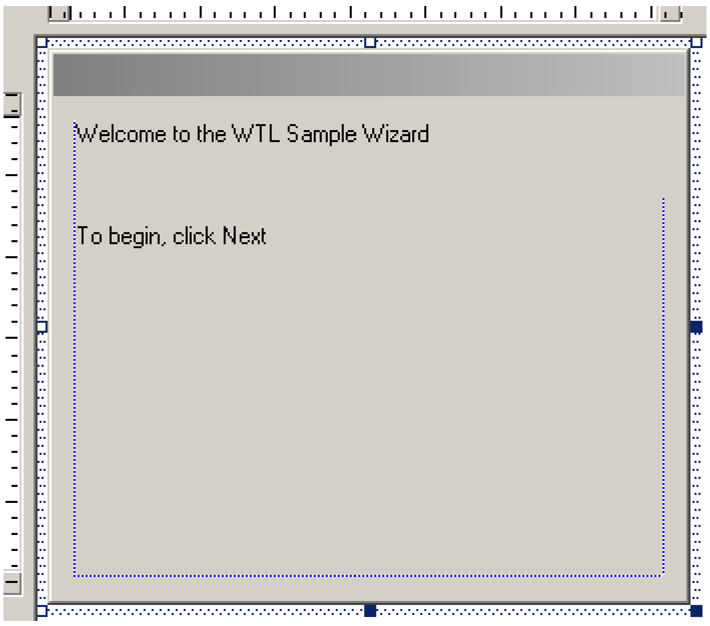
```
COptionsSheet::COptionsSheet (
  _U_STRINGorID title, UINT uStartPage, HWND hWndParent ) :
    CPropertySheetImpl<COptionsSheet>(title, uStartPage, hWndParent)
{
  m_psh.dwFlags |= PSH_NOAPPLYNOW|PSH_NOCONTEXTHELP;

  AddPage ( m_pgBackground );
  AddPage ( m_pgAbout );
}
```

As a result, the `OnOptions()` handler becomes a bit simpler:

```
void CPSheetsView::OnOptions ( UINT uCode, int nID, HWND hwndCtrl )
{
COptionsSheet sheet ( _T("PSheets Options"), 0 );

  sheet.m_pgBackground.m_nColor = m_nColor;
  sheet.m_pgBackground.m_nPicture = m_nPicture;

  if ( IDOK == sheet.DoModal() )
    SetBackgroundOptions ( sheet.m_pgBackground.m_nColor,
                           sheet.m_pgBackground.m_nPicture );
}
```

Creating a Wizard

Creating a wizard is, not surprisingly, similar to creating a property sheet. A little more work is required to enable the *Back* and *Next* buttons; just as in MFC property pages, you override `OnSetActive()` and call `SetWizardButtons()` to enable the appropriate buttons. We'll start with a simple introduction page, with ID `IDD_WIZARD_INTRO`:


Welcome to the WTL Sample Wizard

To begin, click Next

Notice that the page has no caption text. Since every page in a wizard usually has the same title, I prefer to set the text in the CPropertySheetImpl constructor, and have every page use the same string resource. That way, I can just change that one string and every page will reflect the change.

The implementation of this page is done in the `CWizIntroPage` class:

```
class CWizIntroPage : public CPropertyPageImpl<CWizIntroPage>
{
public:
  enum { IDD = IDD_WIZARD_INTRO };

  // Construction

  CWizIntroPage();

  // Maps

  BEGIN_MSG_MAP(COptionsWizard)
    CHAIN_MSG_MAP(CPropertyPageImpl<CWizIntroPage>)
  END_MSG_MAP()

  // Notification handlers

  int OnSetActive();
};
```

The constructor sets the page's text by referencing a string resource ID:

```
CWizIntroPage::CWizIntroPage() :
  CPropertyPageImpl<CWizIntroPage>(IDS_WIZARD_TITLE)
{
}
```

The string IDS_WIZARD_TITLE ("PSheets Options Wizard") will appear in the wizard's caption bar when this page is the current page. OnSetActive() enables just the *Next* button:

```
int CWizIntroPage::OnSetActive()
{
  SetWizardButtons ( PSWIZB_NEXT );
  return 0;
}
```

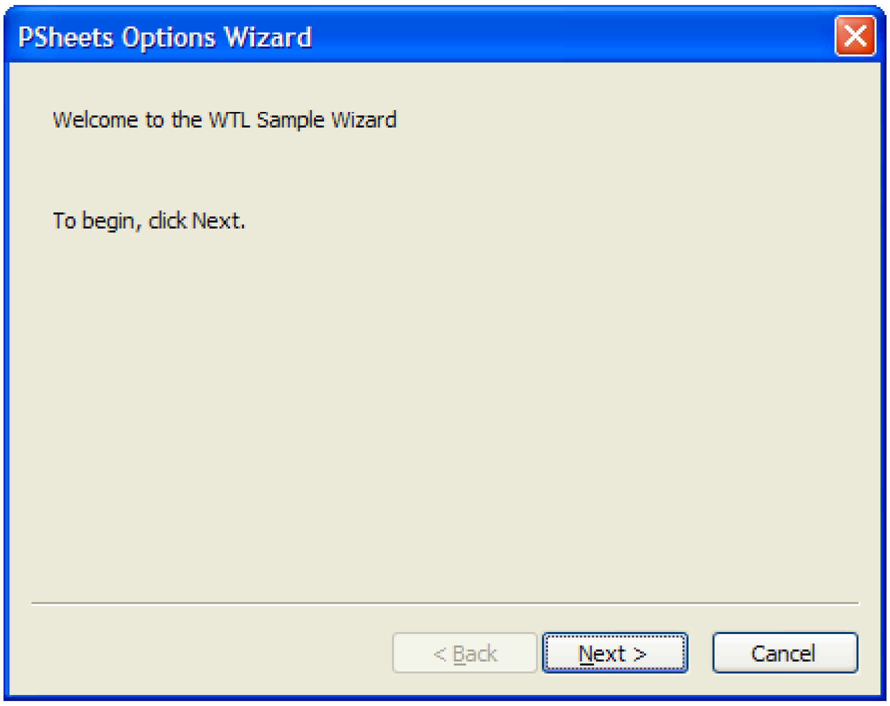
To implement the wizard, we'll create a class COptionsWizard, and a *Tools|Wizard*menu option in the app's menu. The COptionsWizard constructor is pretty similar to COptionsSheet's, in that it sets any necessary style bits or flags, and adds pages to the sheet.

```
class COptionsWizard : public CPropertySheetImpl<COptionsWizard>
{
public:
  // Construction

  COptionsWizard ( HWND hWndParent = NULL );

  // Maps

  BEGIN_MSG_MAP(COptionsWizard)
    CHAIN_MSG_MAP(CPropertySheetImpl<COptionsWizard>)
  END_MSG_MAP()

  // Property pages

  CWizIntroPage m_pgIntro;
};

COptionsWizard::COptionsWizard ( HWND hWndParent ) :
  CPropertySheetImpl<COptionsWizard> ( 0U, 0, hWndParent )
{
  SetWizardMode();
  AddPage ( m_pgIntro );
}
```

Then the handler for the *Tools|Wizard* menu looks like this:

```
void CPSheetsView::OnOptionsWizard ( UINT uCode, int nID, HWND hwndCtrl )
{
COptionsWizard wizard;

  wizard.DoModal ( GetTopLevelParent() );
}
```

And here's the wizard in action:

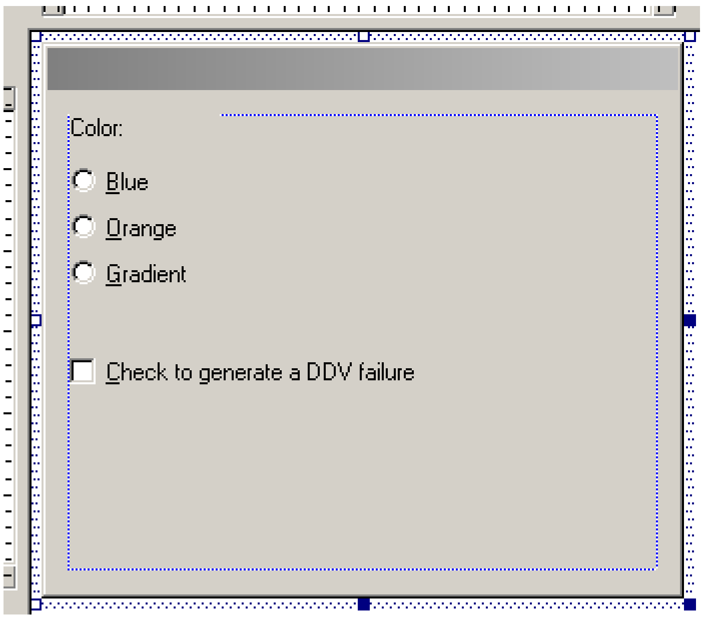


Welcome to the WTL Sample Wizard

To begin, click Next.

Adding More Pages, Handling DDV

To make this a useful wizard, we'll add a new page for setting the view's background color. This page will also have a checkbox for demonstrating handling a DDV failure and preventing the user from continuing on with the wizard. Here's the new page, whose ID is `IDD_WIZARD_BKCOLOR`:



The implementation for this page is in the class CWizBkColorPage. Here are the parts relating to

```
class CWizBkColorPage :
  public CPropertyPageImpl<CWizBkColorPage>,
  public CWinDataExchange<CWizBkColorPage>
{
public:
    //...

  BEGIN_DDX_MAP(CWizBkColorPage)
    DDX_RADIO(IDC_BLUE, m_nColor)
    DDX_CHECK(IDC_FAIL_DDV, m_bFailDDV)
  END_DDX_MAP()

  // Notification handlers

  int OnSetActive();
  BOOL OnKillActive();

  // DDX vars

  int m_nColor;

protected:
  bool m_bFailDDV;
};
```

`OnSetActive()` works similarly to the intro page, it enables both the *Back* and *Next*buttons. `OnKillActive()` is a new handler, it invokes DDX then checks the value of `m_bFailDDV`. If that variable is `true`, meaning the checkbox was checked, `OnKillActive()` prevents the wizard from going to the next page.

```
int CWizBkColorPage::OnSetActive()
{
  SetWizardButtons ( PSWIZB_BACK | PSWIZB_NEXT );
  return 0;
}

int CWizBkColorPage::OnKillActive()
{
  if ( !DoDataExchange(true) )
    return TRUE;    // prevent deactivation


  if ( m_bFailDDV )
    {
    MessageBox (
        _T("Error box checked, wizard will stay on this page."),
        _T("PSheets"), MB_ICONERROR );

    return TRUE;    // prevent deactivation

    }

  return FALSE;       // allow deactivation

}
```

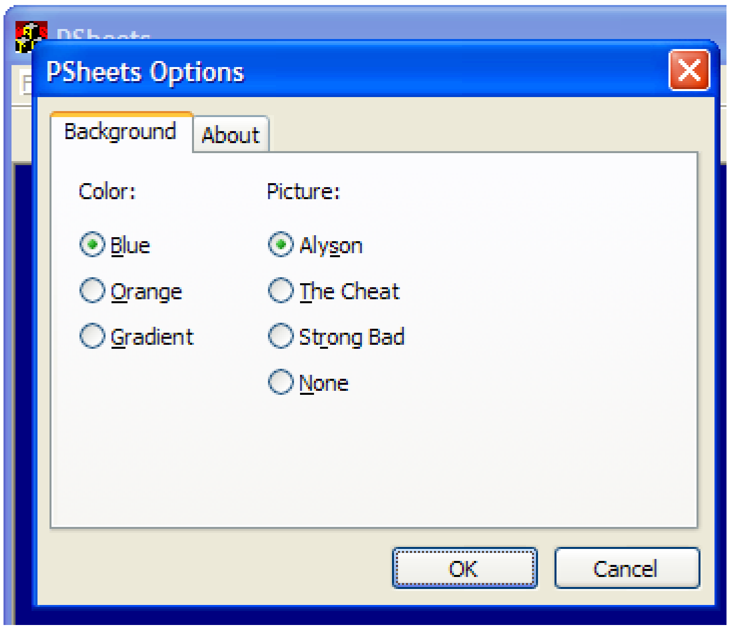
Note that the logic in `OnKillActive()` could certainly have been put in `OnWizardNext()`instead, since both handlers have the ability to keep the wizard on the current page. The difference is that `OnKillActive()`is called when the user clicks *Back* or *Next*, while `OnWizardNext()` is only called when the user clicks *Next* (as the name signifies). `OnWizardNext()` is also used for other purposes; it can direct the wizard to a different page instead of the next one in order, if some pages need to be skipped.

The wizard in the sample project has two more pages, `CWizBkPicturePage` and `CWizFinishPage`. Since they are similar to the two pages above, I won't cover them in detail here, but you can check out the sample code to get all the details.

Other UI Considerations

Centering a sheet

The default behavior of sheets and wizards is to appear near the upper-left corner of their parent window:



This looks rather sloppy, but fortunately we can remedy it. Thanks to the folks on the forum who provided the code to do this; the previous version of the article did it in a much more complicated way.

The property sheet or wizard class can handle the WM_SHOWWINDOW message. The wParamsent with WM_SHOWWINDOW is a boolean indicating whether the window is being shown. If wParamis true, and it's the first time the window is being shown, then it can call CenterWindow().

Here is the code that we can add to COptionsSheet to center the sheet. The m_bCenteredmember is how we keep track of whether the sheet has already been centered.

```
class COptionsSheet : public CPropertySheetImpl<COptionsSheet>
{
//...

  BEGIN_MSG_MAP(COptionsSheet)
    MSG_WM_SHOWWINDOW(OnShowWindow)
    CHAIN_MSG_MAP(CPropertySheetImpl<COptionsSheet>)
  END_MSG_MAP()

  // Message handlers

  void OnShowWindow(BOOL bShowing, int nReason);

protected:
  bool m_bCentered;  // set to false in the ctor

};

void COptionsSheet::OnShowWindow(BOOL bShowing, int nReason)
{
  if ( bShowing && !m_bCentered )
    {
    m_bCentered = true;
    CenterWindow ( m_psh.hwndParent );
    }
}
```

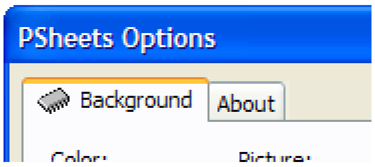
Adding icons to pages

To use other features of sheets and pages that are not already wrapped by member functions, you'll need to access the relevant structures directly: the `PROPSHEETHEADER` member `m_psh` in `CPropertySheetImpl`, or the `PROPSHEETPAGE` member `m_psp` in `CPropertyPageImpl`.

For example, to add an icon to the *Background* page of the options property sheet, we need to add a flag and set a couple other members in the page's `PROPSHEETPAGE` struct:

```
CBackgroundOptsPage::CBackgroundOptsPage()
{
  m_psp.dwFlags |= PSP_USEICONID;
  m_psp.pszIcon = MAKEINTRESOURCE(IDI_TABICON);
  m_psp.hInstance = _Module.GetResourceInstance();
}
```

And here's the result:



Up Next

In Part 9, I'll cover WTL's utility classes, and its wrappers for GDI object and common dialogs.

Part IX - GDI Classes, Common Dialogs, and Utility Classes

Introduction

WTL contains many wrappers and utility classes that haven't gotten full coverage yet in this series, such as`CString` and `CDC`. **WTL** has a nice system for wrapping GDI objects, some useful functions for loading resources, and classes that make it easier to use some of the Win32 common dialogs. Here in **Part** IX, I'll cover some of the most commonly-used utility classes.

This article discusses four categories of features:

1. GDI wrapper classes
2. Resource-loading functions
3. Using the file-open and choose-folder common dialogs
4. Other useful classes and global functions

GDI Wrapper Classes

WTL takes a rather different approach to its GDI wrappers than **MFC**. **WTL**'s approach is to have one template class for each type of GDI object, each template having one `bool` parameter called `t_bManaged`. This parameter controls whether an instance of that class "manages" (or owns) the wrapped GDI object. If `t_bManaged` is `false`, the C++ object does not manage the lifetime of the GDI object; the C++ object is a simple wrapper around the GDI object handle. If `t_bManaged` is `true`, two things change:

1. The destructor calls `DeleteObject()` on the wrapped handle, if it is not NULL.
2. `Attach()` calls `DeleteObject()` on the wrapped handle, if it is not NULL, before attaching the C++ object to the new handle.

This design is in line with the ATL window classes, where `CWindow` is a plain wrapper around an`HWND`, and `CWindowImpl` manages the lifetime of a window.

The GDI wrapper classes are defined in *atlgdi.h*, with the exception of `CMenuT`, which is in *atluser.h*. You don't have to include these headers yourself, because *atlapp.h* always includes them for you. Each class also has typedefs with easier-to-remember names:

Wrapped GDI object	Template class	Typedef for managed object	Typedef for plain wrapper
Pen	`CPenT`	`CPen`	`CPenHandle`
Brush	`CBrushT`	`CBrush`	`CBrushHandle`
Font	`CFontT`	`CFont`	`CFontHandle`
Bitmap	`CBitmapT`	`CBitmap`	`CBitmapHandle`
Palette	`CPaletteT`	`CPalette`	`CPaletteHandle`
Region	`CRgnT`	`CRgn`	`CRgnHandle`
Device context	`CDCT`	`CDC`	`CDCHandle`
Menu	`CMenuT`	`CMenu`	`CMenuHandle`

I like this approach, compared to **MFC** which passes around pointers to objects. You never have to worry about getting a NULL pointer (the wrapped handle might be NULL, but that's another matter), nor do you have any special cases where you get a temporary object that you can't hang on to for more than one function call. It is also very cheap to create an instance of any of these classes since they only have one member variable, the handle being wrapped. As is the case with `CWindow`, there is no problem passing a wrapper class object between threads, since **WTL** keeps no thread-specific maps like **MFC**.

There are additional device context wrapper classes for use in special drawing scenarios:

- `CClientDC`: Wraps calls to `GetDC()` and `ReleaseDC()`, used to draw in a window's client area
- `CWindowDC`: Wraps calls to `GetWindowDC()` and `ReleaseDC()`, used to draw anywhere in a window.
- `CPaintDC`: Wraps calls to `BeginPaint()` and `EndPaint()`, used in a `WM_PAINT` handler.

Each of these classes takes a `HWND` in the constructor, and behaves like the **MFC** classes of the same name. All three are derived from `CDC`, so these classes all manage their device contexts.

Common functions in the wrapper classes

The GDI wrapper classes follow the same design. To be concise, I'll cover the methods in `CBitmapT`here, but the other classes work similarly.

The wrapped GDI object handle

Each class keeps one public member variable that holds the GDI object handle that the C++ object is associated with. `CBitmapT` has an `HBITMAP` member called `m_hBitmap`.

Constructor

The constructor has one parameter, an `HBITMAP`, which defaults to NULL. `m_hBitmap` is initialized to this value.

Destructor

If `t_bManaged` is true, and `m_hBitmap` is not NULL, then the destructor calls `DeleteObject()` to destroy the bitmap.

`Attach()` and `operator =`

These methods both take an `HBITMAP` handle. If `t_bManaged` is `true`, and `m_hBitmap` is not NULL, these methods call `DeleteObject()` to destroy the bitmap that the `CBitmapT` object is managing. Then they set `m_hBitmap` to the `HBITMAP` that was passed in as the parameter.

`Detach()`

`Detach()` sets `m_hBitmap` to NULL, then returns the value that was in `m_hBitmap`. After `Detach()` returns, the `CBitmapT` object is no longer associated with a GDI bitmap.

Methods for creating a GDI object

`CBitmapT` has wrappers for the Win32 APIs that create a bitmap: `LoadBitmap()`, `LoadMappedBitmap()`, `CreateBitmap()`, `CreateBitmapIndirect()`, `CreateCompatibleBitmap()`, `CreateDiscardableBitmap()`, `CreateDIBitmap()`, and `CreateDIBSection()`. These methods will assert if `m_hBitmap` is not NULL; to reuse a `CBitmapT` object for a different GDI bitmap, call `DeleteObject()` or `Detach()` first.

`DeleteObject()`

`DeleteObject()` destroys the GDI bitmap object, then sets `m_hBitmap` to NULL. This method should be called only if `m_hBitmap` is not NULL; it will assert otherwise.

`IsNull()`

`IsNull()` returns `true` if `m_hBitmap` is NULL, or `false` otherwise. Use this method to test whether the `CBitmapT` object is currently associated with a GDI bitmap.

`operator HBITMAP`

This converter returns `m_hBitmap`, and lets you pass a `CBitmapT` object to a function or Win32 API that takes an `HBITMAP` handle. This converter is also called when a `CBitmapT` is evaluated in a boolean context, and evaluates to the logical opposite of `IsNull()`. Therefore, these two if statements are equivalent:

```
CBitmapHandle bmp = /* some HBITMAP value */;

if ( !bmp.IsNull() ) { do something... }
if ( bmp ) { do something more... }
```

GetObject() wrappers

CBitmapT has a type-safe wrapper for the Win32 API GetObject(): GetBitmap(). There are two overloads: one that takes a LOGBITMAP* and calls straight through to GetObject(); and one that takes a LOGBITMAP& and returns a bool indicating success. The latter version is the easier one to use. For example:

```
CBitmapHandle bmp2 = /* some HBITMAP value */;
LOGBITMAP logbmp = {0};
bool bSuccess;

if ( bmp2 )
  bSuccess = bmp2.GetLogBitmap ( logbmp );
```

Wrappers for APIs that operate on the GDI object

CBitmapT has wrappers for Win32 APIs that take an HBITMAP parameter: GetBitmapBits(), SetBitmapBits(), GetBitmapDimension(), SetBitmapDimension(), GetDIBits(), and SetDIBits(). These methods will assert if m_hBitmap is NULL.

Other utility methods

CBitmapT has two useful methods that operate on m_hBitmap: LoadOEMBitmap() and GetSize().

Using CDCT

CDCT is a bit different from the other classes, so I'll cover the differences separately.

Differences in methods

The method to destroy a DC is called DeleteDC() instead of DeleteObject().

Selecting objects into a DC

One aspect of **MFC**'s CDC that is prone to errors is selecting objects into a DC. **MFC**'s CDChas several overloaded SelectObject() functions that each take a pointer to a different kind of GDI wrapper class (CPen*, CBitmap*, and so on). If you pass a C++ object to SelectObject(), instead of a pointer to a C++ object, the code ends up calling the undocumented overload that accepts an `HGDIOBJ`handle, and this is what causes the problems.

WTL's `CDCT` takes a better approach, and has several select methods, each of which works with just one type of GDI object:

```
HPEN SelectPen(HPEN hPen)
HBRUSH SelectBrush(HBRUSH hBrush)
HFONT SelectFont(HFONT hFont)
HBITMAP SelectBitmap(HBITMAP hBitmap)
int SelectRgn(HRGN hRgn)
HPALETTE SelectPalette(HPALETTE hPalette, BOOL bForceBackground)
```

In debug builds, each method asserts that `m_hDC` is not NULL, and that the parameter is a handle to the correct type of GDI object. They then call the `SelectObject()` API and cast the `SelectObject()`return value to the appropriate type.

There are also helper methods that call `GetStockObject()` with a given constant, and then select the object into the DC:

```
HPEN SelectStockPen(int nPen)
HBRUSH SelectStockBrush(int nBrush)
HFONT SelectStockFont(int nFont)
HPALETTE SelectStockPalette(int nPalette, BOOL bForceBackground)
```

Differences from the MFC wrapper classes

Fewer constructors: The wrappers classes lack constructors that create a new GDI object. For example,**MFC**'s `CBrush` has constructors that create a solid or patterned brush. With the **WTL** classes, you must use a method to create the GDI object.

Selecting objects into a DC is done better: See the *Using CDCT* section above.

No `m_hAttribDC`: **WTL**'s `CDCT` does not have a `m_hAttribDC`member.

Minor parameter differences in some methods: For example, `CDC::GetWindowExt()` returns a`CSize` object in **MFC**; while in **WTL** the method returns a `bool`, and the size is returned via an output parameter.

Resource-Loading Functions

WTL has several global functions that are helpful shortcuts for loading various types of resources. We'll need to know about one utility class before getting on to the functions: `_U_STRINGorID`.

In Win32, most types of resources can be identified by a string (`LPCTSTR`) or an unsigned integer (`UINT`). APIs that take a resource identifier take an `LPCTSTR` parameter, and if you want to pass a `UINT`, you need to use the `MAKEINTRESOURCE` macro to convert it to an `LPCTSTR`.`_U_STRINGorID`, when used as the type of a resource identifier parameter, hides this distinction so that the caller can pass either a `UINT` or `LPCTSTR` directly. The function can then use a `CString` to load the string if necessary:

```
void somefunc ( _U_STRINGorID id )
{
CString str ( id.m_lpstr );

  // use str...

}

void func2()
{
  // Call 1 - using a string literal

  somefunc ( _T("Willow Rosenberg") );

  // Call 2 - using a string resource ID

  somefunc ( IDS_BUFFY_SUMMERS );
}
```

This works because the `CString` constructor that takes an `LPCTSTR` checks whether the parameter is actually a string ID. If so, the string is loaded from the string table and assigned to the `CString`.

In VC 6, `_U_STRINGorID` is provided by **WTL** in *atlwinx.h*. In VC 7, `_U_STRINGorID`is **part** of ATL. Either way, the class definition will always be included for you by other ATL/**WTL** headers.

The functions in this section load a resource from the resource instance handle kept in the `_Module`global variable (in VC 6) or the `_AtlBaseModule` global (in VC 7).

Using other modules for resources is beyond the scope of this article, so I will not be covering it here. Just remember that by default, the functions look in the EXE or DLL that the code is running in. The functions do nothing more than call through to APIs, their utility is in the simplified resource identifier handling provided by `_U_STRINGorID`.

```
HACCEL AtlLoadAccelerators(_U_STRINGorID table)
```

Calls through to `LoadAccelerators()`.

```
HMENU AtlLoadMenu(_U_STRINGorID menu)
```

Calls through to `LoadMenu()`.

```
HBITMAP AtlLoadBitmap(_U_STRINGorID bitmap)
```

Calls through to `LoadBitmap()`.

```
HCURSOR AtlLoadCursor(_U_STRINGorID cursor)
```

Calls through to `LoadCursor()`.

```
HICON AtlLoadIcon(_U_STRINGorID icon)
```

Calls through to `LoadIcon()`. Note that this function - like `LoadIcon()` - can only load 32x32 icons.

```
int AtlLoadString(UINT uID, LPTSTR lpBuffer, int nBufferMax)
bool AtlLoadString(UINT uID, BSTR& bstrText)
```

Call through to `LoadString()`. The string can be returned in either a `TCHAR` buffer, or assigned to a `BSTR`, depending on which overload you use. Note that these functions only accept a `UINT` as the resource ID, because string table entries cannot have string identifiers.

This group of functions wrap calls to `LoadImage()`, and take additional parameters that are passed on to `LoadImage()`.

```
HBITMAP AtlLoadBitmapImage(
        _U_STRINGorID bitmap, UINT fuLoad = LR_DEFAULTCOLOR)
```

Calls LoadImage() with the IMAGE_BITMAP type, passing along the fuLoad flags.

```
HCURSOR AtlLoadCursorImage(
        _U_STRINGorID cursor,
        UINT fuLoad = LR_DEFAULTCOLOR | LR_DEFAULTSIZE,
        int cxDesired = 0, int cyDesired = 0)
```

Calls LoadImage() with the IMAGE_CURSOR type, passing along the fuLoad flags. Since one cursor resource can contain several different-sized cursors, you can pass dimensions for the cxDesired and cyDesired parameters to load a cursor with a **part**icular size.

```
HICON AtlLoadIconImage(
       _U_STRINGorID icon,
       UINT fuLoad = LR_DEFAULTCOLOR | LR_DEFAULTSIZE,
       int cxDesired = 0, int cyDesired = 0)
```

Calls LoadImage() with the IMAGE_ICON type, passing along the fuLoad flags. The cxDesired and cyDesired parameters are used as in AtlLoadCursorImage().

This group of functions wrap calls to load system-defined resources (for example, the standard hand cursor). Some of these resource IDs (mostly the ones for bitmaps) are not included by default; you need to #definethe OEMRESOURCE symbol in your *stdafx.h* in order to reference them.

```
HBITMAP AtlLoadSysBitmap(LPCTSTR lpBitmapName)
```

Calls LoadBitmap() with a NULL resource handle. Use this function to load any of the OBM_* bitmaps listed in the LoadBitmap() documentation.

```
HCURSOR AtlLoadSysCursor(LPCTSTR lpCursorName)
```

Calls LoadCursor() with a NULL resource handle. Use this function to load any of the IDC_* cursors listed in the LoadCursor() documentation.

```
HICON AtlLoadSysIcon(LPCTSTR lpIconName)
```

Calls LoadIcon() with a NULL resource handle. Use this function to load any of the IDI_* icons listed in the LoadIcon() documentation. Note that this function - like LoadIcon() - can only load 32x32 icons.

```
HBITMAP AtlLoadSysBitmapImage(
        WORD wBitmapID, UINT fuLoad = LR_DEFAULTCOLOR)
```

Calls LoadImage() with a NULL resource handle and the IMAGE_BITMAP type. You can use this function to load the same bitmaps as AtlLoadSysBitmap().

```
HCURSOR AtlLoadSysCursorImage(
        _U_STRINGorID cursor,
        UINT fuLoad = LR_DEFAULTCOLOR | LR_DEFAULTSIZE,
        int cxDesired = 0, int cyDesired = 0)
```

Calls LoadImage() with a NULL resource handle and the IMAGE_CURSOR type. You can use this function to load the same cursors as AtlLoadSysCursor().

```
HICON AtlLoadSysIconImage(
       _U_STRINGorID icon,
       UINT fuLoad = LR_DEFAULTCOLOR | LR_DEFAULTSIZE,
       int cxDesired = 0, int cyDesired = 0)
```

Calls LoadImage() with a NULL resource handle and the IMAGE_ICON type. You can use this function to load the same icons as AtlLoadSysIcon(), but you can also specify a different size such as 16x16.

Finally, this group of functions are type-safe wrappers for the GetStockObject() API.

```
HPEN AtlGetStockPen(int nPen)
HBRUSH AtlGetStockBrush(int nBrush)
HFONT AtlGetStockFont(int nFont)
HPALETTE AtlGetStockPalette(int nPalette)
```

Each function checks that you're passing in a sensible value (e.g., AtlGetStockPen() only acceptsWHITE_PEN, BLACK_PEN, and so on), then calls through to GetStockObject().

Using Common Dialogs

WTL has classes that make using the Win32 common dialogs easier. Each class handles messages and callbacks that the common dialog sends, and in turn calls overridable functions. This is the same design used in property sheets, where you write handlers for individual property sheet notifications (e.g., OnWizardNext()

for handling`PSN_WIZNEXT`) that are called by `CPropertyPageImpl` when necessary.

WTL contains two classes for each common dialog; for example, the Choose Folder dialog is wrapped by `CFolderDialogImpl`and `CFolderDialog`. If you need to change any defaults or write handlers for any messages, you derive a new class from `CFolderDialogImpl` and make the changes in that class. If the default behavior of`CFolderDialogImpl` is sufficient, you can use `CFolderDialog`.

The common dialogs and their corresponding **WTL** classes are:

Common dialog	**Corresponding Win32 API**	**Implementation class**	**Non-customizable class**
File Open and File Save	`GetOpenFileName()`, `GetSaveFileName()`	`CFileDialogImpl`	`CFileDialog`
Choose Folder	`SHBrowseForFolder()`	`CFolderDialogImpl`	`CFolderDialog`
Choose Font	`ChooseFont()`	`CFontDialogImpl`, `CRichEditFontDialogImpl`	`CFontDialog`, `CRichEditFontDialog`
Choose Color	`ChooseColor()`	`CColorDialogImpl`	`CColorDialog`
Printing and Print Setup	`PrintDlg()`	`CPrintDialogImpl`	`CPrintDialog`
Printing (Windows 2000 and later)	`PrintDlgEx()`	`CPrintDialogExImpl`	`CPrintDialogEx`
Page Setup	`PageSetupDlg()`	`CPageSetupDialogImpl`	`CPageSetupDialog`
Text find and replace	`FindText()`, `ReplaceText()`	`CFindReplaceDialogImpl`	`CFindReplaceDialog`

Since writing about all those classes would make this article far too long, I'll cover just the first two, which are the ones you'll likely use most often.

CFileDialog

CFileDialog, and its base CFileDialogImpl, are used to show File Open and File Save dialogs. The two most important data members in CFileDialogImpl are m_ofn and m_szFileName.m_ofn is an OPENFILENAME that CFileDialogImpl sets up for you with some meaningful default values; just as in **MFC**, you can change the data in this struct directly if necessary. m_szFileNameis a TCHAR array that holds the name of the selected file. (Since CFileDialogImpl only has this one string for holding a filename, you'll need to provide your own buffer when you use a multiple-select open file dialog.)

The basic steps in using a CFileDialog are:

1. Construct a CFileDialog object, passing any initial data to the constructor.
2. Call DoModal().
3. If DoModal() returns IDOK, get the selected file from m_szFileName.

Here is the CFileDialog constructor:

```
CFileDialog::CFileDialog (
    BOOL bOpenFileDialog,
    LPCTSTR lpszDefExt = NULL,
    LPCTSTR lpszFileName = NULL,
    DWORD dwFlags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
    LPCTSTR lpszFilter = NULL,
    HWND hWndParent = NULL )
```

bOpenFileDialog should be true to create a File-Open dialog (CFileDialogwill call GetOpenFileName() to show the dialog), or false to create a File-Save dialog (CFileDialog will call GetSaveFileName()). The remaining parameters are stored directly in the appropriate members of the m_ofn struct, but they are optional since you can access m_ofndirectly before calling DoModal().

A significant difference between **MFC**'s CFileDialog is that the lpszFilter parameter must be a null-character-delimited string list (that is, the format documented in the OPENFILENAMEdocs), instead of a pipe-separated list.

Here is an example of using a CFileDialog with a filter that selects Word 12 files (*.docx):

```
CString sSelectedFile;
CFileDialog fileDlg ( true, _T("docx"), NULL,
                      OFN_HIDEREADONLY | OFN_FILEMUSTEXIST,
                    _T("Word 12 Files\0*.docx\0All Files\0*.*\0") );

  if ( IDOK == fileDlg.DoModal() )
    sSelectedFile = fileDlg.m_szFileName;
```

CFileDialog isn't very localization-friendly, since the constructor uses LPCTSTR parameters. That filter string is also a bit hard to read at first glance. There are two solutions, either set up m_ofnbefore calling DoModal(), or derive a new class from CFileDialogImpl that has the improvements we want. We'll take the second approach here, and make a new class that has the following changes:

1. The string parameters in the constructor are _U_STRINGorID instead of LPCTSTR.
2. The filter string can use pipes to separate the fields, as in **MFC**, instead of null characters.
3. The dialog will be automatically centered relative to its parent window.

We'll start by writing a class whose constructor takes parameters similar to the CFileDialogImplconstructor:

```
class CMyFileDialog : public CFileDialogImpl<CMyFileDialog>
{
public:
  // Construction

  CMyFileDialog ( BOOL bOpenFileDialog,
                  _U_STRINGorID szDefExt = 0U,
                  _U_STRINGorID szFileName = 0U,
                  DWORD dwFlags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
                  _U_STRINGorID szFilter = 0U,
                  HWND hwndParent = NULL );

protected:
  LPCTSTR PrepFilterString ( CString& sFilter );
  CString m_sDefExt, m_sFileName, m_sFilter;
};
```

The constructor initializes the three CString members, loading strings if necessary:

```
CMyFileDialog::CMyFileDialog (
  BOOL bOpenFileDialog, _U_STRINGorID szDefExt, _U_STRINGorID szFileName,
  DWORD dwFlags, _U_STRINGorID szFilter, HWND hwndParent ) :
    CFileDialogImpl<CMyFileDialog>(bOpenFileDialog, NULL, NULL, dwFlags,
                                   NULL, hwndParent),
    m_sDefExt(szDefExt.m_lpstr), m_sFileName(szFileName.m_lpstr),
    m_sFilter(szFilter.m_lpstr)
{
}
```

Note that the string parameters are all NULL in the call to the base class constructor. This is because the base class constructor is always called before member initializers. To set up the string data in `m_ofn`, we add some code that duplicates the initialization steps that the `CFileDialogImpl` constructor would do:

```
CMyFileDialog::CMyFileDialog(...)
{
  m_ofn.lpstrDefExt = m_sDefExt;
  m_ofn.lpstrFilter = PrepFilterString ( m_sFilter );

  // setup initial file name

  if ( !m_sFileName.IsEmpty() )
    lstrcpyn ( m_szFileName, m_sFileName, _MAX_PATH );
}
```

`PrepFilterString()` is a helper method that takes a pipe-delimited filter string, changes the pipes to null characters, and returns a pointer to the beginning of the string. The result is a string list that's in the proper format for use in an `OPENFILENAME`.

```
LPCTSTR CMyFileDialog::PrepFilterString(CString& sFilter)
{
LPTSTR psz = sFilter.GetBuffer(0);
LPCTSTR pszRet = psz;

  while ( '\0' != *psz )
    {
    if ( '|' == *psz )
      *psz++ = '\0';
    else
      psz = CharNext ( psz );
    }

  return pszRet;
}
```

Those changes make the string-handling easier. To implement automatic centering, we'll override the OnInitDone() notification. This requires us to add a message map (so we can chain notification messages to the base class), and our OnInitDone() handler:

```
class CMyFileDialog : public CFileDialogImpl<CMyFileDialog>
{
public:
  // Construction

  CMyFileDialog(...);

  // Maps

  BEGIN_MSG_MAP(CMyFileDialog)
    CHAIN_MSG_MAP(CFileDialogImpl<CMyFileDialog>)
  END_MSG_MAP()

  // Overrides

  void OnInitDone ( LPOFNOTIFY lpon )
  {
    GetFileDialogWindow().CenterWindow(lpon->lpOFN->hwndOwner);
  }

protected:
    LPCTSTR PrepFilterString ( CString& sFilter );
    CString m_sDefExt, m_sFileName, m_sFilter;
};
```

The window attached to the CMyFileDialog object is actually a child of the File Open dialog. Since we need the top-most window in the stack, we call GetFileDialogWindow() to get that window.

CFolderDialog

CFolderDialog, and its base CFolderDialogImpl, are used to show a Browse For Folder dialog. While the dialog supports browsing anywhere within the shell namespace, CFolderDialog is only capable of browsing within the file system. The two most important data members in CFolderDialogImpl are m_bi and m_szFolderPath. m_bi is an BROWSEINFO that CFolderDialogImpl manages and passes to the SHBrowseForFolder() API; you can change the data in this struct directly if necessary. m_szFolderPath is a TCHAR array that holds the name of the selected folder.

The basic steps in using a CFolderDialog are:

1. Construct a CFolderDialog object, passing any initial data to the constructor.
2. Call DoModal().
3. If DoModal() returns IDOK, get the path to the selected folder from m_szFolderPath.

Here is the CFolderDialog constructor:

```
CFolderDialog::CFolderDialog (
    HWND hWndParent = NULL,
    LPCTSTR lpstrTitle = NULL,
    UINT uFlags = BIF_RETURNONLYFSDIRS )
```

hWndParent is the owner window for the browse dialog. You can either set it here in the constructor, or in the DoModal() call. lpstrTitle is a string that will be shown above the tree control in the dialog. uFlags are flags that control the dialog's behavior, and should always include BIF_RETURNONLYFSDIRSso the tree only shows file system directories. Other values for uFlags that you can use are listed in the docs for BROWSEINFO, but remember that some flags may not produce good results, such as BIF_BROWSEFORPRINTER. UI-related flags like BIF_USENEWUI will work fine. Note that the lpstrTitle parameter has the same usability problems as the strings in the CFileDialog constructor.

Here is an example of selecting a directory using CFolderDialog:

```
CString sSelectedDir;
CFolderDialog fldDlg ( NULL, _T("Select a dir"),
                       BIF_RETURNONLYFSDIRS|BIF_NEWDIALOGSTYLE );

  if ( IDOK == fldDlg.DoModal() )
    sSelectedDir = fldDlg.m_szFolderPath;
```

To demonstrate customizing CFolderDialog, we'll derive a class from CFolderDialogImpland set the initial selection. This dialog's callbacks don't use window messages, so the class doesn't need a message map. Instead, we override the OnInitialized() method, which gets called when the base class receives the BFFM_INITIALIZED notification. OnInitialized() calls CFolderDialogImpl::SetSelection()to change the selection in the dialog.

```
class CMyFolderDialog : public CFolderDialogImpl<CMyFolderDialog>
{
public:
  // Construction

  CMyFolderDialog ( HWND hWndParent = NULL,
                   _U_STRINGorID szTitle = 0U,
                   UINT uFlags = BIF_RETURNONLYFSDIRS ) :
     CFolderDialogImpl<CMyFolderDialog>(hWndParent, NULL, uFlags),
     m_sTitle(szTitle.m_lpstr)
  {
        m_bi.lpszTitle = m_sTitle;
  }

  // Overrides

  void OnInitialized()
  {
    // Set the initial selection to the Windows dir.

    TCHAR szWinDir[MAX_PATH];

    GetWindowsDirectory ( szWinDir, MAX_PATH );
    SetSelection ( szWinDir );
  }

protected:
  CString m_sTitle;
};
```

Other useful classes and global functions

Struct wrappers

WTL has the classes `CSize`, `CPoint`, and `CRect`, that wrap the `SIZE`,`POINT`, and `RECT` structs respectively. They work like their **MFC** counter**part**s.

Classes for handling dual-typed arguments

As mentioned earlier, you can use the _U_STRINGorID type for a function parameter that can be a numeric or string resource ID. There are two other classes that work similarly:

- _U_MENUorID: This type can be constructed from a UINT or HMENU, and is meant to be used in CreateWindow() wrappers. The hMenu parameter to CreateWindow() is actually a window ID when the window being created is a child window, so _U_MENUorID hides the distinction between the two usages. _U_MENUorID has one member m_hMenu, which can be passed as the hMenu parameter to CreateWindow() or CreateWindowEx().
- _U_RECT: This type can be constructed from a LPRECT or RECT&, and lets the caller pass in a RECT struct, pointer to a RECT, or a wrapper class like CRect that provides a converter to RECT.

As with _U_STRINGorID, _U_MENUorID and _U_RECT are always included for you by other headers.

Other utility classes

CString

WTL's CString works just like **MFC**'s CString, so I won't be covering it in detail here.**WTL**'s CString has many extra methods that are used when you build with _ATL_MIN_CRT defined. These methods, like _cstrchr(), _cstrstr(), are replacements for the corresponding CRT functions, which aren't available when _ATL_MIN_CRT is defined.

CFindFile

CFindFile wraps the FindFirstFile() and FindNextFile() APIs, and is a bit easier to use than **MFC**'s CFileFind. The general pattern of usage goes like this:

```
CFindFile finder;
CString sPattern = _T("C:\\windows\\*.exe");

  if ( finder.FindFirstFile ( sPattern ) )
    {
    do
      {
      // act on the file that was found
      }
    while ( finder.FindNextFile() );
    }

  finder.Close();
```

If `FindFirstFile()` returns `true`, at least one file matched the pattern. Inside the`do` loop, you can access the public `CFindFile` member `m_fd`, which is a `WIN32_FIND_DATA`struct that holds the info about the file that was found. The loop continues until `FindNextFile()`returns `false`, indicating that all files have been enumerated.

`CFindFile` has methods that return the data from `m_fd` in easier-to-use forms. These methods return meaningful values only after a successful call to `FindFirstFile()` or `FindNextFile()`.

```
ULONGLONG GetFileSize()
```

Returns the file size as a 64-bit unsigned integer.

```
BOOL GetFileName(LPTSTR lpstrFileName, int cchLength)
CString GetFileName()
```

Returns the filename and extension of the file that was found (copied from `m_fd.cFileName`).

```
BOOL GetFilePath(LPTSTR lpstrFilePath, int cchLength)
CString GetFilePath()
```

Returns the full path to the file that was found.

```
BOOL GetFileTitle(LPTSTR lpstrFileTitle, int cchLength)
CString GetFileTitle()
```

Returns just the file title (that is, the filename with no extension) of the file that was found.

```
BOOL GetFileURL(LPTSTR lpstrFileURL, int cchLength)
CString GetFileURL()
```

Creates a `file://` URL that contains the full path to the file.

```
BOOL GetRoot(LPTSTR lpstrRoot, int cchLength)
CString GetRoot()
```

Returns the directory that contains the file.

```
BOOL GetLastWriteTime(FILETIME* pTimeStamp)
BOOL GetLastAccessTime(FILETIME* pTimeStamp)
BOOL GetCreationTime(FILETIME* pTimeStamp)
```

These methods return the `ftLastWriteTime`, `ftLastAccessTime`, and `ftCreationTime`members from `m_fd` respectively.

`CFindFile` also has some helper methods for checking the attributes of the file that was found.

```
BOOL IsDots()
```

Returns `true` if the found file is the "`.`" or "`..`" directory.

```
BOOL MatchesMask(DWORD dwMask)
```

Compares the bits in `dwMask` (which should be the `FILE_ATTRIBUTE_*` constants) with the attributes of the file that was found. Returns `true` if all the bits that are on in `dwMask`are also on in the file's attributes.

```
BOOL IsReadOnly()
BOOL IsDirectory()
BOOL IsCompressed()
BOOL IsSystem()
BOOL IsHidden()
BOOL IsTemporary()
BOOL IsNormal()
BOOL IsArchived()
```

These methods are shortcuts that call `MatchesMask()` with a **part**icular `FILE_ATTRIBUTE_*`bit. For example, `IsReadOnly()` calls `MatchesMask(FILE_ATTRIBUTE_READONLY)`.

Global functions

WTL has several useful global functions that you can use to do things like DLL version checks and show message boxes.

```
bool AtlIsOldWindows()
```

Returns true if the operating system is Windows 95, 98, NT 3, or NT 4.

```
HFONT AtlGetDefaultGuiFont()
```

Returns the value of `GetStockObject(DEFAULT_GUI_FONT)`. In English Windows 2000 and later (and other single-byte languages that use the Latin alphabet), this font's face name is "MS Shell Dlg". This is usable as a dialog box font, but not the best choice if you are creating your own fonts for use in your UI. MS Shell Dlg is an alias for MS Sans Serif, instead of the new UI font, Tahoma. To avoid getting MS Sans Serif, you can get the font used for message boxes with this code:

```
NONCLIENTMETRICS ncm = { sizeof(NONCLIENTMETRICS) };
CFont font;

  if ( SystemParametersInfo ( SPI_GETNONCLIENTMETRICS, 0, &ncm, false ) )
    font.CreateFontIndirect ( &ncm.lfMessageFont );
```

An alternative is to check the face name of the font returned by `AtlGetDefaultGuiFont()`. If the name is "MS Shell Dlg", you can change it to "MS Shell Dlg 2", an alias that resolves to Tahoma.

```
HFONT AtlCreateBoldFont(HFONT hFont = NULL)
```

Creates a bold version of a given font. If `hFont` is NULL, `AtlCreateBoldFont()` creates a bold version of the font returned by `AtlGetDefaultGuiFont()`.

```
BOOL AtlInitCommonControls(DWORD dwFlags)
```

This is a wrapper for the `InitCommonControlsEx()` API. It initializes an `INITCOMMONCONTROLSEX`struct with the given flags, then calls the API.

```
HRESULT AtlGetDllVersion(HINSTANCE hInstDLL, DLLVERSIONINFO* pDllVersionInfo)
HRESULT AtlGetDllVersion(LPCTSTR lpstrDllName, DLLVERSIONINFO* pDllVersionInfo)
```

These functions look in a given module for an exported function called `DllGetVersion()`. If the function is found, it is called. If `DllGetVersion()` is successful, it returns the version information in a `DLLVERSIONINFO` struct.

```
HRESULT AtlGetCommCtrlVersion(LPDWORD pdwMajor, LPDWORD pdwMinor)
```

Returns the major and minor versions of comctl32.dll.

```
HRESULT AtlGetShellVersion(LPDWORD pdwMajor, LPDWORD pdwMinor)
```

Returns the major and minor versions of shell32.dll.

```
bool AtlCompactPath(LPTSTR lpstrOut, LPCTSTR lpstrIn, int cchLen)
```

Truncates a file path so it is less than `cchLen` characters in length, adding an ellipsis at the end if the path is too long. This works similarly to the `PathCompactPath()` and `PathSetDlgItemPath()` functions in shlwapi.dll.

```
int AtlMessageBox(HWND hWndOwner, _U_STRINGorID message,
               _U_STRINGorID title = NULL,
               UINT uType = MB_OK | MB_ICONINFORMATION)
```

Displays a message box, like `MessageBox()`, but uses `_U_STRINGorID` parameters so you can pass string resource IDs. `AtlMessageBox()` handles loading the strings if necessary.

Macros

There are various preprocessor macros that you'll see referenced in the **WTL** header files. Most of these macros can be set in the compiler settings to change behavior in the **WTL** code.

These macros are predefined or set by build settings, you'll see them referenced throughout the **WTL** code:

`_WTL_VER`

Defined as `0x0710` for **WTL** 7.1.

`_ATL_MIN_CRT`

If defined, ATL does not link to the C runtime library. Since some **WTL** classes (notably `CString`) normally use CRT functions, special code is compiled that replaces the code that would normally be imported from the CRT.

_ATL_VER

Predefined as `0x0300` for VC 6, `0x0700` for VC 7, and `0x0800` for VC 8.

_WIN32_WCE

Defined if the current compilation is for a Windows CE binary. Some **WTL** code is disabled when the corresponding features are not available in CE.

The following macros are not defined by default. To use a macro, `#define` it before all `#include`statements in *stdafx.h*.

_ATL_NO_OLD_NAMES

This macro is only useful if you are maintaining **WTL** 3 code. It adds some compiler directives to recognize two old class names: `CUpdateUIObject` becomes `CIdleHandler`, and `DoUpdate()` becomes `OnIdle()`.

_ATL_USE_CSTRING_FLOAT

Define this symbol to enable floating-point support in `CString`; `_ATL_MIN_CRT` must **not** also be defined. You need to define this symbol if you plan to use the `%I64` prefix in a format string that you pass to `CString::Format()`. Defining `_ATL_USE_CSTRING_FLOAT` results in `CString::Format()` calling `_vstprintf()`, which understands the `%I64` prefix.

_ATL_USE_DDX_FLOAT

Define this symbol to enable floating-point support in the DDX code; `_ATL_MIN_CRT` must **not** also be defined.

_ATL_NO_MSIMG

Define this symbol to prevent the compiler from seeing a `#pragma comment(lib, "msimg32")` line; also disables code in `CDCT` that uses msimg32 functions: `AlphaBlend()`, `TransparentBlt()`, `GradientFill()`.

_ATL_NO_OPENGL

Define this symbol to prevent the compiler from seeing a `#pragma comment(lib, "opengl32")` line; also disables code in `CDCT` that uses OpenGL.

_WTL_FORWARD_DECLARE_CSTRING

Obsolete, use `_`**WTL**`_USE_CSTRING` instead.

_WTL_USE_CSTRING

Define this symbol to forward-declare `CString`. This way, code in headers that are normally included before *atlmisc.h* will be able to use `CString`.

_WTL_NO_CSTRING

Define this symbol to prevent usage of `WTL::CString`.

`_WTL_NO_AUTOMATIC_NAMESPACE`

Define this symbol to prevent automatic execution of a `using namespace WTL` directive.

`_WTL_NO_AUTO_THEME`

Define this symbol to prevent `CMDICommandBarCtrlImpl` from using XP themes.

`_WTL_NEW_PAGE_NOTIFY_HANDLERS`

Define this symbol to use newer `PSN_*` notification handlers in `CPropertyPage`. Since the old **WTL** 3 handlers are obsolete, this symbol should always be defined unless you are maintaining **WTL** 3 code that can't be updated.

`_WTL_NO_WTYPES`

Define this symbol to prevent the **WTL** versions of `CSize`, `CPoint`, and `CRect` from being defined.

`_WTL_NO_THEME_DELAYLOAD`

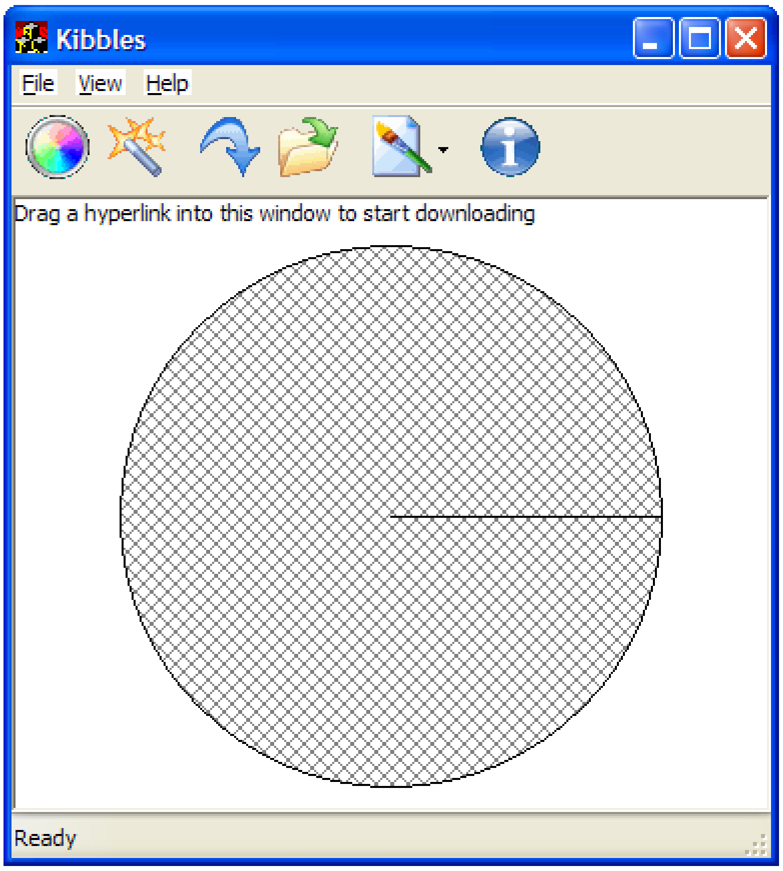
When building with VC 6, define this symbol to prevent *uxtheme.dll* from being automatically marked as a delay-load DLL.

NOTE: If neither `_WTL_USE_CSTRING` nor `_WTL_NO_CSTRING` is defined, then `CString` can be used at any point after *atlmisc.h* is included.

The Sample Project

The demo project for this article is a downloader application called Kibbles that demonstrates the various classes that have been covered in this article. It uses the BITS (background intelligent transfer service) component that you can get for Windows 2000 and later; since this app only runs on NT-based OSes, I also made it a Unicode project.

The app has a view window that shows the download progress, using various GDI calls including `Pie()` which draws the pie chart shapes. When the app is first run, you'll see the UI in its initial state:

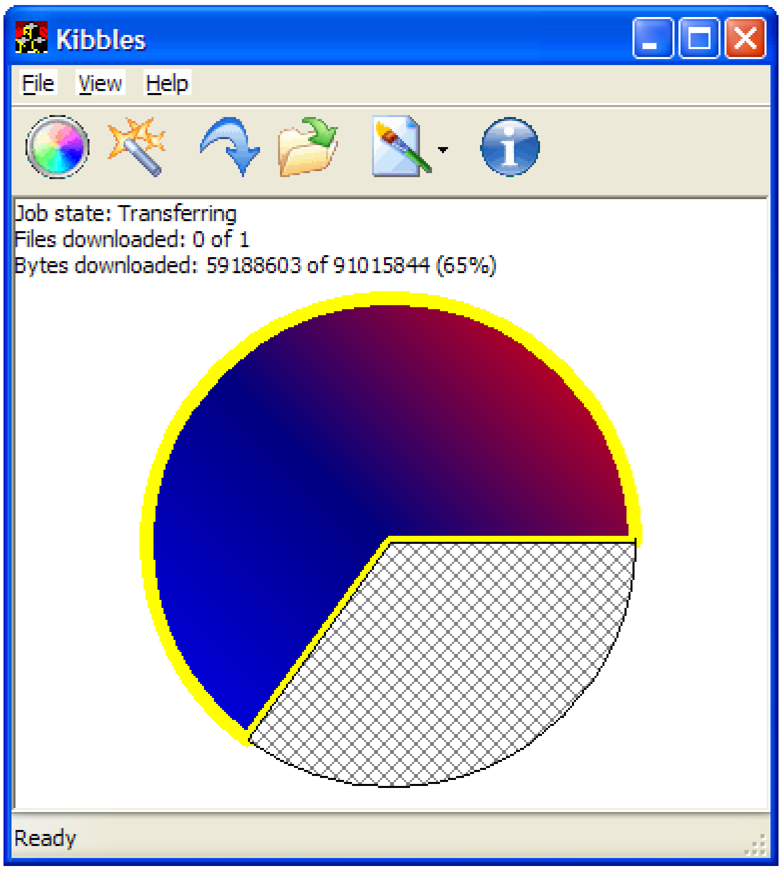


Drag a hyperlink into this window to start downloading

Ready

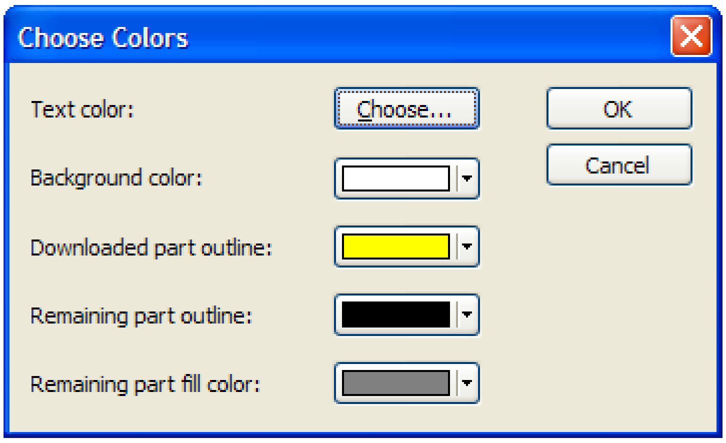
You can drag a link from a browser into the window to create a new BITS job that will download the target of the link to your My Documents folder. You can also click the third toolbar button to add any URL that you want to the job. The fourth button lets you change the default download directory.

When a download job is in progress, Kibbles shows some details about the job, and shows the download progress like so:



Job state: Transferring
Files downloaded: 0 of 1
Bytes downloaded: 59188603 of 91015844 (65%)

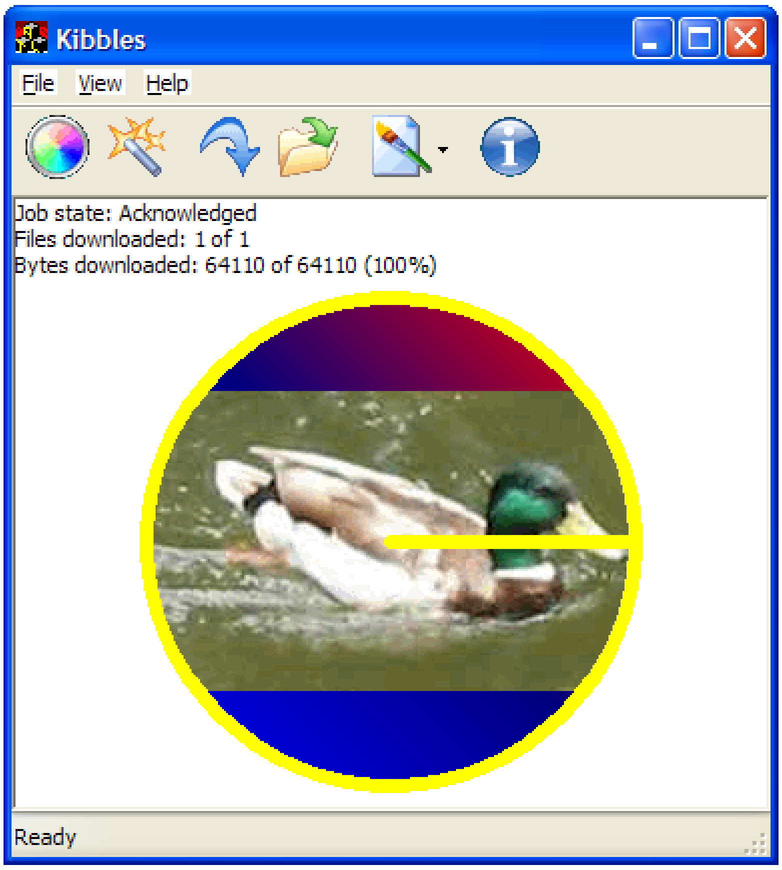
The first two buttons in the toolbar let you change the colors used in the progress display. The first button opens an options dialog where you can set the colors used for various **part**s of the display:



Text color:
Background color:
Downloaded part outline:
Remaining part outline:
Remaining part fill color:

The dialog uses the great button class from Tim Smith's article Color Picker for **WTL** with XP themes; check out the `CChooseColorsDlg` class in the Kibbles project to see it in action. The *Text color* button is a regular button, and the `OnChooseTextColor()` handler demonstrates how to use the **WTL** class `CColorDialog`. The second toolbar button changes all the colors to random values.

The fifth button lets you set a background picture, which will be drawn in the **part** of the pie that shows how much has been downloaded. The default picture is included as a resource, but if you have any BMP files in your My Pictures directory, you can select one of those as well.



Job state: Acknowledged
Files downloaded: 1 of 1
Bytes downloaded: 64110 of 64110 (100%)

`CMainFrame::OnToolbarDropdown()` contains the code that handles the button press event and shows a popup menu. That function also uses `CFindFile` to enumerate the contents of the My Pictures directory. You can check out `CKibblesView::OnPaint()` to see the code that does the various GDI operations that draw the UI.

An important note about the toolbar: The toolbar uses a 256-color bitmap, however the VC toolbar editor only works with 16-color bitmaps. If you ever edit the toolbar using the editor, VC will reduce the bitmap to 16 colors. What I suggest is keeping a high-color version of the bitmap in a separate directory, making changes to it directly using a graphics program, then saving a 256-color version in the `res` directory.

Part X - Implementing a Drag and Drop Source

Introduction

Drag and drop is a feature of many modern applications. While implementing a drop target is rather straightforward, the drop source is much more complicated. MFC has the classes `COleDataObject` and `COleDropSource`that assist in managing the data that the source must provide, but WTL has no such helper classes. Fortunately for us WTL users, Raymond Chen wrote an MSDN article ("The Shell Drag/Drop Helper Object Part 2") back in 2000 that has a plain C++ implementation of `IDataObject`, which is a huge help in writing a complete drag and drop source for a WTL app.

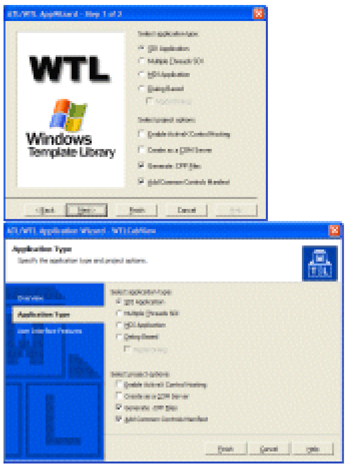
This article's sample project is a CAB file viewer that lets you extract files from a CAB by dragging them from the viewer to an Explorer window. The article will also discuss some new frame window topics such as File-Open handling and data management analogous to the document/view framework in MFC. I'll also demonstrate WTL's MRU (most-recently-used) file list class, and some new UI features in the version 6 list view control.

Important: You will need to download and install the CAB SDK from Microsoft to compile the sample code. A link to the SDK is in KB article Q310618. The sample project assumes the SDK is in a directory called "cabsdk" that is in the same directory as the source files.

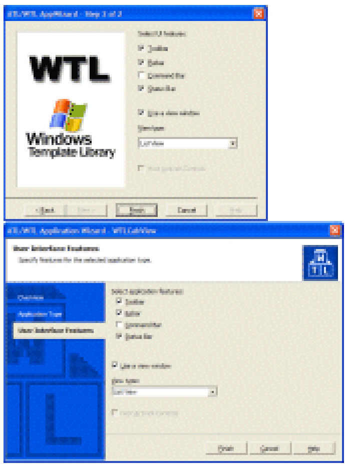
Remember, if you encounter any problems installing WTL or compiling the demo code, read the readme section of Part I before posting questions here.

Starting the Project

To begin our CAB viewer app, run the WTL AppWizard and create a project called *WTLCabView*. It will be an SDI app, so choose SDI Application on the first page:



On the next page, uncheck *Command Bar*, and change the *View Type* to *List View*. The wizard will create a C++ class for our view window, and it will derive from `CListViewCtrl`.



The view window class looks like this:

```
class CWTLCabViewView :
  public CWindowImpl<CWTLCabViewView, CListViewCtrl>
{
public:
  DECLARE_WND_SUPERCLASS(NULL, CListViewCtrl::GetWndClassName())

  // Construction

  CWTLCabViewView();

  // Maps

  BEGIN_MSG_MAP(CWTLCabViewView)
  END_MSG_MAP()

  // ...

};
```

As with the view window we used in Part II, we can set the default window styles using the third template parameter of `CWindowImpl`:

```
#define VIEW_STYLES \
  (LVS_REPORT | LVS_SHOWSELALWAYS | \
   LVS_SHAREIMAGELISTS | LVS_AUTOARRANGE )
#define VIEW_EX_STYLES (WS_EX_CLIENTEDGE)

class CWTLCabViewView :
  public CWindowImpl<CWTLCabViewView, CListViewCtrl,
                      CWinTraitsOR<VIEW_STYLES,VIEW_EX_STYLES> >
{
//...

};
```

Since there is no document/view framework in WTL, the view class will do double-duty as both the UI and the place where information about the CAB is held. The data structure passed around during a drag and drop operation is called `CDraggedFileInfo`:

```
struct CDraggedFileInfo
{
  // Data set at the beginning of a drag/drop:

  CString sFilename;      // name of the file as stored in the CAB

  CString sTempFilePath;  // path to the file we extract from the CAB

  int nListIdx;          // index of this item in the list ctrl



  // Data set while extracting files:

  bool bPartialFile;  // true if this file is continued in another cab

  CString sCabName;   // name of the CAB file

  bool bCabMissing;   // true if the file is partially in this cab and

                    // the CAB it's continued in isn't found, meaning

                    // the file can't be extracted


  CDraggedFileInfo ( const CString& s, int n ) :
    sFilename(s), nListIdx(n), bPartialFile(false),
    bCabMissing(false)
  { }
};
```

The view class also has methods for initialization, managing the list of files, and setting up a list of `CDraggedFileInfo`at the start of a drag and drop operation. I don't want to get too sidetracked on the inner workings of the UI, since this article is about drag and drop, so check out *WTLCabViewView.h* in the sample project for all the details.

File-Open Handling

To view a CAB file, the user uses the *File-Open* command and selects a CAB file. The wizard-generated code for `CMainFrame` includes a handler for the *File-Open* menu item:

```
  BEGIN_MSG_MAP(CMainFrame)
    COMMAND_ID_HANDLER_EX(ID_FILE_OPEN, OnFileOpen)
  END_MSG_MAP()
```

`OnFileOpen()` uses the `CMyFileDialog` class, the enhanced version of WTL's `CFileDialog`introduced in Part IX, to show a standard file open dialog.

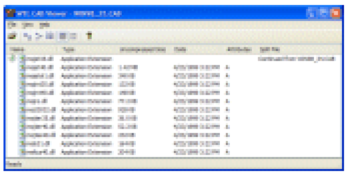
```
void CMainFrame::OnFileOpen (
  UINT uCode, int nID, HWND hwndCtrl )
{
CMyFileDialog dlg ( true, _T("cab"), 0U,
                    OFN_HIDEREADONLY|OFN_FILEMUSTEXIST,
                    IDS_OPENFILE_FILTER, *this );

  if ( IDOK == dlg.DoModal(*this) )
```

```
    ViewCab ( dlg.m_szFileName );
}
```

OnFileOpen() calls the helper function ViewCab():

```
void CMainFrame::ViewCab ( LPCTSTR szCabFilename )
{
  if ( EnumCabContents ( szCabFilename ) )
    m_sCurrentCabFilePath = szCabFilename;
}
```

EnumCabContents() is rather complex, and uses the CAB SDK calls to enumerate the contents of the file that was selected in OnFileOpen() and fill the view window. While ViewCab() doesn't do much right now, we will add code to it later to support the MRU list. Here's what the viewer looks like when showing the contents of one of the Windows 98 CAB files:



EnumCabContents() uses two methods in the view class to fill the UI: AddFile() andAddPartialFile(). AddPartialFile() is called when a file is only partially stored in the CAB, because it began in a previous CAB. In the screen shot above, the first file in the list is a partial file. The remaining items were added with AddFile(). Both of these methods allocate a data structure for the file being added, so the view knows all the details about each file that it's showing.

If EnumCabContents() returns true, all of the enumeration and UI setup completed successfully. If we were writing a simple CAB viewer, we would be done, although the app wouldn't be all that interesting. To make it really useful, we'll add drag and drop support so the user can extract files from the CAB.

The Drag Source

A drag and drop source is a COM object that implements two interfaces: IDataObject and IDropSource.IDataObject is used to store whatever data the client wants to transfer during the drag and drop operation; in our case this data will be an HDROP struct that lists the files being extracted from the CAB. TheIDropSource methods are called by OLE to notify the source of events during the drag and drop operation.

Drag Source Interfaces

The C++ class that implements our drop source is CDragDropSource. It begins with the IDataObjectimplementation from the MSDN articleI mentioned in the introduction. You can find all the details about that code in the MSDN article, so I won't repeat them here. We then add IDropSource and its two methods to the class:

```
class CDragDropSource :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CDragDropSource>,
  public IDataObject,
  public IDropSource
{
public:
  // Construction

  CDragDropSource();

  // Maps

  BEGIN_COM_MAP(CDragDropSource)
    COM_INTERFACE_ENTRY(IDataObject)
    COM_INTERFACE_ENTRY(IDropSource)
  END_COM_MAP()

  // IDataObject methods not shown...


  // IDropSource

  STDMETHODIMP QueryContinueDrag (
                BOOL fEscapePressed, DWORD grfKeyState );
  STDMETHODIMP GiveFeedback ( DWORD dwEffect );
};
```

Helper Methods for the Caller

CDragDropSource wraps the IDataObject management and drag/drop communication using a few helper methods. A drag/drop operation follows this pattern:

1. The main frame is notified that the user is beginning a drag/drop operation.
2. The main frame calls the view window to build a list of the files being dragged. The view returns this info in a `vector<CDraggedFileInfo>`.
3. The main frame creates a `CDragDropSource` object and passes it that vector so it knows what files to extract from the CAB.
4. The main frame beings the drag/drop operation.
5. If the user drops on a suitable drop target, the `CDragDropSource` object extracts the files.
6. The main frame updates the UI to indicate any files that could not be extracted.

Steps 3-6 are handled by helper methods. Initialization is done with the `Init()` method:

```
bool Init(LPCTSTR szCabFilePath, vector<CDraggedFileInfo>& vec);
```

`Init()` copies the data into protected members, fills in an `HDROP` struct, and stores that struct in the data object with the `IDataObject` methods. `Init()` also does another important step: it creates a zero-byte file in the TEMP directory for each file being dragged. For example, if the user drags *buffy.txt* and *willow.txt* from a CAB file, `Init()` will make two files with those same names in the TEMP directory. This is done in case the drag target validates the filenames it reads from the `HDROP`; if the files were not present, the target might reject the drop.

The next method is `DoDragDrop()`:

```
HRESULT DoDragDrop(DWORD dwOKEffects, DWORD* pdwEffect);
```

`DoDragDrop()` takes a set of `DROPEFFECT_*` flags in `dwOKEffects`, indicating which actions the source will allow. It queries for the necessary interfaces, then calls the `DoDragDrop()` API. If the drag/drop succeeds, `*pdwEffect` is set to the `DROPEFFECT_*` value that the user wanted to perform.

The last method is `GetDragResults()`:

```
const vector<CDraggedFileInfo>& GetDragResults();
```

The `CDragDropSource` object maintains a `vector<CDraggedFileInfo>` that is updated as the drag/drop operation progresses. When a file is found that is continued in another CAB, or can't be extracted, the `CDraggedFileInfo` structs are updated as necessary. The main frame calls `GetDragResults()` to get this vector, so it can look for errors and update the UI accordingly.

IDropSource Methods

The first `IDropSource` method is `GiveFeedback()`, which notifies the source of what action the user wants to do (move, copy, or link). The source can also change the cursor if it wants to. `CDragDropSource`keeps track of the action, and tells OLE to use the default drag/drop cursors.

```
STDMETHODIMP CDragDropSource::GiveFeedback(DWORD dwEffect)
{
  m_dwLastEffect = dwEffect;
  return DRAGDROP_S_USEDEFAULTCURSORS;
}
```

The other `IDropSource` method is `QueryContinueDrag()`. OLE calls this method as the user moves the cursor around, and tells the source which mouse buttons and keys are pressed. Here is the boilerplate code that most `QueryContinueDrag()` implementations use:

```
STDMETHODIMP CDragDropSource::QueryContinueDrag (
    BOOL fEscapePressed, DWORD grfKeyState )
{
  // If ESC was pressed, cancel the drag.

  // If the left button was released, do drop processing.

  if ( fEscapePressed )
    return DRAGDROP_S_CANCEL;
  else if ( !(grfKeyState & MK_LBUTTON) )
    {
    // If the last DROPEFFECT we got in GiveFeedback()

    // was DROPEFFECT_NONE, we abort because the allowable

    // effects of the source and target don't match up.

    if ( DROPEFFECT_NONE == m_dwLastEffect )
      return DRAGDROP_S_CANCEL;

    // TODO: Extract files from the CAB here...


    return DRAGDROP_S_DROP;
    }
  else
    return S_OK;
}
```

When we see that the left button has been released, that's the point where we extract the selected files from the CAB.

```
STDMETHODIMP CDragDropSource::QueryContinueDrag (
    BOOL fEscapePressed, DWORD grfKeyState )
{
  // If ESC was pressed, cancel the drag.

  // If the left button was released, do the drop.

  if ( fEscapePressed )
    return DRAGDROP_S_CANCEL;
  else if ( !(grfKeyState & MK_LBUTTON) )
    {
    // If the last DROPEFFECT we got in GiveFeedback()

    // was DROPEFFECT_NONE, we abort because the allowable

    // effects of the source and target don't match up.

    if ( DROPEFFECT_NONE == m_dwLastEffect )
      return DRAGDROP_S_CANCEL;

    // If the drop was accepted, do the extracting here,

    // so that when we return, the files are in the temp dir

    // and ready for Explorer to copy.

    if ( ExtractFilesFromCab() )
      return DRAGDROP_S_DROP;
    else
      return E_UNEXPECTED;
    }
  else
    return S_OK;
}
```

`CDragDropSource::ExtractFilesFromCab()` is another complex bit of code that uses the CAB SDK to extract the files to the TEMP directory, overwriting the zero-byte files we created earlier. When `QueryContinueDrag()` returns `DRAGDROP_S_DROP`, that tells OLE to complete the drag/drop operation. If the drop target is an Explorer window, Explorer will copy the files from the TEMP directory into the folder where the drop happened.

Dragging and Dropping from the Viewer

Now that we've seen the class that implements the drag/drop logic, let's look at how our viewer app uses that class. When the main frame window receives an `LVN_BEGINDRAG` notification message, it calls the view to get a list of the selected files, and then sets up a `CDragDropSource` object:

```
LRESULT CMainFrame::OnListBeginDrag(NMHDR* phdr)
{
vector<CDraggedFileInfo> vec;
CComObjectStack<CDragDropSource> dropsrc;
DWORD dwEffect = 0;
HRESULT hr;

  // Get a list of the files being dragged (minus files

  // that we can't extract from the current CAB).

  if ( !m_view.GetDraggedFileInfo(vec) )
    return 0;   // do nothing


  // Init the drag/drop data object.

  if ( !dropsrc.Init(m_sCurrentCabFilePath, vec) )
    return 0;   // do nothing


  // Start the drag/drop!

  hr = dropsrc.DoDragDrop(DROPEFFECT_COPY, &dwEffect);

  return 0;
}
```

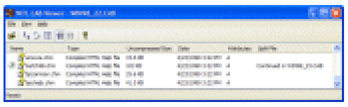
The first call is to the view's GetDraggedFileInfo() method to get the list of selected files. This method returns a vector<CDraggedFileInfo>, which we use to initialize the CDragDropSourceobject. GetDraggedFileInfo() may fail if all of the selected files are ones we know we can't extract (such as files that are partially stored in a different CAB file). If this happens, OnListBeginDrag()fails silently, and returns without doing anything. Finally, we call DoDragDrop() to start the operation, and let CDragDropSource handle the rest.

Step 6 in the list above mentioned updating the UI after the drag/drop is finished. It is possible for a file at the end of a CAB to be only partially stored in that CAB, with the rest being in a subsequent CAB. (This is quite common in the Windows 9x setup files, where the CABs are sized to fit on floppy disks.) When we try extracting such a file, the CAB SDK will tell us the name of the CAB that has the remainder of the file. It will also look for the CAB in the same directory as the initial CAB, and extract the rest of the file if the subsequent CAB is present.

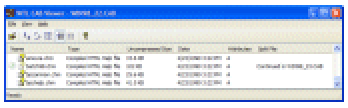
Since we want to indicate partial files in the view window, OnListBeginDrag() checks the drag/drop results to see if any partial files were found:

```
LRESULT CMainFrame::OnListBeginDrag(NMHDR* phdr)
{
//...


  // Start the drag/drop!

  hr = dropsrc.DoDragDrop(DROPEFFECT_COPY, &dwEffect);

  if ( FAILED(hr) )
    ATLTRACE("DoDragDrop() failed, error: 0x%08X\n", hr);
  else
    {
    // If we found any files continued into other CABs, update the UI.

    const vector<CDraggedFileInfo>& vecResults = dropsrc.GetDragResults();
    vector<CDraggedFileInfo>::const_iterator it;

    for ( it = vecResults.begin(); it != vecResults.end(); it++ )
      {
      if ( it->bPartialFile )
        m_view.UpdateContinuedFile ( *it );
      }
    }
  return 0;
}
```

We call `GetDragResults()` to get an updated `vector<CDraggedFileInfo>` that reflects the outcome of the drag/drop operation. If the `bPartialFile` member of a struct is `true`, then that file was only partially in the CAB. We call the view method `UpdateContinuedFile()`, passing it the info struct, so it can update the file's list view item accordingly. Here's how the app indicates a partial file, when the subsequent CAB was found:



If the subsequent CAB cannot be found, the app indicates that the file can't be extracted by setting the `LVIS_CUT`style, so the icon appears ghosted:



To be on the safe side, the app leaves the extracted files in the TEMP directory, instead of cleaning them up immediately after the drag/drop is finished. As `CDragDropSource::Init()` is creating the zero-byte temp files, it also adds each file name to a global vector `g_vecsTempFiles`. The temp files are deleted when the main frame window closes.

Adding an MRU List

The next doc/view-style feature we'll look at is a most-recently-used file list. WTL's MRU implementation is the template class `CRecentDocumentListBase`. If you don't need to override any of the default MRU behavior (and the defaults are usually sufficient), you can use the derived class `CRecentDocumentList`.

The `CRecentDocumentListBase` template has these parameters:

```
template <class T, int t_cchItemLen = MAX_PATH,
        int t_nFirstID = ID_FILE_MRU_FIRST,
        int t_nLastID = ID_FILE_MRU_LAST> CRecentDocumentListBase
```

`T`

The name of the derived class that is specializing `CRecentDocumentListBase`.

`t_cchItemLen`

The length in `TCHAR`s of the strings to be stored in the MRU items. This must be at least 6.

`t_nFirstID`

The lowest ID in the range of IDs to use for the MRU items.

`t_nLastID`

The highest ID in the range of IDs to use for the MRU items. This must be greater than `t_nFirstID`.

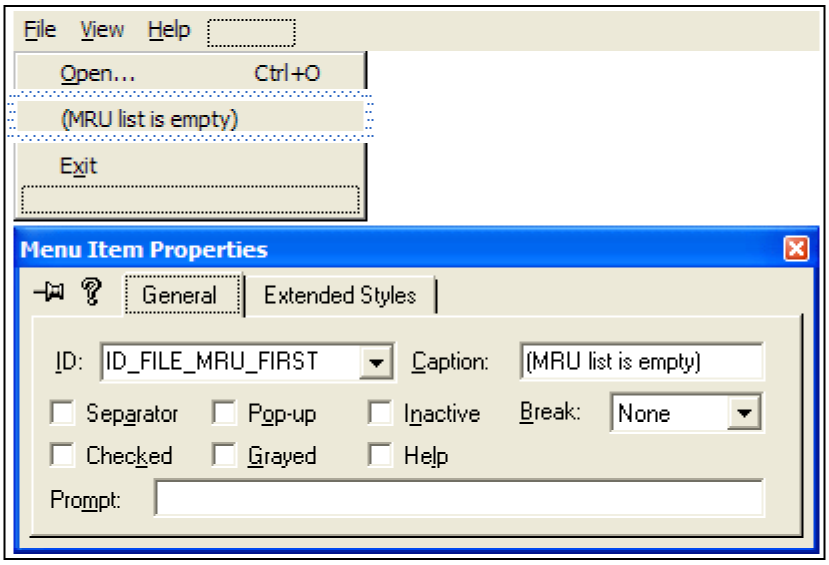
To add the MRU feature to our app, we need to follow a few steps:

1. Insert a menu item with ID `ID_FILE_MRU_FIRST` in the place that we want the MRU items to appear. This item's text will be shown if the MRU list is empty.
2. Add a string table entry with ID `ATL_IDS_MRU_FILE`. This string is used for the flyby help when an MRU item is selected. If you use the WTL AppWizard, this string is already created for you.
3. Add a `CRecentDocumentList` object to `CMainFrame`.
4. Initialize the object in `CMainFrame::Create()`.
5. Handle `WM_COMMAND` messages where the command ID is between `ID_FILE_MRU_FIRST` and `ID_FILE_MRU_LAST` inclusive.
6. Update the MRU list when a CAB file is opened.
7. Save the MRU list when the app closes.

Remember that you can always change the ID range if `ID_FILE_MRU_FIRST` and `ID_FILE_MRU_LAST`are unsuitable for your app, by making a new specialization of `CRecentDocumentListBase`.

Setting Up the MRU Object

The first step is to add a menu item that indicates where the MRU items will go. The usual place is the *File*menu, and that's what we'll use in our app. Here's our placeholder menu item:



ID: ID_FILE_MRU_FIRST Caption: (MRU list is empty)

Separator Pop-up Inactive Break: None

Checked Grayed Help

Prompt:

The AppWizard already added the string ATL_IDS_MRU_FILE to our string table; we'll change it to read "Open this CAB file". Next, we add a CRecentDocumentList member variable to CMainFramecalled m_mru, and initialize it in OnCreate():

```
#define APP_SETTINGS_KEY \
    _T("software\\Mike's Classy Software\\WTLCabView");

LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )
{
HWND hWndToolBar = CreateSimpleToolBarCtrl(...);

  CreateSimpleReBar ( ATL_SIMPLE_REBAR_NOBORDER_STYLE );
  AddSimpleReBarBand ( hWndToolBar );

  CreateSimpleStatusBar();

  m_hWndClient = m_view.Create ( m_hWnd, rcDefault );
  m_view.Init();

  // Init MRU list

CMenuHandle mainMenu = GetMenu();
CMenuHandle fileMenu = mainMenu.GetSubMenu(0);

  m_mru.SetMaxEntries(9);
  m_mru.SetMenuHandle ( fileMenu );
  m_mru.ReadFromRegistry ( APP_SETTINGS_KEY );

  // ...
}
```

The first two methods set the number of items we want in the MRU (the default is 16), and the menu handle that contains the placeholder item. `ReadFromRegistry()` reads the MRU list from the registry. It takes the key name we pass it, and creates a new key under it to hold the list. In our case, the key is `HKCU\Software\Mike's Classy Software\WTLCabView\Recent Document List`.

After loading the file list, `ReadFromRegistry()` calls another `CRecentDocumentList`method, `UpdateMenu()`, which finds the placeholder menu item and replaces it with the actual MRU items.

Handling MRU Commands and Updating the List

When the user selects an MRU item, the main frame receives a `WM_COMMAND` message with the command ID equal to the menu item ID. We can handle these commands with one macro in the message map:

```
  BEGIN_MSG_MAP(CMainFrame)
    COMMAND_RANGE_HANDLER_EX(
        ID_FILE_MRU_FIRST, ID_FILE_MRU_LAST, OnMRUMenuItem)
  END_MSG_MAP()
```

The message handler gets the full path of the item from the MRU object, then calls `ViewCab()` so the app shows the contents of that file.

```
void CMainFrame::OnMRUMenuItem (
  UINT uCode, int nID, HWND hwndCtrl )
{
CString sFile;

  if ( m_mru.GetFromList ( nID, sFile ) )
    ViewCab ( sFile, nID );
}
```

As mentioned earlier, we'll expand `ViewCab()` to be aware of the MRU object, and update the file list as necessary. The new prototype is:

```
  void ViewCab ( LPCTSTR szCabFilename, int nMRUID = 0 );
```

If nMRUID is 0, then ViewCab() is being called from OnFileOpen(). Otherwise, the user selected one of the MRU menu items, and nMRUID is the command ID that OnMRUMenuItem() received. Here's the updated code:

```
void CMainFrame::ViewCab ( LPCTSTR szCabFilename, int nMRUID )
{
  if ( EnumCabContents ( szCabFilename ) )
    {
    m_sCurrentCabFilePath = szCabFilename;

    // If this CAB file was already in the MRU list,

    // move it to the top of the list. Otherwise,

    // add it to the list.

    if ( 0 == nMRUID )
      m_mru.AddToList ( szCabFilename );
    else
      m_mru.MoveToTop ( nMRUID );
    }
  else
    {
    // We couldn't read the contents of this CAB file,

    // so remove it from the MRU list if it was in there.

    if ( 0 != nMRUID )
      m_mru.RemoveFromList ( nMRUID );
    }
}
```

When EnumCabContents() succeeds, we update the MRU differently depending on how the CAB file was selected. If it was selected with *File-Open*, we call AddToList() to add the filename to the MRU list. If it was selected with an MRU menu item, we move that item to the top of the list with MoveToTop(). If EnumCabContents() fails, we remove the filename from the MRU list with RemoveFromList(). All of those methods call UpdateMenu() internally, so the *File* menu will be updated automatically.

Saving the MRU List

When the app closes, we save the MRU list back to the registry. This is simple, and just takes one line:

```
  m_mru.WriteToRegistry ( APP_SETTINGS_KEY );
```

This line goes in the `CMainFrame` handlers for the `WM_DESTROY` and `WM_ENDSESSION`messages.

Other UI Goodies

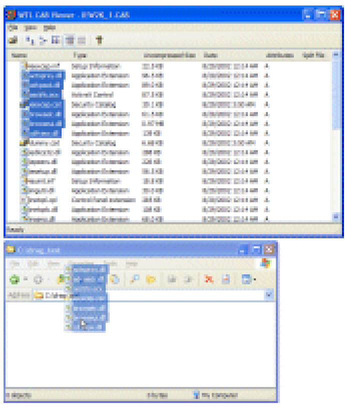
Transparent Drag Images

Windows 2000 and later have a built-in COM object called the drag/drop helper, whose purpose is to provide a fancy transparent drag image during drag/drop operations. The drag source uses this object via the `IDragSourceHelper`interface. Here is the additional code, indicated in bold, we add to `OnListBeginDrag()` to use the helper object:

```
LRESULT CMainFrame::OnListBeginDrag(NMHDR* phdr)
{
NMLISTVIEW* pnmlv = (NMLISTVIEW*) phdr;
CComPtr<IDragSourceHelper> pdsh;
vector<CDraggedFileInfo> vec;
CComObjectStack<CDragDropSource> dropsrc;
DWORD dwEffect = 0;
HRESULT hr;

  if ( !m_view.GetDraggedFileInfo(vec) )
    return 0;   // do nothing


  if ( !dropsrc.Init(m_sCurrentCabFilePath, vec) )
    return 0;   // do nothing


  // Create and init a drag source helper object

  // that will do the fancy drag image when the user drags
  // into Explorer (or another target that supports the

  // drag/drop helper interface).

  hr = pdsh.CoCreateInstance ( CLSID_DragDropHelper );

  if ( SUCCEEDED(hr) )
    {
    CComQIPtr<IDataObject> pdo;

    if ( pdo = dropsrc.GetUnknown() )
      pdsh->InitializeFromWindow ( m_view, &pnmlv->ptAction, pdo );
    }

  // Start the drag/drop!

  hr = dropsrc.DoDragDrop(DROPEFFECT_COPY, &dwEffect);

  // ...

}
```

We start by creating the drag/drop helper COM object. If that succeeds, we call `InitializeFromWindow()` and pass three parameters: the `HWND` of the drag source window, the cursor location, and an `IDataObject` interface on our `CDragDropSource` object. The drag/drop helper uses this interface to store its own data, and if the drag target also uses the helper object, that data is used to generate the drag image.

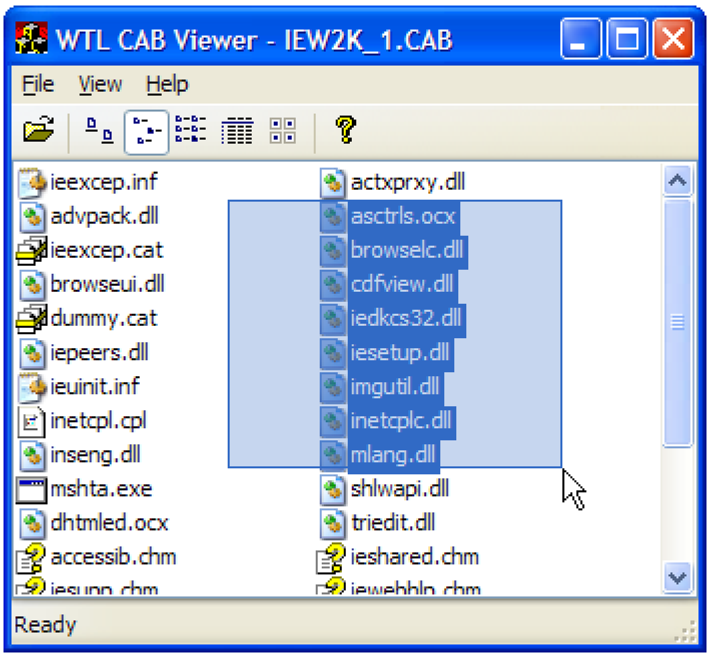
For `InitializeFromWindow()` to work, the drag source window needs to handle the `DI_GETDRAGIMAGE` message, and in response to that message, create a bitmap to be used as the drag image. Fortunately for us, the list view control supports this feature, so we get the drag image with very little work. Here's what the drag image looks like:



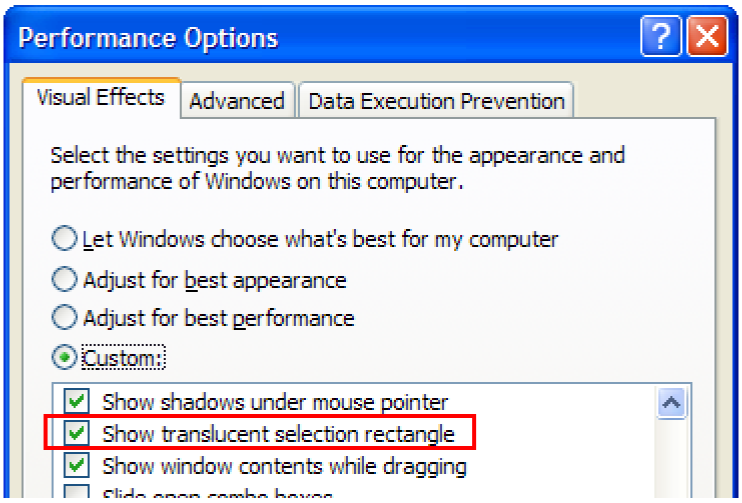
If we were using some other window as our view class, one that didn't handle `DI_GETDRAGIMAGE`, we would create the drag image ourselves and call `InitializeFromBitmap()` to store the image in the drag/drop helper object.

Transparent Selection Rectangle

Starting with Windows XP, the list view control can display a transparent selection marquee. This is turned off by default, but it can be enabled by setting the `LVS_EX_DOUBLEBUFFER` style on the control. Our app does this as part of the view window initialization in `CWTLCabViewView::Init()`. Here's the result:



If the transparent marquee isn't showing up for you, check your system properties and make sure the feature is enabled there:



Performance Options

Select the settings you want to use for the appearance and performance of Windows on this computer.

- ◯ Let Windows choose what's best for my computer
- ◯ Adjust for best appearance
- ◯ Adjust for best performance
- ● Custom:

- ✓ Show shadows under mouse pointer
- ✓ Show translucent selection rectangle
- ✓ Show window contents while dragging

Indicating the Sorted Column

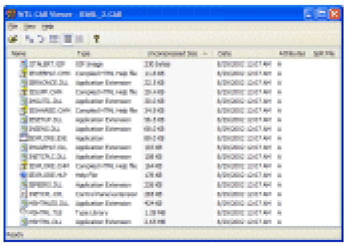
On Windows XP and later, a list view control in report mode can have a selected column, which is shown with a different background color. This feature is normally used to indicate that a column is being sorted, and this is what our CAB viewer does. The header control also has two new formatting styles that make the header show an up- or down-pointing arrow in a column. This is normally used to show the direction of the sort.

The view class handles sorting in the `LVN_COLUMNCLICK` handler. The code for showing the sorted column is highlighted in bold:

```
LRESULT CWTLCabViewView::OnColumnClick ( NMHDR* phdr )
{
int nCol = ((NMLISTVIEW*) phdr)->iSubItem;

  // If the user clicked the column that is already sorted,

  // reverse the sort direction. Otherwise, go back to

  // ascending order.

  if ( nCol == m_nSortedCol )
    m_bSortAscending = !m_bSortAscending;
  else
    m_bSortAscending = true;

  if ( g_bXPOrLater )
    {
    HDITEM hdi = { HDI_FORMAT };
    CHeaderCtrl wndHdr = GetHeader();

    // Remove the sort arrow indicator from the

    // previously-sorted column.

    if ( -1 != m_nSortedCol )
      {
      wndHdr.GetItem ( m_nSortedCol, &hdi );
      hdi.fmt &= ~(HDF_SORTDOWN | HDF_SORTUP);
      wndHdr.SetItem ( m_nSortedCol, &hdi );
      }

    // Add the sort arrow to the new sorted column.

    hdi.mask = HDI_FORMAT;
    wndHdr.GetItem ( nCol, &hdi );
    hdi.fmt |= m_bSortAscending ? HDF_SORTUP : HDF_SORTDOWN;
    wndHdr.SetItem ( nCol, &hdi );
    }

  // Store the column being sorted, and do the sort

  m_nSortedCol = nCol;

  SortItems ( SortCallback, (LPARAM)(DWORD_PTR) this );

  // Indicate the sorted column.

  if ( g_bXPOrLater )
    SetSelectedColumn ( nCol );

  return 0;
}
```

The first section of highlighted code removes the sort arrow from the previously-sorted column. If there was no sorted column, this part is skipped. Then, the arrow is added to the column that the user just clicked on. The arrow points up if the sort is ascending, or down if the sort is descending. After the sort is done, we call `SetSelectedColumn()`, a wrapper around the `LVM_SETSELECTEDCOLUMN` message, to set the selected column to the column we just sorted.

Here's how the list control appears when the files are sorted by size:



Using Tile View Mode

On Windows XP and later, the list view control has a new style called *tile view mode*. As part of the view window's initialization, if the app is running on XP or later, it sets the list view mode to tile mode using`SetView()` (a wrapper for the `LVM_SETVIEW` message). It then fills in a `LVTILEVIEWINFO`struct to set some properties that control how the tiles are drawn. The `cLines` member is set to 2, meaning 2 additional lines of text will appear beside each tile. The `dwFlags` member is set to `LVTVIF_AUTOSIZE`, which makes the control resize the tile area as the control itself is resized.

```
void CWTLCabViewView::Init()
{
  // ...


  // On XP, set some additional properties of the list ctrl.

  if ( g_bXPOrLater )
    {
    // Turning on LVS_EX_DOUBLEBUFFER also enables the

    // transparent selection marquee.

    SetExtendedListViewStyle ( LVS_EX_DOUBLEBUFFER,
                               LVS_EX_DOUBLEBUFFER );

    // Default to tile view.

    SetView ( LV_VIEW_TILE );

    // Each tile will have 2 additional lines (3 lines total).

    LVTILEVIEWINFO lvtvi = { sizeof(LVTILEVIEWINFO),
                             LVTVIM_COLUMNS };
    lvtvi.cLines = 2;
    lvtvi.dwFlags = LVTVIF_AUTOSIZE;
    SetTileViewInfo ( &lvtvi );
    }
}
```

Setting up the tile view image list

For tile view mode, we'll use the extra-large system image list (which has 48x48 icons in the default display settings). We get this image list using the `SHGetImageList()` API. `SHGetImageList()` is different from `SHGetFileInfo()` in that it returns a COM interface on an image list object. The view window has two member variables for managing this image list:

```
  CImageList m_imlTiles;        // the image list handle

  CComPtr<IImageList> m_TileIml; // COM interface on the image list
```

The view window gets the extra-large image list in `InitImageLists()`:

```
HRESULT (WINAPI* pfnGetImageList)(int, REFIID, void**);
HMODULE hmod = GetModuleHandle ( _T("shell32") );

  (FARPROC&) pfnGetImageList = GetProcAddress(hmod, "SHGetImageList");

  hr = pfnGetImageList ( SHIL_EXTRALARGE, IID_IImageList,
                         (void**) &m_TileIml );

  if ( SUCCEEDED(hr) )
    {
    // HIMAGELIST and IImageList* are interchangeable,

    // so this cast is OK.

    m_imlTiles = (HIMAGELIST)(IImageList*) m_TileIml;
    }
```

If `SHGetImageList()` succeeds, we can cast the `IImageList*` interface to an `HIMAGELIST`and use it just like any other image list.

Using the tile view image list

Since the list control doesn't have a separate image list for tile view mode, we need to change the image list at runtime when the user chooses large icon or tile view mode. The view class has a `SetViewMode()` method that handles changing the image list and the view styles:

```
void CWTLCabViewView::SetViewMode ( int nMode )
{
  if ( g_bXPOrLater )
    {
    if ( LV_VIEW_TILE == nMode )
      SetImageList ( m_imlTiles, LVSIL_NORMAL );
    else
      SetImageList ( m_imlLarge, LVSIL_NORMAL );

    SetView ( nMode );
    }
  else
    {
    // omitted - no image list changing necessary on

    // pre-XP, just modify window styles

    }
}
```

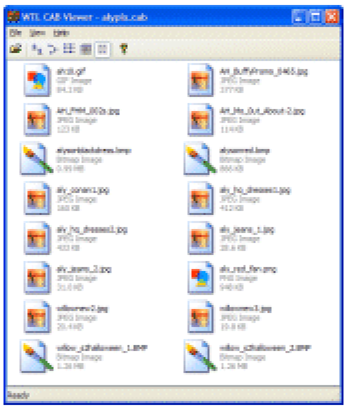
If the control is going into tile view mode, we set the control's image list to the 48x48 one, otherwise we set it to the 32x32 one.

Setting the additional lines of text

During initialization, we set up the tiles to show two additional lines of text. The first line is always the item text, just as in the large icon and small icon modes. The text shown in the two additional lines are taken from subitems, similarly to the columns in report mode. We can set the subitems for each tile individually. Here is how the view sets the text in `AddFile()`:

```
  // Add a new list item.

int nIdx;

  nIdx = InsertItem ( GetItemCount(), szFilename, info.iIcon );
  SetItemText ( nIdx, 1, info.szTypeName );
  SetItemText ( nIdx, 2, szSize );
  SetItemText ( nIdx, 3, sDateTime );
  SetItemText ( nIdx, 4, sAttrs );

  // On XP+, set up the additional tile view text for the item.

  if ( g_bXPOrLater )
    {
    UINT aCols[] = { 1, 2 };
    LVTILEINFO lvti = { sizeof(LVTILEINFO), nIdx,
                        countof(aCols), aCols };

    SetTileInfo ( &lvti );
    }
```

The `aCols` array holds the subitems whose text should be shown, in this case we show subitem 1 (the file type) and 2 (the file size). Here's what the viewer looks like in tile view mode:



Note that the additional lines will change after you sort a column in report mode. When a selected column is set with `LVM_SETSELECTEDCOLUMN`, that subitem's text is always shown first, overriding the subitems we passed in the `LVTILEINFO` struct.