WTL

Developer's Guide

By Eamon O'Tuathail (www.clipcode.biz)

Table Of Contents

Chapter 1
Overview of WTL

Objectives

The objectives of this chapter are to:

- See where WTL fits into the "big picture" of VC++ development
- Examine its aims
- Contrast WTL with other UI development techniques
- Explain the installation of WTL
- Describe what is installed with WTL
- List the available development resources
- Describe the following chapters

Overview

In the past Visual C++ developers often choose MFC as it provided functionality in a very wide range of technologies, while the template libraries covered limited areas. More recently the range and quality of the template libraries have been improving and now the template answer is more and more often chosen.

The design goal of ATL is to provide fast compact COM components. ISO C++'s STL provides collections. The VC++ OLE DB Data Consumer and Provider Templates provide database support. In contrast, MFC provides a single library of C++ classes, which provide a reasonable range of features in COM creation, collection classes, database classes and user interface.

Most advanced developers prefer the newer template-route to development. MFC is monolithic, bulky, not very thread-friendly and, well, is basically old-fashioned! The template approach is fast (when designed correctly), flexible, covers all the latest techniques and for new development is definitely the way to go.

Up to now the one major problem for template enthusiasts was how to create the graphical user interface. ATL does provide lightweight wrappers around Win32/64 windowing, but it certainly does not cover all UI needs. In the past, for anything other than trivial UI, ATL developers had to resort to MFC UI programming or developing a VB front-end to their ATL COM components, neither of which was totally satisfactory.

Enter the Windows Template Library. The WTL is an advanced set of wrappers and productivity enhancement templates, which sit above the Win32/64 API, and provide

comprehensive support for a wide variety of graphical user interface requirements. Keeping true to the template library tradition, WTL is small, ultra-fast and non-intrusive. It covers the latest UI concepts. It works well with multithreading. You can use WTL on its own or along with any combination of ATL, STL, VC++ Data Templates, third-party template libraries and your own custom templates libraries, incorporating just the features you need in each application. WTL usually has no dependencies on external DLLs – you only need to ship to end-users the EXE for the WTL application.

Aims of WTL

WTL is to the user interface what ATL is to COM or STL is to collections. Like its cousins, WTL takes a little while to learn, but when mastered there is no better way of developing the most advanced applications.

Essentially, WTL accomplishes three significant tasks:

- Providing an application framework
- Aggregating UI functionality
- Wrapping windowing controls, system dialogs and GDI objects

Let's examine each of these.

Providing An Application Framework

WTL provides a lightweight yet comprehensive application framework, which automatically furnishes applications based on it with many desirable facilities. The goal is something less than the impenetrable MFC framework, and something easier than starting to code `WinMain` manually.

WTL comes in the form of a series of C++ templates and a WTL AppWizard. The AppWizard asks the application developer a few questions and then generates a VC++ project and application source code, much of which instantiates or derives from the WTL templates and classes. In general, what application developers might wish to change is in the AppWizard-generated application source code, and the "boiler-plate" style code, which rarely needs to be changed, is inside the WTL templates. Where necessary these templates may be derived from and custom functionality provided.

WTL has a class to manage a module (a DLL or EXE). This class is instantiated in the AppWizard-generated application code and within the generated WinMain it is initialized and terminated. A WTL application can optionally act as a COM server, thus supporting programmable applications. WTL provides message-handling support and includes message filtering and on-idle functionality. Change notifications from `WM_SETTINGCHANGE` messages may be handled.

MFC provides a document-frame-view architecture. WTL does provide frames (i.e. a top-level window, containing a menu bar, toolbar and statusbar) and views, but does not support documents at all. The MFC approach often got in the way of more advanced developers, and its documents were based on serialized binary data. In the modern Internet-world, document formats based on XML/XSL are becoming popular and there

are demands for flexible storage mechanisms – it might not be the local hard disk; documents might need to be stored using a variety of remoting architectures, such as WebDAV or FTP. WTL provides no functionality regarding data formats or storage mechanisms. Application developers will have to write their own code for these tasks, e.g. using the fast Win32 APIs `WriteFileGather` / `ReadFileScatter` or an XML parser.

WTL supports applications based on dialog-box, single document interface (SDI) or a multi-document interface (MDI). For SDI, it also supports multi-threaded SDI, with one thread per SDI window. It is expected that most advanced applications will use this architecture (e.g. following the Word 2000 approach). WTL does not support a multi-threaded MDI architecture with one thread per MDI child window. This is sensible, as this avoids a variety of problems with the interaction between threads and MDI parent-child windows.

Aggregating UI Functionality

WTL provides a range of "must-have" features for applications requiring a modern user interface.

A frame window is provided which manages a rebar control containing a command bar (enhanced menubar) and a toolbar, a status bar and one or more views. The views may be based on a generic empty window, or on controls such as richedit, listview or treeview. If more than one view is used, then the WTL splitter functionality can be employed to render multiple views with a moveable splitter divider. The views can also be scrolled. The status bar can be single-paned or multi-paned.

WTL does support traditional menu bars and toolbars, but its main menu presentation concept is based around command bars. A command bar contains menus and toolbar icons, and can be displayed and moved around within a rebar control. If a menu item has the same command id as a toolbar icon, then when the menu is rendered the equivalent icon is rendered next to each item. WTL supports neither docking nor floating command bars. WTL's command-bars are similar to those in Internet Explorer, and not like the (much more desirable!) command-bars in Word 2000 or Visual Studio. WTL does not support shortened menus based on usage-data (e.g. WORD-2000).

The concept of dynamic data exchange allows the transfer in both directions of values between on-screen user interface controls and C++ data members. This functions quite similarly to MFC's DDX.

Dialog boxes may be created using ATL, including those, which support ActiveX, control containment. Visual C++'s ResourceView can be used to layout standard controls, common controls and ActiveX controls in the dialog resource template. The ATL Window Message Handler Wizard can be used to map incoming messages for these controls to message handlers.

WTL provides templates to manage property sheets and property page construction and wizard construction.

Printing is supported through a printer device context, print preview, devmode handling and print job management functionality.

Enhanced metafiles are supported using a special device context, file management and enhanced metafile information classes.

Wrapping Windowing Controls, System Dialogs and GDI objects

ATL provides access to generic window functionality, but provides no special support for windows based on different windows classes. Developers had to resort to manually coding `SendMessage` calls as needed. So whether communicating with an edit or a treeview control, they had to use ATL's `CWindow` and call Win32's `SendMessage` with `EM_LIMITTEXT` or `VM_SETITEM` and ensure that the parameters were correct for each message (no type checking was performed). When applications received them they were in the raw format – `wParam` and `lParam` as sent on the message queue. Applications needed code to convert these to appropriate data types; again making sure the correct conversion was made for each message type.

Some ATL developers were using code extracted from the ATLCON sample, which provide some wrappers for Windows UI elements. This has evolved into a full set of comprehensive wrappers for all the standard and common window controls, the system dialogs and all the GDI objects, and much more. There are WTL classes for edit, button, listbox, treeview, listview etc. There are wrappers for the common file dialog, the color dialog, the font selection dialog, etc. There are wrappers for the device context, pen, brush, region, font etc.

In addition, a full set of message crackers is provided. Incoming messages are mapped to message handlers. With ATL these message handlers were passed `lParam` and `wParam`. With WTL's message crackers, the message handlers' input is specific to the incoming message. For example, the handler for `WM_LBUTTONDOWN` is passed in a `CPoint` parameter (`CPoint` is a WTL wrapper for the Win32 `POINT` structure – WTL also has wrappers for such common structures). Note that the ATL Windows Message Handler Wizard uses ATL's raw message maps, not WTL's message crackers. It is likely wizard support will be improved in the next version of Visual C++.

Many developers choose to use MFC solely for its string-handling support. WTL eliminates this need by providing its own implementation of `CString`, which is a comprehensive clone of MFC's `CString`. The string formats WTL's `CString` supports include ASCII, MBCS, UNICODE and Automation's BSTR. It supports conversions between all these formats. It supports string manipulation such as concatenation, trimming and comparison. Its supports printf-like string formatting. It supports flexible memory management.

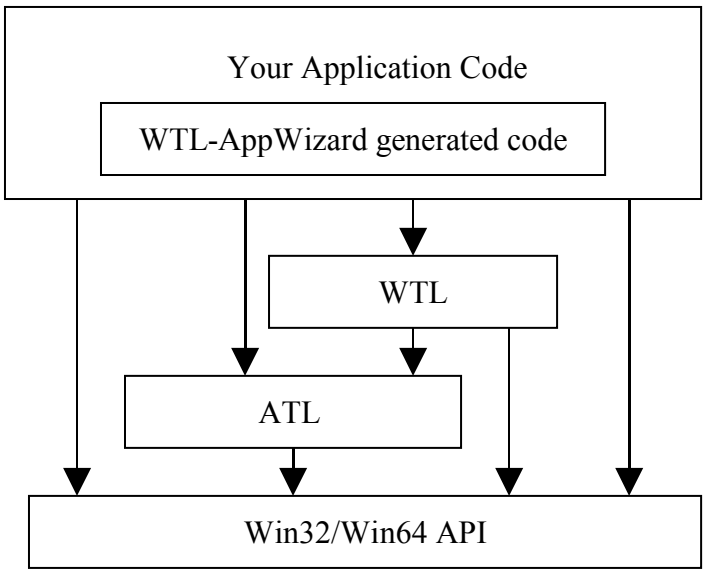
Relationship Between WTL, ATL and Win32/Win64 API

WTL is based on ATL and Win32/Win64, and ATL is based on Win32/Win64. When considering their relationships, we need to examine the source code view and the binary deliverable view.

Source Code

The Win32/Win64 API is a set of thousands of C functions, covering a vast range of topics – including two that interest us here, COM and windowing.

ATL is a set of C++ templates, which mostly are dedicated to COM programming, but also include comprehensive low-level support for windowing functionality.



WTL is a set of C++ templates, which focus purely on higher-level windowing functionality. WTL is independent of COM, but can be use along with COM in an application. Provided you do not select the "*Com Server*" support in the WTL AppWizard, then `CoInitialize` is not actually called.

WTL uses the ATL windowing services, so therefore to use WTL you must have access to the ATL templates.

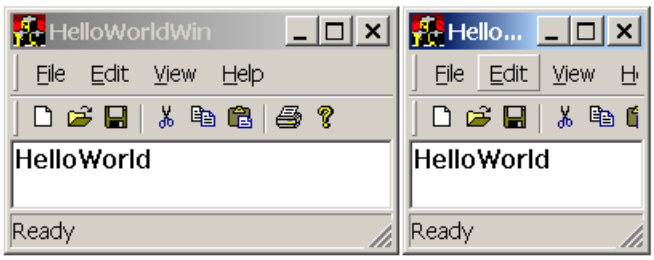
ATL makes calls to the Win32/Win64 API. WTL makes calls to the ATL templates and the Win32/Win64 API. Your application code makes calls to WTL, ATL and the Win32/Win64 API.

When you run the WTL AppWizard, it produces a number of source files, which become part of your application source code. The generated source files make calls into WTL and sets up the framework for the application.

WTL supports Windows 2000, Windows Me, Windows NT 4, Windows 98, Windows 95 OSR 2 and "Classic" Windows 95. WTL support of these is not of the "lowest common denominator" variety. Instead it uses some #defines (e.g. `_RICHEDIT_VER, _WIN32_IE, _WIN32_WINNT` and `WINVER`) to determine which Win32/Win64 features to use. WTL does not auto-detect the installed versions of the OS or Internet Explorer (e.g. it does not use the `LoadLibrary / GetProceAddress` or `GetVersionEx` APIs). Instead, the application developer must specify the `#defines`, and the application during compilation will assume they are available on the client machine and will not run if absent.

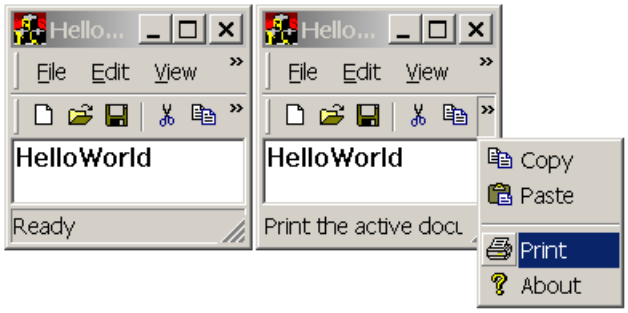
As an example of this, take chevrons in menus. When a command-bar is shortened due to a window resize and chevrons are not used, then the piece of the command-bars, which are outside the windows boundaries, are unceremoniously cut off. These two

screen dumps are an example of a WTL application with a window in full size and reduced – you note the help menu title and the paste toolbar icon are chopped in half.



The chevron is a symbol comprising two greater-than signs. When pressed, it displays a pop-down menu containing the items, which have been chopped off the end of the command-bar and toolbar due to resizing. Chevrons are supported if Internet Explorer 5 is included, and are not supported otherwise.

In a WTL application, if `_WIN32_IE` is set to `0x0500` or later (e.g. in stdafx.h) then the chevron is displayed when needed. These two screen dumps show a WTL application with a resized window, with the chevron not pressed and pressed.



Note that if Internet Explorer 5 or later is installed, but `_WIN32_IE` is not set as `0x0500` or later, then chevrons are not supported in the application, even though the installed OS does support them.

WTL does not support PocketPC or X-Box, though it would be nice to see this in future and there are no technical reasons preventing it. It does seem that WTL is prepared for Win64, as consideration for 64-bit programming is obvious from some of the WTL source files.

Binary Deliverable

ATL and WTL are delivered as a set of header files containing C++ templates. ATL does also have a small number of C++ classes – and these may be used as a separate DLL but the vast majority of application developers also include them in each project, and with WTL applications this is also recommended. WTL does not have separate DLL option. So as needed, the WTL C++ classes (there are a few of them) will be included in each application.

When you compile a WTL application you will end up with an EXE or a DLL. That is all you have to deliver to the end-user of the application. There are no dependencies on

external libraries, apart from OS libraries such as Kernel32.DLL, which are always present (if not, then Windows itself would have problems starting).

One exception to this "no-dependencies" rule is if you need to use floating-point numbers. If this is the case, then you will need to ship the C-Runtime Library (CRT).

Alternatives to WTL – ATL, MFC, VB, Java, DHTML or …

As an application developer, you will certainly find plenty of options available when you need to construct a graphical user interface. No matter which you choose, you will often get other developers saying "why did you not use <insert favorite UI toolkit name here>, it is so much better, because of XYZ". The development platforms wars will continue for along time. What is noticeable is that there are multiple options now, more than there were in the past and it is extremely likely to be the case well into the future. At any one time there are some options, which are very popular, some becoming popular and some waning in popularity. The best option for a particular project is constantly in flux.

Issues to consider when selecting your UI development technology include:

- Quality of implementation (how well does it work)
- Ease of use (what do developers need to know)
- The development environment and third party products
- Available journals and books
- Industry knowledge (how many software houses use it)
- Future prospects (will it be here in three years' time)
- The requirements of the project
- Your available development resources (e.g. what do your existing developers know)
- Your access to new development resources (e.g. how easy is it to get developers who knows a particular technology).

The ordering of the desirability of UI toolkits for one criterion might be the direct opposite for another. For example, in terms of ease of programming, DHTML is easiest, then VB and then one of the C++ options (which include use raw Win32 API, MFC, ATL or WTL). In terms of pure functionality and performance, the C++ options are best, then VB and then DHTML.

DHTML is the use of HTML, especially its forms capabilities, together with scripting, to produce web-based user interface. As HTML offers fewer UI elements than Win32/ Win64, the user interface is much simpler (both in the positive and negative senses of that word). For the general public, who has no interest in becoming techno-nerds (e.g. they have mastered the channel selection and volume control buttons on their TV remote control but would have difficulty explaining the other buttons), the vast array of

UI widgets in Win32 is a problem. They do not understand how they work. For them, it needs to be much simpler and DHTML is ideal. There is less clutter on-screen. There are fewer options. It is easy to navigate. However, if you need a UI to do something more complex than just accepting name and address text fields, you can quickly get into problems. It is possible to develop custom ActiveX controls and embed these within DHTML pages. This makes sense in some cases but also moves back to Win32 programming, but it would not be as easy as developing complex applications fully in Win32. DHTML does have problems when used to construct full UIs for complete complex applications – the problems are not insurmountable – indeed it would be an interesting project to tackle.

The latest trend is to use DHTML for desktop applications. The application hosts a WebBrowser ActiveX control, and then it renders DHTML pages. These can for example be stored as resources. To get remoting with UI toolkits other than DHTML, one could consider the Windows Terminal Server option.

DHTML would be very suitable for content delivery (e.g. delivering training courses,) but not that good at content creation (e.g. developing a training course editing suite), which would require a more complex UI.

MFC and VB are excellent tools for user interface development if the design you require closely matches that which these frameworks support well. However, if you wish something even slightly out of the ordinary then using MFC or VB can involve lots of unexpected twists and turns. It can take a considerable amount of work manipulating MFC or VB to meet you specialist needs. MFC and VB also have the major disadvantage of requiring a very large run-time (MFC DLLs and VBRUN).

MFC's support for Active Document is absent from WTL.

ATL applications and components often need to display windows and dialog boxes. Though it mainly concentrates on supporting COM, ATL does provide a good level of windowing functionality. Windowing in ATL is not as easy to use as MFC or VB. ATL's approach is flexible and highly configurable, but for windowing it is too-low level. If you are using ATL, and need windowing, then you should really move up to WTL.

WTL is very new. Few developers know it. More mainstream development groups will in due course follow the early adopters. Currently there are far more developers using the other UI options. WTL documentation (apart from the document you are currently reading) is very sketchy. All the other UI options are better documented. Any new technology will suffer these disadvantages over established technologies.

Why Choose WTL?

WTL is much less cumbersome than MFC and the resulting applications are less bloated. For advanced applications WTL gives you all the benefits of using the raw Win32 APIs but saves you a lot of time.

The Win32 API can be used directly, but it has a bewildering range of seemingly independent functions and sometimes it is difficult to determine how they fit together. WTL provides more homogeneous access to the available functionality.

Template libraries in general are becoming increasingly popular because the method that gets called is selected at compile time, rather than run-time, with benefits in terms of efficiency and code size.

Installation of WTL

WTL is evolving rapidly and is likely to see changes in its delivery mechanism over time.

Future Delivery Mechanism

It is likely that WTL will be folded into the Visual C++ product in the next release of Visual Studio. Hence it probably will automatically be installed along with ATL, STL and the Data Templates.

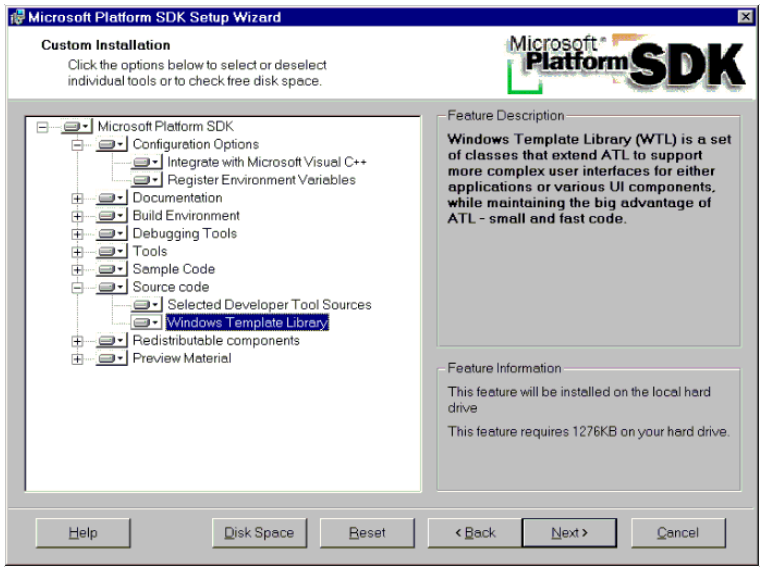
Current Installation Mechanism

Currently, WTL is released as part of the Microsoft Platform SDK (January 2000 or later). Installation involves three manual steps; though it is likely these will be automated in some future update.

First Step - Getting the Files

The first step is to get the relevant WTL files onto your hard disk. The Microsoft Platform SDK is available as part of the MSDN Subscription (disk 3 in the Development Platform set of CDs) or from the http://msdn.microsoft.com website. The Platform SDK using the Microsoft Installer to put the select files on your hard disk. You may choose to install/download the entire Platform SDK or subsections. The Platform SDK is large – but contains lots of goodies and it is recommended that all advanced developers have it fully installed on their hard-disks (we do all have 100GB+ hard-disks nowadays – don't we). To get the WTL files onto your hard-disk you may either install the whole Platform SDK or any selection of sections, provided they include the "Windows Template Library", which is located under the Source Code node in the tree in the installation wizard.

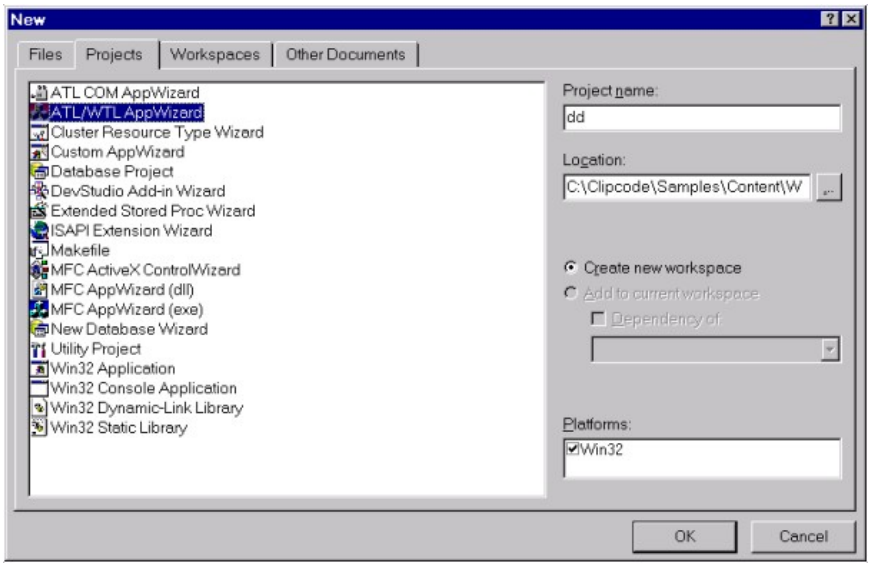
The installed size of the WTL developer files is quite small – the exact size depends on the sector sizes on your disk – on one demonstration machine it was only 1,276KB. (Note that apps built with WTL can be tiny – even less than 25K). The WTL files will be installed on your disk under <SDK>\Src\WTL, where <SDK> is the installation path you provided (e.g. c:\Program Files\Microsoft Platform SDK).



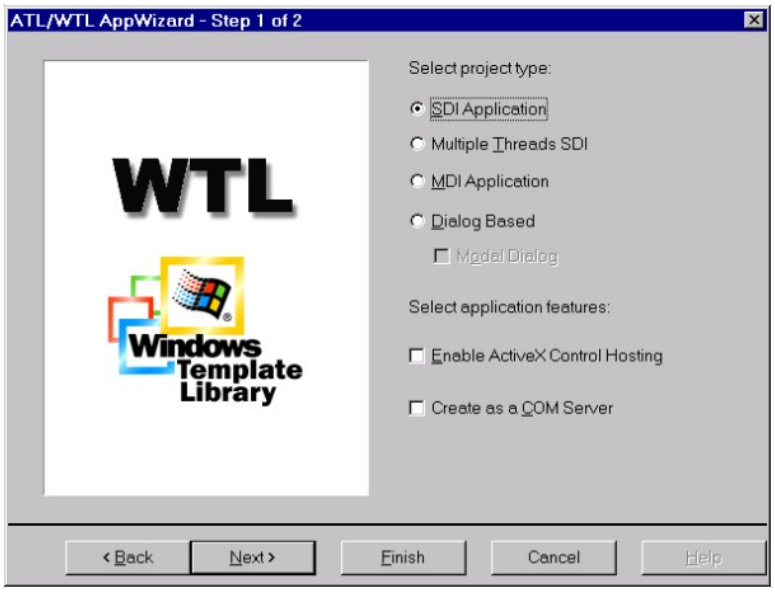
Second Step – The WTL AppWizard

The second step is to move the WTL AppWizard to the appropriate location. You need to manually move it from <SDK>\Src\WTL\AppWizard to <VC>\Common\MSDev98\Template, where <VC> is the installation path of Visual C++. Note that to use WTL you must have Visual C++ 6.0 installed. It is recommended, though not required, that you also have service pack 3 for Visual C++ 6 installed.

When you now start Visual C++ and select New from the File menu, you will see a new entry under the "projects" tab called ATL/WTL AppWizard.



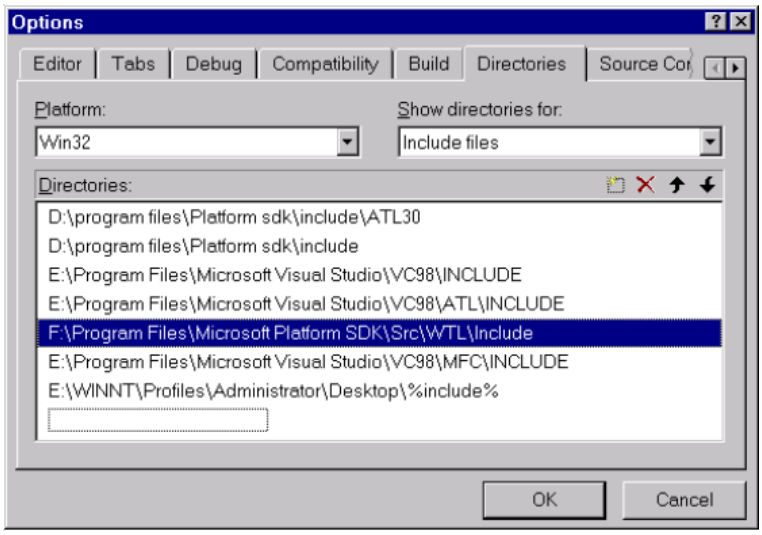
When clicked, it displays an AppWizard, which lets you select a few options, and then produces the initial project source files for you.



Third Step – Updating the Path

The third step is to set the path so that the compiler can find the WTL header files.

WTL is a template library consisting of C++ header files (there are no .cpp files). When compiling, the compiler needs to be able to find these files. The files are located under <SDK>\Src\WTL\include. One option would be to include the full pathname in your application's source files, to which most developers will respond in shock horror tones (and they are quite right too!). The second option is to put them in a directory which is already on the path – such as inside the ATL include directory. However, this is changing directories associated with the development environment, and it is usually best to leave them alone (but for here it will work). The third and recommended option is to configure the Visual C++ compiler on each machine to look in the WTL directory when searching for include files. This can be done by selecting the Options entry from DevStudio's Tools menu, then selecting the Directories tab, then selecting "Include Files" from the "Show Directories For" drop-down list, and then append an entry stating where the WTL files can be found. Do not forget to include the 'include' part of the path.



The ordering of the WTL entry in this list of directories is not important.

Contents Of WTL's Installation

The installation of WTL contains a top-level directory, <SDK>\Src\WTL\, and this in turn contains a readme.txt, and three subdirectories, \Include, \Examples and \AppWizard.

The readme.txt is the only documentation on WTL available currently from Microsoft – and it is certainly sparse. It is about 6 pages in length, and briefly introduces WTL, provides a list of files in the installation, and provides a one-line (!) description of each template/class in WTL, and then mentions the WTL AppWizard.

The \AppWizard directory contains a single AWX file, and as we have seen this needs to be manually copied to the VC++'s installation path.

The \Include directory contains 15 WTL header files – this directory can be considered the core of WTL.

The \Examples directory contains three examples showing how to use WTL. They are:

- MTPad – Notepad using multiple threads
- MDIDocVw - Document View sample
- GuidGen – ATL implementation of GuidGen (calls the Win32 API CoCreateGuid)

The MTPad sample is a SDI application that emulates Notepad.

The MDIDocVw shows how to use the MDI support and how frames and views interact.

The GuidGen sample looks exactly like the GUIDGEN utility COM developers have used for years. It uses a single dialog box class to manage the various UI features of generating GUIDs.

As a test of the installation, it is recommended to load up one of the examples in DevStudio, build it and run it.

Possible Problems with the Installation

There are a small number of common installation problems / questions. Here we will try and provide appropriate answers.

I cannot find the WTL AppWizard.

You must first manually copy it to the appropriate location – it is not done automatically during installation. Note that in DevStudio's *New Project* dialog it is called the *ATL/WTL AppWizard*.

The Examples GuidGen and MdiDocVw work fine, but MTPad compiles and links OK, but crashes when it runs.

Tut, tut, tut: you are not reading the output from the Build!

When you compile stdatl.cpp, the following message is display in the Build tab of the output window:

```
NOTE: WINVER has been defined as 0x0500 or
greater which enables Windows 2000
features.

Caution: When building applications with
WINVER set to 0x0500 or greater, the
resulting binary may not run as expected
on earlier platforms, such Windows NT 4.0
or Windows 95 & Windows 98

See the Platform SDK release notes for
more information.
```

What is happening is that stdatl.h in the MTPad example has `WINVER` set to `0x0500`, and this is used in various header files to determine if Windows 2000-specific material should be included. The above comment is outputted due to some code in windows.h, which checks `WINVER` and outputs the message if it is as `0x0500` or greater. This means that MTPad will run fine on Windows 2000 but will fail on older OSes, such as Windows NT4, where the problem will manifest itself as a failed `ATLASSERT` just after a call to the Win32 API `::GetMenuItemInfo` in the WTL header file AtlCtrlw.h. You can comment out the block of code calling `::GetMenuItemInfo`, in which case the menu will not be displayed on Windows NT 4, but apart from that the MTPad application will run correctly (it does have a toolbar for also accessing commands). Alternatively, comment out the lines:

```
// #define WINVER       0x0500
// #define _WIN32_WINNT 0x0500
```

from stdatl.h file, and the application will then run on older OSes. Actually you really only have to remove the `WINVER` define, and MTPad will then run fine, but it makes sense to remove both: check out the MSDN article titled "Header File Conventions", which states *"In general, applications expected to run on Windows 95 should be built without defining _WIN32_WINNT."*

In the MTPad example no Windows 2000-specific features are used, so these defines are not needed. However, note that for some applications you will wish to use Windows 2000 features, and therefore will definitely wish to include these lines. You will need to decide to either have a special code path in your application for older OSes, and another for Windows 2000 and later OSes, or you will decide you will only run on Windows 2000 (for example, for other reasons it might be necessary – e.g. you are using the thread pool functions which are exclusive to Windows 2000 or later). In the latter case it would be a good idea at application startup to detect if the OS can run the application, and if not to display a simple error message, rather than crashing mid-way through, as in the case of the default MTPad on non-Windows 2000 OSes.

I have not added WTL's include directory to DevStudio's directory list, yet the samples compile and run fine?

Yippee, it works, so what's the problem! However, you are quite right to be concerned, as such "magical" solutions often have a nasty way of coming back to haunt you with difficult-to-detect problems (usually within a few hours of a your product's release!). The files must be coming from somewhere, but where?

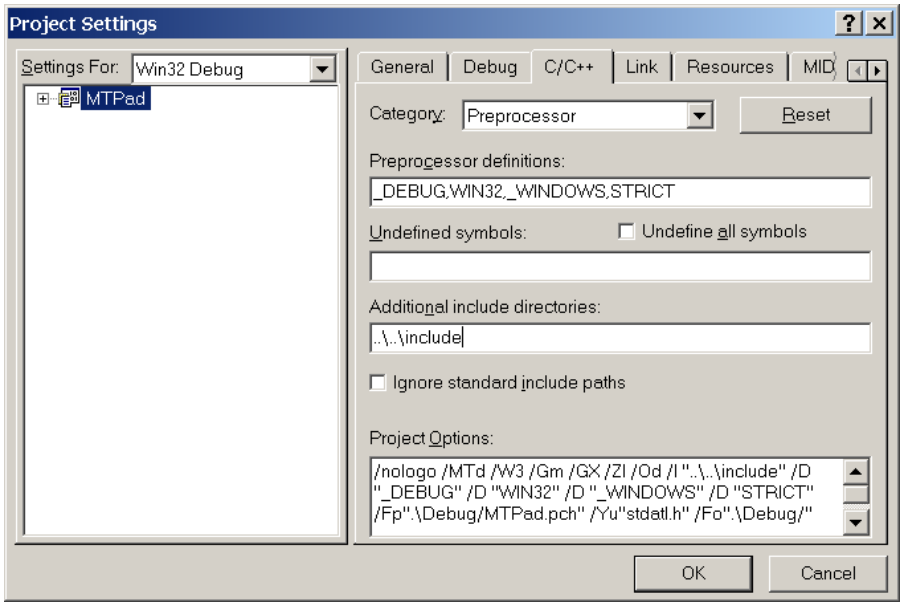
Your first worry might be it is coming from either the Platform SDK's own Include directory or one of the include directories under VC++, such as ATL's.

It is a worry because you might be getting an old version of the headers and you definitely want the latest. Also, going forward, it is vital that you are sure which version and the location of the header files you are using, as new releases of WTL are likely. All members of a development team using code based on WTL should be building off the same version of the WTL headers.

So you will start a search of your entire hard disk, looking for say, WTL's "atlapp.h" file. (All WTL's header files are named beginning with "atl", which provides a good hint as to the likely packaging arrangements when Visual Studio 7 is launched).

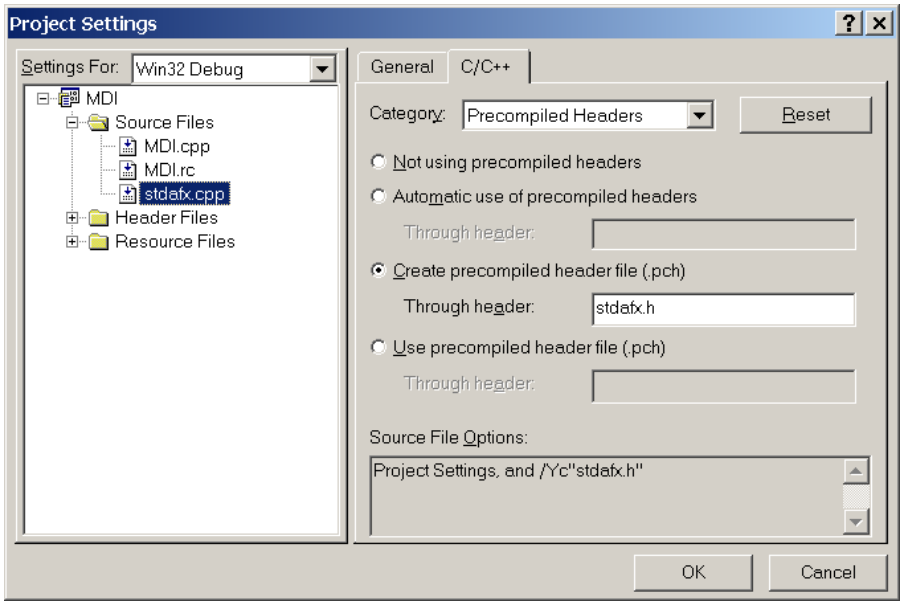
You will not find the WTL files in any location other than <SDK>\Src\WTL\include. What has happened with the examples is that per example, they have a relative include path configured, pointing to the location of the WTL includes relative to the WTL examples.

This solution is fine for the WTL examples, but it is highly unlikely that all your code will be located under <SDK>\Src\WTL\examples, so you should follow the suggestion of modifying DevStudio's options as explained earlier. Also, it means that inside DevStudio's text editor, when you select a WTL header file in source code, right click, and select *Open Document <atlXXX.h>,* that with the ..\..\include method the file will not be displayed, but with the DevStudio's options techniques it does work, and the appropriate WTL header file is displayed.

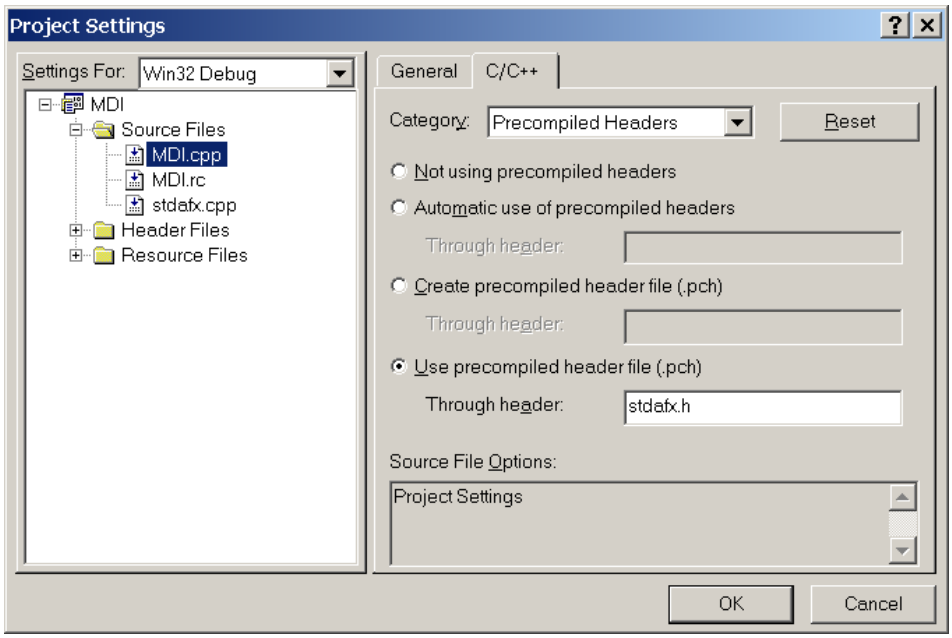


The GUIDGEN and MTPAD examples do not use stdafx.cpp & stdafx.h

The title AFX is a historical naming artifact, and has no coding significance.



AFX started off as the project name for the earliest MFC project, before it was christened MFC. The AFX name has survived through later MFC versions by being embedded in the stdafx.cpp and stdafx.h files and is part of the name of certain MFC global functions. By MFC tradition, stdafx.cpp contains boilerplate code which only needs to be added once and stdafx.h contains includes of other framework header files, general application header files and definitions of versions (e.g. `WINVER`). For each project these files are slightly different, and therefore the files are part of the application source files as opposed to the framework's. Both files tend to be quite small. Most developers know what should go in each.



Stdafx.cpp and stdafx.h are also traditionally the files used to manage precompiled headers. One .cpp file (in this case stdafx.cpp) is selected for creating the PCH file

based on a selected header file (stdafx.h), and then all other files use the PCH, based on the same header.

When ATL got started, it also used stdafx.cpp and stdafx.h for the same purposes, though obviously their contents were quite different and contained absolutely no MFC material.

Now with WTL, we also see the use of stdafx.cpp and stdafx.h in AppWizard generated code. (Is AFX a virus or what!).

So, to return to the question, why do WTL's examples MTPad and GuidGen not use stdafx.cpp and stdafx.h? The answer is for no particular reason, but probably just that the example developers wanted to break with Visual C++ tradition – the WTL AppWizard has obviously much greater respect for our coding heritage. In brief, it does not matter. So long as the appropriate WTL header get included and the pre-compiler header files get set up correctly, you may choose any file names (it is suggested that you use stdafx.[h/cpp], as that is what the WTL AppWizard uses).

Resources

WTL currently has few available resources, but this is likely to change shortly.

Development Tools

The Microsoft Platform SDK contains the WTL itself, and apart from that there are currently no other development tools available. It is likely that many tool vendors who currently based their products on MFC will migrate to WTL.

Internet Sites

Some Internet sites with interesting WTL material are:

- http://www.develop.com/dm/dev_resources.asp (very comprehensive)
- http://www.idevresource.com/wtl
- http://www.codeproject.com/wtl
- http://www.argsoft.com/Wtl/DocView.html
- http://www.sellsbrothers.com/tools/index.htm#wtl

Newsgroups and Discussion List

There are no newsgroups or discussion lists dedicated specifically to WTL programming. However, those targeted at ATL have many WTL-related postings.

The best ATL discussion list is:

- http://discuss.microsoft.com/archives/atl.html

The best ATL newsgroup is:

- news://msnews.microsoft.com/microsoft.public.vc.atl

Books

Apart from the WTL Developer's Guide which you are currently reading, there is no other WTL book available. It is expected that most of the major software engineering book publishers will provide books on WTL in the future as its popularity increases.

Many ATL books cover ATL windowing, and this is a crucial foundation for WTL. The best ATL books are:

- "Professional ATL COM Programming", Dr. Richard Grimes, Wrox Press, 1998, ISBN: 1-861001-40-1 (he also has a beginner's guide and a reference manual published with Wrox)
- "ATL Internals", Brent Rector and Chris Sells, Addison-Wesley, 1999, ISBN: 0-201-69589-8
- "Creating Lightweight Components with ATL", Jonathan Bates, SAMS, 1999, ISBN: 0-672-31535-1 [interesting discussion of adding support for Active Documents to ATL applications]
- "Inside ATL", George Shepherd and Brad King, Microsoft Press, 1999, 1-57231-858-9

Journals

Visual C++ Developer Journal (http://www.vcdj.com) published an article on WTL in its April 2000 issue.

Target Audience For This Developer's Guide

This developer's guide is aimed at experienced software developers who have a good knowledge of C++ and some exposure to C++ templates, ATL, Win32 windowing and MFC UI programming.

Chapter Contents

Chapter 2 – Win32 SDK Windowing reviews core concepts of Win32 windowing, which must be clearly understood before moving on to ATL windowing and WTL.

Chapter 3 – ATL Windowing examines the functionality within ATL relating to windowing. This includes constructing windows and dialogs, handling messages and command processing through message maps and super/subclassing. WTL does not replace this; rather it extends the range of functionality already available with ATL windowing.

Chapter 4 – WTL Quick Tour provides an overview of developing with WTL. It introduces the WTL build process, application architecture and describes a number of development issues, such as CRT usage. It has a list of WTL's templates and classes, a description of WTL's `CString`, and explains the macros used.

Chapter 5 – WTL AppWizard walks through the coded generated from the various AppWizard options. This chapter examines each AppWizard option and analyses the code changes that it effects.

Chapter 6 – Dialogs and Controls looks at how dialog boxes and their contents work with WTL. It looks at how a variety of standard, common and ActiveX controls can be used, along with common system dialogs and property pages. It looks at the message crackers. It describes DDX, data validation and how to extend these.

Chapter 7 – Graphical Primitives looks at outputting graphics. It introduces the WTL templates for GDI objects, such as `CDC`, `CPen`, `CBrush`, `CFont`, `CBitmap`, `CPalette` and `CRgn`. It explains the role of `CDC` derivatives, for painting client-area rendering, complete window rendering and enhanced metafiles.

Chapter 8 – Internals of WTL walks through the WTL header files and examines the design for these templates.

Chapter 9 – Bugs and Suggestions lists known bugs in the current version of WTL and makes some small and large suggestions above extending its capabilities.

Chapter 2
Win32 SDK Windowing

Objectives

The objectives of this chapter are to:

- Review the fundamentals of windowing concepts
- Introduce windowing terms
- Explain how the user interface is structured in the Windows OS
- Describe how the Windows OS handles concepts such as message queues, subclassing, superclassing, the message loop and general windows management
- Examine how threads and windows interact
- Describe how graphics primitives work

Fundamental Windowing Concepts

WTL is built above ATL windowing, which in turn is built above Win32 SDK windowing. Before exploring WTL, it is useful to review how windowing works within the Windows family of operating systems, which we do in this chapter, and how ATL handles windowing, which we will do in the next chapter. There are a number of important Win32 concepts and if clearly understood they will facilitate understanding why ATL and WTL are structured the way they are.

Types of Windows

At the most basic level, everything that appears on screen on a PC running the Windows OS is either a window or a graphical primitive drawn within a window. Win32 APIs are provided to create, manage and destroy windows and to render graphics inside them. Each user interface construct lives within a separate window – known as a control.

The end-user might think there are many types of windows – graphical output windows, user interface standard controls (pushbutton, radio-button, etc.), common controls (listview, treeview) and ActiveX controls. (A special case of ActiveX control, known as windowless - can operate without its own window – it lives within the window of its container). Fundamentally, they are all instances of windows that work according to the same set of rules.

Hierarchies of Windows

Windows are arranged on screen in a hierarchy. The root desktop window is a listview control, in large-icon mode. Applications' top-level windows and their child windows form nodes within this hierarchy, which is maintained by the OS.

When you create a window, you must specify its parent. The new window will be located within the hierarchy under the node of the parent window. When a window is destroyed, all its child windows are automatically destroyed.

Window Classes

A window class is a description of important aspects of how a window works. Each window you create must be of a particular window class. The operating system provides window classes for the standard controls, such as Button, Edit and List Box and the common controls, such as listview, treeview and calendar picker.

A window class is registered using the Win32 `RegisterClassEx` function, and unregistered using the `UnregisterClass` function.

```
ATOM RegisterClass( //Return is an Atom identifier for the class
 CONST WNDCLASS *lpWndClass // IN: pointer to window
                            // class structure
);
BOOL UnregisterClass( //Return states whether it succeeded
  LPCTSTR lpClassName,  // lpszClassName field from WNDCLASS
  HINSTANCE hInstance   // handle to application instance
);
```

Sometimes a window class is referred to as a "template" for a window. In programming terms, a window class is a Win32 structure (not a C++ class). The most important aspect of a window class is the window procedure. Other aspects include an icon when the application is minimized, background brush, default cursor and the `hInstance` of the owner (EXE or DLL containing the WndProc). A process should create a window class and then instantiate windows based on it.

```
typedef struct _WNDCLASS {
    UINT       style; // Bitmask containing additional settings
                      // (e.g. CS_NOCLOSE - Do not show CLOSE
    WNDPROC    lpfnWndProc;//The windows procedure function
    int        cbClsExtra; // Extra bytes for this class
    int        cbWndExtra; // Extra bytes for each window
    HINSTANCE  hInstance;// The instance handle of the EXE/DLL
                        // containing the window procedure
    HICON      hIcon;   // Icon for the class
    HCURSOR    hCursor;       // Cursor for the class
    HBRUSH     hbrBackground; // How the background should
                              // be drawn
    LPCTSTR    lpszMenuName; // Resource name for the menu
    LPCTSTR    lpszClassName; // Class name
} WNDCLASS, *PWNDCLASS;
```

Important styles include:

- `CS_SAVEBITS` – copy the pixels in a window before it is obscured, and replace the pixels when the window becomes visible again (use for small windows only)
- `CS_NOCLOSE` – Disable the *close* button on the window menu
- `CS_CLASSDC` – use a unique device context to be shared among all windows of this class (NOT RECOMMENDED for multithreading)
- `CS_OWNDC` – use a unique device context for each window of this class
- `CS_PARENTDC` – enables a child window to draw into its parent (a common DC from the system's cache is used)

Extra application-specific bytes can be stored with the class (`cbClsExtra`) and each window of the class (`cbWndExtra`)

The class name is used later when creating windows – as one argument to CreateWindow[Ex] states the name of the window class that the new window should be based upon.

The Window Procedure

Window messages are constantly being sent to windows to be processed, and they handle these in a window procedure. This is a function specific to the application and class that detects messages of relevance to the application and responds, usually by calling another function within the application. Messages not of interest are forwarded to the default handler (`DefWindowProc`). It is quite normal that multiple windows could be based on the same window class, which means that multiple windows could have their messages handled by the same window procedure (a `HWND` parameter identifies the window to which the message refers).

The Window

Calling CreateWindowEx creates a window. The name of a window class that has already been registered must be provided (this parameter may not be `NULL`).

```
HWND CreateWindow(
  LPCTSTR lpClassName,  // Name of window class
  LPCTSTR lpWindowName, // Name of window, to appear in titlebar
  DWORD dwStyle,        // Bitmask of style of window
  int x,                // X&Y position of new window compared
  int y,                // to parent window
  int nWidth, nHeight,  // dimensions of window
  HWND hWndParent,      // parent window
  HMENU hMenu,          // menu handle
  HINSTANCE hInstance,  // Module associated with the window
  LPVOID lpParam        // custom data to be sent with WM_CREATE
);
```

STEP 1: Application provides a WndProc function

```
LRESULT CALLBACK MyWndProc(HWND hwnd,
UINT uMsg,WPARAM wParam,LPARAM lParam ){

switch (uMsg) {

        case WM_CREATE:
            // …;    return 0;

        case WM_PAINT:
            // …;    return 0;
    ...
```

STEP 2: Application calls RegisterClassEx

Application fills in the `WNDCLASSEX` structure, including setting the class name to a application-defined string (e.g. "MyFirstClassName") and setting the `WNDPROC` field to point to an application defined function (e.g. `MyWndProc`) whose signature is `WNDPROC` – then the application calls `RegisterClassEx` to register it.

This results in an atom getting added to an atom table based and the class name, and the system using the atom identifier to locate the window class for it

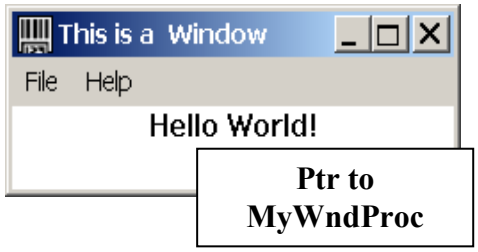
Atom table for registered window classes	
MyFirstClassName	**Atom Identifier X**
. . .	**Atom Identifier Y**
. . .	**Atom Identifier Z**

Window Class Stored information for atom identfiier X
WndProc=ptr to MyWndProc
. . .

STEP 3: Application calls CreateWindowEx

Application calls `CreateWindowEx` and among it parameters is the name of the window class to use

This results in a new window being created. Each window has certain pieces of hidden data, one of which is a pointer to a wndproc function. This is initialized to point to the `wndproc` identified in the window class



STEP 4: Messages are processed for window

When messages are received for a window, its ptr to a `wndproc` is used to process the message

Sub-Classing

When a window is created based on a window class, the window records a function pointer to the window procedure specified in the window class. Subclassing is used to modify the behavior of a class, without having to rewrite the class entirely, possibly duplicating lots of code. Subclassing involves an application replacing the window procedure of a window with a different window procedure, which can process some messages it receives and pass them onto the original window procedure. This can be used for window classes you write yourself or those which are provided by the operating system for predefined controls.

STEP 1: Application calls RegisterClassEx

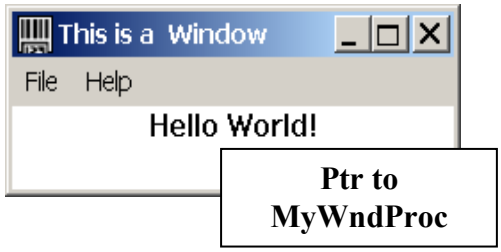
Application calls `RegisterClassEx` as before

Atom table for registered window classes	
MyFirstClassName	**Atom Identifier X**
. . .	**Atom Identifier Y**
. . .	**Atom Identifier Z**

Window Class Stored information for atom identfiier X
WndProc=ptr to MyWndProc
. . .

STEP 2A: Instance Subclassing

With instance subclassing, the window needs to be created as before using `CreateWindowEx`. It gets its wndproc from the registered wndproc value for the window class



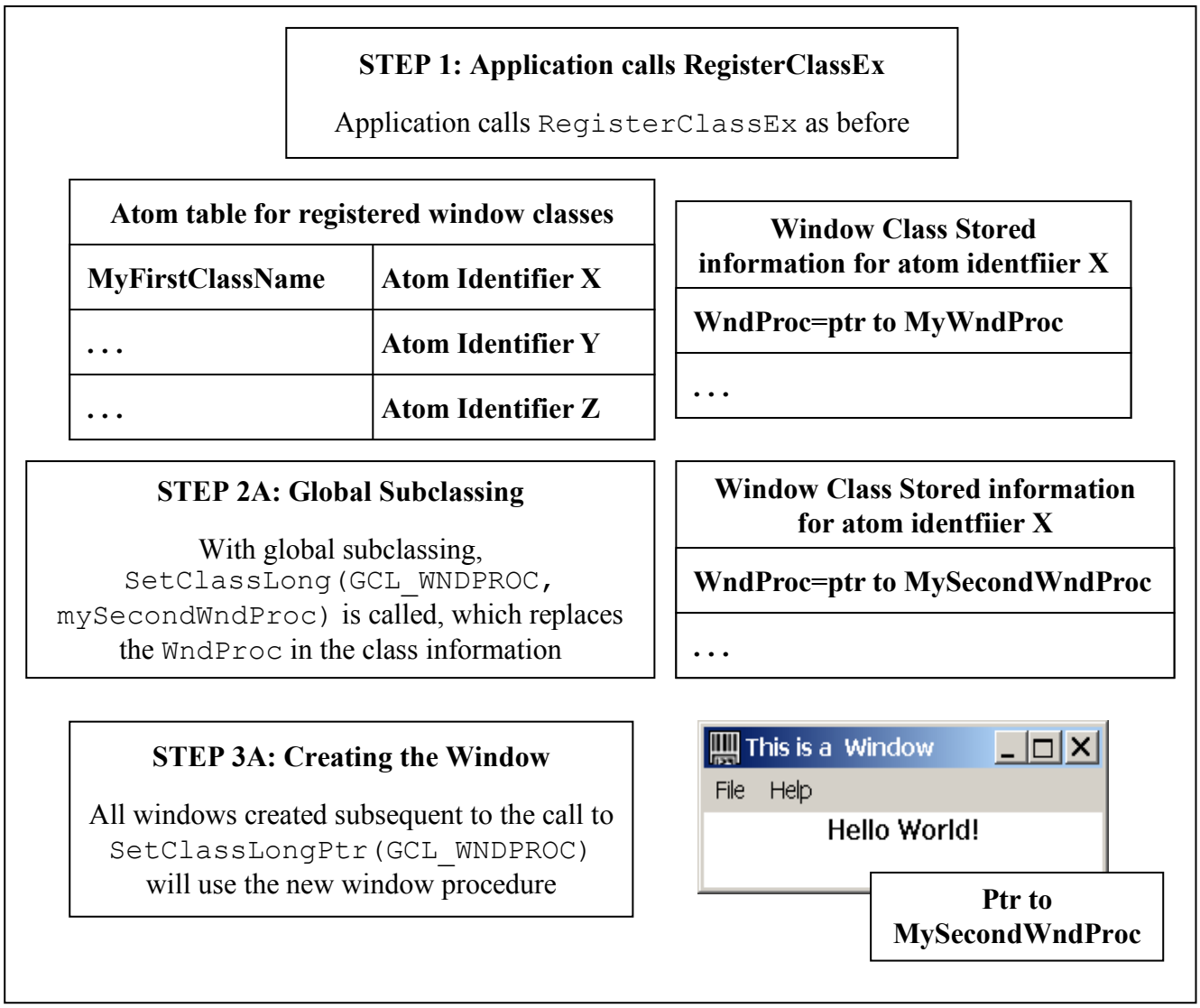
STEP 3A: Instance Subclassing

Then, to carry out instance subclassing, write a separate window procedure (e.g. `mySecondWndProc`) and then call `SetWindowLongPtr(GWL_WNDPROC, mySecondWndProc);` Note that the return value from this call will the existing wndproc in use. In the implementation of `mySecondWndProc`, for certain messages one could call `myWndProc` if desired).

Instance subclassing is used for a single instance of an existing window. This can be done by calling `SetWindowLong(GWL_WNDPROC)` or `SetWindowLongPtr(GWL_WNDPROC)` for an existing window (the difference

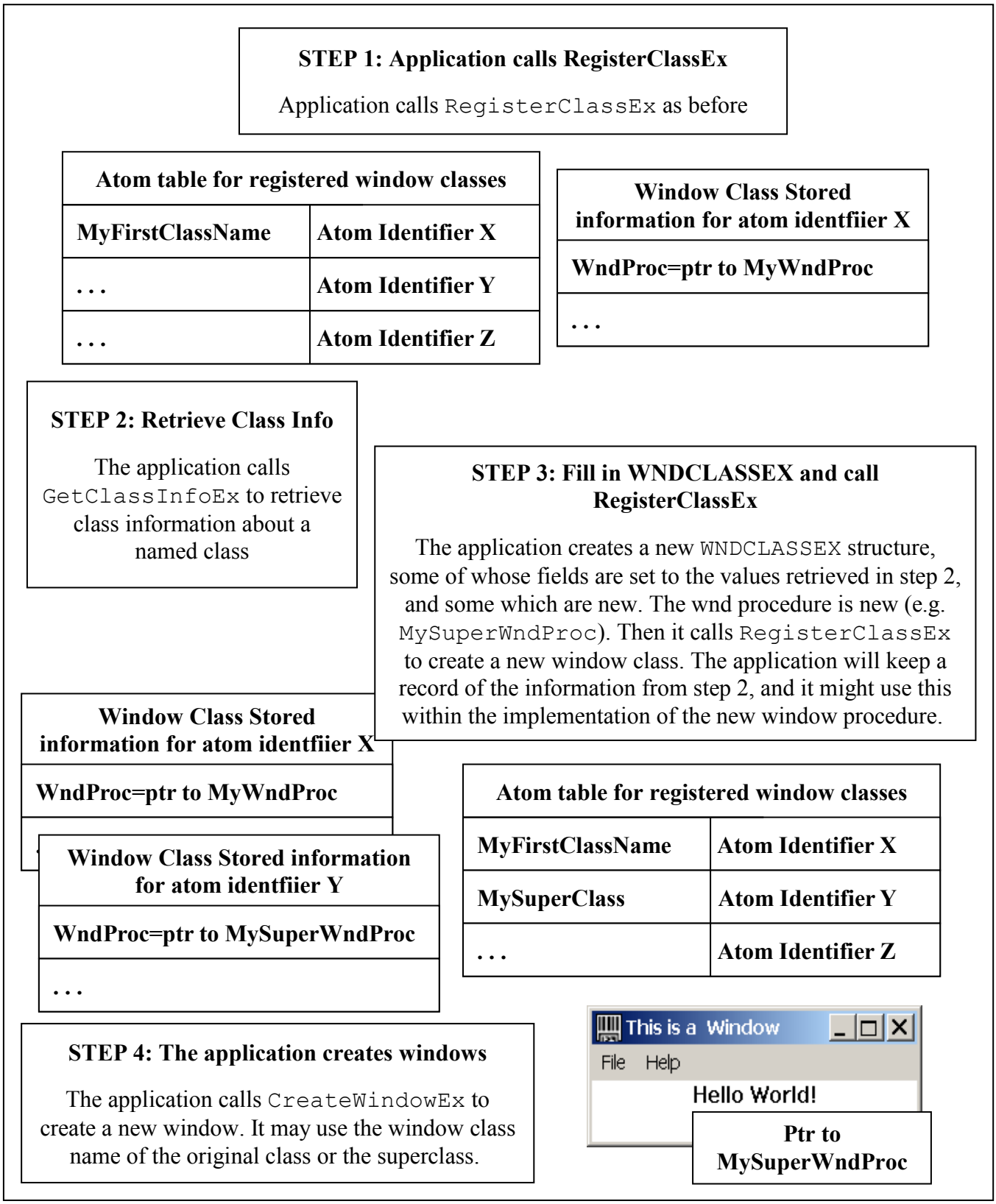
between the functions is that `SetWindowLong` is compatible with Win32 only, whereas `SetWindowLongPtr` is compatible with both Win32 and Win64, so it is better to use `SetWindowLongPtr`).

Global subclassing is to replace the window procedure in the window class, which will come into effect for all subsequent window creation based on that window class. This is done by calling `SetClassLongPtr(GCL_WNDPROC)` to replace the window procedure stored in the window class. All new windows created after this will new the new windows procedure. Windows created before this will continue to use the initial window procedure.



Super-Classing

Superclassing involves extending the functionality of a base window class. In the superclass' own window procedure, when it receives a message it may process it directly or invoke the window procedure of the base class.

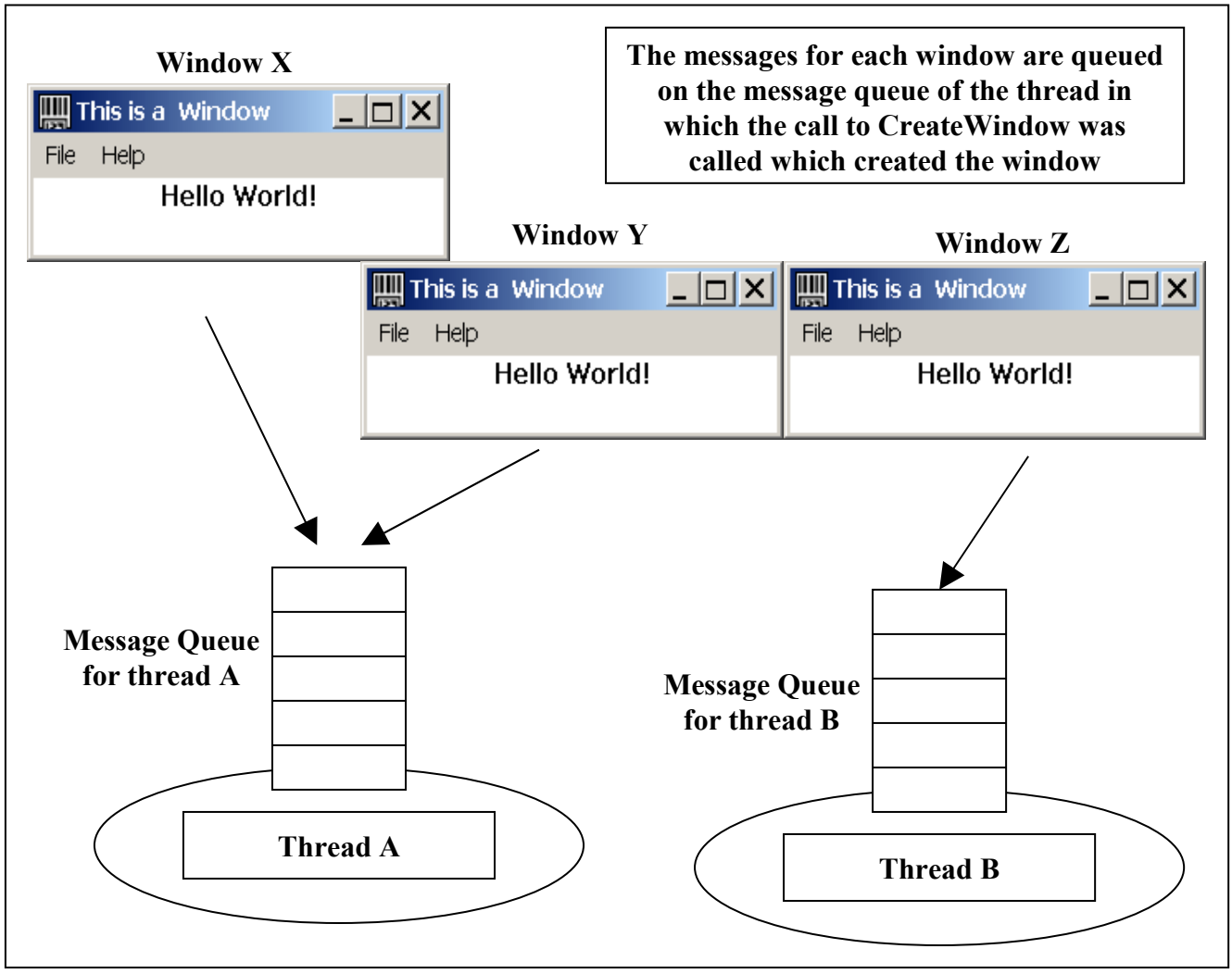


This is used by calling `GetClassInfo` to retrieve information about the window class of the base class, modify the data returned (such as the window procedure) and then calling `RegisterClassEx` to register the superclass.

You will often use subclassing and superclassing when trying to modify the behavior of predefined windows controls, such as edit boxes and list boxes. Subclassing and superclassing are allowed only within a process.

Message Queues

Each thread that invokes Win32 windowing functions has its own message queue. The message queue of the thread in which a window is created is used to deliver messages concerning that window to the application.



Message loop

Each thread that has a message queue should process incoming messages by means of a message loop. This calls the `GetMessage` function, which blocks until a message becomes available, and then calls `DispatchMessage`, which results in the appropriate window procedure getting called.

```
MSG msg;
while (GetMessage(&msg, 0, 0, 0))
    DispatchMessage(&msg);
```

The message loop is also the basis of single-threaded apartments in COM. The idea is that if a component is designed to be called from only has a single thread, and multiple threads inside or outside the application wish to call it, a hidden window is created, and

all functions calls for a particular COM component are posted as messages to this window. The object can then extract them one by one and process them.

Capture and Focus

Normally messages are sent to the window in which they occurred. For user input this is the focus window. For mouse messages, many applications wish to detect the corresponding mouse up message for every mouse down message they receive. Calling SetCapture does this.

Window Styles (Traits)

Application developers may set a wide variety of window styles when creating windows and these influences the look and behavior of the window. There are two types of styles – standard styles (e.g. `WS_CHILD, WS_CLIPCHILDREN, WS_VISIBLE`) and extended styles (e.g. `WS_EX_PALETTEWINDOW, WS_EX_APPWINDOW, WS_EX_CLIENTEDGE`).

Another term for "window styles" used within WTL/ATL is "traits".

Windows Properties

The operating system can maintain a set of named properties for each window. A property name is defined by the application and the property value is a `HANDLE`, usually a pointer to memory that the application allocates and fills in with data relevant to the window.

Sample – WindowingWithSDK

This sample explores programming windowing concepts using raw Win32 calls. Later, when we use ATL and WTL, it will be instructive to look back and see exactly how it could be done manually with the SDK calls. The WindowingWithSDK workspace contains five small projects.

The WindowsWithSDK project is a simple "Hello World" project, generated in Developer Studio using the *New Workspace* command, then selecting *Win32 Application*, and then selecting *"A typical "Hello World!" application"*. The code is all auto-generated by Developer Studio. It simply initializes a `WNDCLASSEX` structure and calls `RegisterClassEx` to register the window class. Then it calls `CreateWindow` to create a window based on the window class. It has a message loop which processing messages, and forwards them to the widow procedure.

The InstanceSubclassing project was generated in the same way and modified to show instance subclassing. An additional window procedure was written, called `MySubclassedWndProc`. This processes `WM_LBUTTONDOWN` messages and forwards all other calls to the original window procedure, using `CallWindowProc`.

```
LRESULT CALLBACK MySubclassedWndProc(HWND hWnd, UINT message, WPARAM
wParam, LPARAM lParam){
      switch (message) {
            case WM_LBUTTONDOWN:
            OutputDebugString(
               TEXT("Mouse Down detected in subclassed WndProc\n"));
```

```
                break;
            default:
return CallWindowProc(MyWndProc, hWnd,
                message, wParam, lParam);
   }
   return 0;
}
```

The subclassing is put into effect by a call to `SetWindowLongPtr`, after first creating the window.

```
      hWnd = CreateWindow(szWindowClass, szTitle,
            WS_OVERLAPPEDWINDOW,CW_USEDEFAULT, 0,
            CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
SetWindowLongPtr(hWnd, GWL_WNDPROC,
                        (LONG_PTR)MySubclassedWndProc);
```

The GlobalSubclassing project changes the window procedure in the registered class. The class is first registered as before. To change class settings with `SetClassLongPtr`, an existing window based on that class must be used (which indirectly provides access to the window class). This first window is created, and then `SetClassLongPtr` is called to change the window procedure to use for all new windows based on that class. (Note that the first window will continue to use the original window procedure and is not affected by the call to `SetClassLongPtr`).

```
hWnd1 = CreateWindow(szWindowClass, szTitle,
            WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0,
            CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
SetClassLongPtr(hWnd1, GCLP_WNDPROC,
            (LONG_PTR)MySubclassedWndProc);
hWnd2 = CreateWindow(szWindowClass, szTitle,
            WS_OVERLAPPEDWINDOW,CW_USEDEFAULT, 0,
            CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
```

The SuperClassing project demonstrates superclassing of an existing window class. The size field of the `WNDCLASSEX` is first correctly initialized. It extracts the window class of an existing class by calling `GetClassInfoEx`. It copies certain fields to a new `WNDCLASSEX` structure, and sets the window procedure to point to a new function. Then it calls `RegisterClassEx` to register the new window class. In the window procedure certain calls are forwarded to the original window procedure.

```
      wcex_base.cbSize = sizeof(WNDCLASSEX);
      GetClassInfoEx(hInstance, szWindowClass, &wcex_base);
      CopyMemory(&wcex_superclass, &wcex_base, sizeof(WNDCLASSEX));
wcex_superclass.lpfnWndProc   =
(WNDPROC)MySuperclassWndProc;
wcex_superclass.lpszClassName =
TEXT("SuperClassName");
      RegisterClassEx(&wcex_superclass);
      hWnd = CreateWindow(TEXT("SuperClassName"), szTitle,
                  WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0,
                  CW_USEDEFAULT, 0, NULL, NULL, hInstance,
                  NULL);
```

The MessagesAndThreads project creates a window in the original thread and then starts a new thread.

```
   hWnd = CreateWindow(szWindowClass, szTitle,
            WS_OVERLAPPEDWINDOW,CW_USEDEFAULT, 0,
            CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
   HANDLE hMyThread;
   DWORD MyThreadID;
   hMyThread = CreateThread(NULL, 0,
             (LPTHREAD_START_ROUTINE) MyThreadStartProc, hInstance,
            0, &MyThreadID);
```

In the new thread it creates a new window. Both threads maintain their own message loop. The rule is that the thread in which the window was created is used to queue messages concerning that window for the application. If one thread for example stopped processing messages (e.g. put in a long call to `Sleep`), then the messages for the windows created in that thread would not be processed.

```
DWORD MyThreadStartProc(LPVOID hInstance){
      HWND hWnd;
      MSG msg;
      hWnd = CreateWindow(szWindowClass, szTitle,
            WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0,
      CW_USEDEFAULT, 0, NULL, NULL,
            (HINSTANCE)hInstance, NULL);
      ShowWindow(hWnd, SW_SHOWNORMAL);
      UpdateWindow(hWnd);
      // Main message loop:
      while (GetMessage(&msg, NULL, 0, 0)) {
      if (!TranslateAccelerator(msg.hwnd,
                  hAccelTable, &msg)) {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
            }
      }
      return 0;// thread functioned correctly
}
```

Messages

The messages that are placed on message queues are identified by integers, and contain two parameters, `wParam` and `lParam`. The senders of messages could be the kernel code that manages the desktop/keyboard/mouse, the implementation of window controls, your application code or ActiveX controls. The recipient of messages is usually the window procedure associated with the window in which the action that precipitated the message occurred, but in some circumstances it can be the parent window.

Message IDs

The Platform SDK defines a range of messages related to generic windows, which are named `WM_XXX`. This will work with normal windows and the standard/common controls, which based on normal windows. So when a mouse is moved inside a window, the window procedure for that window gets a `WM_MOUSEMOVE` message. This is the case for all messages that begin with `WM_`, with three exceptions. `WM_COMMAND`, `WM_NOTIFY` and the various `WM_CTLCOLORXXX` messages are sent to the parent of the window in which the action took place.

Applications can send custom messages for application-specific communication. Values should be added to `WM_USER` to form a message id, and then the message sent to a window. The window procedure for that window will extract the message, subtract `WM_USER` and interpret the message.

To shut down an application, the `WM_QUIT` message is used.

Standard Controls vs. Common Controls

Windows provides a set of standard controls that are built into the kernel and are always available. These are buttons, combo-boxes, edit, listbox, rich-edit, scroll-bar and static control.

Note that the Rich-Edit control is a standard control, not a common control. In comes in three versions, each with varying degrees of support for rich text. Version 1 is supported in "classic" Windows 95 and all later OSes. Version 2 is supported with Windows 98 and Windows NT 4 and later, and may be installed on Windows 95 manually. Version 3 is available with Windows 2000 and Windows Me.

Windows also provides more advanced controls, known as common controls, which are available to applications if they call the Platform SDK APIs `InitCommonControls` or `InitCommonControlsEx` and if the system DLL Comctl32.dll with the correct version is available. "Classic" Windows 95 came with a core set of common controls, such as treeview, listview, and property pages. Later versions of Comctl32.dll, such as that delivered with IE 4 (version 4.71) contains additional controls, such as IPAddress, DateTime Picker, and MonthCalendar. IE5 does not add additional common controls, but it does add some new messages for existing controls (e.g. `TVM_GETLINECOLOR` for a treeview).

Getting/Setting Information

There is no real API to these controls. Instead, application code can send message to them to set data, get data and perform other actions. For example, to limit the number of characters permitted in an edit-box, an application could send it an `EM_SETLIMITTEXT` message, with the limit as the `wParam`. To retrieve this limit, the application can send `EM_GETLIMITTEXT` and the limit will be the return value from the `SendMessage` function.

Commands and Notifications

Applications often wish to use the standard and common controls and be informed of some of the messages for these controls. We have seen that an application cans subclass and superclass windows, including those used with these controls. In this way application code, rather than the default window proc for these controls can get first access to all the incoming messages and pass on any messages not of interest to the default window procedure.

For specialist cases this is fine, but it is considerable amount of work when you have many dialog boxes, and many controls inside each. In general, we wish to use the controls "as supplied". What would be nice is if the dialog box (the parent of theses controls in the window hierarchy) would be told of significant actions inside the

controls, such as a click in a button or text entry in an edit-box. Luckily for all of use, this is exactly how the OS behaves. It sends `WM_COMMAND` and `WM_NOTIFY` messages to the parent window with a notification message stating what happened in the child.

For each standard and common control, the OS decides that the parent of a control would be informed that certain ranges of actions have occurred. The control itself might be receiving a `WM_LBUTTONDOWN` message, but it passes a `WM_COMMAND` message to the parent, with a `BN_CLICKED` notification message inside it. The standard controls and the Animation common control use `WM_COMMAND`. All the other common controls use `WM_NOTIFY`.

Which Windows are in the Hierarchy

One good piece of advice when programming windowing code is to be clear in your own mind the hierarchy with which you are working. It can be the case that there are additional or fewer windows present than you actually think – and messages that are sent to parent windows might end up going to windows that are not expecting them. Two sample scenarios where this can occur concern ActiveX controls. Such controls can be implemented as "windowless", in which case the control itself does not have its own window, rather it rendered in the window of the dialog. When controls are hosted in a client development environment, a host window class can wrap them, which itself manages a window for the site, and this site window is situated in the window hierarchy below the dialog window (it is a child of the dialog window) and above the control window (it is the control window's parent).

Ample use of the Spy++ utility can help you clarify the hierarchy arrangements and save many hours of debugging.

Threads and Windowing

Threads and windows interact at a variety of levels. It is important we have a clear understanding of these interactions, to avoid a number of potential problems and to optimize performance.

On Win32, there is one and only one type of thread. It executes user-interface and non-user interface code alike inside a threadproc and functions called from the threadproc. There is no such thing as a "user-interface thread" from an OS perspective. Where the difference can arise is in application space because you as the application developer have decided that some code with UI calls will execute in some particular threads, and other code, without UI calls, will execute in different threads. It is a design decision you make and the OS is not interested. (The same argument applies to COM. There is no such thing as a COM thread, just a Win32 thread, which happens to execute COM, calls. What is different is that the rules of COM apartments also apply.)

Win32 lets you execute UI calls on any thread. You may decide for design reasons to be more selective and limit UI calls to certain threads. How window operations work in the context of threads needs to be appreciated so that appropriate design decisions may be made.

Three excellent books with coverage of threads and windowing are Richter's "Programming Applications for Microsoft Windows" – ISBN: 1-57231-996-8, chapter

26; Berveridge's/Wiener's "Multithreading Applications in Win32", ISBN: 0-201-44234-5, chapter 11 and Cohen's/Woodring's "Win32 Multithreaded Programming", ISBN: 1-56592-296-4, chapter 12.

Who Owns the Objects?

An application can create objects such as windows, pens, brushes etc. It is considered good programming practice to delete these when they are no longer necessary. Even if this is not done, the kernel will step in and at thread deletion time or process deletion time will delete the objects that are still in existence. Window objects (and hook objects) are owned by the thread in which they were created. When the thread exits then each window created within that thread will be closed immediately. All other objects (pens, brushes, regions, device contexts, bitmaps, fonts, etc.) are owned by the process, and regardless of which thread upon which they were created, they remain in existence until the object is explicitly deleted or the process (not the creator thread) is terminated.

Threads and Windows and Message Queues

The golden rule of Win32 windowing/thread interaction is that the thread in which a window was created (via a call to Win32's `CreateWindow[Ex]`) is also the thread that processes messages for that window. Most other aspects of Win32 windows/thread interaction flow from this rule.

There are a number of questions for which we need to find answers, to understand the implications of this rule. If a thread is going to receive window messages, then does it need a message queue? How does this get constructed? How are messages placed on the message queue? If a thread has a message queue, then it needs a message pump to extract messages from the queue and process them. How is this implemented? When a thread dies, then the windows associated with the thread might as well also die, as no further messages can be sent to them – how is this handled? What happens if the thread that has a message pump also wishes to wait on OS kernel objects, such as a mutex?

Some threads need a message queue and some do not. Some threads will never be used for windowing, and it would be inefficient to allocate memory for a message queue when it will not be used. Therefore, when the OS creates a thread, it does not automatically create a message queue. The very first time a call is made in a new thread to any of the Win32 windowing APIs, the OS creates the message queue. This is done transparently to application code. Once the message queue is in existence, it remains so until the thread dies.

The application needs to know which threads have a message queue and hence which threads it will need to add the familiar `GetMessage / DispatchMessage` message pump functions. The application knows this because the message queue only gets created when windowing calls are made in that thread. Therefore it is a design issue to decide where to make such calls and if present then to add the message pump.

Sending and Posting Messages

The two main techniques, which result in the window procedure getting executed, are known as sending and posting. Sending is done via:

```
LRESULT SendMessage( // return is result of message processing
```

```
  HWND hWnd,      // Window to receive the message
  UINT Msg,       // Message to send
  WPARAM wParam,  // Parameter to message
  LPARAM lParam   // Parameter to message
);
```

You can think of sending as synchronous. The caller thread will block until the message has actually being processed, and then the next line after the `SendMessage` in the thread will be executed. The `hWnd` parameter intrinsically identifies the thread that has the window procedure that needs to be executed. If this is the same thread as that on which `SendMessage` was called, then `SendMessage` internally simply calls the window procedure directly – without a thread context switch, and without placing the message on the message queue. If a different thread provides the window procedure, then a message is placed on its message queue. Some time later it processes the message, and sends back the reply, and then the thread in which `SendMessage` was called awakes and continues processing. It is possible to call `SendMessage` from one thread to send a message to a window created on a separate thread but the one major problem that can arise is if that separate thread is not servicing its message pump regularly, the first thread can block.

Posting is done via one of three functions.

```
BOOL PostMessage(// Return is result of posting
  HWND hWnd,      // Window to receive the message
  UINT Msg,       // Message to send
  WPARAM wParam,  // Parameter to message
  LPARAM lParam   // Parameter to message
);
```

This places the message on the message queue associated with the thread in which the identified window was created.

```
BOOL PostThreadMessage( // Return is result of posting
  DWORD idThread, // Thread to receive the message
  UINT Msg,       // Message to send
  WPARAM wParam,  // Parameter to message
  LPARAM lParam   // Parameter to message
);
```

This places the message on the message queue associated with the identified thread. The window parameter when received will be 0.

```
VOID PostQuitMessage( // no return
  int nExitCode   // Exit-code
);
```

This places a `WM_QUIT` on the message queue of the thread in which this call was made. Sometime later, when the thread retrieves the `WM_QUIT` message off the queue, it should exit gracefully.

Posting is asynchronous. The message is posted on the message queue and the function returns immediately. It does not wait and the message may or may not be correctly processed.

This normally works fine and enables different threads to communicate with each other, without requiring synchronization. The recipient thread can, at a time convenient to itself, take the message off its queue and process it in an orderly fashion.

One point of concern is what happens if the message queue does not exist – will a call to one of the Post functions from a different thread create it. The answer is no. The message queue for a particular thread will only get created inside that thread, when it calls any windowing function. This is important in the context of WTL's Multiple Threads SDI Application, which we will discuss further in detail in the upcoming WTL AppWizard chapter.

Threads and Kernel Objects

The `GetMessage` API is a blocking call, waiting for the next message to come in on the message queue. The `WaitForSingleObject` or `WaitForMultipleObjects` are blocking calls, waiting until one or more kernel objects become signaled. What happens it we wish to wait on both kernel objects and incoming window messages? For this the Win32 APIs `MsgWaitForMultipleObjects` and `MsgWaitForMultipleObjectsEx` can be used.

```
DWORD MsgWaitForMultipleObjects(
  DWORD nCount,          // handle count
  CONST HANDLE pHandles, // array of handles
  BOOL fWaitAll,         // what waiting needs to be done
  DWORD dwMilliseconds,  // Timeout
  DWORD dwWakeMask       // What input to detect
);
DWORD MsgWaitForMultipleObjectsEx(
  DWORD nCount,
  CONST HANDLE pHandles,
  DWORD dwMilliseconds,
  DWORD dwWakeMask,
  DWORD dwFlags          // Additional Flags
);
```

They will wait on an array of kernel object handles and detect incoming messages. They also have a timeout.

Chapter 3
ATL Windowing

Objectives

The objectives of this chapter are to:

- Describe ATL's windowing classes and templates
- Cover ATL message maps and chaining
- Explain how subclassing and superclassing work in ATL
- Introduce the concept of a contained window
- Explore ActiveX control containment within ATL's dialogs
- Examine the use of GDI functions with ATL windows

Overview

ATL provides comprehensive support for the full range of core windowing functionality. This includes sub-classing, super-classing, handling windows messages and notifications in a wide variety of means, and creating modal and modeless dialog boxes (including ActiveX control containment).

ATL does not offer a C++ class for each type of Windows standard control (button, combo box, edit-box, etc) or common control (list view, rebar, tree view, property sheets). Rather, it offers classes and templates to construct generic windows and allows you to specify the class name, which can be that of one of the standard or common controls. The concept of message maps is used to dispatch incoming messages to C++ member functions.

ATL windowing is the foundation for the Windows Template Library (WTL). WTL provides effective MFC UI-like C++ classes for each standard and common controls, templates for GDI drawing and higher-level user interface application features, such as toolbars, frames, views, statusbars, splitters and printing.

ATL provides limited wizard support. The ATL Object Wizard permits the creation of dialog boxes with support ActiveX control containment. All other window creation – dialog boxes without ActiveX control containment and generic windows, need to be created manually in code by instantiating the appropriate ATL templates. Dialog resources may be constructed and edited using Developer Studio's standard ResourceView. Through ClassView, you can access a wizard called *New Windows Message and Event Handler* that enables you to add handler functions for particular types of windows messages.

Windowing with ATL

ATL windowing provides an efficient layer of classes and templates above the low-level Win32 windowing APIs. The aim is to provide a C++ interface to windowing that is compatible and well integrated with other aspects of ATL. You can use ATL windowing on its own, but more likely it will be part of an application which also exposes COM functionality. ATL windowing is needed to display standard windows (e.g. displaying graphics or word-processing output), for dialog boxes (including but not limited to those which contain ActiveX controls) and for the visual aspects of an ActiveX control and its design-mode property sheets.

CWindow

The `CWindow` class is the main API between the client (the user of the window) and the implementation.

`CWindow` manages a `HWND` and provides type-safe member function wrappers to virtually all the APIs related to `HWND`s.

A typical member function of `CWindow` is implemented as:

```
BOOL SetWindowText(LPCTSTR lpszString){
      ATLASSERT(::IsWindow(m_hWnd));
      return ::SetWindowText(m_hWnd, lpszString);
}
```

The `m_hWnd` member needs to be initialized. This is done in one of the `Create` member function, which the client must call with appropriate parameters.

```
HWND Create(LPCTSTR lpstrWndClass, HWND hWndParent,
 RECT& rcPos, LPCTSTR szWindowName = NULL, DWORD dwStyle = 0,
 DWORD dwExStyle = 0, UINT nID = 0, LPVOID lpCreateParam=NULL){
           m_hWnd = ::CreateWindowEx(dwExStyle,
                            lpstrWndClass, szWindowName,
                            dwStyle, rcPos.left,rcPos.top,
                            rcPos.right - rcPos.left,
                            rcPos.bottom - rcPos.top,
                            hWndParent, (HMENU)nID,
                            _Module.GetModuleInstance(),
                            lpCreateParam);
           return m_hWnd;
     }
     HWND Create(LPCTSTR lpstrWndClass, HWND hWndParent,
                 LPRECT lpRect = NULL,
                 LPCTSTR szWindowName = NULL,
                 DWORD dwStyle = 0,
                 DWORD dwExStyle = 0,
                 HMENU hMenu = NULL,
                 LPVOID lpCreateParam = NULL) {
           if(lpRect == NULL)
                 lpRect = &rcDefault;
           m_hWnd = ::CreateWindowEx(dwExStyle,
                            lpstrWndClass, szWindowName,
                            dwStyle, lpRect->left,
                            lpRect->top,
                            lpRect->right - lpRect->left,
                            lpRect->bottom - lpRect->top,
                            hWndParent, hMenu,
```

```
                                _Module.GetModuleInstance(),
                                lpCreateParam);
        return m_hWnd;
    }
```

Effectively, the `Create` member functions are very thin wrappers around the Win32 API `CreateWindowEx`.

If we assume the client has already registered a window class with the name `szWindowClass` using `RegisterClassEx`, a `CWindow` may be instantiated using:

```
CWindow wnd;

wnd.Create(szWindowClass, 0, 0,
     TEXT("CWindow Demo"),
     WS_OVERLAPPEDWINDOW | WS_VISIBLE);
```

As we have seen, this results in a call to `CreateWindowEx`.

Sometimes a client has an existing `HWND`, and to use this via the `CWindow` wrappers it can pass in the `hwnd` as a parameter to the `CWindow` constructor, and not bother to call `Create` (the window already exists, so there is no need for an additional call to `CreateWindowEx`).

```
CWindow wnd(hwnd);
wnd.SetWindowText(TEXT("New Text"));
```

When handling ActiveX control containment, an additional template, called `CAxWindowT`, is used to provide the extra support needed. We will cover this template later.

Note that when a `CWindow` goes out of scope the `HWND` attached to it is NOT destroyed (The `CWindow` class actually does not have a destructor). If you wish to destroy the `HWND` associated with a `CWindow` you must directly call `CWindow::DestroyWindow`.

From a client's perspective, the use of `CWindow` is fine as it provides us with access to the features we require.

Sample - WindowingWithATL

The WindowingWithATL project is a simple example showing how to use `CWindow`. It was created using the ATL AppWizard and *"Executable"* was selected in the project type options. No COM objects were inserted as we are focusing exclusively on the windowing features. The AppWizard generated a file WindowingWithATL.cpp, and this contained a `WinMain` with a message loop.

To use windowing we first need to include `atlwin.h`. A good place to put this in `stdafx.h`, somewhere after the inclusion of `<atlbase.h>`.

```
// stdafx.h
#include <atlcom.h>
#include <atlwin.h>
```

In our `WinMain` function we add code to register a window class using the Win32 SDK API `RegisterClassEx` and then we create a window based on it using `CWindow`.

```
TCHAR szWindowClass[]=TEXT("myapp");
WNDCLASSEX wcex;
wcex.cbSize       = sizeof(WNDCLASSEX);
wcex.style        = CS_HREDRAW | CS_VREDRAW;
wcex.lpfnWndProc  = (WNDPROC)MyWndProc;
wcex.cbClsExtra   = 0;
. . .
RegisterClassEx(&wcex);
CWindow wnd;
wnd.Create(szWindowClass, 0, 0, TEXT("CWindow Demo"),
                          WS_OVERLAPPEDWINDOW | WS_VISIBLE);
```

We must create a traditional window procedure, and to access the window through the `CWindow` API we first instantiate a `CWindow` based on the `HWND` passed to our message loop.

```
LRESULT CALLBACK MyWndProc(HWND hWnd, UINT message, WPARAM wParam,
LPARAM lParam){
      PAINTSTRUCT ps;
      HDC hdc;
      TCHAR szHello[]=TEXT("Hello");
      switch (message) {
            case WM_PAINT:
                  . . . ;
            case WM_LBUTTONDOWN:{
                        CWindow wnd(hWnd);
                        wnd.SetWindowText(TEXT("New Text"));
                  }
                  break;

            . . .
   }
   return 0;
}
```

From an implementer's perspective (and the person who implements a window is very often the same person who is going to use it as a client), we have two problems: the window class and the window procedure are still handled as with SDK programming.

Window Construction

A group of classes, templates and macros based around `CWindowImpl` and its parent, `CWindowImplBaseT`, are provided to facilitate the implementation of more advanced windowing functionality, such as window classes and window procedures.

With `CWindowImpl`, the implementer may add as much or as little functionality as s/he desires, and leave the remaining to default to that provided by `CWindowImpl`. Each important piece of functionality is separately represented and may easily be configured as needed.

Window Traits

When creating windows numerous styles such as `WS_CHILD` or `WS_OVERLAPPEDWINDOW` may be specified in the call to `CreateWindowEx`. Instead of having to specify the same set of styles again and again throughout your code, ATL provides a template called `CWinTraits` that stores the styles to use, and they may be passed as a parameter to `CWindowImpl` when needed.

```
template <DWORD t_dwStyle = 0, DWORD t_dwExStyle = 0>
class CWinTraits{
public:
      static DWORD GetWndStyle(DWORD dwStyle)   {
      return dwStyle == 0 ? t_dwStyle : dwStyle;
      }
      static DWORD GetWndExStyle(DWORD dwExStyle)     {
      return dwExStyle == 0 ? t_dwExStyle:dwExStyle;
      }
};
```

You specify the set of standard and extended styles when instantiating the template. The two functions, `GetWndStyle` and `GetWndExStyle` return these values. Note that both values default to 0 if not specified.

There are a number of frequently used sets of styles, and for these ATL provides predefined `CWinTraits`. It also provides a `NULL` trait.

```
typedef CWinTraits<WS_CHILD | WS_VISIBLE |
            WS_CLIPCHILDREN | WS_CLIPSIBLINGS, 0>
            CControlWinTraits;
typedef CWinTraits<WS_OVERLAPPEDWINDOW |
            WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
            WS_EX_APPWINDOW | WS_EX_WINDOWEDGE>
            CFrameWinTraits;
typedef CWinTraits<WS_OVERLAPPEDWINDOW | WS_CHILD |
            WS_VISIBLE | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
            WS_EX_MDICHILD>
            CMDIChildWinTraits;
typedef CWinTraits<0, 0> CNullTraits;
```

There is an extra template called `CWinTraitOR` that logically ORs additional styles with those already set in a `CWinTraits`.

```
template <DWORD t_dwStyle = 0, DWORD t_dwExStyle = 0,
            class TWinTraits = CControlWinTraits>
class CWinTraitsOR{
public:
      static DWORD GetWndStyle(DWORD dwStyle)   {
            return dwStyle | t_dwStyle |
                  TWinTraits::GetWndStyle(dwStyle);
      }
      static DWORD GetWndExStyle(DWORD dwExStyle){
            return dwExStyle | t_dwExStyle |
                  TWinTraits::GetWndExStyle(dwExStyle);
      }
};
```

The word "trait" is used in the ISO C++ standard when modeling separately a collection of attributes of some other class/template. An example of its use is `char_traits` which is a parameter to the string template `basic_string`. This explains why the

term is used in ATL when naming CWinTraits (CWinTraits is the only use of the term in ATL in the current version, but this could change in future). The term is also used in the Windows Template Library.

Window Class Information

ATL provides a class to manage the window class information. This is called _ATL_WNDCLASSINFO and is implemented as follows:

```
struct _ATL_WNDCLASSINFOW{
      WNDCLASSEX m_wc;
      LPCTSTR m_lpszOrigName;
      WNDPROC pWndProc;
      LPCTSTR m_lpszCursorID;
      BOOL m_bSystemCursor;
      ATOM m_atom;
      TCHAR m_szAutoName[13];
      ATOM Register(WNDPROC* p){
            return AtlModuleRegisterWndClassInfoW(
                                          &_Module, this, p);
      }
};
```

The string CWndClassInfo maps to an ASCII or Unicode version of _ATL_WNDCLASSINFO as needed. Note that the third member, pWndProc, is a pointer to the window procedure to use.

The Register member function first checks to see if the named class has already been registered, and if not, then calls the Win32 API RegisterClassEx to register a new class. If the specified class name is NULL then a new one is generated.

Application developers normally do not use CWndClassInfo directly – rather they use one of three macros, DECLARE_WND_CLASS, DECLARE_WND_CLASS_EX or DECLARE_WND_SUPERCLASS. The CWindowImpl template has DECLARE_WND_CLASS(NULL) entry, so this is used if an application's windowing class does not have its own CWndClassInfo.

One of these macros can be added to a window implementation class. It provides a function called GetClassInfo, which returns a reference to a static CWndClassInfo structure defined within the function. The reason for three macros is to allow you to influence how the CWndClassInfo gets initialized.

DECLARE_WND_CLASS creates a data member called wc of type CWndClassInfo and fills in all the fields in the structure with default values, and fills in the name with the supplied parameter. This name may be NULL, in which case ATL will generate a name by combining the string "ATL:" and the string-ified address of the m_wc field.

```
#define DECLARE_WND_CLASS(WndClassName) \
static CWndClassInfo& GetWndClassInfo() { \
      static CWndClassInfo wc = { \
            { sizeof(WNDCLASSEX), CS_HREDRAW | CS_VREDRAW|\
                  CS_DBLCLKS, StartWindowProc, 0, 0, NULL,\
                  NULL, NULL, (HBRUSH)(COLOR_WINDOW + 1),\
                  NULL, WndClassName, NULL }, \
            NULL, NULL, IDC_ARROW, TRUE, 0, _T("") \
```

```
      }; \
      return wc; \
}
```

DECLARE_WND_CLASS_EX takes additional style and background color members and uses them when filling in the structure.

```
#define DECLARE_WND_CLASS_EX(
            WndClassName, style, bkgnd) \
      static CWndClassInfo& GetWndClassInfo() { \
            static CWndClassInfo wc = { \
            { sizeof(WNDCLASSEX), style, StartWindowProc, \
            0, 0, NULL, NULL, NULL, (HBRUSH)(bkgnd + 1),\
            NULL, WndClassName, NULL }, \
            NULL, NULL, IDC_ARROW, TRUE, 0, _T("") \
      }; \
      return wc; \
}
```

DECLARE_WND_SUPERCLASS creates a new window class which is a superclass of an existing class.

```
#define DECLARE_WND_SUPERCLASS(WndClassName,
       OrigWndClassName) \
static CWndClassInfo& GetWndClassInfo() { \
      static CWndClassInfo wc = { \
      { sizeof(WNDCLASSEX), 0, StartWindowProc, \
        0, 0, NULL, NULL, NULL, NULL, NULL, WndClassName,
       NULL }, OrigWndClassName,NULL,NULL,TRUE, 0,_T("") \
      }; \
      return wc; \
}
```

The third member is the window procedure to use and it is set to StartWindowProc.

If an application wishes to provide specific settings, and these are not supported through the macros, then there is no problem adding a direct implementation of GetWndClassInfo. We will see this in action in the later ATLSplitter sample, which uses a specific class cursor for the window (IDC_SIZEALL). It defines GetWndClassInfo as follows:

```
      // We wish to use a specific class cursor (IDC_SIZEALL),
      // so we define our own implementation of GetWndClassInfo
      static CWndClassInfo& GetWndClassInfo() {
            static CWndClassInfo wc = {
                  { sizeof(WNDCLASSEX), CS_HREDRAW | CS_VREDRAW|\
                        CS_DBLCLKS, StartWindowProc, 0, 0, NULL,
                        NULL, NULL, (HBRUSH)(COLOR_WINDOW + 1),
                        NULL, TEXT("MYSPLITTER"), NULL },
                        NULL, NULL, IDC_SIZEALL, TRUE, 0, _T("")
                  };
            return wc;
      }
```

Window Procedure using CMessageMap

To facilitate the processing of messages, ATL provides an abstract class called CMessageMap and a range of macros to set up message maps. CMessageMap declares one pure virtual function called ProcessWindowMessage.

```
class ATL_NO_VTABLE CMessageMap{
      public:
            virtual BOOL ProcessWindowMessage(HWND hWnd,
                  UINT uMsg, WPARAM wParam, LPARAM lParam,
                  LRESULT& lResult, DWORD dwMsgMapID) = 0;
};
```

If you think `ProcesssWndMessage` looks similar to the declaration of a window procedure, you are quite right. Within the implementation of this function each windowing class will have to react to incoming window messages, such as `WM_LBUTTONDOWN` and `WM_CHAR`. The range of messages, which a window implementation will wish, to track and their reaction to them varies according to the role of the window and therefore each window which derives from `CMessageMap` must implement its own version of this function.

If none are needed, then the `DECLARE_EMPTY_MSG_MAP` macro may be used.

```
#define DECLARE_EMPTY_MSG_MAP() \
      public: BOOL ProcessWindowMessage(HWND, UINT, \
       WPARAM, LPARAM, LRESULT&, DWORD) { return FALSE; }
```

Most applications will wish to detect certain messages, and we will shortly examine how ATL supports this using message map macros. For now, think of these as setting up a large switch statement, similar to those found in normal window procedures, which test for a particular message type (e.g. `WM_MOUSEMOVE`) and call a C++ member function when the message type is received.

As we have seen from the various `DECLARE_WND_XXX` macros, the window procedure is specified as `StartWindowProc`. So how do we get our newly created window to call its own `ProcessWindowMessage`?

StartWindowProc and Thunking

The window procedure is called `StartWindowProc` and is defined as a static function. C++ static functions cannot directly call instance member functions (how could they decide which of the potentially many instances to call?), but this is exactly what we want our window procedure to do. To get around this, the idea of thunking is used. We have a `HWND` and we need to map it to the object pointer, so we can calls its non-static `ProcessWindowMessage`. To understand how this works we need to take a small tour of the innards of ATL.

The `CComModule` maintains a linked list of `_AtlCreateWndData`, which maintains pairs of object pointers and thread ids.

```
struct _AtlCreateWndData{
      void* m_pThis;
      DWORD m_dwThreadID;
      _AtlCreateWndData* m_pNext;
};
```

`CComModule` has member functions `AddCreateWndData` to add to this linked list and `ExtractCreateWndData` to retrieve the object pointer for the current thread. Only one value per thread may be maintained at any one time.

`CWindowImplBaseT` is part of the inheritance tree when we use `CWindowImpl`. One of its data members is:

```
      CWndProcThunk m_thunk;
```

It is implemented as:

```
class CWndProcThunk{
      union {
            _AtlCreateWndData cd;
            _WndProcThunk thunk;
      };
      void Init(WNDPROC proc, void* pThis);
}
```

The reason a union is used here is to conserve space. At any one time either the `cd` or `thunk` field will be needed – they are never needed together.

To create a window, the `Create` member function of `CWindowImplBaseT` is called, and this results in a call to:

```
AddCreateWndData(&m_thunk.cd, this);
```

Note that the `m_thunk.cd` parameter is memory for the node that becomes part of the linked list, and `this` is the data that is put into it.

Then a call to the Win32 API `CreateWindowEx` is made. As the window procedure for the window class is `StartWindowProc`, it is called with the first window creation message (important: this is guaranteed to occur before the call to `CreateWindowEx` returns).

Within `StartWindowProc` it calls `CComModule::ExtractCreateWndData` to retrieve the object pointer for the current thread (and remove the node from the linked list). The application code running in this thread is in a call to `CreateWindowEx`, so there is no possibility that two windows within one thread might be in the process of being created at the same time. As a window procedure, `StartWindowProc` will receive the `HWND` as a parameter. It stores this value in the object's `m_hwnd` field.

The thunk is initialized, so that when called it will replace the `HWND` parameter on the stack with a pointer to the object instance, and jump to the `CWindowImplBaseT::WindowProc` method, which is also a static member function of `CWindowImplBaseT`. Then using `SetWindowLong(GWL_WNDPROC)` [or `SetWindowLongPtr` in the 32/64-bit version of ATL] the window procedure is changed to call the thunk. Then `StartWindowProc` returns. `StartWindowProc` will not be called again for this instance of the window object.

For all further messages for the window, the thunk will be called, and it replaces the `HWND` on the stack (we have already stored a copy of it once in the object's `m_Hwnd`), and then jumps to executing `WindowProc`.

`CWindowImplBaseT` has a (non-static) method called `WindowProc`, which when called will result in the class' `ProcessWindowMessage` method being called, which as we have seen results in the message map being processed. the `CWindowImplBaseT::WindowProc` looks like a window procedure, and its first parameter is by definition a `HWND`. However, because of the thunking, the value on the

stack was replaced with the pointer to the `CWindowImplBaseT` object. After a suitable cast the object's `ProcessWindowMessage` is called.

```
CWindowImplBaseT< TBase, TWinTraits >* pThis =
            (CWindowImplBaseT< TBase, TWinTraits >*)hWnd;
BOOL bRet = pThis->ProcessWindowMessage(pThis->m_hWnd, uMsg,
            wParam, lParam, lRes, 0);
```

CWindowImpl

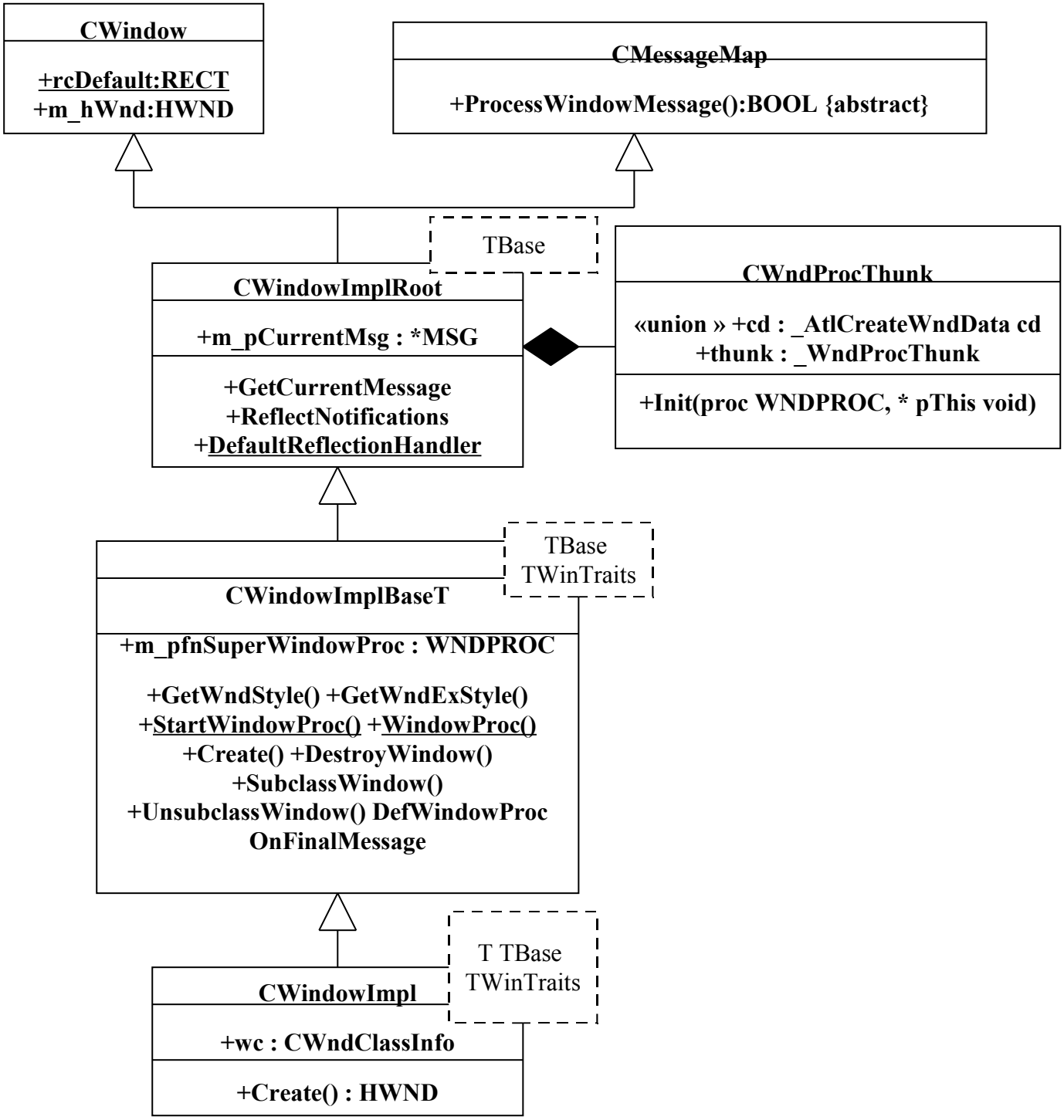
To help facilitate the creation of C++ classes for windowing, ATL provide the `CWindowImpl` template. This derives from `CWindowImplBaseT`, which in turn derives from `CWindowImplRoot`, which finally derives from both `CMessageMap` and `CWindow`. Note that the `ProcessWindowMessage` member function in `CMessageMap` is abstract and therefore your implementation class must supply this function.

The ATL Object Wizard does not provide wizard support to use `CWindowImpl` (but it does for dialog boxes). It is easy enough to use it manually. You must create a C++ class which derives from `CWindowImpl`, write a constructor which calls `Create` and a destructor which calls `DestroyWindow`, and add a message map which implements `ProcessWindowMessage`.

There is a small syntactical issue you should be aware of. As your class is deriving from the template `CWindowImpl`, and it final parameter is itself the templated `CWinTraits`, at the end of your declaration you will have two template terminators '>'. You must leave a space between them – C++ has a quite different understanding of the symbols '>' and ' >>'. If you forget don't worry – the compiler will kindly remind you!

```
class CMyATLWindow : public CWindowImpl< CMyATLWindow, CWindow,
      CWinTraits < WS_CAPTION | WS_POPUPWINDOW | WS_VISIBLE, 0> >{
public:
      CMyATLWindow (){
      RECT r={0, 0, 510, 510};
      Create(0, r, TEXT("My Name"));
}
~ CMyATLWindow (){
      if (m_hWnd)
            DestroyWindow();
}
      BEGIN_MSG_MAP(CMyATLWindow)
      MESSAGE_HANDLER(WM_CHAR, OnChar)
      END_MSG_MAP()
```

```
};
```



Message Maps

Now that we have seen how our object's `ProcessWindowMessage` member function gets called, we next want to examine how it reacts to incoming messages in this function. ATL does not "crack" messages, so the same prototype is used for all handlers. [Note that WTL does add crackers and has an additional macro, `BEGIN_MESSAGE_MAP_EX`.]

`ProcessWindowMessage` will receive a very large number of messages and we wish to detect only a small number of these and pass the remainder onto

DefWindowProc or equivalent for default processing. Within ProcessWindowMessage we need to build a switch statement which for each message we in which we are interested, we call a member function we write (e.g. for WM_CHAR we might call MyOnCharFnc).

```
ProcessWindowMessage(HWND hwnd, UINT uMsg, . . .){
      switch (msg) {
            case WM_CHAR: MyOnCharFnc (); break;
            case WM_PAINT: MyOnPaintFnc (); break;
            default: defWindowProc();
      }
}
```

For organizational reasons, we often wish a C++ class representing a parent window to react to messages that were sent to a child window. This saves us having to write separate C++ classes for each window, and is useful for subclassing and superclassing, which we will cover later.

```
ProcessWindowMessage(HWND hwnd, UINT uMsg, . .,
                                          DWORD dwMsgMapID){
      switch (dwMsgMapID) {
            case 0: // handle messages for this window
                  if (uMsg == WM_CHAR) MyOnCharFnc();
                  if (uMsg == WM_PAINT) MyOnPaintFnc();
                  break;
            case 1: // handle messages for the first child window
                  if (uMsg == WM_LBUTTONDOWN) MyOnButtonDownFnc();
            default: defWindowProc();
      }
}
```

Constructing the Message Map

We do not wish to manually build up such a big switch statement. The macros BEGIN_MSG_MAP and END_MSG_MAP set up an empty message map.

```
#define BEGIN_MSG_MAP(theClass) \
public: \
      BOOL ProcessWindowMessage(HWND hWnd, UINT uMsg, WPARAM
wParam,LPARAM lParam,LRESULT& lResult,DWORD dwMsgMapID = 0){ \
            BOOL bHandled = TRUE; \
            //Following line avoids possible "not used" warnings
            hWnd; uMsg; wParam; lParam; lResult; bHandled;
            switch(dwMsgMapID) { \
            case 0:
#define END_MSG_MAP() \
                  break; \
            default: \
                  ATLTRACE2(atlTraceWindowing, 0,
                  _T("Invalid message map ID (%i)\n"), dwMsgMapID);\
                  ATLASSERT(FALSE); \
                  break; \
            } \
            return FALSE; \
      }
```

Note that the last line of BEGIN_MSG_MAP is "case 0:" which indicates we are switching on the message map id. The macro ALT_MSG_MAP sets up an alternative message map id, after which we could place handlers for other windows.

```
#define ALT_MSG_MAP(msgMapID) \
            break; \
            case msgMapID:
```

Message Handlers

Next we need to populate our message maps with handlers, which can be done using a number of macros. The simplest of these is MESSAGE_HANDLER, which takes two parameters, a message and a function pointer, and if the message parameter is equal to the messageID parameter to ProcessWindowMessage, then the function pointer is called.

```
#define MESSAGE_HANDLER(msg, func) \
      if(uMsg == msg) { \
            bHandled = TRUE; \
            lResult = func(uMsg, wParam, lParam, bHandled); \
            if(bHandled) \
                  return TRUE; \
      }
```

If we wish to call a handler if any one of a contiguous range of messages was received, we could use MESSAGE_RANGE_HANDLER:

```
#define MESSAGE_RANGE_HANDLER(msgFirst, msgLast, func) \
      if(uMsg >= msgFirst && uMsg <= msgLast) { \
            bHandled = TRUE; \
            lResult = func(uMsg, wParam, lParam, bHandled); \
            if(bHandled) \
                  return TRUE; \
      }
```

For each handler specified in the message map, you must create a handler function, with the following prototype:

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

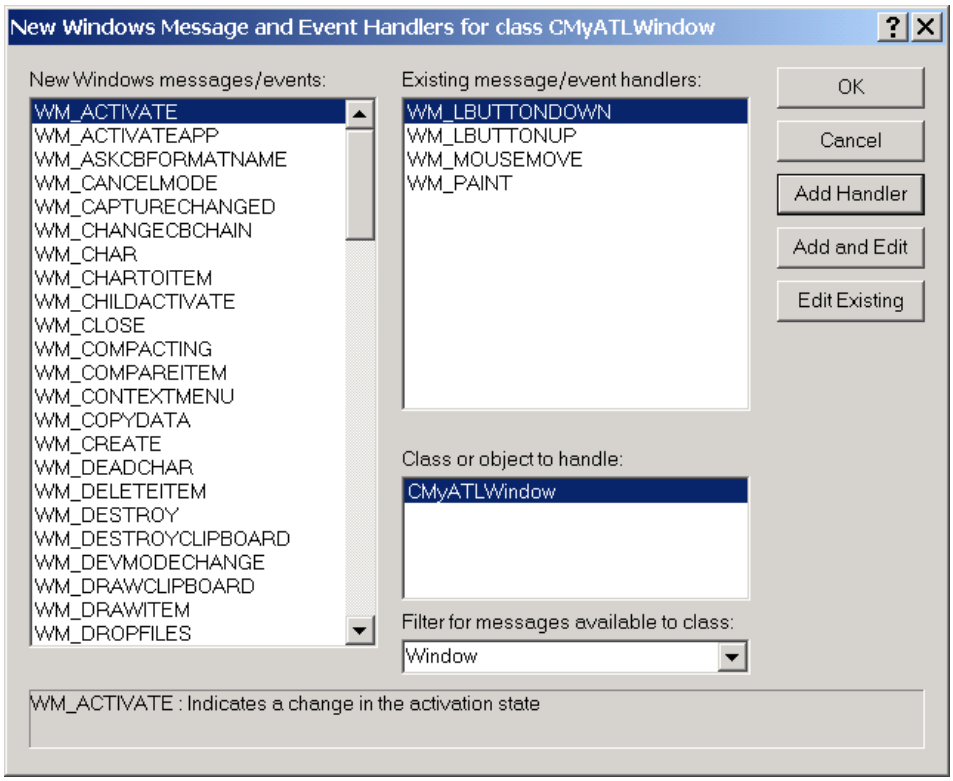
The return value from this function becomes the return value from the window procedure. The final Boolean parameter, bHandled, can be set to FALSE to state that the message has not been handled, and that the search for message handlers in the message map should continue. Before your handler is called, the initial value of bHandled is set to TRUE, which is what most handlers want, so if you are happy with this there is no need to set it within your handler. Note that normally ATL does not unpack the parameters to the window procedure. The wParam and lParam values contain important information, which is particular to each message type, and you will have to examine the Win32 Platform SDK documentation to discover what they represent. ATL does not un-wrap them and present them to your function nicely converted to specific data types.

The *New Windows Message and Event Handlers* Dialog

Though Developer Studio does not offer any Wizard support to create `CWindowImpl`-derived classes, it does provide one dialog box to help you populate their message maps.

You will first have to manually create a class deriving it from `CWindowImpl`. Then in ClassView (not the **MFC** ClassWizard!) select the class and right click to bring up the context menu. Select the *Add Windows Message Handler* menu item, and the *New Windows Message and Event Handlers* dialog will be displayed. This dialog enables you to add handlers for different window messages and events.

The list of messages depends on the type of window that is selected. For example, if a pushbutton is selected, handlers may be added to detect single and double button clicks. The most common notifications are supported. For others, you will have to manually add them to the message maps in code.



Rendering with ATL

ATL itself does not provide any templates to help with GDI programming. This is one of the added benefits of using WTL. If developing windowing code using ATL only, you will have to use the Win32 GDI functions such as `LineTo` etc.

Sample: ATLWithCWindowImpl

The ATLWithCWindowImpl project examines how to implement a C++ class that is derived from `CWindowImpl` and has a message map containing a number of message handlers.

The project enables to user to draw lines in a window, and including rubber banding as the line is being constructed.

It was started by running the ATL AppWizard and creating an EXE. Then a new file was created and the code for our class `CMyATLWindow` added.

```
class CMyATLWindow  : public CWindowImpl< CMyATLWindow, CWindow,
CWinTraits < WS_CAPTION | WS_POPUPWINDOW | WS_VISIBLE, 0> >{
private:
      CSimpleArray<POINT> m_pointList;
      bool m_bAddingLine;
      POINT m_StartPoint;
      POINT m_CurrentPoint;
public:
      CMyATLWindow (){
      RECT r={0, 0, 510, 510};
      Create(0, r, TEXT("Draw Lines"));
      bAddingLine = false;
}
~CMyATLWindow (){
      if (m_hWnd)
            DestroyWindow();
}
void OnFinalMessage(HWND hWnd){
      ::PostQuitMessage(0);
}
```

The `m_pointList` data member maintains an array of points of the lines in the picture. This will be used when the picture needs to be redrawn. The `m_bAddLine` Boolean data member is used while detecting mouse moves, and states if we are in the middle of adding a line (and rubber-banding effect is needed). The `m_StartPoint` is the beginning point in the line, and the `m_CurrentPoint` is the last point entered.

The constructor and destructor create and destroy the window, and the `OnFinalMessage` function causes the entire single-threaded application to shut down when this window is closed (for more substantial applications, you certainly will not want to add this to every window!).

The message map detects four messages and calls suitable member functions for each.

```
      BEGIN_MSG_MAP(CMyWindows)
      MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
      MESSAGE_HANDLER(WM_MOUSEMOVE, OnMouseMove)
      MESSAGE_HANDLER(WM_LBUTTONUP, OnLButtonUp)
      MESSAGE_HANDLER(WM_PAINT, OnPaint)
      END_MSG_MAP()
```

The functions `OnButtonDown`, `OnMouseMove` and `OnButtonUp` keep track of the line as it is being constructed and perform rubber banding.

```
      LRESULT OnLButtonDown(UINT uMsg, WPARAM wParam,
                              LPARAM lParam, BOOL& bHandled){
       m_StartPoint.x = LOWORD(lParam);
       m_StartPoint.y = HIWORD(lParam);
```

```
 m_CurrentPoint.x = LOWORD(lParam)+20;
 m_CurrentPoint.y = HIWORD(lParam)+20;
 HDC hdc = ::GetDC(m_hWnd);
 SetROP2(hdc, R2_NOT);
 MoveToEx(hdc, m_StartPoint.x, m_StartPoint.y, NULL);
 LineTo(hdc, m_CurrentPoint.x, m_CurrentPoint.y);
 m_bAddingLine = true;
 ::ReleaseDc(m_hWnd, hdc);
 return 0;
}
LRESULT OnMouseMove(UINT uMsg, WPARAM wParam,
                        LPARAM lParam, BOOL& bHandled){
      if (m_bAddingLine){
            HDC hdc = ::GetDC(m_hWnd);
            SetROP2(hdc, R2_NOT);
            // First erase last line drawn
            MoveToEx(hdc, m_StartPoint.x, m_StartPoint.y,NULL);
            LineTo(hdc, m_CurrentPoint.x, m_CurrentPoint.y);
            // Update curent point
            m_CurrentPoint.x = LOWORD(lParam);
            m_CurrentPoint.y = HIWORD(lParam);
            // Draw new line
            MoveToEx(hdc, m_StartPoint.x, m_StartPoint.y,NULL);
            LineTo(hdc, m_CurrentPoint.x, m_CurrentPoint.y);
       ::ReleaseDc(m_hWnd, hdc);
      }
 return 0;
}
LRESULT OnLButtonUp(UINT uMsg, WPARAM wParam,
                        LPARAM lParam, BOOL& bHandled){
 HDC hdc = ::GetDC(m_hWnd);
 SetROP2(hdc, R2_NOT);
 // First erase last line drawn
 MoveToEx(hdc, m_StartPoint.x, m_StartPoint.y, NULL);
 LineTo(hdc, m_CurrentPoint.x, m_CurrentPoint.y);
 // Reset drawing mode to black
 SetROP2(hdc, R2_BLACK);
 m_bAddingLine = false;
 // Add points to array
 m_CurrentPoint.x = LOWORD(lParam);
 m_CurrentPoint.y = HIWORD(lParam);
 m_pointList.Add(m_StartPoint);
 m_pointList.Add(m_CurrentPoint);
 // Draw newly added line
 MoveToEx(hdc, m_StartPoint.x, m_StartPoint.y, NULL);
 LineTo(hdc, m_CurrentPoint.x, m_CurrentPoint.y);
 ::ReleaseDc(m_hWnd, hdc);
 return 0;
}
```

The `OnPaint` member function is called whenever the picture in the window needs to be refreshed. It simply runs through the list of points, and for each pair moves to the first and calls `LineTo` to the second.

```
      LRESULT OnPaint(UINT uMsg, WPARAM wParam,
                      LPARAM lParam, BOOL& bHandled)
      {
       PAINTSTRUCT ps;
       HDC hdc = (wParam != NULL) ?
                   reinterpret_cast<HDC>(wParam) :
                   ::BeginPaint(m_hWnd, &ps);
       int numLines = m_pointList.GetSize() / 2;
       int counter=0;
       while (counter< numLines)
       {
             POINT  ptbegin = m_pointList[counter*2];
             POINT  ptend = m_pointList[counter*2+1];
             MoveToEx(hdc, ptbegin.x, ptbegin.y, NULL);
             LineTo(hdc, ptend.x, ptend.y);
             counter++;
       }
       if (wParam == NULL)
             ::EndPaint(m_hWnd, &ps);
             return 0;
       }
};
```

To instantiate your new class add this line to your `WinMain`:

```
      CMyATLWindow wnd;
```

Command and Notification Handlers

In general ATL does not "crack" messages for us (WTL does!), but ATL does offer some special help for commands (`WM_COMMAND`) and notifications (`WM_NOTIFY`). Commands are sent to parent windows when, for example, menu items in the window menu are selected or child pushbuttons are pressed. Notifications are used by the common controls to inform the application in details about what is happening with these controls.

You write your command handlers as follows:

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

Note that here ATL does break out the notification code and the command id.

ATL provides four macros to support commands. Each macro tests that the `uMsg` field is set to `WM_COMMAND` and another condition, and if so it calls the final parameter, which is your command handler. The "other condition" is what distinguishes the four macros from each other.

```
#define COMMAND_HANDLER(id, code, func) \
      if(uMsg == WM_COMMAND && id == LOWORD(wParam)
            && code == HIWORD(wParam)) { \
            bHandled = TRUE; \
            lResult = func(HIWORD(wParam), LOWORD(wParam),
```

```
                                        (HWND)lParam, bHandled); \
            if(bHandled) \
                  return TRUE; \
      }
```

COMMAND_HANDLER maps a control id and notification to a command handler.

```
#define COMMAND_ID_HANDLER(id, func) \
      if(uMsg == WM_COMMAND && id == LOWORD(wParam)) { \
            bHandled = TRUE; \
            lResult = func(HIWORD(wParam), LOWORD(wParam), \
                                        (HWND)lParam, bHandled); \
            if(bHandled) \
                  return TRUE; \
      }
```

COMMAND_ID_HANDLER maps the control id only.

```
#define COMMAND_CODE_HANDLER(code, func) \
      if(uMsg == WM_COMMAND && code == HIWORD(wParam)) { \
            bHandled = TRUE; \
            lResult = func(HIWORD(wParam), LOWORD(wParam), \
                                        (HWND)lParam, bHandled); \
            if(bHandled) \
                  return TRUE; \
      }
```

COMMAND_CODE_HANDLER maps the notification code only.

```
#define COMMAND_RANGE_HANDLER(idFirst, idLast, func) \
      if(uMsg == WM_COMMAND && LOWORD(wParam) >= idFirst \
                                        && LOWORD(wParam) <= idLast) { \
            bHandled = TRUE; \
            lResult = func(HIWORD(wParam), LOWORD(wParam), \
                                        (HWND)lParam, bHandled); \
            if(bHandled) \
                  return TRUE; \
}
```

COMMAND_RANGE_HANDLER maps a range of control ids.

Notifications are normally used by the common controls to tell their parent of significant events. The prototype for notification handlers is as follows:

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

Note that the third parameter is of type pointer to NMHDR. Depending on the control involved, this can be cast to a larger control-specific header. For example, the DateTimePicker notification header would be:

```
typedef struct tagNMDATETIMECHANGE {
NMHDR nmhdr;
DWORD dwFlags;
SYSTEMTIME st;
} NMDATETIMECHANGE;
```

As the first field it NMHDR if can safely be accessed through a pointer to NMHDR and if such a pointer were cast to a pointer to NMDATETIMECHANGE, the other fields could be accessed. Hence very rich information can be supplied to the application. The four notification macros are similar to the command macros, in that the first checks for a control id and a notification code, a control id only, a notification only and a range of control ids.

```
#define NOTIFY_HANDLER(id, cd, func) \
      if(uMsg == WM_NOTIFY && id == ((LPNMHDR)lParam)->idFrom && cd ==
```

```
((LPNMHDR)lParam)->code) { \
            bHandled = TRUE; \
      lResult = func((int)wParam, (LPNMHDR)lParam, bHandled); \
            if(bHandled) return TRUE; \
      }
#define NOTIFY_ID_HANDLER(id, func) \
      if(uMsg == WM_NOTIFY && id == ((LPNMHDR)lParam)->idFrom) { \
            bHandled = TRUE; \
            lResult = func((int)wParam, (LPNMHDR)lParam, bHandled); \
            if(bHandled) \
                  return TRUE; \
      }
#define NOTIFY_CODE_HANDLER(cd, func) \
      if(uMsg == WM_NOTIFY && cd == ((LPNMHDR)lParam)->code) { \
            bHandled = TRUE; \
            lResult = func((int)wParam, (LPNMHDR)lParam, bHandled); \
            if(bHandled) \
                  return TRUE; \
      }
#define NOTIFY_RANGE_HANDLER(idFirst, idLast, func) \
      if(uMsg == WM_NOTIFY && ((LPNMHDR)lParam)->idFrom >= idFirst &&
((LPNMHDR)lParam)->idFrom <= idLast) { \
            bHandled = TRUE; \
            lResult = func((int)wParam, (LPNMHDR)lParam, bHandled); \
            if(bHandled) \
                  return TRUE; \
      }
```

Sample : ATLCommandsAndNotifications

The ATLCommandsAndNotifications project demonstrates the use of command and notification handlers. It uses a menu to generate the commands and a tree view to generate the notifications.

```
class CMyCommandAndNotifWindow  : public CWindowImpl<
CMyCommandAndNotifWindow, CWindow,
CWinTraits < WS_CAPTION | WS_POPUPWINDOW | WS_VISIBLE, 0> >{
public:
      CWindow m_ctlTreeview;
      CMyCommandAndNotifWindow (){
      RECT r={0, 0, 510, 510};
      HMENU hMenu = LoadMenu(_Module.GetResourceInstance(),
                                    MAKEINTRESOURCE(IDR_MY_MENU));
      Create(0, r,
         TEXT("ATL Command and Notification Demo"),0,0,(UINT)hMenu);

      RECT rc={20,20,150,300};
      m_ctlTreeview.Create(WC_TREEVIEW, m_hWnd, &rc,
            TEXT("CWindow Demo"),
            WS_CHILD | WS_VISIBLE | TVS_HASLINES | TVS_LINESATROOT);
      RECT rc1={200,20,500,50};
      TV_INSERTSTRUCT TreeCtrlItem;
      TreeCtrlItem.hInsertAfter = TVI_LAST;
      TreeCtrlItem.item.mask = TVIF_TEXT | TVIF_PARAM;
      TreeCtrlItem.item.pszText = "First Entry";
      TreeCtrlItem.item.lParam = 0;
      TreeCtrlItem.hParent = TVI_ROOT;
      m_ctlTreeview.SendMessage(TVM_INSERTITEM, 0,
                                    (LPARAM)&TreeCtrlItem);
```

```
      TreeCtrlItem.item.pszText = "Second Entry";
      TreeCtrlItem.hParent = TVI_ROOT;
      m_ctlTreeview.SendMessage(TVM_INSERTITEM, 0,
                                (LPARAM)&TreeCtrlItem);
      }
      BEGIN_MSG_MAP(CMyCommandAndNotifWindow)
            COMMAND_ID_HANDLER(ID_BEEP, OnBeep)
            COMMAND_ID_HANDLER(ID_RELOCATE, OnRelocate)
            NOTIFY_CODE_HANDLER(TVN_SELCHANGED, OnSelChange)
      END_MSG_MAP()
```

Simple implementations of the handlers are provided to verify that they are executed. The `OnBeep` handler calls the Win32 `Beep` API, the `OnRelocate` handler calls `CWindow::MoveWindow` to reposition the window and the `OnSelChange` handler outputs some debug text to show that the notification has been received.

```
      LRESULT OnBeep(WORD wNotifyCode, WORD wID, HWND hWndCtl,
             BOOL& bHandled){
            OutputDebugString("OnBeep called\n");
            Beep(100,100);
            return 0;
      }
      LRESULT OnRelocate(WORD wNotifyCode, WORD wID, HWND hWndCtl,
             BOOL& bHandled){
            OutputDebugString("OnRelocate called\n");
            MoveWindow( 200, 200, 600, 600, TRUE );
            return 0;
      }
```

Message Map Chaining

One of the great features of the MFC architecture is the way command and message routing occurs. The class representing the window in which a message is received need not be the one that handles that message. It is possible to forward to message to another class, which might be better suited. With MFC much of this is done automatically.

ATL offers similar functionality and it is known as message map chaining. A number of macros are provided and these can be inserted manually as appropriate. ATL provides five macros and one C++ class (`CDynamicChain`) that are used to allow implementers to flexibly route messages to different classes.

`CHAIN_MSG_MAP` and `CHAIN_MSG_MAP_ALT` redirect messages from a derived class to be processed in the message map of a parent class. `CHAIN_MSG_MAP` redirects the message to the parent's default message map, and `CHAIN_MSG_MAP_ALT (<number>)` redirects messages to the alternate message map identified by the number parameter. Be clear that `CHAIN_MSG_MAP` should appear in the derived class' message map, and the parameter to the macro, `theChainClass`, must be a base class.

```
#define CHAIN_MSG_MAP(theChainClass) { \
            if(theChainClass::ProcessWindowMessage(hWnd, uMsg,\
                  wParam, lParam, lResult)) \
                return TRUE; \
      }
```

```
#define CHAIN_MSG_MAP_ALT(theChainClass, msgMapID) { \
            if(theChainClass::ProcessWindowMessage(hWnd, uMsg,\
             wParam, lParam, lResult, msgMapID)) \
                  return TRUE; \
      }
```

CHAIN_MSG_MAP_MEMBER and CHAIN_MSG_MAP_ALT_MEMBER are used when a class wishes to redirect incoming messages to another class that is a data member of it. (This other class must itself implement ProcessWindowMessage).

CHAIN_MSG_MAP_MEMBER redirects to the default message map whereas CHAIN_MSG_MAP_ALT_MEMBER(<number>) redirects to the specified alternate.

```
#define CHAIN_MSG_MAP_MEMBER(theChainMember) { \
            if(theChainMember.ProcessWindowMessage(hWnd, uMsg,\
                   wParam, lParam, lResult)) \
                  return TRUE; \
      }
#define CHAIN_MSG_MAP_ALT_MEMBER(theChainMember,msgMapID){\
            if(theChainMember.ProcessWindowMessage(hWnd, uMsg,\
                  wParam, lParam, lResult, msgMapID)) \
                  return TRUE; \
      }
```

With these four chaining macros there must be either an inheritance or a containment relationship between the classes. What happens if we wish to direct messages between classes that are not related, or if we wish to change the destination class on the fly? This is supported through the use of dynamic chaining, which uses the CHAIN_MSG_MAP macro and the CDynamicChain class.

```
#define CHAIN_MSG_MAP_DYNAMIC(dynaChainID) { \
            if(CDynamicChain::CallChain(dynaChainID, hWnd, uMsg,\
                  wParam, lParam, lResult)) \
                  return TRUE; \
      }
```

CDynamicChain is a simple class that manages an array of mapping from a chain id (an integer identifier) to a pointer to a CMessageMap and a map id (which will be 0 if the default map is to be used, or a greater number if an alternate map should be used). Entries can be added to the array using CDynamicChain::SetChainEntry and removed with CDynamicChain::RemoveChainEntry. The CDynamicChain::CallChain finds the CMessageMap object associated with a chain id, and calls its ProcessWindowMessage function.

The class you write which is to receive the redirected message must be derived from CMessageMap and have its own message map. The class you write which receives the message must be derived from CDynamicChain and must call CDynamicChain::SetChainEntry passing in the recipient class and a chain id.

Then when CHAIN_MSG_MAP_DYNAMIC is used in the message map, the chain id is used to lookup the CDynamicChain array and the ProcessWindowMessage function called in the appropriate recipient.

Sample: WindowMessageChaining

The WindowMessageChaining project explores the use of ATL message map chaining.

Base class chaining is performed by creating two classes, CMyBaseWindow and derived from it, CMyDerivedWindow. Both implement messages maps – usually these would contain different messages but to clearly indicate which piece of code handles the message, we have both of them handling WM_LBUTTONDOWN, and the handler for this prints out a string to the output window (when you are running it in the debugger).

The message map for the derived class contains two entries, one specifying a base chaining to the base class, and the other specifies a normal message handler. When the derived class receives a WM_LBUTTONDOWN message, whichever of these entries comes first will get to process it, so if CHAIN_MSG_MAP(CMyBaseWindow) is first the output window shows "*CMyBaseWindow::OnLButtonDown called"* whereas if the MESSAGE_HANDLER (WM_LBUTTONDOWN, OnLButtonDown) entry came first the output window would show *"CMyDerivedWindow::OnLButtonDown called"*.

```
class CMyBaseWindow  : public CWindowImpl< CMyBaseWindow ,
 CWindow, CWinTraits <WS_CAPTION|WS_POPUPWINDOW|WS_VISIBLE, 0>>{
public:
      BEGIN_MSG_MAP(CMyBaseWindow)
      MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
      END_MSG_MAP()
      LRESULT OnLButtonDown(UINT uMsg, WPARAM wParam,
LPARAM lParam, BOOL& bHandled){
            OutputDebugString(TEXT(
                  "CMyBaseWindow::OnLButtonDown called\n"));
            return 0;
      }
};
class CMyDerivedWindow  : public CmyBaseWindow {
public:
      BEGIN_MSG_MAP(CMyDerivedWindow)
      CHAIN_MSG_MAP(CMyBaseWindow)
      MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
      END_MSG_MAP()
      LRESULT OnLButtonDown(UINT uMsg, WPARAM wParam,
                              LPARAM lParam, BOOL& bHandled){
     OutputDebugString(TEXT(
                  "CMyDerivedWindow::OnLButtonDown called\n"));
      return 0;
      }
};
```

Chaining to a contained class member is performed using the CMyContainedClass and CMyContainerWindow classes. CMyContainedClass derives from CMessageMap, and its map detects WM_LBUTTONDOWN and the handler prints out a string for this. CMyContainerWindow is a normal window and as a data member has an instance of CMyContainedClass. It its message map it calls CHAIN_MSG_MAP_MEMBER. When CMyContainerWindow is instantiated and the mouse button clicks in the window, the string *"CMyContainedClass::OnLButtonDown called"* is output.

```
class CMyContainedClass : public CMessageMap {
public:
```

```
      BEGIN_MSG_MAP(CMyContainedClass)
      MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
      END_MSG_MAP()
      LRESULT OnLButtonDown(UINT uMsg, WPARAM wParam, LPARAM lParam,
BOOL& bHandled){
            OutputDebugString(TEXT(
                  "CMyContainedClass::OnLButtonDown called\n"));
            return 0;
      }
};
class CMyContainerWindow : public CWindowImpl< CMyContainerWindow ,
CWindow,
      CWinTraits < WS_CAPTION | WS_POPUPWINDOW | WS_VISIBLE, 0> >
{
public:
      CMyContainedClass contained;
      BEGIN_MSG_MAP(CMyContainerWindow)
      CHAIN_MSG_MAP_MEMBER(contained)
      END_MSG_MAP()

      LRESULT OnLButtonDown(UINT uMsg, WPARAM wParam, LPARAM lParam,
BOOL& bHandled){
            OutputDebugString(TEXT(
                  "CMyContainerWindow::OnLButtonDown called\n"));
            return 0;
      }
};
```

Dynamic chaining is shown using the classes `CMyDestinationClass` (which is derived from `CMessageMap`) and `CMySourceWindow` (which is derived from `CDynamicChain`). These two classes are not related by inheritance or containment, as with the other chaining techniques. To set up the connection between instances of the classes, the following code is added (e.g. in `WinMain`)

```
            CMySourceWindow wnd3;
            CMyDestinationClass DynamicDestination;
            wnd3.SetChainEntry(1, &DynamicDestination, 0);
```

Note that the first parameter to `SetChainEntry` is the chain id – and if multiple calls are may to `SetChainEntry` with the same id then the latter call replaces the previous setting. The third parameter is the message map id, 0 for the default or a higher number for alternates. If you use a higher number and no alternate of that number exists, an `ASSERT` fails within the ATL code.

The implementation of the `CMyDestinationClass` consists of a message map that detects `WM_LBUTTONDOWN` and the handler that outputs a string when it is detected.

```
class CMyDestinationClass : public CMessageMap {
public:
      BEGIN_MSG_MAP(CMyDestinationClass)
      MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
      END_MSG_MAP()

      LRESULT OnLButtonDown(UINT uMsg, WPARAM wParam,
            LPARAM lParam, BOOL& bHandled){
```

```
         OutputDebugString(TEXT(
            "CMyDestinationClass::OnLButtonDown called\n"));
         return 0;
      }
};
```

The implementation of `CMySourceWindow` contains a message map which dynamic chains chainID 1. Its `OnLButtonDown` is never called.

```
class CMySourceWindow : public CWindowImpl< CMySourceWindow,
 CWindow, CWinTraits <WS_CAPTION|WS_POPUPWINDOW|WS_VISIBLE,0>>,
 public CDynamicChain {
public:
      BEGIN_MSG_MAP(CMySourceWindow)
      CHAIN_MSG_MAP_DYNAMIC (1)
      END_MSG_MAP()
      LRESULT OnLButtonDown(UINT uMsg, WPARAM wParam,
      LPARAM lParam, BOOL& bHandled){
            OutputDebugString(TEXT(
                  "CMySourceWindow::OnLButtonDown called\n"));
            return 0;
      }
};
```

When this code is run and the mouse clicked in the window the output is *CMyDestinationClass::OnLButtonDown called,* which shows that the chaining has worked.

Subclassing and Superclassing

`CWindowImpl` supports superclassing and instance subclassing.

To superclass a window you use the `DECLARE_WND_SUPERCLASS` instead of `DECLARE_WND_CLASS` to define the class information. `DECLARE_WND_SUPERCLASS` takes two class names as parameters – the name of the new class, and the name of an existing class. During class registration (which is initiated in `CWindowImpl::Create`) the class information for the existing class is accessed, and its window procedure recorded in the data member `CWindowImplBaseT::m_pfnSuperWindowProc`.

Later, when messages are being processed, if a particular message has no entry in the message map it is forwarded to the `CWindowImplBaseT::m_pfnSuperWindowProc` for processing by the base class.

Subclassing of an existing window is performed in `CWindowImplBaseT::SubclassWindow(HWND hWnd)`, where the `HWND` parameter represents the window to be subclassed. A `CWindowImpl` expects to manage a single `HWND`, so either `SubclassWindow` (which uses an existing window) or `Create` (which creates a new one without subclassing) should be called, but it is an error to call both. The implementation of `SubclassWindow` calls `SetWindowLong(hWnd, GWL_WNDPROC)` to replace the window procedure in use for the existing window. The old window procedure is returned, and this is stored in

`CWindowImplBaseT::m_pfnSuperWindowProc` , and is called if messages are not processed by our message map.

ATL does not support global subclassing (a search of the ATL source tree for `SetClassLongPtr` or `SetClassLong` finds no matches).

Sample: SubClassAndSuperClass

The SubclassAndSuperclass project experiments with superclassing and subclassing in ATL. The `CMySuperWindow` class superclasses the BUTTON control. It detects `WM_LBUTTONDOWN` messages and prints out a string when they arrive.

```
class CMySuperWindow  : public CWindowImpl< CMySuperWindow ,
CWindow,CWinTraits<WS_CAPTION|WS_POPUPWINDOW | WS_VISIBLE, 0> >{
public:
      DECLARE_WND_SUPERCLASS ("MySuperWindow", "BUTTON")
      CMySuperWindow (){
            RECT r={0, 0, 510, 510};
            Create(0, r, TEXT("MY SUPERCLASSED WINDOW"));
      }
      BEGIN_MSG_MAP(CMySuperWindow)
      MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
      END_MSG_MAP()
      LRESULT OnLButtonDown(UINT uMsg, WPARAM wParam,
                              LPARAM lParam, BOOL& bHandled){
            OutputDebugString(TEXT(
                  "CMySuperWindow::OnLButtonDown called\n"));
            return 0;
      }
};
```

To show subclassing we first have to have an existing window. For this we write `CMyChildWindow`. Note that it detects `WM_LBUTTONDOWN` and `WM_LBUTTONUP` messages. Also note that its constructor calls `Create`, to really create a new window, whose parent is a `HWND` we pass in.

```
class CMyChildWindow  : public CWindowImpl< CMyChildWindow , CWindow,
CWinTraits < WS_CHILD | WS_VISIBLE, 0> >{
public:
      CMyChildWindow (HWND hParent){
            RECT r={20, 20, 150, 150};
            Create(hParent, r, TEXT("MY CHILD WINDOW"));
      }
      BEGIN_MSG_MAP(CMyChildWindow)
      MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
      MESSAGE_HANDLER(WM_LBUTTONUP, OnLButtonUp)
      END_MSG_MAP()
      LRESULT OnLButtonDown(UINT uMsg, WPARAM wParam,
                  LPARAM lParam, BOOL& bHandled){
            OutputDebugString(TEXT(
                  "CMyChildWindow::OnLButtonDown called\n"));
            return 0;
      }
```

```
      LRESULT OnLButtonUp(UINT uMsg, WPARAM wParam,
                  LPARAM lParam, BOOL& bHandled){
            OutputDebugString(TEXT(
                  "CMyChildWindow::OnLButtonUp called\n"));
            return 0;
      }
};
```

To subclass `CMyWindow` we write another class called `CMySubClassedWindow`. In its constructor it takes in a `HWND` and calls `SubclassWindow` to subclass the window identified by the `HWND` parameter. The message map for `CMySubClassedWindow` detects the `WM_LBUTTONDOWN` message.

```
class CMySubclassedWindow  : public CWindowImpl< CMySubclassedWindow ,
CWindow,    CWinTraits < WS_CHILD | WS_VISIBLE, 0> >{
public:
      CMySubclassedWindow (HWND hSubclassThisWindow){
            RECT r={20, 20, 150, 150};
            SubclassWindow(hSubclassThisWindow);
      }
      BEGIN_MSG_MAP(CMySubclassedWindow)
      MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
      END_MSG_MAP()
      LRESULT OnLButtonDown(UINT uMsg, WPARAM wParam,
                              LPARAM lParam, BOOL& bHandled){
            OutputDebugString(TEXT(
                  "CMySubclassedWindow::OnLButtonDown called\n"));
            return 0;
      }
};
```

To use both these classes we need to add the following (e.g. in `WinMain`):

```
            CMyChildWindow wnd1(hWnd);
            CMySubclassedWindow wnd2(wnd1.m_hWnd);
```

We first instantiate `CMyChildWindow` and then we subclass it in `CMySubclassedWindow`. Now when we run this program and press the mouse button down and up in the window, the output is:

```
CMySubclassedWindow::OnLButtonDown called
CMyChildWindow::OnLButtonUp called
```

Which is what we expect.

CContainedWindow

When you wish to process messages concerning a child window in the message map of a parent window, you can use `CContainedWindow`. You may either create the window within `CContainedWindow` or subclass an existing window. When initializing the contained window you must specify an object that implements `CMessageMap` and which will act as the parent window. You must also specify an alternative message map id, which will be used when passing child messages to the parent.

When using `CContainedWindow` you do not derive a class from it, rather you create a data member of type `CContainedWindow` in another class (usually the parent class), and then you initialize the data member and call Create (which super-classes the window) or SubclassWindow.

To initialize a `CContainedWindow`, you call `Create`. `CContainedWindow` has a number of constructors and a number of `Create` functions with different signatures. It is important that you use a combination of one of each that supplies a pointer to the parent object which implements `CMessageMap`, the class name of the contained window and the alternate message map id.

Sample: ContainedWindowSample

The ContainedWindowSample shows the use of `CContainedWindow`. It creates two instances of `CContainedWindow`, one using superclassing and the other using subclassing. Both redirect their messages to their parent's message map, but to different map ids.

```
class CMyWindow  : public CWindowImpl< CMyWindow , CWindow,
      CWinTraits < WS_CAPTION | WS_POPUPWINDOW | WS_VISIBLE,
WS_EX_WINDOWEDGE> > {
public:
      HWND m_hToSubclassAsContainedWindow;
      CMyWindow (HWND hToSubclassAsContainedWindow)
        :Subclassed(this, 2){
            RECT r={0, 0, 510, 510};
            m_hToSubclassAsContainedWindow =
                        hToSubclassAsContainedWindow;
            Create(0, r, TEXT("MY WINDOW"));
      }
      CContainedWindow SuperClassed;
      CContainedWindow Subclassed;
      BEGIN_MSG_MAP(CMyWindow)
      MESSAGE_HANDLER(WM_CREATE, OnCreate)
      ALT_MSG_MAP(1)
      MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDownSuperClassed)
      ALT_MSG_MAP(2)
      MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDownSubclassed)
      END_MSG_MAP()
      LRESULT OnLButtonDownSuperClassed(UINT uMsg, WPARAM wParam,
                  LPARAM lParam, BOOL& bHandled)      {
            OutputDebugString(TEXT(
              "CMyWindow::OnLButtonDown called for superclassed\n"));
            return 0;
      }
      LRESULT OnLButtonDownSubclassed(UINT uMsg, WPARAM wParam,
                  LPARAM lParam, BOOL& bHandled)      {
            OutputDebugString(TEXT(
               "CMyWindow::OnLButtonDown called for subclassed\n"));
            return 0;
      }
      LRESULT OnCreate(UINT uMsg, WPARAM wParam,
                        LPARAM lParam, BOOL& bHandled){
```

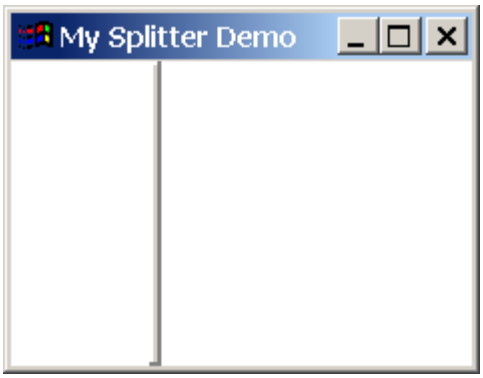
```
            RECT r={50, 50, 210, 210};
            OutputDebugString("OnCreate\n");
            SuperClassed.Create("edit", this, 1, m_hWnd, &r);
         Subclassed.SubclassWindow(
                              m_hToSubclassAsContainedWindow);
      return 0;
      }
};
```

Higher-level UI

It is possible to build more sophisticated functionality above the ATL windowing foundation. Potential areas of interest could be splitter windows, command bars, hyperlink functionality or advanced controls such as bitmap buttons.

ATL does provide you with all the facilities to build such UI constructs. We will now briefly look at one sample, to show it is possible. But the question must be asked - why bother with this extra effort? WTL provides this type of functionality in a well-thought out library of templates, and hence instead of coding such UI yourself, it would be much better to use WTL directly.

Sample: ATLSplitter



The ATLSplitter sample shows how a splitter window could be implemented. It consists of two windows, a workspace window, and as a child, a splitter, which is used to divide the workspace.

The project contains a splitter class that has methods to react to button down and up events and mouse menus, similar to what we saw in the ATLWithCWindowImpl sample.

```
      class CMySplitter : public CWindowImpl< CMySplitter, CWindow,
                  CWinTraits < WS_CHILD | WS_VISIBLE> >
{
public:
      enum Orientation {
            ORIENTATION_VERT,
            ORIENTATION_HORIZ
      };
private:
      RECT  m_rcLocation;     // dimensions of splitter itself
      RECT  m_rcBoundingBox;// dimensions of workspace being split
```

```
      int   m_CenterPoint;    // offset from 0 in parent's
                           // co-ordinates for center of splitter
      Orientation m_Orientation;// whether splitter is vert or horiz
      HWND  m_hParentWnd;     // Handle to parent window
      bool  m_bBeingMoved;    // Are we in the middle of a move
                              // WM_MOUSEMOVE handling is only done
                           // when true
      int   m_parentCoord;    // Location within parent of splitter

public:

      BEGIN_MSG_MAP(CMySplitter)
      MESSAGE_HANDLER(WM_PAINT, OnPaint)
      MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
      MESSAGE_HANDLER(WM_LBUTTONUP, OnLButtonUp)
      MESSAGE_HANDLER(WM_MOUSEMOVE, OnMouseMove)
      END_MSG_MAP()
      . . .
};
```

The workspace class contains one instance of the splitter class and must initialize it to the correct dimensions.

```
      CMySplitter m_VerticalSplitter;
 . . .
      m_VerticalSplitter.Init(m_hWnd,CMySplitter::ORIENTATION_VERT,
        200, rcBoundingBox);
```

When the workspace is resized, it must inform the splitter.

```
 LRESULT OnSize(UINT uMsg, WPARAM wParam, LPARAM lParam,
              BOOL& bHandled){
      int ClientAreaWidth = LOWORD(lParam);
      int ClientAreaHeight = HIWORD(lParam);
      if (m_VerticalSplitter.IsWindowVisible()) {
            RECT rcBoundingBox;
            GetClientRect(&rcBoundingBox);
            m_VerticalSplitter.SetBoundingBox(rcBoundingBox);
            m_VerticalSplitter.GetFirstPane(rcBoundingBox);
           // here would need to update contents of workspace
      }
 return 0;
 }
```

Dialog Boxes

In addition to providing classes that manage windows, ATL also provides a range of classes for dialog boxes. `CDialogImpl` is used to manage dialog boxes and just like with `CWindowImpl`, you must derived your class from it manually. You must also manually create a dialog template using Developer Studio's ResourceView. The ATL Object Wizard lets you create a class derived from `CAxDialogImpl` and automatically adds a default dialog template for it. The difference between `CDialogImpl` and `CAxDialogImpl` is that the latter supports ActiveX control

containment, which is an excellent feature if you plan to add ActiveX controls to the dialog, but is unnecessary overhead (though not much) if not needed.

Once you have a class derived from one of the ATL dialog templates, you may add message handlers to it and expand the message map, and edit the dialog templates with ResourceView.

`CSimpleDialog` can be used where you need to construct a dialog based on a dialog template, initialize fields within `OnInitDialog`, then display the dialog, react to messages, and then handle OK and Cancel selections. `CSimpleDialog` has its own predefined message map, in which the `WM_INITDIALOG` causes `OnInitDialog` to be called, and either `IDOK` or `IDCANCEL` causes `::EndDialog` to be executed. If you wish to handle other message or commands you can either add your own message map in your derived class or you can use `CDialogImpl/CAxDialogImpl`.

An important point to note if adding your message map to your derived class is that by default the message map in `CSimpleDialog` itself will then not be used – if you are interested in detecting `WM_INITDIALOG`, `IDOK` or `IDCANCEL`, you must either add appropriate entries to your message map or chain the one in `CSimpleDialog`.

`CDialogImpl` is available for more sophisticated needs. It does not have a default message map.

Sample : SimpleModalAndModeless

This project shows how to use `CSimpleDialog`, and in both modal and modeless forms how to use `CDialogImpl`.

The `MySimpleDialog` class is derived from `CSimpleDialog` and contains its own message map. It has a checkbox that in `OnInitDialog` is initialized to true, and in `OnOk` the current value is retrieved.

```
class MySimpleDialog : public CSimpleDialog<IDD_SIMPLE_DIALOG>{
public:
      MySimpleDialog(): m_bButton(BST_CHECKED){}
      BEGIN_MSG_MAP(CMainDialog)
      COMMAND_ID_HANDLER(IDOK, OnOk)
      COMMAND_ID_HANDLER(IDCANCEL, OnCloseCmd)
      MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
      END_MSG_MAP()
      UINT m_bButton;
      LRESULT OnInitDialog(UINT uMsg, WPARAM wParam,
                           LPARAM lParam, BOOL& bHandled){
            CheckDlgButton(IDC_SIMPLE_CHECKBOX, m_bButton);
            return 0;
      }
      LRESULT OnOk(WORD wNotifyCode, WORD wID, HWND hWndCtl,
                  BOOL& bHandled){
            m_bButton = IsDlgButtonChecked(IDC_SIMPLE_CHECKBOX);
            OnCloseCmd(wNotifyCode, wID, hWndCtl, bHandled);
            return 0;
      }
```

```
};
```

This dialog is displayed with the following calls:

```
MySimpleDialog dlg;
dlg.DoModal();
```

Note that the windows for the controls in the dialog do not exist immediately after the dialog's C++ class is constructed – we do get a chance to change them programmatically in `OnInitDialog`, which is called after they are created but before the dialog is rendered on screen.

The `MyModalDialog` is a modal dialog, whose main feature is a pushbutton, which when pressed relocates itself!

```
class MyModalDialog : public CDialogImpl<MyModalDialog>{
      public:
            MyModalDialog(){}
            enum {IDD=IDD_MODAL_DIALOG};
            BEGIN_MSG_MAP(MyModalDialog)
            COMMAND_ID_HANDLER(IDOK, OnOk)
            COMMAND_ID_HANDLER(IDC_CTL_RELOCATE_BUTTON, OnRelocate)
            END_MSG_MAP()
            LRESULT OnRelocate(WORD wNotifyCode, WORD wID,
                  HWND hWndCtl, BOOL& bHandled){
            RECT r;
            CWindow wnd(GetDlgItem(IDC_CTL_RELOCATE_BUTTON));
            wnd.GetWindowRect(&r);
            r.left +=1;r.top+=1;r.right +=1;r.bottom+=1;
            wnd.MoveWindow(&r, TRUE);
            return 0;
      }
      LRESULT OnOk(WORD wNotifyCode, WORD wID, HWND hWndCtl,
                  BOOL& bHandled)   {
            EndDialog(IDOK);
            return 0;
      }
};
```

This dialog is displayed with the following calls:

```
MyModalDialog dlg;
dlg.DoModal();
```

The `MyModelessDialog` is a modeless dialog which is implemented as follows:

```
#define WM_MODELESSCHILD_DESTROYED WM_USER+1
class MyModelessDialog : public CDialogImpl<MyModelessDialog>{
      public:
            CWindow m_ParentWnd;
            MyModelessDialog(){}
            enum {IDD=IDD_MODELESS_DIALOG};
      BEGIN_MSG_MAP(MyModelessDialog)
      COMMAND_ID_HANDLER(IDOK, OnOk)
      COMMAND_ID_HANDLER(IDCANCEL, OnCancel)
      END_MSG_MAP()
```

```
      LRESULT OnOk(WORD wNotifyCode, WORD wID, HWND hWndCtl,
                  BOOL& bHandled){
            m_ParentWnd = GetParent();
            DestroyWindow();
            return 0;
      }
      LRESULT OnCancel(WORD wNotifyCode, WORD wID, HWND hWndCtl,
             BOOL& bHandled){
            m_ParentWnd = GetParent();
            DestroyWindow();
            return 0;
      }
      void OnFinalMessage(HWND hWnd){
            m_ParentWnd.PostMessage(WM_MODELESSCHILD_DESTROYED,0,0);
      }
};
```

One issue that requires some application-specific design is how to destroy modeless dialogs – both the window and the C++ class which represents it.

The modeless dialog itself can detect when it should disappear – when the user clicks in the *OK* or *Cancel* buttons, or the *Close* button in the window frame (the "X" in the top right corner of the window). Some developers have the class which created the modeless dialog (e.g. the main application window class / mainframe) maintain a data member which stores an instance of a modeless dialog class. The lifetime of this instance mirrors the lifetime of the entire application – its visibility is turned on or off as appropriate. This technique works fine and is often highly suitable.

Other developers wish to maintain an instance of a modeless dialog class only so long at its window is displayed. Once its window is dismissed they wish both the window and the C++ class instance to be destroyed. The modeless dialog itself can detect when to disappear – and its `OnOK` or `OnCancel` methods can call `DestroyWindow`. This will destroy the on-screen window, but the instance of the C++ class is still in memory. If `delete` is called on it in `OnOK` or `OnCancel` it will crash – as the instance is not finished processing. Instead, the piece of code that created the modeless dialog class instance should be told to destroy it, using a custom window message. A good place to put this is inside `OnFinalMessage`. The only problem here is to get the parent window's handle. By the time `OnFinalMessage` is called the window of the modeless dialog is destroyed, so it we called `GetParent` directly there it would fail. Therefore we call it elsewhere, and use a member variable to store the value, and we use this value in `OnFinalMessage`. The final task is to detect the custom message arriving in the parent and react appropriately. The following code shows how the parent creates and destroyed the modeless dialog class.

```
BEGIN_MSG_MAP(CMainDialog)
COMMAND_ID_HANDLER(ID_SHOW_MODELESS, OnShowModeless)
MESSAGE_HANDLER(WM_MODELESSCHILD_DESTROYED, OnDestroyModeless)
END_MSG_MAP()
LRESULT OnShowModeless(WORD wNotifyCode, WORD wID,
                  HWND hWndCtl, BOOL& bHandled){
            if (!m_dlgModeless){
                  m_dlgModeless=new (MyModelessDialog);
```

```
                m_dlgModeless->Create(m_hWnd);
                m_dlgModeless->ShowWindow(SW_SHOW);
            }
            return 0;
        }
        LRESULT OnDestroyModeless(UINT uMsg, WPARAM wParam,
                LPARAM lParam, BOOL& bHandled){
            if (m_dlgModeless){
            OutputDebugString(TEXT("Destroying modeless class\n"));
            delete m_dlgModeless;
            m_dlgModeless=NULL;
        }
        return 0;
}
```

Windowing for ActiveX Controls

ATL provides good support for ActiveX control development and client containment. Here we will examine a number of windowing issues related to ActiveX controls from an ATL perspective.

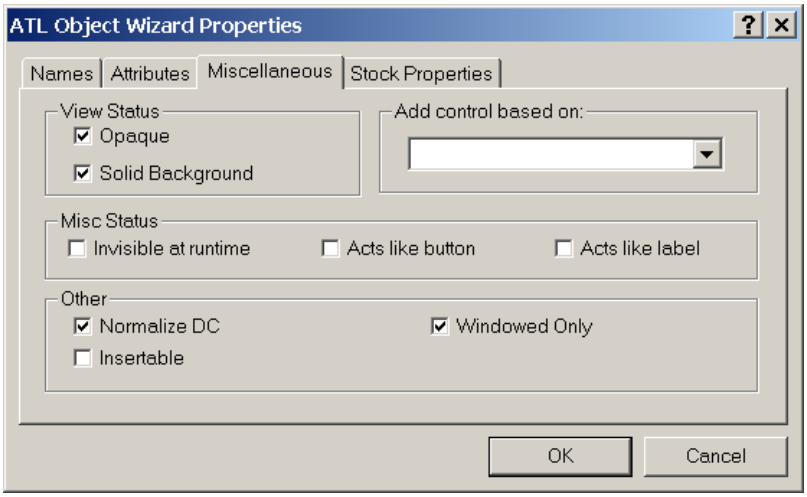
Windowed or Windowless Controls

When creating ActiveX controls within ATL Object Wizard, in the Miscellaneous tab there is an option called *"Windowed Only"*. By default, this checkbox is unselected.

In the past every ActiveX control had its own window, which became a child window of the container (e.g. dialog box) in which the ActiveX control was placed. When one or two controls were used, this was not a problem. When fifty controls were used, it was noticeably slower. The idea behind windowless controls is that the control does not have its own window; rather it lives within the window of its container, which improves performance. Some containers mandate that the control have its own window, and some will allow the control not to. Hence even a control that is capable of being windowless might not always be created without a window. If the *"Windowed Only"* checkbox is selected in the ObjectWizard, then the control will definitely be created with a window. Some controls might have specific requirements that mandate this. In the generated code, it results in this line being added to the control class' constructor:

```
            m_bWindowOnly = TRUE;
```

In the ATL templates, this variable is used inside a member function called `CComControlBase::InPlaceActivate`, where it results in `CComControl::CreateControlWindow` being called, which calls `CWindow::Create`. The main class in your ATL ActiveX project will derive from ATL's `CComControl`, which in turn derives from ATL's `CComControlBase`.



Basing an ActiveX Control on a Standard or Common Control

In addition to being based on a generic window (which needs to be rendered in `OnDraw`), an ATL ActiveX control may also be based or an existing Standard or common control. The Miscellaneous tab also has an *Add control based on:* combo-box. This contains a list of some standard and common controls and one of them may be chosen or another name entered. Note that if the selection in this combo-box is not empty then the *Windowed Only* option is disabled. The ATL ActiveX control will superclass the standard or common control.

Assuming we superclass the listview, the effect this has on the generated code is that a `CContainedWindow` variable is added and this is initialized in the constructor (note the last parameter is the message map id of 1):

```
CContainedWindow m_ctlSysListView32;
CLISTVIEWDEMO() :m_ctlSysListView32(
                    _T("SysListView32"),this,1){
      m_bWindowOnly = TRUE;
}
```

The message map has this entry removed:

```
DEFAULT_REFLECTION_HANDLER()
```

And these entries added:

```
ALT_MSG_MAP(1)
      // Replace this with message map entries for
  // superclassed SysListView32
END_MSG_MAP()
```

Messages that were destined for the listview are now sent to the parent, with a message map id of 1. This alternative message map could be populated with message handlers that we wish to control.

ActiveX Control Message Reflection

With parent-child relationships among windows, the child window based on a control often sends notification messages to the parent, which the parent will detect in its window procedure. The normal mechanism of communication from an ActiveX control to its container is via ActiveX events. The ActiveX control fires these events and the

container traps them. As an ActiveX control could be used with a variety of containers, it would be much better to handle messages related to the ActiveX control within the control itself, and not in its parent, and also when the control has something significant to tell the parent, to so this via ActiveX events.

This concept is supported through the use of message reflection. The parent will reflect back to the child any window messages it receives regarding the child. The child will process these in its own window procedure, and if needed fire ActiveX events.

The ActiveX control container provides message reflection. It is an optional feature – a control can at runtime determine if it is being contained in a container that supports it by examining the ambient property MessageReflect – if it is true then the feature is supported. If it is not supported, then there are three options. Either the control is not very functional, or the parent has to do a lot of control-specific work, or the control could create an extra window of its own to sit between the parent and the main control window (the control window's immediate parent would be the intermediate window, which is also owned by the control and messages sent to it from children would be processed with the control itself.

To enable the control to determine that the messages have been reflected back from the parent as opposed to being sent directly to the control, the value `OCM_BASE` is added to the message id.

With ATL control containment, there is support for message reflection, using two macros and two member function of `CWindowImplRoot`. The two macros may be used in messages maps, and in their implementation simply call the equivalent member function. `REFLECT_NOTIFICATIONS` is used in the parent window and it calls `ReflectNotifications` in `CWindowImplRoot`, which adds `OCM_BASE` to the message value and calls `SendMessage` to send it to the child window.

The `DEFAULT_REFLECTION_HANDLER` macro is used in the control and it calls `DefaultReflectionHandler` in `CWindowImplRoot`, which removes the added `OCM_BASE` from the message value and calls `DefWindowProc`.

Rendering within ActiveX Controls

If your ActiveX control is superclassing a Windows standard or common control, normally you need to do nothing to redraw it. The control itself will refresh as needed. You may influence what it draws by setting values on the Windows control, such as the text to display. For other ActiveX controls, your code will be called by the control-hosting infrastructure when the control needs to be redrawn.

If the control is windowed, `CComBase` detects the `WM_PAINT` message and calls `CComControlBase::OnDrawAdvanced`, which finally calls your `OnDraw` method, which you will need to supply. If the control is windowless, the container will call `IViewObject::Draw`, which is implemented in ATL's `CComControlBase::IViewObject_Draw`; it then calls `CComControlBase::OnDrawAdvanced`, which calls your `OnDraw`. It is passed a structure of type `ATL_DRAWINFO`. The ATL ObjectWizard automatically generates this default implementation of `OnDraw`:

```
HRESULT OnDraw(ATL_DRAWINFO& di){
      RECT& rc = *(RECT*)di.prcBounds;
      Rectangle(di.hdcDraw, rc.left, rc.top,
                          rc.right, rc.bottom);
      SetTextAlign(di.hdcDraw, TA_CENTER|TA_BASELINE);
      LPCTSTR pszText = _T("ATL 3.0 : FULL_CONTROL");
      TextOut(di.hdcDraw,
       (rc.left + rc.right)/2, (rc.top + rc.bottom)/2,
       pszText, lstrlen(pszText));
      return S_OK;
}
```

The two important fields in the `ATLDRAWINFO` structure are `prcBounds`, which is a rectangle stating where to draw, and `hdcDraw` that is a device context, which can be used to actually carry out the drawing.

Composite Controls

A composite control is an ActiveX control which can manage a dialog template that contains standard and common controls, and other ActiveX controls. To the container, a composite control behaves as a single ActiveX control. It is windowed. The ATL template `CComCompositeControl` is the foundation for it.

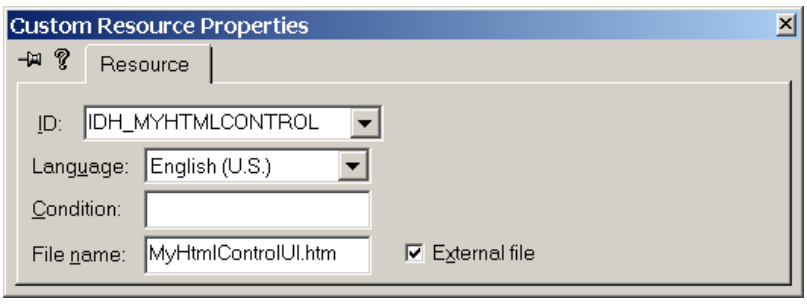
```
CComCompositeControl : public CComControl<T,CAxDialogImpl<T> >
```

Note that it uses `CAxDialogImpl`, which is also the foundation for normal dialog boxes that can contain ActiveX controls. Composite controls can be used when you need to supply somewhat more functionality than a single superclassed standard/common control or a simple drawing control, yet less than a complete dialog box. Think of them as pre-built blocks of functionality ready to slot in when needed.

HTML ActiveX Controls

ATL supports the creation of a type of ActiveX control known as an HTML control. This control hosts the WebBrowser control from Internet Explorer. Naturally the client application could host the web browser control directly, so it only makes sense creating an HTML control if there is some "added-value" which is needed by the client, and you do not wish to implement it directly within the client.

The HTML control generated by the ATL ObjectWizard by default renders an HTML page that is stored as a resource (loaded from an external file during the build).



The control's `OnCreate` method has this call:

```
HRESULT hr = wnd.CreateControl(IDH_MYHTMLCONTROL);
```

which is implemented inside ATL as:

```
HRESULT CAxWindows::CreateControl(DWORD dwResID,
     IStream* pStream=NULL, IUnknown** ppUnkContainer=NULL){
     TCHAR szModule[_MAX_PATH];
     GetModuleFileName(_Module.GetModuleInstance(),
      szModule, _MAX_PATH);
     CComBSTR bstrURL(OLESTR("res://"));
     bstrURL.Append(szModule);
     bstrURL.Append(OLESTR("/"));
     TCHAR szResID[11];
     wsprintf(szResID, _T("%0d"), dwResID);
     bstrURL.Append(szResID);
     ATLASSERT(::IsWindow(m_hWnd));
     return AtlAxCreateControl(bstrURL, m_hWnd, pStream,
       ppUnkContainer);
}
```

The WebBrowser control understands a Windows specific protocol, called `res://` in addition `to http://` and the other standard protocols. With `res://`, the data following the protocol description contains the pathname to a DLL and a resource id of an HTML page, which is located inside the resources of that DLL. If our control were installed under C:\HtmlControlDemo.dll and the resource id of the HTML page were 102, then the full resource string would be:

```
res://C:\HtmlControlDemo.dll/102
```

ActiveX Control Containment

ATL provides a family of templates that support ActiveX control containment. The naming convention is to use `CAx` as a prefix - e.g. `CAxDialogImpl`. To give an indication of the complexity involved, the implementation of most of the ATL functionality for control containment is in the atlhost.h file that is over 2,500 lines long. Luckily most of the work is done for us, so as component and client developers we only have to concern ourselves with the "added-value" of our solution. Note that ATL's control containment functionality works through the official ActiveX control COM interfaces, and they totally disregard with which development environment the control itself was created – VB, MFC, ATL or maybe even in low-level C. ATL control containment works equally well with them all. No special tricks are performed if the ActiveX control itself was also developed with ATL.

The AtlAxWin Window and CAxWindow / CAxHostWindow Templates

To contain ActiveX controls, an application must support a wide variety of COM interfaces and a hosting window, which is used as the parent of the ActiveX control window if it is windowed, and as the actual window for rendering if the ActiveX control is windowless. In implementations of control containment with some other development environments, the hosting window would be a top-level window or more likely a dialog-box. With ATL, the hosting window is created specifically for each hosted control, and this hosting window itself is usually a child of a dialog-box. It is an intermediary between the ActiveX control and the dialog. This hosting window is of the

window class `AtlAxWin`. ATL provides the `CAxHostWindow` and `CAxWindow` templates for client-side management of `AtlAxWin` windows.

If you place an ActiveX control inside a dialog, ATL control containment code will automatically place AtlAxWin windows between the dialog and the ActiveX control. This allows the ATL C++ templates, `CAxHostWindow` and `CAxWindow`, to manage the `AtlAxWin` to handle most of the complex interaction with the ActiveX control, leaving nothing or little for the client application developer to do.

For example, if you have a dialog box with three ActiveX controls, there will be an ATL dialog box C++ class and its window, and three instances of C++ classes based on `CAxHostWindow` and another three windows. In addition, if the ActiveX controls are *windowed-only*, then there will be a further three windows, giving a total of seven windows for this sample dialog. ATL control containment does support windowless controls, so if the ActiveX controls being used also support windowless, only four windows will be used for the dialog. If we assume the dialog has *OK* and *Cancel* standard pushbuttons and three ActiveX controls, one of which is *Windowed-Only*, the window hierarchy as displayed in SPY++ is:

Creating the Control

When `CAxWindow` is instantiated it creates the "`AtlAxWin`" window and then it needs to internally use `CoCreateInstance` to instantiate a COM component, the ActiveX control, and make it live within the "`AtlAxWin`" window. This cannot be done until the "`AtlAxWin`" window actually exists. The easiest time to perform these tasks (and what ATL actually does) is during the processing of `WM_CREATE` for `AtlAxWin`.

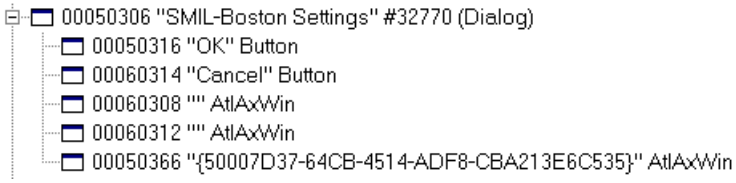
The next question is which CLSID to use? The client code creating `CAxWindow` will know and it somehow needs to pass this information into the handler for `WM_CREATE`. Your first idea might be to use the `LPCREATESTRUCT` that is passed in to `CreateWindow`. The client code can be written to instantiate `CAxWindow` directly, but more often it instantiates a `CAxDialogImpl`, which is based on a dialog template in the resources, which contains embedded ActiveX controls. The resource file will also contain properties for each ActiveX control, and these also need to be passed to the creation code. ATL uses the `LPCREATESTRUCT` field for a pointer to a stream for this property information, so it is not available for the CLSID. The alternative approach, which ATL takes, is to set the window name to a CLSID or information from which the CLSID can be deduced. The "`AtlAxWin`" is never displayed with a title bar, so end-users will never see it. Once extracted, it is set to `NULL`. Hence the previous SPY++ screen dump showed empty titles for the `AtlAxWin` windows.

To see this is action, set up a break point in your debugger on ATL's `AtlAxWindowProc` in the atlhost.h header, run the application. When it hits the breakpoint, examine the window hierarchy in SPY++.

```
static LRESULT CALLBACK AtlAxWindowProc(HWND hWnd,
                    UINT uMsg, WPARAM wParam, LPARAM lParam){
      switch(uMsg){
      case WM_CREATE:
            . . .
       // Property stream is lpCreate->lpCreateParams
       // This value is 0 when creating AtlAxWin directly, and
       // is set to poitner to an IStream based on global memory
       // initialised from resources for dialog-based controls
       CREATESTRUCT* lpCreate = (CREATESTRUCT*)lParam;

       // Get string defining CLSID to use from window title
            int nLen = ::GetWindowTextLength(hWnd);
            LPTSTR lpstrName = (LPTSTR)_alloca((nLen + 1)
                                        * sizeof(TCHAR));
            ::GetWindowText(hWnd, lpstrName, nLen + 1);
            ::SetWindowText(hWnd, _T(""));
```

As each ActiveX control is created, the debugger stops at the `GetWindowText` call. In SPY++, we see a window of type AtlAxWin with a name of the CLSID. After this is extracted, the call to `SetWindowText` NULLs the title. This happens for each of the three controls. The output from SPY++ when the third and final control is created is:



We should emphasize again, that the AtlAxWin is an extra window to the dialog itself and (if windowed only) the ActiveX control. If the control is *windowed only*, then inside `CComControlBase::InPlaceActivate` which calls `CComControlBase::CreateControlWindow` which actually creates the window as a child of AtlAxWin. Once this is done, the output from SPY++ is as in the previous screen dump.

Which control to create?

We have seen how the window title is used to pass in a string to the `WM_CREATE` handling code. This string is usually a CLSID of the ActiveX control to create. This is the case when the ActiveX control is created based on a dialog template from the resource file. However, when used from client application code, the string can be set to other values, such as a PROGID or a URL or even the pathname to a MS-WORD document, and an appropriate ActiveX control for these will be created. ATL uses the following ordered set of rules to decided which CLSID to use in `CoCreateInstance`:

- Firstly, if window name is NULL, or first character of window name is 0, do not create any control and return
- Secondly, if the initial seven characters of the window name are "MSHTML:" (case-insensitive), then use CLSID_HTMLDocument (set a flag to note it can handle HTML) and return
- Thirdly, if the string is shorter than 255 characters, then do the following. If the string begins with "{" use CLSIDFromString to determine if it is a CLSID, otherwise use CLSIDFromProgId to determine if it is a ProgID – if either of these succeeded, then use CoCreateInstance with the CLSID of whichever technique worked, and return
- Finally, if none of the above worked, then call CoCreateInstance with CLSID_WebBrowser (set a flag to note it can handle HTML)

This is implemented in the static function CreateNormalizedObject in atlhost.h.

Note that if a non-NULL string is passed in which does not contain "MSHTML:", a string-ified CLSID or a ProgID, then CLSID_WebBrowser is used – it is the default. Hence we could pass in a pathname to a file from MS-WORD or any other application which CLSID_WebBrowser can access through its Active Document support, or any files that CLSID_WebBrowser can directly understand, such as files with the suffixes *.htm or *.txt .

If the string was a CLSID or a ProgID, once CoCreateInstance was called ATL is finished with it. Otherwise there is a bit more to do, as CLSID_HTMLDocument or CLSID_WebBrowser have been instantiated, but now they need to be told what to display. The AtlControlEx method does this.

CLSID_HTMLDocument is Internet Explorer's rendering engine for HTML. ATL uses it if the window title started with the seven characters "MSHTML:". The rest of the window title can be raw HTML. This needs to be passed to the CLSID_HTMLDocument instance to render. The relevant snippet of code from CreateControlEx is as follows:

```
CComPtr<IHTMLDocument2> spHTMLDoc2;
hr = spUnk->QueryInterface(IID_IHTMLDocument2,
  (void**)&spHTMLDoc2);
if (SUCCEEDED(hr)){
 CComPtr<IHTMLElement> spHTMLBody;
hr = spHTMLDoc2->get_body(&spHTMLBody);
      if (SUCCEEDED(hr))
        hr = spHTMLBody->put_innerHTML(
                    CComBSTR(lpszTricsData + 7));
}
```

We see it passes the window title from the eighth character onwards to the put_innerHMTL method of the BODY. A valid message could be:

```
MSHTML:<P>Hello World</P>
```

CLSID_WebBrowser is Internet Explorer's full engine. It incorporates the rendering engine for HTML, Active Document support, scripting and dynamic HTML. If CLSID_WebBrowser was instantiated, then the full window title is passed in to the instance via its IWebBrowser2::Navigate2 method, and it can decide what to do with it. The relevant snippet of code from CreateControlEx is as follows:

```
        CComPtr<IWebBrowser2> spBrowser;
        spUnk->QueryInterface(IID_IWebBrowser2, (void**)&spBrowser);
        if (spBrowser){
              CComVariant ve;
              CComVariant vurl(lpszTricsData);
              spBrowser->put_Visible(VARIANT_TRUE);
              spBrowser->Navigate2(&vurl, &ve, &ve, &ve, &ve);
  }
```

If the window title is a URL, the web browser will access it over HTTP and render the contents. If the title is another protocol that Internet Explorer understands (e.g. mailto:info@example.com), then it will be used. If the title is a filename (e.g. a full pathname to a Word 2000 document), then the ActiveX Document functionality comes into play.

Enthusiastic Windows shell programmers will note that Navigate2 is used. In addition to accepting an URL/pathname, this method also accepts a PIDL to a location in the Windows Shell. So if we set the window title that that of a PIDL, can we browse the shell? Well, no. As is, the code here initializes the VARIANT parameter to Navigate2 as a string, and when you look at the IWebBrowser2::Navigate2 in the MSDN, you will see that strings are processed as if they are URLs. To browse the shell, we have to pass in a VARIANT of type VT_ARRAY|VT_UI1 containing a SAFEARRAY with the PIDL. The ATL code does not do this and it is a bad idea to modify it. However, there is nothing stopping us calling IWebBrowser2::Navigate2 sometime later.

The later chapter that discusses the views in WTL applications has samples of using all the possible window title formats and a PIDL.

CAxDialogImpl - ActiveX Controls within Dialog Boxes

Dialog boxes that do not contain ActiveX controls are managed by ATL's CDialogImpl template. Classes deriving from this must be manually created – the ATL ObjectWizard does not provide any help here. ATL's CAxDialogImpl manages dialog boxes that do support ActiveX controls. The ATL ObjectWizard, in its miscellaneous section, does support creating these. As there is some extra processing inside CAxDialogImpl, if you do not plan to host ActiveX controls, you would be (slightly) better off using CDialogImpl. If you decide later that you do need ActiveX controls, you can simply switch CAxDialogImpl for CDialogImpl in the header file. The ObjectWizard is only available if DevStudio thinks it is editing an ATL project. This is of course the case if you have created the project with the ATL AppWizard. This is not the case it you have created the project with the WTL AppWizard (and not selected the *COM Server* option). There is a way to fix this – which is discussed in the later chapter on dialogs and controls.

With the ResourceView, you can edit the resource template used as the basis for a dialog. The control CLSID, its id and its location in the dialog are stored in the DIALOG section of the RC file. Additional properties for each ActiveX control are stored in the `DLGINIT` section. When `CAxDialogImpl` is constructing a dialog, it needs to process both sections in order to create the specified dialog.

```
/////////////////////////////////////////////////////////////
// Dialog

IDD_MYACTIVEXDIALOG DIALOG DISCARDABLE 0, 0, 186, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Dialog"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "OK",1,129,7,50,14
    PUSHBUTTON      "Cancel",2,129,24,50,14
    CONTROL         "",IDC_MEDIAPLAYER1,
                    "{22D6F312-B0F6-11D0-94AB-0080C74C7E95}",
                        WS_TABSTOP,10,0, 118,75
END

/////////////////////////////////////////////////////////////
Dialog Info

IDD_MYACTIVEXDIALOG DLGINIT
BEGIN
    IDC_MEDIAPLAYER1, 0x376, 340, 0
0x0000, 0x0000, 0x0001, 0x0000, 0x1383, 0x0000, 0x0c67, 0x0000,
0x0003, 0xffff, 0xffff, 0x000b, 0x0000, 0x000b, 0xffff, 0x000b,
. . .
0x0000, 0x000b, 0x0000, 0x0003, 0xfda8, 0xffff, 0x000b, 0x0000,
    0
END

/////////////////////////////////////////////////////////////
```

During dialog construction, ATL creates an "`AtlAxWin`" for each control, and puts the `DLGINIT` data for the control in a stream based on global memory. As discussed earlier, this is passed into the handler for `WM_CREATE` in its `LPCREATESTRUCT` structure, and from there the contents of the stream are parsed by the control (e.g. if the control was created using ATL, its property map is traversed).

The final issue to consider is once constructed and initialized, how does the containing application programmatically communicate with the control at run-time. MFC has its `COleDispatchDriver` class. VB internal run-time dispatching functionality, which the application developer can influence using the VB *References* dialog box. What happens with ATL? Like the other solutions, it is better that ATL is told at compile time about the ActiveX controls it will be using. This can be done using the #import construct. We have to #import the DLL containing the ActiveX control.

Question: To use #import, we have to load in the CRT, something we often wish to avoid – is there any better way to access the type information for ActiveX controls, which does not consume hundreds of lines of code, yet gives us access to the dual interfaces of the ActiveX control?

We can use `CWindow::GetDlgControl` to extract an interface pointer to the control and then we can use it like any other COM interface.

Internally, what happens is that the CAxHostWindow/CAxWindow instance representing the ActiveX control maintains a data-member of the IUnknown interface to the control, which it received from CoCreateInstance (the riid passed into it is IID_IUnknown). It has a message handler for an ATL custom message type WM_ATLGETCONTROL, and it responds to it by add-ref'ing the IUnknown pointer it has and returning it as the handler result.

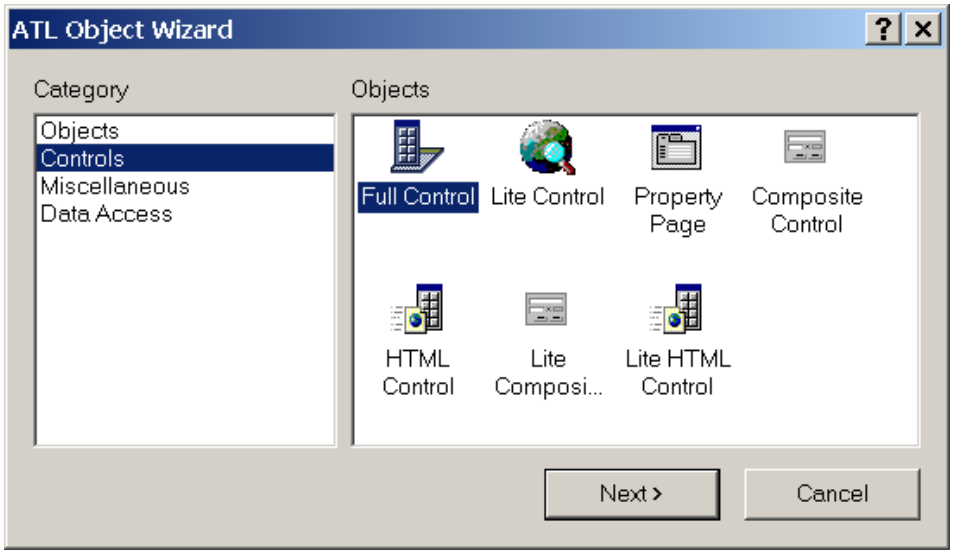
```
MESSAGE_HANDLER(WM_ATLGETCONTROL, OnGetControl)
LRESULT OnGetControl(UINT /*uMsg*/, WPARAM /*wParam*/,
            LPARAM /*lParam*/, BOOL& /*bHandled*/){
      IUnknown* pUnk = m_spUnknown;
      if (pUnk)
            pUnk->AddRef();
      return (LRESULT)pUnk;
}
```

When CWindow::GetDlgControl is called, it converts the dialog item id into a window handler with the GetDlgItem call (this window will be of type "AtlAxWin"), and then it calls AtlAxControl helper function, which causes a WM_ATLGETCONTROL message to be sent to the AtlAxWin window.

```
HRESULT GetDlgControl(int nID, REFIID iid, void** ppUnk){
      HRESULT hr = E_FAIL;
      HWND hWndCtrl = GetDlgItem(nID);
      if (hWndCtrl != NULL){
            *ppUnk = NULL;
            CComPtr<IUnknown> spUnk;
            hr = AtlAxGetControl(hWndCtrl, &spUnk);
            if (SUCCEEDED(hr))
                  hr = spUnk->QueryInterface(iid, ppUnk);
      }
      return hr;
}
ATLINLINE ATLAPI AtlAxGetControl(HWND h, IUnknown** pp){
      *pp = (IUnknown*)SendMessage(h, WM_ATLGETCONTROL, 0, 0);
      return (*pp) ? S_OK : E_FAIL;
}
```

Sample : ATLControlDemo

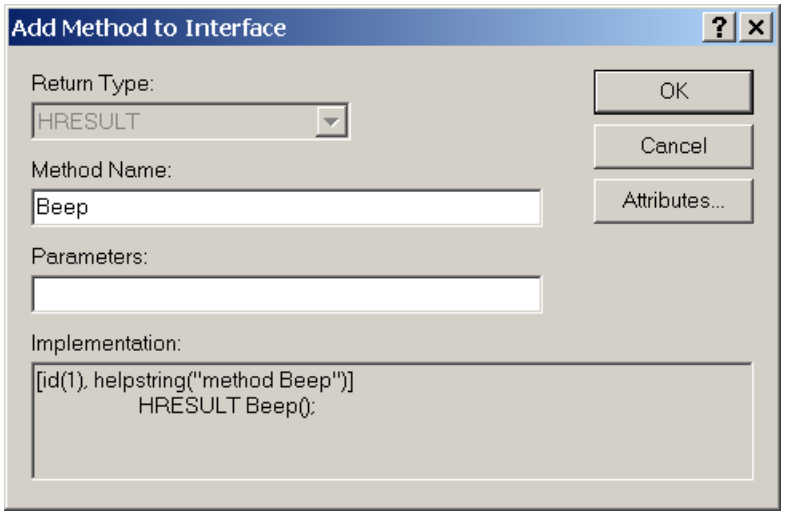
The ATLControlDemo sample shows how to create a control in one project and host it in a different project.



The one workspace is used for both. The fact that both projects are created with ATL does not have an impact on how they communicate – they will still use the rules of ActiveX controls. Use the ATL AppWizard to create the workspace and first project named ATLControlDemo. The project should be of type DLL. Use the ATL Object Wizard to insert a Full Control. Name it "MyAtlControl".

In ClassView, select `IMyAtlControl`, and right-click to bring up the context menu, and select *Add Method*.

Use the dialog to add a method called `Beep`. This adds the method to the IDL file and adds a default implementation to the `CMyAtlControl` class.



Change it to call the `Beep` API.

```
STDMETHODIMP CMyAtlControl::Beep(){
      ::Beep(200,200);
      return S_OK;
}
```

That is the ActiveX control finished. Build the project, which automatically registers it on the development machine. It also adds type information to the DLL.

Now in FileView, select the Workspace root, right click and select *Add New Project to Workspace.* Select the ATL AppWizard and create a new EXE-based project called ATLControlContainer.

Use the ObjectWizard to insert a new dialog box. In ResourceView, select the dialog template, right-click and select Add ActiveX Control. Select the `MyAtlControl` from the list. If it does not appear in the list it is probably because you have not actually compiled the AtlControlDemo project (which would have automatically registered the control). In Properties, assign the id `IDC_MYATLCONTROL` to the control.

In the implementation file for the dialog box, add this import statement:

```
#import "..\debug\ATLControlDemo.dll" no_namespace
```

Change the `OnInitDialog` method to call the exposed `Beep` method in the ActiveX control:

```
      LRESULT OnInitDialog(UINT uMsg, WPARAM wParam, LPARAM lParam,
BOOL& bHandled){
            CComPtr<IMyAtlControl> m_pIMyAtlControl;
            GetDlgControl(IDC_MYATLCONTROL, __uuidof(IMyAtlControl),
                        (void **) &m_pIMyAtlControl);
            m_pIMyAtlControl->Beep();
            return 1; // Let the system set the focus
      }
```

In the application's main cpp file, ATLControlContainer.cpp, include the header file for the dialog class and instantiate the dialog and call its `DoModal`. That's it - compile and run. You should hear a beep should as the dialog is being displayed.

Chapter 4
WTL Quick Tour

Objectives

The objectives of this chapter are to:

- Examine the templates and classes in WTL
- List the features which are not part of WTL
- Describe issues such as the WTL namespace, CRT and error handling
- List the WTL macros
- See how debugging can be improved with AUTOEXP.DAT
- Review the use of CRT and WTL
- Compare MFC and WTL
- Cover WTL's CString class

The WTL Distribution

There are three parts to the WTL distribution – the fifteen C++ header files in the include directory, the WTL AppWizard and the sample projects. The core functionality is in the include directory. The AppWizard generates projects whose source code makes calls into the templates/classes defined in the header files. The sample projects should how to use them. You might spend a few hours examining what the WTL AppWizard produces and the sample projects, but you will spend many weeks learning all about the templates/classes defined in the header files.

WTL provides a large number of templates and classes dedicated to user interface development. They may be used in tandem with, rather than competing with, other template libraries, such as ATL, STL and VC++ OLE DB Data templates. All these template libraries together may be considered as a competitor to MFC and VB.

In most parts, WTL is a very thin layer on the low level Win32 APIs. In a small number of areas (e.g. scroll-bars, splitters, print preview) there is substantial new functionality.

Apart from the WTL AppWizard, there are no specific WTL wizards integrated with DevStudio. One can use the ATL Windows Message Handler Wizard to add handlers, but these do not use WTL's message crackers.

Templates and Classes

WTL is a collection of C++ templates and classes. There are some isolated cases of a small group of classes using inheritance (e.g. `CDC, CPaintDC`, `CWindowDC` and `CPrintDC`), but by and large WTL's templates and classes are independent of each other and they share no common root, hence there is no such thing as a hierarchy diagram for WTL. There simply is no large hierarchy.

In this section we will provide an overview of the functional areas covered by WTL and list the templates and classes available. Each functional area is neatly contained within its own WTL header file, which we also list.

Every WTL application must include atlapp.h, which internally includes atluser.h and atlgdi.h. The other WTL header files only need to be included if the functionality they contain will be used in the application.

Application Services (atlapp.h)

These classes provide the foundation for the module (EXE or DLL) and control the message loop. They also support message filtering and idle handling.

```
class CAppModule : public CComModule
class CServerAppModule : public CAppModule
class CMessageLoop
class CMessageFilter
class CIdleHandler
```

Standard Controls and Common Controls (atlctrls.h)

WTL provides a wrapper template for each standard control and common control in the Windows OS. The template is typically used with ATL's `CWindow`, so an appropriate typedef is also provided for each control. In code, you will normally use the typedef'ed value, and not the template directly.

Each control is defined as follows:

```
template <class TBase> class CStaticT;
typedef CStaticT<CWindow>     CStatic;
```

Here is the full list of controls:

```
CButtonT          CButton          CTrackBarCtrlT         CTrackBarCtrl
CListBoxT         CListBox         CUpDownCtrlT           CUpDownCtrl
CComboBoxT        CComboBox        CProgressBarCtrlT      CProgressBarCtrl
CEditT            CEdit            CHotKeyCtrlT           CHotKeyCtrl
CScrollBarT       CScrollBar       CAnimateCtrlT          CAnimateCtrl
CToolTipCtrlT     CToolTipCtrl     CRichEditCtrlT         CRichEditCtrl
CHeaderCtrlT      CHeaderCtrl      CReBarCtrlT            CReBarCtrl
CListViewCtrlT    CListViewCtrl    CMonthCalendarCtrlT    CMonthCalendarCtrl
CTreeViewCtrlT    CTreeViewCtrl    CDateTimePickerCtrlT   CDateTimePickerCtrl
CToolBarCtrlT     CToolBarCtrl     CIPAddressCtrlT        CIPAddressCtrl
CStatusBarCtrlT   CStatusBarCtrl   CPagerCtrlT            CPagerCtrl
CTabCtrlT         CTabCtrl
```

These controls can be created using the `Create` method, but more often a dialog template defined in the ResourceView will contain the layout of the dialog and the controls which are to appear on it, and in source code you can instantiate one of these control templates and call its `Attach` method to hook up with the control in the dialog box. So to add a string to a list box in a dialog you might use the following:

```
CListBox m_ListBox;
. . .
      m_ListBox.Attach(GetDlgItem(IDC_LIST_DEMO));
      m_ListBox.AddString(TEXT("One"));
```

Note that ATL already supports ActiveX control containment (as discussed in detail in the previous chapter on ATL Windowing) and WTL adds no additional support.

There are some additional classes/templates for controls that provide extra functionality.

WTL's `CComboBoxEx` is a wrapper for Win32's ComboBoxEx control, which adds support for image lists to the standard combo box. In WTL, `CComboBoxExT` is derived from `CComboBoxT`, and adds extra functionality as needed.

```
template <class TBase> class CComboBoxExT
                              : public CComboBoxT< TBase >
typedef CComboBoxExT<CWindow> CComboBoxEx;
```

Flat scroll-bars are supported in WTL. In functionality they are the same as normal scrollbars, but in appearance, as the name suggests, they are flat, without the raised beveled edge.

```
template <class T> class CFlatScrollBarImpl
template <class TBase> class CFlatScrollBarT : public TBase,
           public CFlatScrollBarImpl<CFlatScrollBarT< TBase > >
typedef CFlatScrollBarT<CWindow> CFlatScrollBar;
```

The treeview control deals with a `HTREEITEM`, which is a handle to a complicated structure containing details of how each node should appear in the tree. To simplify dealing with `HTREEITEM`, WTL has a class called `CTreeItem`, which provides methods to get and set text, image, data and state information, and to walk the tree. The `CTreeViewCtrlExT` template is based on the `CTreeViewCtrlT` template.

```
template <class TBase> class CTreeViewCtrlExT :
                               public CTreeViewCtrlT< TBase >
typedef CTreeViewCtrlExT<CWindow> CTreeViewCtrlEx;
class CTreeItem; // wrapper for HTREEITEM
```

`CTreeViewCtrlExT` furnishes methods that deal with `CTreeItem` in addition to the methods that work with raw `HTREEITEM` handles. For example, `CTreeViewCtrlT` provides this method to insert an item:

```
HTREEITEM InsertItem(LPTV_INSERTSTRUCT lpInsertStruct);
```

whereas `CTreeViewCtrlExT` has this method:

```
CTreeItem InsertItem(LPTV_INSERTSTRUCT lpInsertStruct);
```

Edit boxes and rich-edit boxes must deal with a number of common messages, such as `ID_EDIT_COPY`, `ID_EDIT_CUT`, `ID_EDIT_PASTE` and `ID_EDIT_UNDO`. These two helper classes are provided to automate the processing of such messages.

```
template <class T> class CEditCommands;
template <class T> class CRichEditCommands
                                : public CEditCommands< T >
```

Helper classes are also provided to manage image lists and toolinfo.

```
class CImageList;
class CToolInfo : public TOOLINFO
```

A drag-listbox is similar to a normal listbox, but also offers functionality to drag items within the listbox. `CDragListNotifyImpl` supports message processing during the dragging.

```
template <class TBase> class CDragListBoxT
                                : public CListBoxT< TBase >
typedef CDragListBoxT<CWindow> CDragListBox;
template <class T> class CDragListNotifyImpl;
```

Some common controls can be configured to use custom draw to render their items. They send a `NM_CUSTOMDRAW` message. This WTL template helps with the processing of such messages:

```
template <class T> class CCustomDraw
```

Command Bars (atlctrlw.h)

In the past applications presented commands to end users in standard menubars. Then along came toolbars. Later still came the rebar control, first introduced with Internet Explorer. Some advanced applications, such as Microsoft Word 2000, provided the concept of command bars, which are a uniform hosting of menus and toolbar icons. WTL also supports functionality called command bars, but these are different to those that appear in Word 2000.

In WTL, a command-bar is a more powerful version of a menubar. The toolbar is still used. The command bar is hosted in one band and the toolbar in a different band within a single rebar control in the WTL generated application. The bands are movable within that rebar. Command bars (and toolbars) in WTL are not dockable.

A WTL command bar displays menus, and within each menu displays menu-items and the equivalent toolbar icon next to the menu item, thus increasing end-user knowledge of what each toolbar icon means.

WTL provides classes to help with command bars:

```
class CCommandBarCtrlBase : public CToolBarCtrl
template <class T, class TBase = CCommandBarCtrlBase,
            class TWinTraits = CControlWinTraits>
class CCommandBarCtrlImpl
                  : public CWindowImpl< T, TBase, TWinTraits>
class CCommandBarCtrl
                  : public CCommandBarCtrlImpl<CCommandBarCtrl>
```

When generating an application with the WTL AppWizard, and assuming you accept the default settings, then your project will automatically use command-bars and you have no further action to take.

Advanced Controls (atlctrlx.h)

WTL supports a number of more advanced controls.

`CBitmapButton` provides pushbuttons that display an image from an imagelist rather than text.

```
template <class T, class TBase = CButton,
            class TWinTraits = CControlWinTraits>
       class CBitmapButtonImpl
       : public CWindowImpl< T, TBase, TWinTraits>
class CBitmapButton : public CBitmapButtonImpl<CBitmapButton>
```

`CCheckListView` provides a list of checkboxes.

```
template <DWORD t_dwStyle, DWORD t_dwExStyle,
  DWORD t_dwExListViewStyle> class CCheckListViewCtrlImplTraits;
typedef CCheckListViewCtrlImplTraits<WS_CHILD | WS_VISIBLE |
   LVS_REPORT | LVS_SHOWSELALWAYS, WS_EX_CLIENTEDGE,
   LVS_EX_CHECKBOXES | LVS_EX_FULLROWSELECT>
   CCheckListViewCtrlTraits;
template <class T, class TBase = CListViewCtrl,
            class TWinTraits = CCheckListViewCtrlTraits>
       class CCheckListViewCtrlImpl
       : public CWindowImpl<T, TBase, TWinTraits>;
class CCheckListViewCtrl
            : public CCheckListViewCtrlImpl<CCheckListViewCtrl>;
```

`CHyperlink` enables the embedded of web-links within the application. When pressed, a web-browser is started displaying a specific URL.

```
template <class T, class TBase = CWindow,
            class TWinTraits = CControlWinTraits>
class CHyperLinkImpl
            : public CWindowImpl< T, TBase, TWinTraits >
class CHyperLink : public CHyperLinkImpl<CHyperLink>
```

`CWaitCursor` sets the cursor. The actual cursor used depends on an input parameter – which defaults to `IDC_WAIT`.

```
class CWaitCursor
```

`CMultiPaneStatusBarCtrl` provides a status bar with sub-divisions, each of which can contain different text.

```
template <class T, class TBase = CStatusBarCtrl>
class CMultiPaneStatusBarCtrlImpl
            : public CWindowImpl< T, TBase >
class CMultiPaneStatusBarCtrl
        : public CMultiPaneStatusBarCtrlImpl<CMultiPaneStatusBarCtrl>
```

Dynamic Data Exchange (atlddx.h)

DDX is used to transfer data in both directions between C++ data members and dialog box controls. In WTL, DDX involves a single template, `CWinDataExchange`, and many macros, such as `DDX_TEXT`.

```
template <class T> class CWinDataExchange
```

To use DDX, your dialog class should derived from both `CDialogImpl` and `CWinDataExchange`, and it should have a message map, such as:

```
      BEGIN_DDX_MAP(CMyDialogDlg)
            DDX_INT(IDC_AMOUNT_INT, m_nAmount)
```

```
      END_DDX_MAP()
```

Currently this map must be created manually, though it is likely that in a future version of Visual C++ Wizard support will be added.

System Dialogs And Property Sheets (atldlgs.h)

WTL offers a full range of templates which expose the system dialogs.

```
template <class T> class CFileDialogImpl
           : public CDialogImplBase
       class CFileDialog : public CFileDialogImpl<CFileDialog>
template <class T> class CFolderDialogImpl
        class CFolderDialog : public CFolderDialogImpl<CFolderDialog>
class CCommonDialogImplBase : public CWindowImplBase
template <class T> class CFontDialogImpl
           : public CCommonDialogImplBase
class CFontDialog : public CFontDialogImpl<CFontDialog>
class CRichEditFontDialogImpl : public CFontDialogImpl< T >
class CRichEditFontDialog : public
       CRichEditFontDialogImpl<CRichEditFontDialog>
template <class T> class CColorDialogImpl
          : public CCommonDialogImplBase
      class CColorDialog : public CColorDialogImpl<CColorDialog>
template <class T> class CPrintDialogImpl
           : public CCommonDialogImplBase
class CPrintDialog : public CPrintDialogImpl<CPrintDialog> template
<class T> class CPrintDialogExImpl
           : public CWindow, public CMessageMap,
           public IPrintDialogCallback,
           public IObjectWithSiteImpl< T >
class CPrintDialogEx : public CPrintDialogExImpl<CPrintDialogEx>
template <class T> class CPageSetupDialogImpl
           : public CCommonDialogImplBase
class CPageSetupDialog
           : public CPageSetupDialogImpl<CPageSetupDialog>
template <class T> class CFindReplaceDialogImpl
           : public CCommonDialogImplBase
class CFindReplaceDialog
           : public CFindReplaceDialogImpl<CFindReplaceDialog>
```

WTL also offers a range of templates to help with property sheet construction.

```
class CPropertySheetWindow : public CWindow
template <class T, class TBase = CPropertySheetWindow>
class CPropertySheetImpl : public CWindowImplBaseT< TBase >
class CPropertySheet : public CPropertySheetImpl<CPropertySheet>
class CPropertyPageWindow : public CWindow
template <class T, class TBase = CPropertyPageWindow>
      class CPropertyPageImpl : public CDialogImplBaseT< TBase >
template <WORD t_wDlgTemplateID> class CPropertyPage
        : public CPropertyPageImpl<CPropertyPage<t_wDlgTemplateID> >
```

Frames (atlframe.h)

The following classes support frames.

`CFrameWndClassInfo` is the window class for frame windows.

```
class CFrameWndClassInfo
```

`CFrameWindowImpl` is used as the basis of application-level C++ classes that manage the frames.

```
template <class TBase = CWindow, class TWinTraits =
            CFrameWinTraits> class CFrameWindowImplBase
            : public CWindowImplBaseT< TBase, TWinTraits >
template <class T, class TBase = CWindow, class TWinTraits =
       CFrameWinTraits> class CFrameWindowImpl
       : public CFrameWindowImplBase< TBase, TWinTraits >
```

To support MDI, WTL provides these templates:

```
class CMDIWindow : public CWindow
template <class T, class TBase = CMDIWindow, class TWinTraits =
       CFrameWinTraits>
class CMDIFrameWindowImpl : public CFrameWindowImplBase<TBase,
       TWinTraits >
template <class T, class TBase = CMDIWindow, class TWinTraits =
       CMDIChildWinTraits> class CMDIChildWindowImpl
       : public CFrameWindowImplBase<TBase, TWinTraits >
```

The template `COwnerDraw` assists with the implementation of owner draw functionality. It is normally added to a class via multiple inheritance. The difference between `COwnerDraw` and `CCustomDraw` that we mentioned when introducing the controls, is that `CCustomDraw` processes `NM_CUSTOMDRAW` notifications delivered in a `WM_NOTIFY` message. This is supported by the following common controls: header, list view, rebar, toolbar, tooltip, trackbar and tree view controls. `COwnerDraw` processes the `WM_DRAWITEM, WM_MEASUREITEM, WM_COMPAREITEM` and `WM_DELETEITEM` messages, which are typically sent to owner-drawn button, combo box, list box, list view control, or menu items.

```
template <class T> class COwnerDraw
```

`CUpdateUI` is provided to update UI elements dynamically. Menu-items and toolbar buttons can be enabled or disabled using this.

```
class CUpdateUIBase;
template <class T> class CUpdateUI : public CUpdateUIBase
```

GDI (atlgdi.h)

WTL provides templates to manage all the GDI objects. The Boolean template parameter for each of these, `t_bManaged`, specifies whether the GDI object should be deleted in the destructor of the template instance. For each template, WTL has two typedefs, one of which is declared with `t_bManaged` set to `true`, and the other with `t_bManaged` set to `false`.

These templates bring together all the Win32 C APIs which deal with each GDI. These expose typesafe C++ methods which call the underlying C API.

```
template <bool t_bManaged> class CPenT
typedef CPenT<false>          CPenHandle;
typedef CPenT<true>                 CPen;
template <bool t_bManaged> class CBrushT
```

```
typedef CBrushT<false>        CBrushHandle;
typedef CBrushT<true>         CBrush;
template <bool t_bManaged> class CFontT
typedef CFontT<false>         CFontHandle;
typedef CFontT<true>          CFont;
template <bool t_bManaged> class CBitmapT
typedef CBitmapT<false>       CBitmapHandle;
typedef CBitmapT<true>        CBitmap;
template <bool t_bManaged> class CPaletteT
typedef CPaletteT<false>            CPaletteHandle;
typedef CPaletteT<true>       CPalette;
template <bool t_bManaged> class CRgnT
typedef CRgnT<false>          CRgnHandle;
typedef CRgnT<true>                 CRgn;
template <bool t_bManaged> class CDCT
typedef CDCT<false>                 CDCHandle;
typedef CDCT<true>                  CDC;
```

A number of helper classes are derived from `CDC` to provide additional functionality. `CPaintDC` is used to handle `WM_PAINT` messages; it calls Win32's `BeginPaint` in its constructor and `EndPaint` in its destructor. `CClientDC` and `CWindowDC` get the client area and full window DCs of a `HWND` parameter respectively. `CEnhMetaFileDC` is used to render into a enhanced metafile. It calls Win32's `CreateEnhMetaFile` from its constructor and `CloseEnhMetaFile` from its destructor.

```
class CPaintDC : public CDC
class CClientDC : public CDC
class CWindowDC : public CDC
class CEnhMetaFileDC : public CDC
```

Extra functionality is provided for enhanced metafiles. `CEnhMetaFileT` is a wrapper for an enhanced metafile handle. `CEnhMetaFileInfo` provides methods that call Win32's `GetEnhMetaFileBits, GetEnhMetaFileDescription`, `GetEnhMetaFileHeader` and `GetEnhMetaFilePixelFormat`.

```
template <bool t_bManaged> class CEnhMetaFileT
typedef CEnhMetaFileT<false>  CEnhMetaFileHandle;
typedef CEnhMetaFileT<true>   CEnhMetaFile;
class CEnhMetaFileInfo
```

Miscellaneous (atlmisc.h)

Helper classes are provided to assist managing common data types. WTL's `CString` is a perfect emulation of MFC's equivalent. `CSize, CPoint` and `CRect` derived from Win32 data types, and offer method to initialize the values and carry out operations (e.g. summation) on them.

```
class CSize : public tagSIZE
class CPoint : public tagPOINT
class CRect : public tagRECT
class CString;
```

These classes support recent file list and the file searching (using Win32's `FindFirstFile / FindNextFile` APIs).

```
class CRecentDocumentList;
class CFindFile;
```

Printing (atlprint.h)

WTL provides good support for printing.

`PrinterInfo` data is handled via:

```
template <unsigned int t_nInfo> class CPrinterInfo
```

These are provided to manage a handle to a printer:

```
template <bool t_bManaged> class CPrinterT
typedef CPrinterT<false>      CPrinterHandle;
typedef CPrinterT<true>       CPrinter;
```

These are provided to manage printing's `DEVMODE` information:

```
template <bool t_bManaged> class CDevModeT;
typedef CDevModeT<false>      CDevModeHandle;
typedef CDevModeT<true>       CDevMode;
```

When rendering for printing, use this DC type:

```
class CPrinterDC : public CDC
```

To gain information about print jobs, use:

```
class IPrintJobInfo
class CPrintJobInfo : public IPrintJobInfo
class CPrintJob;
```

Print preview window support is provided by:

```
class CPrintPreview;
template <class T, class TBase = CWindow, class TWinTraits =
CControlWinTraits>
class CPrintPreviewWindowImpl
 :public CWindowImpl<T, TBase, TWinTraits>, public CPrintPreview

class CPrintPreviewWindow
            : public CPrintPreviewWindowImpl<CPrintPreviewWindow>
```

Scrolling (atlscrl.h)

Window scrolling is supported through this set of templates:

```
template <class T> class CScrollImpl
template <class T, class TBase = CWindow,
      class TWinTraits = CControlWinTraits>
class CScrollWindowImpl
      :public CWindowImpl<T,TBase,TWinTraits>,public CScrollImpl< T >
template <class T> class CMapScrollImpl: public CScrollImpl< T >
template <class T, class TBase = CWindow,
      class TWinTraits = CControlWinTraits>
class CMapScrollWindowImpl : public CWindowImpl< T, TBase,
            TWinTraits >, public CMapScrollImpl< T >
```

Flat scrollbars are also supported.

```
template <class TBase = CWindow> class CFSBWindowT
 :public TBase, public CFlatScrollBarImpl<CFSBWindowT< TBase > >
typedef CFSBWindowT<CWindow>  CFSBWindow;
```

Splitters (atlsplit.h)

Horizontal and vertical splitters are supported using these templates:

```
template <class T, bool t_bVertical = true> class CSplitterImpl;
template <class T, bool t_bVertical = true, class TBase =
             CWindow, class TWinTraits = CControlWinTraits>
class CSplitterWindowImpl : public CWindowImpl< T, TBase, TWinTraits
>, public CSplitterImpl<CSplitterWindowImpl< T , t_bVertical, TBase,
TWinTraits >, t_bVertical>
template <bool t_bVertical = true> class CSplitterWindowT
       : public CSplitterWindowImpl<
      CSplitterWindowT<t_bVertical>, t_bVertical>
typedef CSplitterWindowT<true>      CSplitterWindow;
typedef CSplitterWindowT<false>     CHorSplitterWindow;
```

Menus (atluser.h)

The `CMenuItemInfo` class is a wrapper for Win32's `MENUITEMINFO` structure and menus are managed through `CMenuT`, which provides typesafe access to all operations that are possible on menus.

```
class CMenuItemInfo : public MENUITEMINFO
template <bool t_bManaged> class CMenuT
typedef CMenuT<false> CMenuHandle;
typedef CMenuT<true> CMenu;
```

What is NOT in WTL

After having considered what are the major pieces of functionality within WTL, now we will examine what is NOT supported.

WTL is solely aimed at developers of desktop user interfaces. It does this exceptionally well. It does not stray into other areas. Competing toolkits, such as VC++'s MFC or VB do provide incredible amounts of functionality covering vast areas. Their supporters might claim these toolkits offer a consistent mutually compatible, well-integrated range of functionality, but WTL supporters will counter with the assertion that MFC and VB are monolithic, slow-changing and simply too big (downloads of apps to be measured in megabytes, compared to WTL downloads which are in kilobytes). The philosophy of template programming – fast, static binding, supporting multiple data types, is still very much alive in WTL and is shared with other template libraries, such as ATL, STL and the data templates. WTL can be used in conjunction with these, or on its own.

In this section we examine features not available with WTL.

WTL does not compete with ATL, STL, Data Templates or COM libraries

Functionality that is provided by other template libraries is not duplicated in WTL. This keeps WTL small and focused, yet developers can access other template libraries when needed.

No Document Support

WTL provides a frame and a view, but no document. Developers coming from MFC wonder why. WTL is concentrating on the user-interface, and the document itself is never seen, so it is considered outside the scope of WTL.

MFC provided binary files that usually were based on structured storage. The latest in document format is basing everything on XML. For Internet content, formats could include XHTML for hypertext, PNG for images, and SVG for vector diagrams and SMIL-Boston for multimedia. PNG is a binary file; all the others are based on XML. Applications that need custom formats should define their own XML schemas and write the appropriate code to load and save XML files based on this schema.

Chapter 16 of "Inside ATL", Sheperd/King, MS-Press, contains an interesting (but short) description of how to manage documents.

No Active Document Support

WTL does not support Active Documents (formerly called OLE Documents) directly. WTL/ATL provides no support for Active Document Servers. ATL does provide some indirect support for Active Document Containment, through the use of Internet Explorer. This could be used in a WTL application, but it certainly does not provide the same level of functionality as MFC's extensive Active Document support. Microsoft has indicated that it currently has no plans to added support to WTL for Active Documents.

However, it should be noted that the popularity of Active Documents is waning. It adds considerably to the complexity of the application. The vast majority of end-users never actually benefit from the functionality as they ignore it.

As we move our data storage formats from binary structured storage files to XML data, this trend will accelerate.

Chapter 15 of "Creating Lightweight Components with ATL", Bates, SAMS, contains a well-designed project which shows Active Document Server and Containment.

One-side note: If planning to become a container, you will have to offer structured storage. The default (provided) implementation sits above binary files. If your data file format is based on XML with your custom tags, then somehow you have to get the data coming from the server in structured storage streams to be converted into data that end up as tags inside your XML file. The way this could be done is to implement a COM component that exposes `ILockBytes`. It might have a method called `WriteAt` that pumps the data to the XML file:

```
HRESULT __stdcall LockBytesObj::WriteAt(
      ULARGE_INTEGER ulOffset, const void *pv,
      ULONG cb,ULONG *pcbWritten){
      *pcbWritten = write-to-xml-file(pv, 1, cb, fp);
      return (S_OK);
}
```

When loading a document, use the Win32 API `StgCreateDocfileOnILockBytes`, and base it one your LockBytes component, rather than the normal binary data file created with `StgCreateDocFile`.

The storage you get back can be passed to other components, which might call:

```
hres = pIStorage->CreateStream(szStreamName,
            STGM_CREATE | STGM_READWRITE | STGM_SHARE_EXCLUSIVE,
            0, 0, &pIStream);
      hres = pIStream->Write(myDataWrite, strlen(myDataWrite)+1,
       &ulLenWritten);
```

No ISAPI Support

WTL does not provide ISAPI functionality. ISAPI is a well-defined API that sits between web servers and extension DLLs that are dynamically loaded to service specialist browser requests. These extension DLLs are written by third-party developers and provide features that are not part of the web server. Their advantage over CGI scripts is that they are only loaded once compared to CGI that requires a new process to be launched for each request.

The ATL Server functionality in Visual C++ version 7 will provide powerful features in the area. Microsoft's MSDN web site has details of this.

No WinInet Support

WTL provides no special WinInet support. Again, this is non-UI, and there is no reason you could not use the WinInet APIs directly.

No Wrappers for Threading or Synchronization

WTL provides no specific wrappers for Win32's thread creation functions or synchronization functions. WTL works extremely well with Win32 threads and synchronization kernel objects and simply needs no additional support. The code generated by the WTL AppWizard Multithreaded SDI option contains an excellent implementation of a multithreaded SDI application – the code related to the threads is actually in the application code, not inside WTL templates.

Rest assured, WTL and threads get on very well together!

No Database Support

WTL does not offer any database support. The VC++ OLE DB Templates offer both consumer and provider templates for data access. These can of course be used from within a WTL application.

MFC offers consumer classes to access existing database tables. It offers no provider functionality. MFC also has the likes of `CRecordView` and `DDX_Field` functions for UI functionality when accessing ODBC databases. WTL does not offer corresponding functionality.

The lack of `CRecordView` is not a problem – indeed, it could be considered a benefit. Many people consider the MFC approach of forcing database record UI into the totally un-related MFC Document/View architecture as less than stellar design. They are

simply different - so let them be different. Very many MFC database applications did not use `CRecordView`.

The lack of `DDX_Field` is more troublesome. The idea behind MFC's `DDX_Field` is that you will have a C++ class for the UI dialog, and a C++ class for the database table. Usually there is a (one-to-one?) mapping from what appears on the dialog to the fields in the database table. It is inefficient to store the data values both in the dialog's C++ class and in the database table's C++ class. MFC's `DDX_Field` eliminates the need for the dialog C++ data-members, and the data is copied directly from the dialog box on screen to the member variables of the C++ database class. Hence it is much more efficient.

WTL does not support this idea of `DDX_Field`. Applications based of WTL, which do not use too many database tables, could simply copy the data from the C++ class representing the dialog to the C++ class representing the database table. Applications that make more substantial use of databases might consider implementing their own version of `DDX_Field`.

Development Issues

To get the most out of WTL there are a number of development issues that need to be well understood.

The Finished Product

The size of applications built with WTL is very small. When you compile the AppWizard generated default SDI app the resulting EXE is only 64K in size. The TinyWTL sample app has extraneous features stripped out (so it does absolutely nothing), but is only 24K in size. A substantial commercial-strength application could be provided in less than 250K. Note that when such a file is compressed, it generally loses two-thirds of its size and hence could be rapidly downloaded over the Internet.

After building the application, you then have to consider how to get it installed on the client's machine. Assuming you are not using the CRT or other libraries such as DirectX, then when you compile a WTL project the resulting EXE or DLL is the only file you will need to deliver to end-users. Hence the size of your installation is quite small. Not only that, it is the only file that can cause problems. The less you deliver, the fewer things that can go wrong. Many applications vendors operate on tight margins and even one support call from an end-user can wipe out the profit on the sale to that customer. WTL applications will have no external dependencies, thus producing savings in this area.

To install the application, you could create an installation script (e.g. based on Windows Installer). For simple app you could get away with just delivering the EXE or DLL. Sometimes you may wish to add entries to the registry. In this case, you can have the end-user execute a command line such as `regsvr32 <your DLL name>` or `<your EXE name> /RegServer`

An alternative would be to make the EXE or DLL auto-sense that it has no registry entries the first time it is run, and to add them at that point.

CRT

The C-Runtime library is a set of C functions built above the Win32 API. It is commonly known as the CRT. The source code for the CRT is provided as part of the Visual C++ distribution (in the src directory). When you examine it you will see that the functions in the CRT are implemented in ANSI C, some call Win32 APIs directly and some call helper functions that in turn call Win32 APIs. The CRT is mostly a collection of functions from the Standard Library in ANSI C (`printf, strtok` etc.), but it is also some Win32-specific functions (the `beginthreadex` CRT function does some CRT initialization and then calls the Win32 function `CreateThread`).

If you program exclusively to the ANSI C Standard library then you code will be portable. There are plenty of multi-million lines of code projects out there that can be equally well compiled on Windows and UNIX and with changes to less than 2% of code it will work just fine. The CRT does offer some extra functionality over the Win32 API in some areas, such as supporting floating-point values in strings. Other benefits of CRT include support for C++ exception handling, STL and the calling of constructors/destructors for global variables.

If you are programming with ATL and WTL, your code will definitely not be portable to UNIX/LINUX platforms, and the question arises, should I link with the CRT? By and large the CRT duplicates functionality already in the Win32 API, so often the answer is no. It is possible to survive without C++ exception handling and the calling of constructors/destructors for global variables. By linking with the CRT, you are making your binary larger. You are also introducing a dependency on an external library (MSVCRT.dll), and multiple versions of this can exist. It is just one extra thing that can cause problems.

The WTL AppWizard produces a project with settings so that the CRT is provided with the debug build (this is needed for the `ATLASSERT` calls), and the CRT is not provided with the release build, as the latter defines the `_ATL_MIN_CRT` macro in the preprocessor. If you are not paying attention here, sometime over the next few months you will develop a project that uses a CRT call, and it will not link in release mode. You will scratch your head, and hopefully come back here and say, aha! To avoid such rigmarole, note now that CRT IS NOT PRESENT IN RELEASE BUILDS BY DEFAULT.

If you do decide to use the CRT, simply remove the `_ATL_MIN_CRT` macro in the preprocessor. If using the CRT, then an interesting point arises with multithreading. There are multiple versions of the CRT – some which are statically bound, and some that are dynamically loaded, and some are for single-thread apps, and some for multi-threaded apps. If you are using the CRT functions from more than one thread, you definitely must use the multithreaded version. It is also imperative that the EXE and all DLLs in the project use the same setting for the CRT. It is to be found in the Project Settings dialog in DevStudio, Code Generation category.

Error Handling

In the past, the recommendation was to examine the return value from every function and carry out remedial action if needed. This often led to code bloating. For every ten

lines of real code you could have forty lines of error checking and correction code and it was often difficult to debug and to follow the logical path through the code.

Then along came C++ exception handling. The functional code was placed within try-catch blocks and when an exception was thrown, either directly within the try-catch block or in a function called from the block, and then the exception handler was activated. This reduced the amount of code in the project and made it clear. However, exceptions are time consuming to handle, do not cross language boundaries and more importantly, require the use of the CRT.

If you are writing code which will run on multiple operating systems it makes sense to use the CRT and its features, including exception handling. However, as discussed earlier, if you are solely targeting Windows, and wish to produce small binaries without external dependencies (e.g. on a particular version of the CRT library - MSVCRT.dll), then you will wish to avoid linking with the CRT, which also means avoiding C++ exceptions. WTL is designed so that the CRT is not needed in release builds – so the obvious question we have to look at is how does WTL handle error situations.

There are two main categories of errors:

- Operational Errors – e.g. hard-disk is full while saving file
- Programming Errors – e.g. divide by zero somewhere

Operational errors will always occur and a well-written application should be able to detect them and respond appropriately – such as informing the user of the problem, suggesting how to fix it (e.g. make space on the disk), and the application should continue running. Programming errors should never occur in a well-written application. If they do occur it is a sign of a defect in the application. Something is seriously wrong. How can an application sensibly handle divide by zero – it is a design defect as user interface code should have only allowed the user enter a number not equal to zero. Such errors should never exist in an application in use by end-users. They will occur in debug builds, when developers are testing their applications. That is expected, but such problems should be eliminated by the time the product is ready to ship.

For operational errors, WTL detects the problems and reports them back to application code, for example as a Boolean return code. This is a sample excerpt from WTL's `CFindFile` class, which calls the Win32 API `FindFirstFile`.

```
BOOL FindFile(LPCTSTR pstrName = NULL){
      Close();
      if(pstrName == NULL)
            pstrName = _T("*.*");
      lstrcpy(m_fd.cFileName, pstrName);
      m_hFind = ::FindFirstFile(pstrName, &m_fd);
      if(m_hFind == INVALID_HANDLE_VALUE)
            return FALSE;
      . . .
```

`CFindFile::FindFile` returns a `BOOL`, which states whether the operation succeeded or not. There can often be problems with pathnames (e.g. network is temporarily down) and applications need to respond appropriately. How this response is

made is left to the application code – WTL just says the error occurred. Note that the last error is often that of the Win32 API that was called, so application code often can get more information from calling `GetLastError` (WTL provides no wrapper for it).

For programming errors, WTL's approach is quite different. The design goal could be stated as: "In debug builds, when we detect programming errors we will invoke an assertion, and we will test the application so well the problems won't occur in release builds". The great benefit of this approach is that the release build is very fast, as it is not clogged up with millions of error-checking lines of code.

`ATLASSERT` statements are liberally scattered through out the WTL source code. `ATLASSERT`s get eliminated during release builds. During debug builds, then evolve into `ASSERTE` calls, which in turn evaluate the Boolean expression passed in as a parameter, and if `FALSE` then it calls `_CrtDbgReport`, which displays a message box with the string-ified version of the expression parameter and breaks in the debugger at the appropriate line.

As an example, the WTL's `CListBox` has this method to insert a string into a listbox:

```
int InsertString(int nIndex, LPCTSTR lpszItem){
      ATLASSERT(::IsWindow(m_hWnd));
      return (int)::SendMessage(m_hWnd,
        LB_INSERTSTRING, nIndex, (LPARAM)lpszItem);
}
```

The window handle data member must be initialized before this call. If not, during the call it invokes an assertion. It is expected that application developers have a comprehensive test plan to cover all the code paths, and coding errors such as calling `CListBox::InsertString` without initializing the `CListBox` will be detected (and fixed) using the debug build, and when finished, shipped as a very fast release build to end-users. That's the theory.

What happens if we … eh … skimped on testing! What happens if an error occurs on a customer site and we are using release builds. WTL carries out no programmer error checking in release builds. The above `CListBox::InsertString` method gets compiled into a straight call to Win32's `SendMessage`. In general, WTL methods that are wrappers just call the underlying Win32 function and WTL methods that perform more elaborate operations work as best they can. If errors occur, then either just the last error value is set, the method returns an error code and the process continues, or the process is terminated.

If it is vital for your application that all error values are checked, you can certainly make the piece of application code which call the likes of `CListBox ::InsertString` itself call `ATLASSERT` before and / or after the method calls or add other error handling code which might even be invoked even during release builds, and handle the error some other way.

The WTL Namespace

The new ISO C++ standard introduced the concept of namespaces. Template and class names all have to be unique with a namespace. In the past there was a single global namespace within a project. This sometimes caused problems as libraries from different vendors often has name clashes. What ISO C++ added was the idea of creating extra namespaces, in addition to the global namespace.

If we have a C++ namespace for WTL and a namespace for ATL and yet another namespace for a third-party library, then all three can happily be used together safe in the knowledge that even if classes from the different libraries use the same names there will be no problems.

Every WTL header file starts with:

```
namespace WTL
{
```

and ends with:

```
}; //namespace WTL
```

Unlike a class definition, which can only be declared once, a namespace can be partially declared in multiple header files, and all they entries become part of its namespace.

When distinguishing between an MFC class and a WTL class of the same name, we can use the full WTL class name, which includes its namespace name. So we can talk about `WTL::CRect`. The namespace is used just for naming purposes – it has no other programmatic significance.

Within your source code, you will have to decide whether you wish to use the long-form, `WTL::CRect` or the short-form, `CRect`. If you are only using WTL, then it is likely adding `WTL::` every time you use a template/class name will be tedious, so you will wish to avoid it. You can do this by adding the "`using namespace WTL;`" line to your source code. However, if using multiple libraries, and there is a possibility of name clashes, then you will have to use the full name.

These lines are at the bottom of the WTL header file, atlapp.h:

```
 #ifndef _WTL_NO_AUTOMATIC_NAMESPACE
using namespace WTL;
#endif //!_WTL_NO_AUTOMATIC_NAMESPACE
```

In the project generated by the WTL AppWizard, `_WTL_NO_AUTOMATIC_NAMESPACE` is not defined, so the using entry is in force, and the generated code does not use `WTL::` prefix. If you want to take this out, just define or set it in the preprocessor definitions.

Template/Class/Method Naming

The naming conventions of WTL deliberately follow those of MFC. This makes sense, as both are based above the Win32 API, and most WTL developers will be coming from a MFC background.

WTL has class names such as `CEdit, CStatic` and `CDC`. Handlers for Windows messages have names such as `OnInitDialog` and `OnOK`. Data exchange is carried out using DDX, and a method called `DoDataExchange`.

Where there is a difference between MFC and Win32 naming, WTL follows Win32, which is sensible. Therefore for the Win32 UpDown control, WTL has a template called `CUpDownCtrlT` whereas MFC has a class called `CSpinButtonCtrl`.

Versions of Windows

One appealing aspect of WTL is its full support of all new UI features in Windows 2000 and Windows Me. WTL can also be used with all older versions of Windows, and the level of functionality it exposes can be controlled via a `#define` of `WINVER`.

Complete list of Macros used in WTL

The implementation of the WTL functionality is controlled from a variety of #defines. They are listed in the following table, along with the default and what impact they have. Note that some of them are preceded and succeeded by one or two underscores – and you better make sure you get it right!

Some of the macros are Win32-based, some are ATL-based and some are new to WTL.

Win32 Macros Which Impact WTL

Macro	Default	Impact
`_WIN64`	Not defined	If defined, the 64-bit friendly version of `_SettingChangeDlgProc` is used; if not defined, the 64/32-bit friendly functions `GetWindowLongPtr` etc. are mapped to the 32-bit only `GetWindowLong` etc.
`_WIN32_WINNT`	Not defined	If defined as `0x400`, assume that the target platform is Windows NT 4 or later; if defined as `05x00`, assume target is Windows 2000 or later
`WINVER`	Set to 0x0400 in stdafx.h	If defined as `0x400`, assume the target platform is one of Windows 95 or Windows NT 4 or later; if defined as `0x0500`, assume the target platform is Windows 98, Windows 2000 or later
`_WIN32_IE`	Set to 0x0400 in stdafx.h	Defines minimum version of Internet Explorer which is installed
`_UNICODE`	Not defined	If defined, `TCHAR`s expect to `wchar_t` etc. and use W versions of OS APIs, if not defined, `TCHAR` expands to `char` etc., and use A version of OS APIs

ATL Macros Which Impact WTL

Macro	Default	Impact
`_ATL_MIN_CRT`	Depends	States whether the C-Runtime Library should be included or not. In the WTL AppWizard generated project settings, it is defined for release builds, and it is not defined for debug builds. When `_ATL_MIN_CRT` is defined, `_ATL_USE_DDX_FLOAT` may not be defined

WTL Macros That You Can Define

Macro	Default	Impact
`_WTL_NO_AUTOMATIC_NAMESPACE`	Not defined	If not defined, then the statement `using namespace WTL;` is added, other wise each WTL construct needs to be preceded with `WTL::`
`_WTL_NO_CSTRING`	Not defined	If defined, states that you do not wish to use `CStrings` anywhere in the app
`_CMDBAR_EXTRA_TRACE`	Not defined	If defined, then extra debug output is supplied for command bars
`_ATL_USE_DDX_FLOAT`	Not defined	If defined, then no floating point DDX support is available. If true then it is available – and it will use the CRT (so this cannot be defined at the same time `_ATL_MIN_CRT` is defined)
`_ATL_NO_REBAR_SUPPORT`	Not defined	If true, do not add support for the rebar control
`_ATL_NO_MSIMG`	Not defined	If not defined, then link with "msimg32.lib" and add these methods to `WTL::CDC`: `AlphaBlend, TransparentBlt` and `GradientFill`
`_ATL_NO_OPENGL`	Not defined	If not defined, then link with "opengl32.lib" and add 15 OpenGL wrapper methods to `WTL:CDC`
`_WTL_NO_WTYPES`	Not defined	If defined, then provide the WTL classes `CSize`, `CPoint` & `CRect`
`_ATL_USE_NEW_PRINTER_INFO`	Not defined	If defined, add support for printer templates based on `PRINTER_INFO_8` and `PRINTER_INFO_9`
`_RICHEDIT_VER`	Set to 0x0100 in stdafx.h	Specifies the level of rich edit support

RICHEDIT	Not defined	If defined, add declarations of CRichEditFontDialogImpl and CRichEditFontDialog, which provide font selection for the Rich Edit control
OEMRESOURCE	Not defined	If defined, add implementations of AtlLoadSysBitmap and AtlLoadSysBitmapImage which call Win32's LoadBitmap and LoadImage with a NULL first parameter, which means they use the predefined Win32 bitmaps
_ATL_NO_COM	Not defined	If not defined, then for many WTL classes, where there is a member function that takes a string parameter, add an overridden member that takes a BSTR. Also, add the member functions AllocSysString and SetSysString to WTL's CString
_OLEAUTO_H_	Not defined	This macro is only evaluated if _ATL_NO_COM is not defined If _OLEAUTO_H_ is defined, then add a member function GetTextBSTR to WTL's CListBoxT, and overridden member functions GetText to CTreeItem and GetTextFace to CDC, both of which retrieve a BSTR

WTL Macros That You May Use But Should Not Define

Macro	Default	Impact
_WTL_VER	0x0300	Version of WTL. It is defined in the WTL header file AtlApp.h, but is not used anywhere. Future versions might need it.
__ATLSTR_H__	Depends	States that WTL's CStrings may be used. By default, this is defined if both atlmisc.h is included and _WTL_NO_CSTRING is not defined

Debugging with WTL

The development environment for Visual C++ includes many excellent productivity features. (It is unfortunate that most developers have not taken the time to explore them).

One of every developer's favorites is while executing an application under the debugger, if you float the cursor above a variable name or look at it in the data window, your see the variable's value. Visual C++ automatically knows how to handle simple variables such as integers and character pointers. But how should it display more complex data structures and classes?

When you install Visual C++, a text file called AUTOEXP.DAT is placed under `<INSTALL_LOCATION>\Common\MSDev98\Bin\`. This file contains instructions, using simple text-based formatting rules, telling the Visual C++ environment how to render specific data structures. The formatting rules are explained at the top of the file.

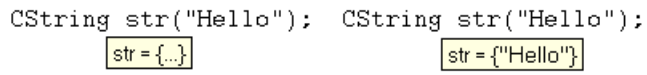
By default, AUTOEXP.DAT contains entries for common Win32 structures, such as `POINT` and `RECT` (but not `SIZE`), and many common MFC classes.

There are no entries for WTL. But that is not a problem, as we can add our own. Open up the file in Notepad and pop the following entries at the bottom of the file.

```
WTL::CString =<m_pchData,st>
WTL::CSize =cx=<cx> cy=<cy>
WTL::CPoint =x=<x> y=<y>
WTL::CRect =top=<top> bottom=<bottom> left=<left> right=<right>
WTL::CDCT<*> =hDC=<m_hDC>
WTL::CClientDC=hDC=<m_hDC> hWnd = <m_hWnd>
WTL::CWindowDC=hDC=<m_hDC> hWnd = <m_hWnd>
WTL::CPaintDC =hDC=<m_hDC> hWnd = <m_hWnd>
WTL::CServerAppModule =m_hEventShutdown=<m_hEventShutdown>
WTL::CPenT<*> =m_hPen=<m_hPen>
WTL::CBrushT<*> = m_hBrush=<m_hBrush>
WTL::CFontT<*> = m_hFont=<m_hFont>
WTL::CBitmapT<*> = m_hBitmap=<m_hBitmap>
WTL::CPaletteT<*> = m_hPalette=<m_hPalette>
WTL::CRgnT<*> = m_hRgn=<m_hRgn>
ATL::CWindowImpl<*>=<,t> hWnd=<m_hWnd>
ATL::CWindow=<,t> hWnd=<m_hWnd>
```

Note that the namespace specification for the last two entries is `ATL::` and not `WTL::`.

Structures and classes that do not have entries in AUTOEXP.DAT are rendered using "…", which is not very informative. The following two screen dumps are from a WTL application running under DevStudio's Debugger, and the cursor is floating over a WTL variable of type `CString`. This is how a WTL `CString` is displayed without (left picture) and with (right picture) the AUTOEXP.DAT entries shown above:



The AUTOEXP.DAT entries are of the format:

```
type=[text]<member[,format]>
```

`Type` indicates the data-type (including the namespace name). When the debugger detects a variable of this type, it renders a string based on the rest of the entry. The `[text]` field is optionally, but if present is rendered as is. The `<member>` field states which data member of the class or structure should be displayed, and the `format` field states how it should be displayed.

So the entry . . .:

```
WTL::CString =<m_pchData,st>
```

. . . says please display a `WTL::CString` as the `m_pchData` field, and display it as a `NULL` terminated string, and the entry. . :

```
WTL::CBrushT<*> = m_hBrush=<m_hBrush>
```

. . . says please display any class based on the `WTL::CBrushT` template as the text string "`m_hBrush=`" and the actual data value of the `m_hBrush` data member (in the default integer format).

If an entry does not exist for a class, and if that class is derived from another class or a structure, then an attempt is made to find an entry for the base class or structure. Because of this, even without the entries for `WTL::CPoint` and `WTL::CRect`, you will see tooltips for them, because they are derived from the Win32 data types, `POINT` and `RECT`, and entries for these structures do exist in the default AUTOEXP.DAT. As there is no entry for Win32's `SIZE` structure, then a tooltip is not displayed for `WTL::CSize`.

To ensure the right match, it is important that the namespace name is correct and you are clear about whether you are matching a class or a template. For a template you must use <*>, and for a class, you may not use it. Note that `CWindow` is a class, not a template, and hence is entered without the <*>. All the standard and common control wrappers are based on `CWindow`.

In addition to seeing the actual value of the variable, this feature is also helpful to know it if it `NULL` (e.g. non-initialized), or some value that is impossible (e.g. illegal memory address), thus indicating memory problems.

Detailed Comparison between MFC and WTL

Most developers starting with WTL will have already developed MFC applications. They will have an excellent knowledge of MFC and have used it in quite substantial projects.

They will wish in some cases to leverage that knowledge to quickly develop WTL applications and sometimes might even need to port from MFC to WTL.

To assist these, here we examine how the functionality of each of the MFC classes can be provided in a WTL application. We also look at what is in WTL that is not supported inside MFC.

We follow the categories in the MFC hierarchy charts.

Application Architecture

- `CCmdTarget` – command routing is done in ATL message maps
- `CWinThread` – use Win32 Multithreading APIs
- `CWinApp` – `CAppModule` or `CServerAppModule`
- `COleControlModule` – ATL ActiveX control infrastructure

- `CDocTemplate, CSingleDocTemplate, CMultiDocTemplate` – WTL has no document support – use XML code
- `COleObjectFactory` – Use ATL's wide range of class factories
- `COleTemplateServer / COleMessageFilter` – WTL has no Active Document support
- `COleDataSource, COleDropSource, COleDropTarget` – WTL has no universal data transfer support – could implement directly
- `CConnectionPoint` – Use ATL's connection point templates and proxy generator
- `CDocument / CHtmlEditDoc / CRichEditDoc` – WTL has no document support – you will need to manually implement these features
- `COleDocument / COleLinkingDoc / COleServerDoc, CDocItem/ COleClientItem/ COleDocObjectItem / COleServerItem / CDocObjectServerItem / CDocObjectServer` – WTL does not support Active Documents
- `CRichEditCntrItem` – WTL provides a template for the rich edit control, but has no support from Active Document server/containment for it, as this MFC class provides
- `COleControlContainer / COleControlSite` – Use ATL's ActiveX control containment functionality (CAxWindow etc.)

Window Support

- `CWnd` – Use ATL's `CWindow, CWindowImpl` and `CContainedWindow`

Frame Window

- `CFrameWnd` – Use WTL's `CFrameWindowImpl`
- `CDockFrameWnd` – WTL provides no docking support
- `CMDIChildWnd` – Use WTL's `CMDIChildWindowImpl`
- `CMDIFrameWnd` – Use WTL's `CMDIFrameWindowImpl`
- `CMDIDockSiteFrame` – WTL provides no docking support
- `COleIPFrameWnd` – WTL provides no in-place activation support
- `CSplitterWnd` – Use WTL's `CSplitterImpl, CSplitterWindowImpl` and `CSplitterWindowT`

Question - CTabMDIChildWnd and CTabFrameWnd are listed in the MFC hierarchy chart, but no doc provided – Any ideas what they are? VC++ v6.1

Property Sheets

- `CPropertySheet` – Use WTL's `CPropertySheetWindow`, `CPropertySheetImpl` and `CPropertySheet`

Dialog Boxes

- `CDialog` – Use WTL's `CDialogImpl`
- `CCommonDialog` – WTL's `CCommonDialogImplBase`
- `CColorDialog` – WTL's `CColorDialogImpl` and `CColorDialog`
- `CFileDialog` – WTL's `CFileDialogImpl` and `CFileDialog`
- `CFindReplaceDialog` – WTL's `CFindReplaceDialog`
- `CFontDialog` – `CFontDialogImpl` and `CFontDialog`
- `COleDialog / COleBusyDialog / COleChangeIconDialog / COleChangeSourceDialog / COleConvertDialog / COleInsertDialog / COleLinksDialog / COleUpdateDialog / COlePasteSpecialDialog / COlePropertiesDialog` – WTL offers no Active Document support at all
- `CPageSetupDialog` – WTL's `CPageSetupDialogImpl` and `CPageSetupDialog`
- `CPrintDialog` – WTL's `CPrintDialogImpl` and `CPrintDialog`
- `COlePropertyPage` – ATL's ActiveX control functionality provides similar support
- `CPropertyPage` – WTL's `CPropertyPageWindow`, `CPropertyPageImpl` and `CPropertyPage`

Question - CDHtmlSinkHandler / CDHtmlEventSink / CDHtmlEventHandler / CDHtmlHostHandler/CDHtmlDialog /CMultiPageDHtmlDialog are listed in the MFC hierarchy chart but no doc provided – Any ideas what they are? VC++ v6.1

Views

- `CView` – WTL offers no special class for the view. The WTL AppWizard generated application code does contain a view class derived from `CWindowImpl` –this simplifies the design immensely
- `CScrollView` – Use WTL's `CScrollImpl, CScrollWindowImpl, CMapScrollImpl, CMapScrollWIndowImpl` and `CFSBWindowT`
- `COleDBRecordView/CRecordView` – WTL's offers no corresponding functionality – which is a good, as trying to force database editing into MFC's Document/View infrastructure was not a good approach – use WTL's `CDialogImpl< MYCLASS >` [form view] when needed

- `CCtrlView` – In MFC, this class derives from `CView` and is the parent of other views based on controls, such `CEditView` and `CTreeView`. WTL does not provide such an intermediary class

WTL offers no specific templates for views based controls, as it has no need for them. It does offer a complete range – they are to be implemented by deriving from the `CWindowImpl` template based on the control in question (This is also supported in step 2 in the WTL AppWizard).

- `CEditView –CWindowImpl< MYCLASS, CEdit>`
- `CListView - CWindowImpl< MYCLASS, CListViewCtrl>`
- `CRichEditView - CWindowImpl< MYCLASS, CRichEditCtrl >`
- `CTreeView - CWindowImpl< MYCLASS, CTreeViewCtrl>`
- `CFormView - CDialogImpl< MYCLASS >`
- `CHtmlEditView` – This class is listed in the MFC hierarchy chart, along with `CHtmlEditBase / CHtmlEditCtrl`, but no class documentation on the MSDN and no implementation in the VC++ installation is available – we assume it will be provided in a future version – and will be based on MSHTML Editing Platform. For more details. See:
  http://msdn.microsoft.com/workshop/browser/mshtmleditplaf.asp
  WTL provides no functionality to edit HTML, but this could be manually implemented
- `CHtmlView – CWindowImpl< MYCLASS, CAxWindow>;` The ATL `CAxWindow` uses the Web Browser control internally

Controls

Both MFC and WTL offer a complete range of wrappers for Window controls. WTL's controls are templates (names ends in T) and a default implementation based on `CWindow` is provided (without the T).

Note that WTL always uses the correct OS name (e.g. UpDown control, TrackBar control), whereas MFC sometimes invents its own (e.g. SpinButtonCtrl or SliderCtrl)

- `CAnimateCtrl - CAnimateCtrlT<CWindow> CAnimateCtrl;`
- `CButton - CButtonT<CWindow> CButton`
- `CBitmapButton - CBitmapButton : public CBitmapButtonImpl`
- `CComboBox - CComboBoxT<CWindow> CComboBox`
- `CComboBoxEx - CComboBoxExT<CWindow> CComboBoxEx`

- `CDataTimeCtrl - CDateTimePickerCtrlT<CWindow>CDateTimePickerCtrl`
- `CEdit - CEditT<CWindow> CEdit` [Note: WTL also offers `CEditCommands`, to assist with the handling of common editing commands such as clear/copy/page/undo]
- `CHeaderCtrl - CHeaderCtrlT<CWindow> CHeaderCtrl`
- `CHotKeyCtrl - CHotKeyCtrlT<CWindow> CHotKeyCtrl`
- `CIPAddressCtrl - CIPAddressCtrlT<CWindow> CIPAddressCtrl`
- `CListBox - CListBoxT<CWindow> CListBox`
- `CCheckListBox – CCheckListViewCtrlImpl/CCheckListViewCtrl`
- `CDragListBox - CDragListBoxT<CWindow> CDragListBox;` [Note: WTL also offers `CDragListNotifyImpl` to help with notifications]
- `CListCtrl - CListViewCtrlT<CWindow> CListViewCtrl`
- `CMonthCalCtrl - CMonthCalendarCtrlT<CWindow> CMonthCalendarCtrl`
- `CProgressCtrl - CProgressBarCtrlT<CWindow> CProgressBarCtrl`
- `CReBarCtrl - CReBarCtrlT<CWindow> CReBarCtrl` [Note that WTL offers extensive commandbar support, based on the rebar]
- `CRichEditCtrl - CRichEditCtrlT<CWindow> CRichEditCtrl` [Note: WTL also offers `CRichEditCommands` to help with editing command processing
- `CScrollBar - CScrollBarT<CWindow> CScrollBar`
- `CSliderCtrl - CTrackBarCtrlT<CWindow> CTrackBarCtrl`
- `CSpinButtonCtrl - CUpDownCtrlT<CWindow> CUpDownCtrl`
- `CStatic - CStaticT<CWindow> CStatic`
- `CStatusBarCtrl - CStatusBarCtrlT<CWindow> CStatusBarCtrl` [Note that WTL offers substantial extra functionality in `CMultiPaneStatusBarCtrlImpl / CMultiPaneStatusBarCtrl`]
- `CTabCtrl - CTabCtrlT<CWindow> CTabCtrl`

- `CToolBarCtrl` - `CToolBarCtrlT<CWindow> CToolBarCtrl`
- `CToolTipCtrl` - `CToolTipCtrlT<CWindow> CToolTipCtrl` [Note that WTL also offers `CToolInfo`, to help with the initialization of Win32's `TOOLINFO` structure]
- `CTreeCtrl` - `CTreeViewCtrlT<CWindow> CTreeViewCtrl` and `CTreeViewCtrlExT:public CTreeViewCtrlT< TBase >` and `CTreeItem`
- `CHtmlEditBase / CHtmlEditCtrl` – See note regarding these classes in the earlier discussion of MFC's `CHtmlEditView`
- `COleControl` – This is MFC's base class for ActiveX controls – ATL offers equivalent (really, far superior) functionality

Exceptions

- CException, CArchiveException, CDaoException, CDBException, CFileException, CInternetException, CMemoryException, CNotSupportException, COleException, CResourceException, CUserException – WTL has no exception classes.

File Services

- `CFile, CMemFile, CSharedFile, COleStreamFile, CMonikerFile, CAsyncMonikerFile, CDataPathProperty, CCachedDataPathProperty, CSocketFIle, CInternetFIle, CGopherFIle, CHttpFile` – WTL offers no handling file support
- `CRecentFileList` – Use WTL's `CRecentDocumentList`

Graphical Drawing

- `CDC` – WTL's `CDC`
- `CClientDC` – WTL's `CClientDC`
- `CMetaFileDC` – WTL's `CEnhMetaFileDC` [Note that WTL also offers a number of additional classes for enhanced metafiles]
- `CPaintDC` – WTL's `CPaintDC`
- `CWindowDC` – WTL's `CWindowDC`

Control Support

- `CDockState` – WTL offers no support for docking
- `CImageList` – WTL's `CImageList`

Graphical Drawing Objects

- `CGdiObject` – WTL has no common base for its GDI templates
- `CBitmap` – WTL's `CBitmap`
- `CBrush` – WTL's `CBrush`
- `CFont` – WTL's `CFont`
- `CPalette` – WTL's `CPalette`
- `CPen` – WTL's `CPen`
- `CRgn` – WTL's `CRgn`

Menu

- `CMenu` – WTL's `CMenu` [WTL also offers `CMenuItemInfo`, a helper to access Win32's `MENUINFO` structure]

Command Line

- `CCommandLineInfo` – WTL's `CServerAppModule` has the `FindOneOf` and `ParseCommandLine` helper functions – [In the WTL AppWizard, step 1, if *Act as a COM Server* is selected, the generated code uses `CServerAppModule`, otherwise it uses `CAppModule`]

MFC's Arrays/Lists/Maps/Typed Template Collections

WTL has no container classes. You may either use the excellent STL (many of whose features are dependent on the CRT) or ATL offers a limited range of collection classes that are not dependent on the CRT.

MFC's ODBC/DAO/Synchronization/Sockets/Internet Services

WTL offers no equivalent support for MFC classes in these areas.

In WTL applications, MFC's ODBC/DAO functionality can be replaced with the VC++ OLE DB Consumer templates. Synchronization can be provided by using the relevant Win32 APIs directly. Sockets functionality can be provided by direct calls to WinSock2. Internet services can be provided by direct calls to the WinInet APIs.

Internet Server API (ISAPI)

MFC offers `CHttpArgList`, `CHtmlStream`, `CHttpFilter`, `CHttpFilterControl`, `CHttpServer`, `CHttpServerContext`, `CInternetSession`, `CInternetConnection`, `CFtpConnection`, `CGopherConnection`, `CHttpConnection`, `CFtpFindFile`, `CGopherFindFile` and `CGopherLocator`.

ATL will cover this area comprehensively in Visual C++ v7, in a technology to be known as ATL Server. For more details, see
http://msdn.microsoft.com/vstudio/nextgen/default.asp

Run-time Object Model Support

- `CArchive` – This is used to serialize data, which might ultimately end up in a document or sent over a socket – WTL offers no similar functionality
- `CDumpContext` – ATL/WTL does not fully cover this, but some of the debugging facilities within ATL partially help – for all the features, you would have to implement a dump function for each class, and walk some collection of class instances and call the function
- `CRuntimeClass` – Use ISO C++ RTTI if needed – but in general it is considered bad practice to use either of these, as they is costly in terms of performance/size and can cause problems in future (if client code has some big switch doing class specific processing, what happens if later a new class is added – then client has to be change – (class specific code should be inside the class!)

Simple Value Types

- `CPoint` – WTL's `CPoint`
- `CRect` – WTL's `CRect`
- `CSize` – WTL's `CSize`
- `CString` – WTL's `CString`
- `CTime` – This is MFC's wrapper for absolute time, stored as `time_t` - WTL offers no similar class
- `CTimeSpan` - This is MFC's wrapper for the interval between times, stored as `time_t` - WTL offers no similar class

Structures

- `CCreateContext` – MFC uses this when creating frame windows and views for a document. WTL provides no document support

Question - CHttpArg is listed in the MFC Hierarchy Chart but no documentation and no implementation is provided – Any ideas what it is? VC++ v6.1

- `CMemoryState` – This is used for debugging memory leaks – WTL/ATL provides no similar functionality, apart from ATL's functionality which traces interface reference counting
- `COleSafeArray` – WTL offers no SafeArray support – one can use the Win32 SafeArray APIs directly [Should this not be part of Automation Types section in the MFC hierarchy?]
- `CPrintInfo` – WTL's `CPrintJob`

Support Classes

- `CCmdUI` – WTL's `CUpdateUI`
- `COleCmdUI` – WTL provides no Active Document Support
- `CDaoFieldExchange` – WTL/ATL provides no DAO support – DAO is a legacy API, and for new projects you will be much better off to using VC++ OLE DB Data Consumer templates, which can be used inside WTL applications
- `CDataExchange` – WTL's `CWinDataExchange`
- `CDbVariant` – This is a MFC wrapper for `VARIANT`s that is not based on OLE – useful for the MFC ODBC data access – For WTL, use the OLE DB data consumer templates
- `CFieldExchange` - WTL/ATL provides no ODBC support – ODBC is a legacy API, and for new projects you will be much better off using VC++ OLE DB Data Consumer Templates, which can be used inside WTL applications

Question - CImage is listed in the MFC Hierarchy Chart but no documentation and no implementation is provided – Any ideas what it is? VC++ v6.1

- `COccManager` – This is a MFC helper class which provides ActiveX control Containment – Use ATL's `CAxWindow`
- `COleDataObject` – Use ATL's `IDataObjectImpl`
- `COleDispatchDriver` – Use ATL's `CComDispatchDriver`
- `CPropExchange` – Use ATL's Active Control properties functionality
- `CRectTracker` – WTL offers no similar class
- `CWaitCursor` – WTL's `CWaitCursor`

Question – CWinDHtmlDDX & CDHtmlDDX are listed in the MFC Hierarchy Chart but no documentation and no implementation is provided – Any ideas what they are? VC++ v6.1

OLE Type Wrappers

- `CFontHolder` – This is a wrapper for Platform SDK `IFontDisp` interface returned as the Font stock property for ActiveX controls. WTL/ATL provides no similar support – you can simply use `IFontDisp` (and `IFont`) directly
- `CPictureHolder` - This is a wrapper for Platform SDK `IPictureDisp` interface returned as the `MouseIcon` and `Picture` stock properties for ActiveX controls. WTL/ATL provides no similar support – you can simply use `IPictureDisp` (and `IPicture`) directly

Automation Types

- `COleCurrency / COleDateTime / COleDateTimeSpan` – These are wrappers for various fields in Automation's `VARIANT` structure. WTL offers no similar support – you can use ATL's `CComVariant` or VC++'s native `_variant_`
- `COleVariant` – `CComVariant` or VC++ native `_variant_`

What is in WTL but not MFC

Unlike WTL, MFC does not provide wrappers for the folder dialog or the rich edit font dialog (WTL provides `CFolderDialogImpl` / `CFolderDialog` and `CRichEditFontDialog` / `CRichEditFontDialogImpl`).

MFC does not support equivalents of WTL's `CEditCommands` or `CRichEditCommands`.

MFC does support menubars and toolbars, but does not provide the extra functionality of WTL's command bars.

MFC does not provide a wrapper for `HTREEITEM`, whereas WTL provides `CTreeItem`.

WTL provides excellent enhanced metafile support, beyond what MFC offers.

WTL offers these additional control classes, which have no direct equivalent in MFC:

- `CPageCtrl`
- `CFlatScrollBarImpl`
- `CCustomDraw`
- `CHyperLink`

WTL's CString

To begin to get a feel for WTL's functionality, we will here examine one of its classes – `CString`.

The WTL `CString` class manages a character buffer. It provides virtually every conceivable operation you would like to carry out on a string. It offers all the functionality of MFC's `CString` class – it is such a perfect emulation of MFC's `CString` that their class documentation is interchangeable! Every method in MFC's `CString` is in WTL's `CString`.

A string in WTL's `CString` is stored in a contiguous array of characters. WTL supports 16-bit (UNICODE) and 8-bit characters (ASCII) and a mixture (MBCS). `CString` offers a number of constructors that takes as the input parameter strings in different formats.

```
CString();
```

```
CString(const CString& stringSrc);
CString(TCHAR ch, int nRepeat = 1);
CString(LPCSTR lpsz);
CString(LPCWSTR lpsz);
CString(LPCTSTR lpch, int nLength);
CString(const unsigned char* psz);
```

The internal buffer will automatically grow and contract as needed. Member functions are provided to get the length of the string, to determine it is `NULL` and to empty it.

```
int GetLength() const;
BOOL IsEmpty() const;
void Empty();
```

`CString`s are reference-counted, so that when passing strings around the reference count is incremented and the entire string is not copied. Only if a string is about to be changed will a separate copy of the shared string data be made. If a `CString` is going to be used from multiple threads, and that use includes writing from at least one thread, then care must be taken with synchronization.

Member functions are provides to access the `CString` as if it were a character array. The `LPCTSTR()` operator means a `CString` may be passed in anywhere a `LPCTSTR` is expected.

```
TCHAR GetAt(int nIndex) const;
TCHAR operator[](int nIndex) const;
void SetAt(int nIndex, TCHAR ch);
operator LPCTSTR() const;
```

A complete range of assignment & concatenation operators is supported.

Member functions are provided for string comparisons, extractions, conversions and trimming.

```
int Compare(LPCTSTR lpsz) const;
int CompareNoCase(LPCTSTR lpsz) const;
int Collate(LPCTSTR lpsz) const;
CString Mid(int nFirst, int nCount) const;
CString Mid(int nFirst) const;
CString Left(int nCount) const;
CString Right(int nCount) const;
CString SpanIncluding(LPCTSTR lpszCharSet) const;
CString SpanExcluding(LPCTSTR lpszCharSet) const;
void MakeUpper();
void MakeLower();
void MakeReverse();
void TrimRight();
void TrimLeft();
void AnsiToOem();
void OemToAnsi();
```

Member functions are provided for string replacement, insertions & appending.

```
int Replace(TCHAR chOld, TCHAR chNew);
int Insert(int nIndex, TCHAR ch);
int Insert(int nIndex, LPCTSTR pstr);
```

```
int Delete(int nIndex, int nCount = 1);
int Find(TCHAR ch) const;
int ReverseFind(TCHAR ch) const;
int FindOneOf(LPCTSTR lpszCharSet) const;
int Find(LPCTSTR lpszSub) const;
const CString& Append(int n)
void __cdecl Format(LPCTSTR lpszFormat, ...);
void __cdecl Format(UINT nFormatID, ...);
BOOL __cdecl FormatMessage(LPCTSTR lpszFormat, ...);
BOOL __cdecl FormatMessage(UINT nFormatID, ...);
```

WTL's `CString` supports loading strings from a resource file:

```
// Windows support
BOOL LoadString(UINT nID);
```

It also supports Automation `BSTR`s.

```
BSTR AllocSysString() const;
BSTR SetSysString(BSTR* pbstr) const;
```

These methods are provided to allow access to the data for direct manipulation

```
LPTSTR GetBuffer(int nMinBufLength);
void ReleaseBuffer(int nNewLength = -1);
LPTSTR GetBufferSetLength(int nNewLength);
void FreeExtra();
```

After calling `GetBuffer`, and changing the data, it is important to later call `ReleaseBuffer`, so that `CString` again takes responsibility for the buffer.

To lock and unlock the buffer for reference counting uses these methods:

```
LPTSTR LockBuffer();
void UnlockBuffer();
```

From just examining this one WTL class, it should become clear that when WTL provides support for a feature, it is very comprehensive.

Chapter 5
The WTL AppWizard

Objectives

The objectives of this chapter are to:

- Use the WTL AppWizard to create all support application types
- Review the generated build settings
- Examine the application features available in Step 1 of the AppWizard – COM Server and ActiveX hosting
- Examine the UI features available in Step 2
- Conduct a code walkthrough of all the generated source code

Just Say "HelloWorld"

In the time-honored Kernighan & Ritchie tradition of computerdom, we had better start our exploration of WTL with the simplest HelloWorld application we can create. We could either display the text in a dialog box or in an SDI or MDI application window. We could display it as ordinary text with a view, or a listbox, richeditbox, treeview, listview or each a HTML page. It is such a difficult design decision; let's try them all!

Naturally we do not wish to type (that's too much effort) so we will use the WTL AppWizard and hopefully if it is any good it won't have too much effort. Of course we wish to examine how to use the AppWizard, but it is very simple to use and only contains two steps – it is almost trivial, but the real magic is what it produces. Our goal here is to comprehensively understand the generated application code. After all, we will use the AppWizard for two minutes at the beginning or a project, but have to spend six months or more building our application above the foundation of the code it generates.

Modal Dialog-Based Application

To use the WTL AppWizard to create a dialog-based application, start DevStudio, select *File/New*, select the ATL/WTL AppWizard, and the following first Wizard step will appear. As we wish to create a dialog-based app, select *Dialog-Based* and select *Modal* (we will examine modeless dialogs shortly).

As we have selected *Dialog-Based*, all step 2 options are disabled (as they relate to standard application windows), so we can select finish at the bottom of step 1. Then we use the ResourceView to edit the dialog `IDD_MAINDLG` and add a static control with the text *"HelloWorld"*. Compile and run.

The wizard-weary gray-haired Visual C++ developer will recall his/her early MFC days and suspiciously think, "WTL is easy, but what is really happening – with MFC there was a million lines of code generated and it was quite difficult to understand".

With WTL, things are actually different, and somewhat easier.

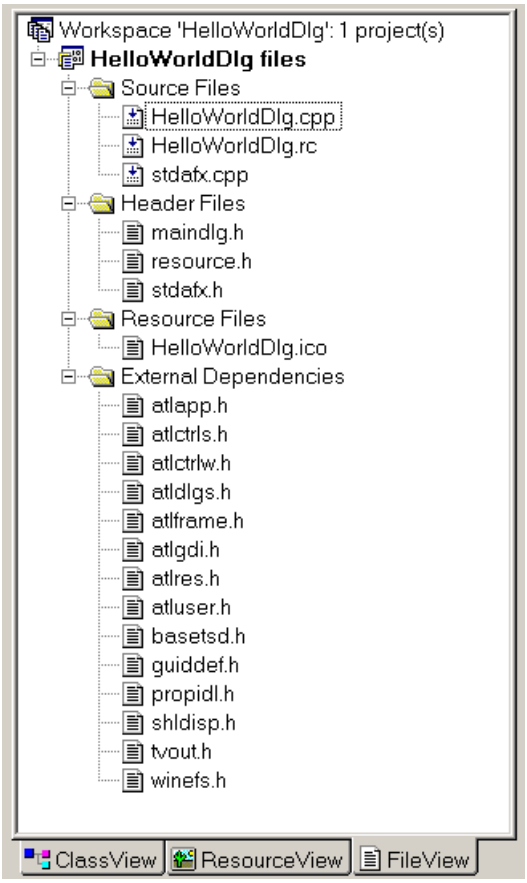
The *External Dependencies* are based directly on the WTL header files. There is no WTL DLL to use – only the header files are needed. The resources are in a file called <projectname>.rc and there is an icon <projectname>.ico. Note that the icon used for the top-left corner defaults to *"ATL"*.

There are three header files, stdafx.h, resources.h and maindlg.h. All of which are quite small. Stdafx.h contains these `#defines` and `#includes`:

```
#define WINVER          0x0400
#define _WIN32_IE       0x0400
#define _RICHEDIT_VER   0x0100
#include <atlbase.h>
#include <atlapp.h>
extern CAppModule _Module;
#include <atlwin.h>
```

Resources.h contains defines for resource Ids, such as:

```
#define IDD_ABOUTBOX    100
#define IDR_MAINFRAME   128
#define IDD_MAINDLG     129
```



Maindlg.h contains the implementation of the `CMainDlg` class. It is the core functionality of this application. It derives from the ATL Windowing class `CDialogImpl`. It contains an ATL Windowing message map, which maps buttons Ids to member functions. It also maps `WM_INITDIALOG` to `OnInitDialog` (all very familiar to MFC developers!).

```
class CMainDlg : public CDialogImpl<CMainDlg> {
public:
      enum { IDD = IDD_MAINDLG };
      BEGIN_MSG_MAP(CMainDlg)
            MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
            COMMAND_ID_HANDLER(ID_APP_ABOUT, OnAppAbout)
            COMMAND_ID_HANDLER(IDOK, OnOK)
            COMMAND_ID_HANDLER(IDCANCEL, OnCancel)
      END_MSG_MAP()
      LRESULT OnInitDialog(UINT /*uMsg*/, WPARAM /*wParam*/,
       LPARAM /*lParam*/, BOOL& /*bHandled*/){
            // center the dialog and set icons
       . . .
            return TRUE;
      }
```

```
      LRESULT OnAppAbout(WORD /*wNotifyCode*/, WORD /*wID*/,
             HWND /*hWndCtl*/, BOOL& /*bHandled*/){
            CSimpleDialog<IDD_ABOUTBOX, FALSE> dlg;
            dlg.DoModal();
            return 0;
      }
      LRESULT OnOK(WORD /*wNotifyCode*/, WORD wID, HWND /*hWndCtl*/,
BOOL& /*bHandled*/)     {
            // TODO: Add validation code
            EndDialog(wID);
            return 0;
      }
      LRESULT OnCancel(WORD /*wNotifyCode*/, WORD wID, HWND
/*hWndCtl*/, BOOL& /*bHandled*/){
            EndDialog(wID);
            return 0;
      }
};
```

Two source files are generated, stdafx.cpp and <projectname>.cpp. Stdafx.cpp contains just two lines, #includes of stdafx.h and the ATL header file, atlimpl.h. The main source file, <projectname>.cpp, #includes various WTL header files and maindlg.h, then create a variable called `_Module` based on the WTL data type `CAppModule`, and provides the following `WinMain`:

```
int WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE
/*hPrevInstance*/, LPTSTR lpstrCmdLine, int nCmdShow){
      HRESULT hRes = ::CoInitialize(NULL);
 ATLASSERT(SUCCEEDED(hRes));
 // initialize the common controls
 . . .
      hRes = _Module.Init(NULL, hInstance);
      ATLASSERT(SUCCEEDED(hRes));
      CMainDlg dlgMain;
      int nRet = dlgMain.DoModal();
      _Module.Term();
      ::CoUninitialize();
      return nRet;
}
```

Note that there are calls to `CoInitialize` and `CoUninitialize`, even though we have not selected the *COM Server* application feature.

That is it. It is easy to really understand your first WTL application within ten minutes.

Default Project Settings

The WTL AppWizard produces project files with two pre-configured build settings – one for release and one for debug. This contrasts with the ATL AppWizard, which produces builds for Debug, Unicode Debug, Release MinSize, ReleaseMinDependency, Unicode Release MinSize and UnicodeMinDependency. At first one might consider ATL to be more helpful – but as we will see very often during our examination of WTL, the minimalist approach that WTL offers is better.

The vast majority of developers just need two builds – one for debugging and one for release. Decisions concerning UNICODE support or the tradeoff of minimum build versus minimum size is made once during development, and it is very rare that developers need to have multiple builds for these alternatives. They make their choice and that is that. Having all these extra build settings hanging around can cause problems and confusion. Project settings changes can incorrectly be made to the wrong build, and they seemingly have no effect. Groups of developers working on the same project have to all know which one to use. In most projects involving ATL, all but two of the build settings become "dormant", are never updated but are not deleted. It just adds to potential problems. Smart developers get rid of these as soon as possible.

Unicode is supported if `_UNICODE` is defined.

The minimum sized binary is achieved if `_ATL_DLL` and `_ATL_MIN_CRT` are defined.

The minimum dependency is achieved if `_ATL_STATIC_REGISTRY` and `_ATL_MIN_CRT` are defined.

When developing with WTL, you should decide which of these settings you desire and simply add they to the release and debug builds. If you really need additional builds (thought it is unlikely) they simply create them.

The default preprocessor definition for a WTL AppWizard generated project is for the DEBUG build:

```
WIN32,_DEBUG,_WINDOWS,_MBCS,STRICT
```

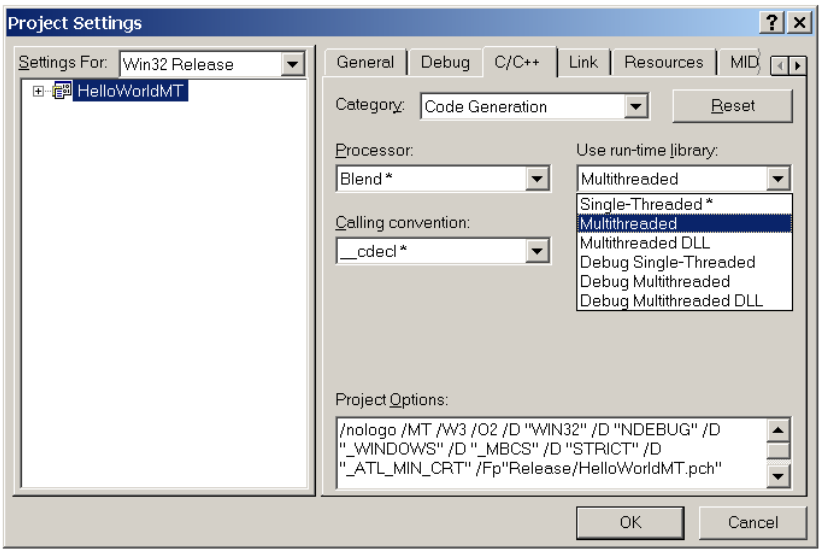
And for the release build:

```
WIN32,NDEBUG,_WINDOWS,_MBCS,STRICT,_ATL_MIN_CRT
```

Precompiled headers are supported through stdafx.h and stdafx.cpp.

Normally WTL does not use the C Run-Time Library. This is enforced when `_ATL_MIN_CRT` is added, which is the default setting for the release build. If you do decide to use CRT, then remove `_ATL_MIN_CRT`. If using the CRT, one point you must be careful with is which version you are using. For efficiency reasons, there are multiple versions of the CRT. Sometimes applications will only have one thread and with the CRT to be as fast as possible. Other times applications will have multiple threads and wish to call CRT functions concurrently, so a special version of the CRT exists with synchronization as appropriate. This is slightly slower so is only used with multi-threaded applications. How does the compiler know which one your application should use? It does not. Instead, it lets you tell it, via a build setting.

In DevStudio, Project menu, select "*Settings ...*" . The Project Settings dialog will be displayed. Select the *C/C++* property page and the *Code Generation* category. The Use run-time library list shows the available options regarding which CRT to use. It is almost always best if an EXE and all its DLLs use the same setting for this. It is often disastrous if an EXE with multiple threads that uses the CRT only links with the single-threaded version of the CRT.



The WTL AppWizard generate projects have this setting set to *Multithreaded*, which makes sense. By default WTL projects do not use the CRT, but if they do they are thread-safe.

Exception handling is enabled for debug build and disabled for release builds.

C++'s RTTI is not enabled for either Debug or Release builds. Most developers do not need RTTI.

In WTL projects that are not COM servers, there are neither custom steps, pre-link steps nor post-build steps. In WTL projects that are COM servers, there is this custom step that registers the EXE if its timestamp is more recent than the dummy trg timestamp file:

```
"$(TargetPath)" /RegServer
echo regsvr32 exec. time > "$(OutDir)\regsvr32.trg"
echo Server registration done!
```

SDI Application

Next, we will try to create a HelloWorld application that is presented as a typical main window in an application. We need to create a new project, which we call HelloWorldWin.

In the WTL AppWizard, select the SDI Application project type. Leave both application features unchecked. Then select next. In the second step of the AppWizard use the default selections.

Open the <projectname>view.h file, and in the `OnPaint` method replace the TODO comment with this one line of code:

```
dc.TextOut(0, 0, TEXT("HelloWorld"));
```

When you build and run the application the following window will appear:

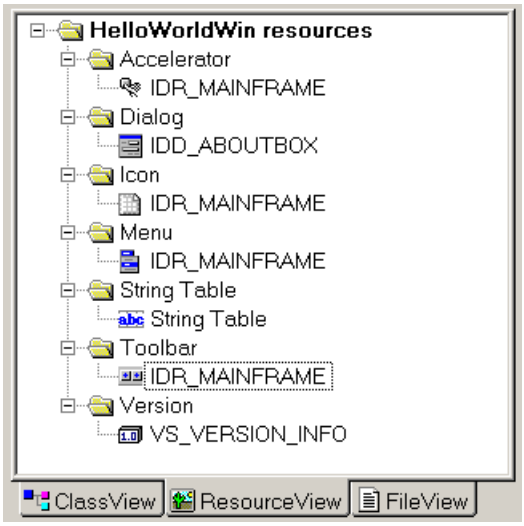
Let's look at what is happening behind the scenes.

There are many more entries in the resources. Additional entries are provided for the menubar, toolbar, and versioning. There is a string table with almost fifty strings, which are used in various places in the application.

The files stdafx.h, stdafx.cpp and resources.h are the same are for the dialog-based sample. The about box code is in a separate class, `CAboutDlg` in a separate file, aboutdlg.h.

The three files that are different in the sample are <projectname>.cpp, <projectname>view.h and mainfrm.h.

<Projectname>.cpp contains two functions – WinMain which is similar to the Dlg implementation, with the change that the `dlg.DoModal` code is replaced by a called to the second function in the file, `Run`.




```
int Run(LPTSTR /*lpstrCmdLine*/ = NULL,
              int nCmdShow = SW_SHOWDEFAULT){
      CMessageLoop theLoop;
      _Module.AddMessageLoop(&theLoop);
      CMainFrame wndMain;
      if(wndMain.CreateEx() == NULL)      {
            ATLTRACE(_T("Main window creation failed!\n"));
            return 0;
      }
      wndMain.ShowWindow(nCmdShow);
```

```
      int nRet = theLoop.Run();
      _Module.RemoveMessageLoop();
      return nRet;
}
```

Run sets up instances of a class within our application called `CMainFrame` (defined in our mainfrm.h) and the WTL class `CMessageLoop`.

`_Module` maintains a map of threadId / messageloop pairs, and calls to `AddMessageLoop` and `RemoveMessageLoop` edit it as needed. Later we will see the use of `GetMessageLoop` that retrieves the message loop for the current thread.

`CMessageLoop` manages arrays of message filters and idle handlers and performs the traditional Win32 message dispatching. Idle handlers are functions that are called when there are no more messages on the queue. They are usually used for background processing. Message filters are functions that can be used to customize how messages get translated, before the main message loop gets at them.

The WTL AppWizard has generated an implementation of a class called `CMainFrame` in mainfrm.h for us. `CMainFrame` manage the main application window and within it must render the menubar, toolbar, statusbar and a separate client area windows called the view. In effect, it performs the same role as the MFC class of the same name.

```
class CMainFrame : public CFrameWindowImpl<CMainFrame>,
            public CUpdateUI<CMainFrame>,
            public CMessageFilter, public CIdleHandler{
public:
      DECLARE_FRAME_WND_CLASS(NULL, IDR_MAINFRAME)
      CHelloWorldWinView m_view;
      CCommandBarCtrl m_CmdBar;

      BEGIN_MSG_MAP(CMainFrame)
            MESSAGE_HANDLER(WM_CREATE, OnCreate)
            COMMAND_ID_HANDLER(ID_APP_EXIT, OnFileExit)
            COMMAND_ID_HANDLER(ID_FILE_NEW, OnFileNew)
            COMMAND_ID_HANDLER(ID_VIEW_TOOLBAR, OnViewToolBar)
            COMMAND_ID_HANDLER(ID_VIEW_STATUS_BAR, OnViewStatusBar)
            COMMAND_ID_HANDLER(ID_APP_ABOUT, OnAppAbout)
            CHAIN_MSG_MAP(CUpdateUI<CMainFrame>)
            CHAIN_MSG_MAP(CFrameWindowImpl<CMainFrame>)
      END_MSG_MAP()

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

      LRESULT OnCreate(. . .) {
            // create command bar window, set up toolbar and call
       // the view's Create function
       . . .
       CMessageLoop* pLoop = _Module.GetMessageLoop();
```

```
        pLoop->AddMessageFilter(this);
        pLoop->AddIdleHandler(this);
            return 0;
        }

        LRESULT OnFileExit(WORD /*wNotifyCode*/, WORD /*wID*/,
         HWND /*hWndCtl*/, BOOL& /*bHandled*/){
            PostMessage(WM_CLOSE);
            return 0;
        }

        LRESULT OnFileNew(WORD /*wNotifyCode*/, WORD /*wID*/,
                          HWND /*hWndCtl*/, BOOL& /*bHandled*/)
        {
            // TODO: add code to initialize document

            return 0;
        }
        // Implementation of the other message handlers
        . . .
};
```

`CMainFrame` derives from the following two WTL templates and two WTL classes.

- `CFrameWindowImpl` – Boilerplate code for manages frames
- `CUpdateUI` – Works in conjunction with an `UPDATE_UI_MAP` to perform data exchange between UI elements (e.g. toolbar & statusbar) and code
- `CMessageFilter` – custom message filtering
- `CIdleHandler` – custom idle handling

`CMainFrame` has a macro called `DECLARE_FRAME_WND_CLASS` that identifies the resource Id to use to when accessing UI resources such as toolbar, accelerator, and menu descriptions. `CMainFrame` instantiates a class called `C<Projectname>View`, which is responsible for the client area.

The `CMainFrame::OnCreate` function instantiates UI elements such as the toolbar, the statusbar and the menubar, and calls `Create` on the View class.

It also gets the message loop for the current thread through a call to `GetMessageLoop` and appends a message filter and idle handler.

There is a message map that maps menubar command Ids (e.g. `ID_APP_EXIT` & `ID_FILE_NEW`) to handler functions (e.g. `OnFileExit` & `OnFileNew`). Note that unlike MFC, where default implementation handlers for common commands are part of the library, with WTL these handlers are directly in the application's code. The WTL AppWizard generated implementations of the handlers provide a full implementation where possible (e.g. `OnFileExit` posts a `WM_CLOSE`, which results in app shutdown) and otherwise the handlers contain a TODO comment (e.g. `OnFileNew` needs to be written by the application developer). Unlike MFC, WTL does not provide

its own serialization code and binary file formats. Instead modern applications are strongly encouraged to base their serialization on XML, and it is easy enough to add your own XML code behind `OnFileNew` etc to call the XML parser.

When `OnFileExit` posts a `WM_CLOSE`, neither the WTL templates nor the WTL AppWizard-generated application code detects this message, so by default it is passed to `DefWindowProc`, which responds by calling the Win32 API `DestroyWindow`, which posting a `WM_DESTROY` message. The application code does not detect this either, but the WTL template `CFrameWindowImplBase` does – by mapping it to a call to its member function `OnDestroy`. It in turn calls the Win32 API `PostQuitMessage`, which pops a `WM_QUIT` message onto the message queue.

```
      LRESULT OnDestroy(UINT, WPARAM, LPARAM, BOOL& bHandled){
            if((GetStyle() & (WS_CHILD | WS_POPUP)) == 0)
                  ::PostQuitMessage(1);
            bHandled = FALSE;
            return 1;
      }
```

The WTL `CMessageLoop::Run` method has a `for` loop taking messages off the message queue and processing them. It calls the Win32 API `GetMessage`, which returns `0` if the message received is `WM_QUIT`. If this is the case, there is a break out of the loop.

```
      bRet = ::GetMessage(&m_msg, NULL, 0, 0);
 . . .
      else if(!bRet){
            ATLTRACE2(atlTraceUI, 0,
              _T("CMessageLoop::Run - exiting\n"));
            break;            // WM_QUIT, exit message loop
      }
```

If you wish to customize shutdown, you could consider detecting the `WM_CLOSE` message and when finished calling `DestroyWindow` manually.

The view class provides a straightforward ATL window that occupies the space from underneath the toolbar to above the statusbar. It has a message map that by default handles one message, `WM_PAINT`, which gets mapped to `OnPaint`. The implementation of `OnPaint` by default contains a TODO comment, but we changed that to call `TextOut`.

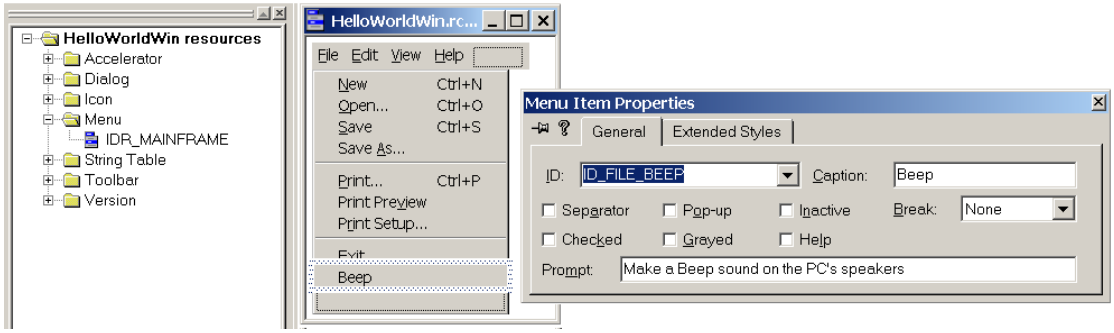
```
class CHelloWorldWinView :
       public CWindowImpl<CHelloWorldWinView>{
public:
      DECLARE_WND_CLASS(NULL)
      BOOL PreTranslateMessage(MSG* pMsg){
            pMsg;
            return FALSE;
      }
      BEGIN_MSG_MAP(CHelloWorldWinView)
            MESSAGE_HANDLER(WM_PAINT, OnPaint)
      END_MSG_MAP()
      LRESULT OnPaint(UINT /*uMsg*/, WPARAM /*wParam*/,
```

```
            LPARAM /*lParam*/, BOOL& /*bHandled*/)    {
        CPaintDC dc(m_hWnd);
        dc.TextOut(0,0, TEXT("HelloWorld"));
        return 0;
    }
};
```

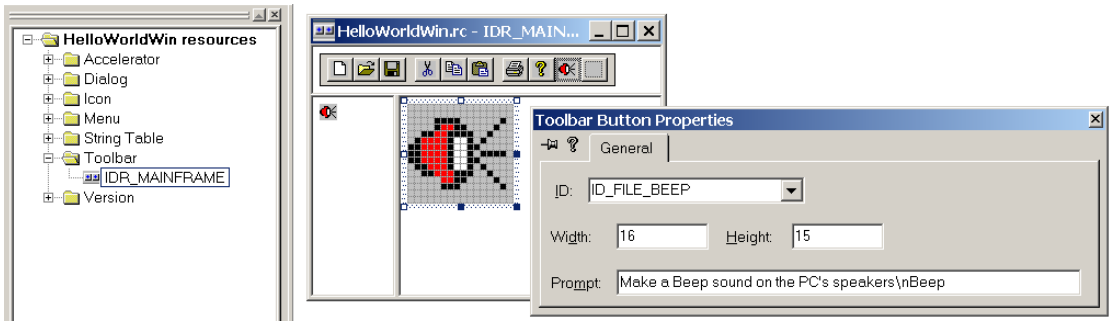
Adding a Menu-item And a Toolbar Icon

The resources generated by the WTL AppWizard include a menu and a toolbar, both identified by `IDR_MAINFRAME`. If we wish to add commands to menus or the toolbar, then we need to extend these resources in ResourceView, assign command ids to them, detect these command ids in the message map for the frame window and map them to handler functions. An example will make this clearer. Imagine we wished to add a menu item and toolbar icon which when pressed, would result in the Win32 `Beep` API being called.

We start by adding a menu item to the resources, giving it an id of `ID_FILE_BEEP`.



We then add a toolbar icon to the toolbar resource. We also give it also the id of `ID_FILE_BEEP`. To support tooltips, append *"\nBeep"* to the prompt.



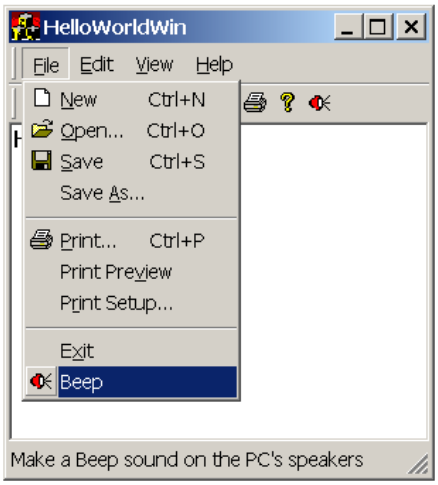
Both the menubar and the toolbar are going to be hosted by the mainframe class. Therefore we add this command handler to its message map:

```
COMMAND_ID_HANDLER(ID_FILE_BEEP, OnFileBeep)
```

And we add this implementation

```
LRESULT OnFileBeep(WORD /*wNotifyCode*/, WORD /*wID*/,
 HWND /*hWndCtl*/, BOOL& /*bHandled*/){
     Beep(200,200);
     return 0;
}
```

When we build and run the application we see the extra Beep menu item and Beep toolbar icon. When either is pressed, we should hear a beep from the PC's speakers. When we hover the mouse over the beep toolbar icon we see its tooltip.



Assuming you accept the defaults for step 2 in the WTL AppWizard, the generated application uses WTL's command bars to display the contents of the menubar and toolbar within rebar controls. One neat trick of WTL command bars is that when rendering menu items that have the same command id as a toolbar icon, that icon is also rendered along with the menu-item name. This greatly increases awareness among end-users of what such icons mean – it is especially useful when very many icons are in use.

MDI Application

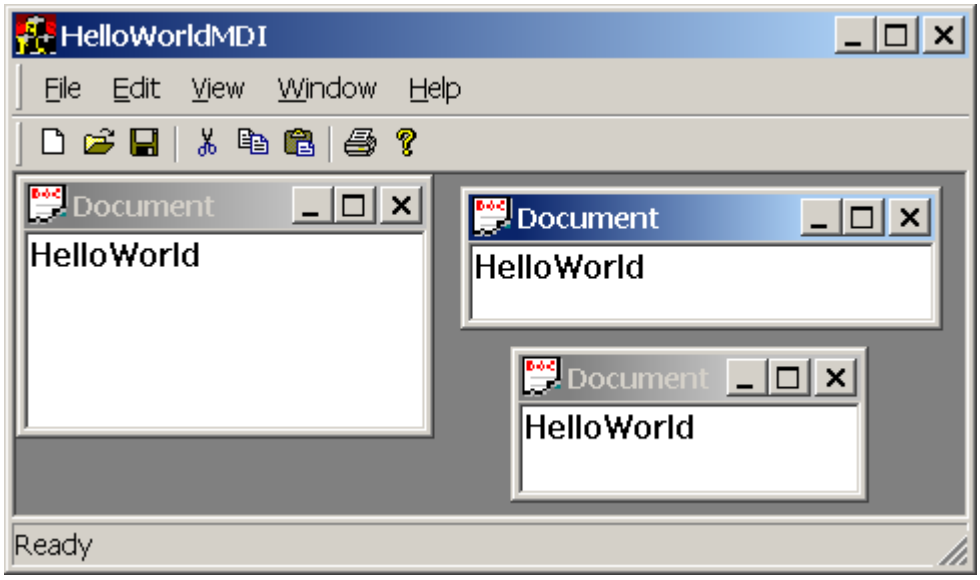
If we wish to edit multiple document types concurrently, the MDI is one approach. The top-level mainframe window contains a number of childframe windows, each of which can edit a document. Visual C++'s DevStudio itself is a classic example.

To examine the MDI support within WTL, create a new project called HelloWorldMDI, and in the WTL AppWizard select the MDI Application option.

As with the SDI Application sample, in the `OnPaint` method in the C<ProjectName>view class add this line:

```
dc.TextOut(0,0, TEXT("HelloWorld"));
```

Compile and run. There is another frame containing the menubar, toolbar, status bar, and in its client area there can be a multiple MDI child window. Each of these has a view and hence during each of their refreshes they output the string *"HelloWorld"*.



The code produced is very similar to that of the SDI Application. All files are exactly the same, except for mainfrm.h. Also, there is an additional file called ChildFrm.h.

The `CMainFrame` class is defined in mainfrm.h as:

```
class CMainFrame : public CMDIFrameWindowImpl<CMainFrame>,
      public CUpdateUI<CMainFrame>, public CMessageFilter,
      public CidleHandler
```

In the SDI sample it derives from `CFrameWindowImpl`, whereas in the MDI sample it derives from `CMDIFrameWindowImpl`, which WTL defines as:

```
template <class T, class TBase = CMDIWindow,
 class TWinTraits = CFrameWinTraits> class CMDIFrameWindowImpl;
```

The message map handles three additional messages:

```
COMMAND_ID_HANDLER(ID_WINDOW_CASCADE, OnWindowCascade)
COMMAND_ID_HANDLER(ID_WINDOW_TILE_HORZ, OnWindowTile)
COMMAND_ID_HANDLER(ID_WINDOW_ARRANGE, OnWindowArrangeIcons)
```

Handler functions for the new messages merely pass the call onto default implementations inside `CMDIFrameWindowImpl`'s `CMDIWindow`. You can change these as needed.

```
LRESULT OnWindowCascade(WORD /*wNotifyCode*/, WORD /*wID*/, HWND
/*hWndCtl*/, BOOL& /*bHandled*/)    {
            MDICascade();
            return 0;
      }
LRESULT OnWindowTile(WORD /*wNotifyCode*/, WORD /*wID*/, HWND
/*hWndCtl*/, BOOL& /*bHandled*/){
            MDITile();
            return 0;
      }
      LRESULT OnWindowArrangeIcons(WORD /*wNotifyCode*/, WORD /*wID*/,
HWND /*hWndCtl*/, BOOL& /*bHandled*/){
            MDIIconArrange();
            return 0;
```

```
      }
```

OnFileNew is also changes to create a new ChildFrame.

```
      LRESULT OnFileNew(WORD /*wNotifyCode*/, WORD /*wID*/, HWND
/*hWndCtl*/, BOOL& /*bHandled*/)
      {
            CChildFrame* pChild = new CChildFrame;
            pChild->CreateEx(m_hWndClient);

            // TODO: add code to initialize document

            return 0;
      }
```

The ChildFrm.h file contains the definition of the new `ChildFrame` class.

It is defined simply as:

```
class CChildFrame : public CMDIChildWindowImpl<CChildFrame>
{
public:
      DECLARE_FRAME_WND_CLASS(NULL, IDR_MDICHILD)
      CHelloWorldMDIView m_view;
      virtual void OnFinalMessage(HWND /*hWnd*/){
            delete this;
      }
      BEGIN_MSG_MAP(CChildFrame)
            MESSAGE_HANDLER(WM_CREATE, OnCreate)
            MESSAGE_HANDLER(WM_FORWARDMSG, OnForwardMsg)
            CHAIN_MSG_MAP(CMDIChildWindowImpl<CChildFrame>)
      END_MSG_MAP()
      LRESULT 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);
        bHandled = FALSE;
        return 1;
      }
LRESULT OnForwardMsg(UINT /*uMsg*/, WPARAM /*wParam*/,
                        LPARAM lParam, BOOL& /*bHandled*/){
            LPMSG pMsg = (LPMSG)lParam;
      if(CMDIChildWindowImpl<CChildFrame>::PreTranslateMessage(pMsg))
                  return TRUE;
            return m_view.PreTranslateMessage(pMsg);
      }
};
```

Multiple Threads SDI

The SDI Application is fine when we wish to edit a single document of a single document type at a time. This happens if the application only wishes to display a single

SDI window, and within it edit a single document (Notepad is a good example). The MDI approach manages multiple child frames within a single top-level main frame.

What the Multiple Threads SDI option in the WTL AppWizard provides is the capability of editing multiple documents in multiple top-level SDI windows (the Notepad-like WTL example MTPad shows this – Microsoft Word 2000 is also a good example). The end-user sees a separate top-level window for each open document, but only a single process is running. There should be a slight performance increase (as multiple threads in one process will work faster than multiple processes) and resources (memory/files/database connection, etc.) can be shared among all the threads.

Now we wish to examine how this works. Create a new project called HelloWorldMT, and in the WTL AppWizard select the Multiple Threads SDI option. When we select this, the *"Create as a COM Server"* option is disabled. This is something we will investigate further later. Select the defaults for step 2 options.

The code produced is quite similar to that from *"SDI Application"* option with 3 areas of difference. In both the resource file and the mainfrm.h file there are a small difference and there is a substantial difference in the <projectname>.cpp file.

The menubar in Multiple Threads SDI has an additional menu item, titled "New Window" which has a command id of `ID_FILE_NEW_WINDOW`.

In mainfrm.h, the message map has a new entry:

```
COMMAND_ID_HANDLER(ID_FILE_NEW_WINDOW, OnFileNewWindow)
```

And an implementation of `OnFileNewWindow` is provided.

```
LRESULT OnFileNewWindow(WORD /*wNotifyCode*/, WORD /*wID*/,
 HWND /*hWndCtl*/, BOOL& /*bHandled*/){
 ::PostThreadMessage(_Module.m_dwMainThreadID, WM_USER, 0, 0L);
return 0;
}
```

What this is doing is posting a private message, `WM_USER`, onto the message queue of the primary thread. (Therefore the developer should not try to use this particular message type for private purposes). We will need to see what happens to that message.

The source file contain `WinMain`, <projectname>.cpp, is quite different.

The `WinMain` implementation has the call to the `Run` function removed, and it is replaced by:

```
CHelloWorldMTThreadManager mgr;
int nRet = mgr.Run(lpstrCmdLine, nCmdShow);
```

The `C<Projectname>ThreadManager` class is completely defined inside the <projectname>.cpp file. The implementation of the `Run` function has been moved into this class and it includes multi-threading functionality.

This class is used to manage a set of threads, each of which in turn manages one SDI window. This class defines one internal structure, and provides a static function that is the threadproc for the per-SDI thread, methods to add and remove threads and a `Run` method.

```
class CHelloWorldMTThreadManager
{
public:
      // thread init param
      struct _RunData
      {
            LPTSTR lpstrCmdLine;
            int nCmdShow;
      };
```

The `RunData` structure stores the command line and the `nCmdShow` parameter. A new instance of this structure is created each time a new thread is needed, and it tells the thread which command line and which `nCmdShow` option to use.

The `threadproc` parameter is a pointer, and we need to pass in a pointer and an int. Therefore we need this extra structure containing a pointer and an integer, and pass in a pointer to it.

Our multithreading experts might say this information is read-only and it is only necessary that it exists once. After all, we are allocating either eight or twelve bytes (depending on whether we are using Win32 or Win64), which are both occupying space and time consuming, and it would better if we could avoid it. The way it is used here is that the first SDI window gets the command line and the `CmdShow` from `WinMain` and that subsequent threads are passed a `NULL` command line and `SW_SHOWNORMAL`.

The wizard-generated code ignores the command line and uses `nCmdShow` parameter to determine the rendering of the mainframe. As your first thread will be receiving the command line and subsequent threads will receive `NULL`, you might consider adding code to determine if the command-line is non-`NULL` and if so to process it – how this is to be is obviously application-specific, but one typical command-line parameter could be a document filename to open.

At this point our multithreading experts will argue (they are a persistent bunch) that surely we could have the structure for the first thread (which needs real data), and for subsequent thread pass in `NULL` as the threadproc parameter, and inside the threadproc if the parameter is `NULL` ignore the command line and use `SW_SHOWNORMAL` when rendering the mainframe. Yes, this will work, but it limits our future flexibility – what happens if we wish to do different things in different threads, then this defaulting will hamper us. In conclusion, the current implementation is fine, and don't worry about a dozen bytes (1GB of RAM does not cost much!). If you are losing a night's sleep over it, then use the `NULL` parameter idea.

```
      static DWORD WINAPI RunThread(LPVOID lpData){
            CMessageLoop theLoop;
            _Module.AddMessageLoop(&theLoop);
            _RunData* pData = (_RunData*)lpData;
            CMainFrame wndFrame;
      if(wndFrame.CreateEx() == NULL){
                  ATLTRACE(_T("Frame window creation failed!\n"));
                  return 0;
            }
            wndFrame.ShowWindow(pData->nCmdShow);
```

```
            ::SetForegroundWindow(wndFrame);    // Win95 needs this
            delete pData;
            int nRet = theLoop.Run();
            _Module.RemoveMessageLoop();
            return nRet;
      }
```

`RunThread` is the threadproc for each SDI thread. Note that it is static, which is a requirement for a C++ class member to be a threadproc. Apart from the code concerning `_RunData`, this is the same as we examine previously in the SDI Application sample.

```
      DWORD m_dwCount;
      HANDLE m_arrThreadHandles[MAXIMUM_WAIT_OBJECTS - 1];
      CHelloWorldMTThreadManager() : m_dwCount(0){ }
```

There are two data members, one to hold the count of and the other to hold the handles of the SDI threads. We need to keep track of these to determine when thread dies, and when the final one exits then we can shutdown the process. The count is initialized to 0.

```
      int Run(LPTSTR lpstrCmdLine, int nCmdShow){
            MSG msg;
            // force message queue to be created
            ::PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE);
            AddThread(lpstrCmdLine, nCmdShow);
            int nRet = m_dwCount;
            DWORD dwRet;
            while(m_dwCount > 0){
                  dwRet = ::MsgWaitForMultipleObjects(m_dwCount,
                        m_arrThreadHandles, FALSE,
                        INFINITE, QS_ALLINPUT);
                  if(dwRet == 0xFFFFFFFF)
                   ::MessageBox(NULL,
               _T("ERROR: Wait for multiple objects failed!!!"),
               _T("HelloWorldMT"), MB_OK);
                  else if(dwRet >= WAIT_OBJECT_0 && dwRet <=
                              (WAIT_OBJECT_0 + m_dwCount - 1))
                        RemoveThread(dwRet - WAIT_OBJECT_0);
                  else if(dwRet == (WAIT_OBJECT_0 + m_dwCount)){
                        ::GetMessage(&msg, NULL, 0, 0);
                        if(msg.message == WM_USER)
                              AddThread("", SW_SHOWNORMAL);
                        else
                              ::MessageBeep((UINT)-1);
                  }
                  else
                        ::MessageBeep((UINT)-1);
            }
            return nRet;
      }
};
```

The `Run` method is called from `WinMain`. It first calls the `AddThread` method to add the initial thread that processes the real command line and displays the first window.

After this it initializes the `nRet` parameter to the value of `m_dwCount`, which has just been set to 1 inside `AddThread`. The last line in `Run` returns `nRet` as the return value of the method. In `WinMain`, the return value from `Run` is used as the return value for the entire process, which means it becomes the process' exit code. Most of the time this is not important – if it is for your application you can change it as needed.

The core of the `Run` method is a while loop, in which a blocking call is make to `MsgWaitForMultipleObjects`, which waits until a message arrives on the message queue for the upon which this API call is made or any kernel handle in the list becomes signaled. The kernel handles are the handles to the thread Ids, and they will become signaled when the thread exits. Threads get added when the user selects *"New Window",* which results in a `WM_USER` message getting popped onto the primary thread's message queue. Threads exit when the end-user closes the SDI window. In both cases it is the call to `MsgWaitForMultipleObjects` that detects these and responds accordingly by calling `AddThread` and `RemoveThread` as needed. When the number of SDI threads hits 0, then no windows are displayed and the application can shutdown.

In the `WaitForMultipleObjects` API, `MAXIMUM_WAIT_OBJECTS` kernel objects can be waited upon simultaneously. `MAXIMUM_WAIT_OBJECTS` is set to 64 on Windows 2000 and the older Windows OS versions. With `MsgWaitForMultipleObjects`, the limit is `MAXIMUM_WAIT_OBJECTS-1`, or `63`. What happened to the last one? The OS itself is adding an event handle to the list and thus taking up one slot. This event gets signaled when messages are detected on the message queue.

The hard coded strings inside the call to `MessageBox` are noted – your internationalization engineers will need to change that to resource strings.

Note there is a call to `PeekMessage` with the comment "`force message queue to be created`". What is happening here is that the primary thread running this method has no window, and yet each SDI thread contains a menu item called *"Add Window",* which when selected causes the SDI thread to pop a `WM_USER` message onto the message queue of the primary thread.

As you will recall from our discussion of Win32 windowing, inside the Windows OS, the kernel maintains message queues for threads that might need them. Not every thread will be using windows (think of all that number-crunching code in the world which has no user interface), so it would be inefficient to automatically create these queues. Instead, what the kernel does is to create a message queue structure the first time calls are made to the windowing APIs on that thread.

As the primary thread has no windows, up to this point the message queue does not exist. The call to `PeekMessage` here within the primary thread will result in the kernel creating it for this thread. The call to `MsgWaitForMultipleObjects` a few lines on will also cause the message queue for the primary thread to be created.

So why is the call to `PeekMessage` needed? If we comment it out and build and run the application, we will notice it seems to run fine. Where problems can occur is when the secondary thread (created in the call to `AddThread`) starts sending messages to the

primary thread before the primary thread has a message queue. This could occur if there is a context switch between the calls to `AddThread` and `MsgWaitForMultipleObjects,` and when the secondary thread starts running, the end-user selected the *New Window* menu item. It is unlikely to happen, but the `PeekMessage` call completely eliminates that possibility. To simulate the problem we could force a context switching by changing this code in the `Run` method . . .:

```
::PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE);
AddThread(lpstrCmdLine, nCmdShow);
```

. . . to these lines:

```
// ::PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE);
   AddThread(lpstrCmdLine, nCmdShow);
   Sleep(20000);
```

When we run this application, the primary thread runs, executes `AddThread` and sleeps. The secondary thread is launched from `AddThread` and causes the SDI window to be displayed – if the end-user then quickly selects "`New Window`" a few times, we see nothing immediately. When the sleep duration expires, the message queue is create, and all messages subsequent to that are processed.

What happens to the messages posted onto the message queue from the SDI threads when the message queue did not exist? On Windows 2000, a solitary message is recorded – so if we press *"Add Window"* three times, then at the end of the sleep period only one new message is created. This is probably different on other versions of Windows.

A word of warning - as with any multithreading coding, you have to be extremely careful with sequencing your calls to the OS. The code generated by WTL AppWizard as presented is correct. However, as it is now part of your application, you could quite easily, and probably will, start making changes to it. Again, this is fine – that is what you are paid to do if it is needed. This warning relates to the call to `MsgWaitForMultipleObjects`. This returns when new messages are received on the message queue for this thread. Messages that have been looked at by PeekMessage are not consider new. In the AppWizard generated code, the sequence of calls is `PeekMessage`, `AddThread` and `MsgWaitForMultipleObjects`. If you change their order to `AddThread, PeekMessage` and `MsgWaitForMultipleObjects`, and if there is a context switch between the first two of these, if could happen that the user clicks *"Add New Window"* in the SDI thread and a `WM_USER` message gets popped onto the primary thread's message queue. Sometime later there is another context switch, and the primary thread resumes executing. It calls `PeekMessage`, which detects the `WM_USER` message on the queue and changes its state so it is not new. Then `MsgWaitForMultipleObjects` executes and it simply ignores the `WM_USER` message because it is not new. To clearly demonstrate the problem, change these two lines in the wizard-generated code:

```
::PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE);
AddThread(lpstrCmdLine, nCmdShow);
```

to these lines:

```
AddThread(lpstrCmdLine, nCmdShow);
Sleep(20000); // give ourselves plenty of time to "Add Window"
```

```
::PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE);
```

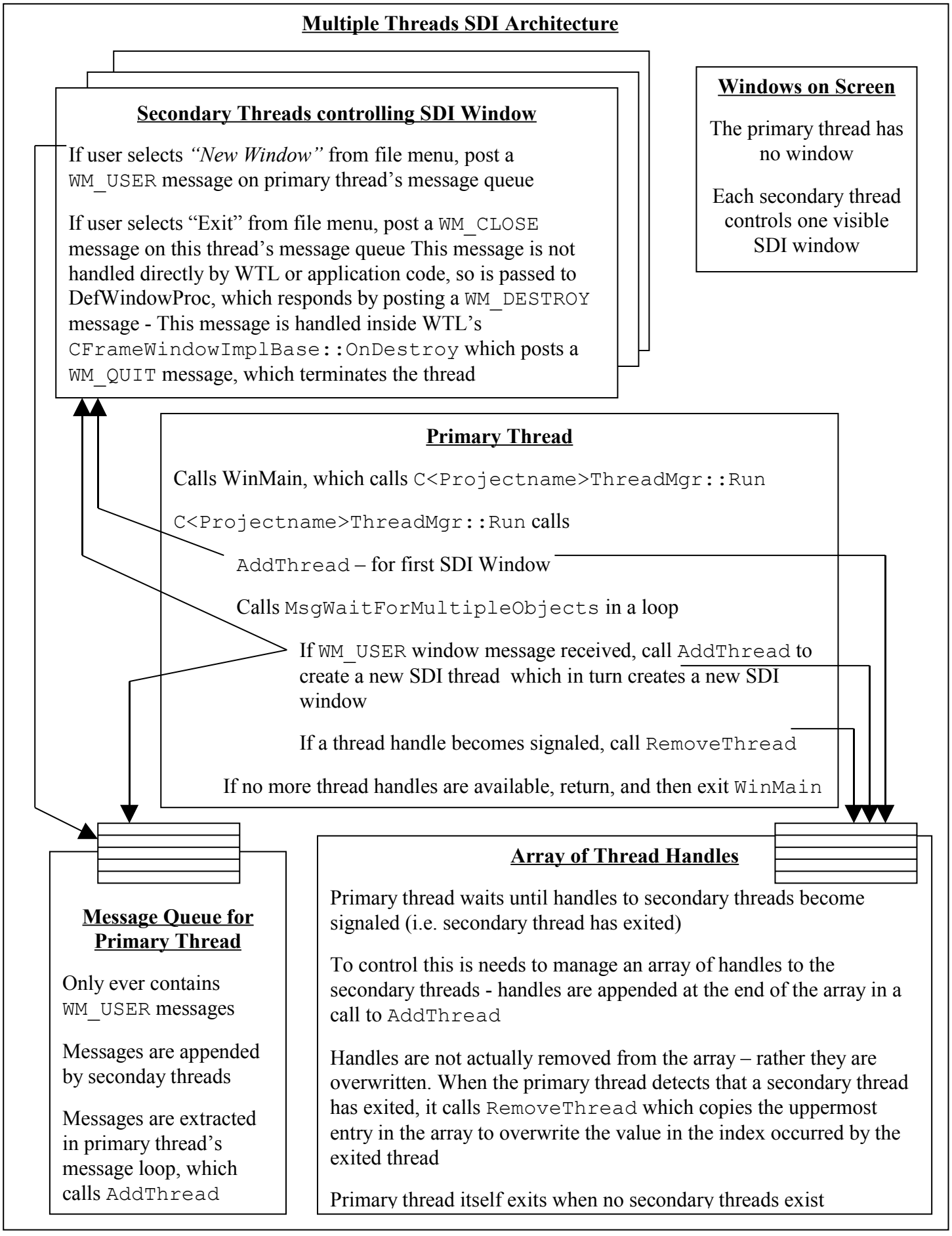
When the application runs, select *New Window* from the *File* menu – you will note that the command is ignored by the application. In the original code, a new window would be displayed.

To avoid the problem, you should either leave the code generated by the WTL AppWizard as is, remove the call to `PeekMessage` or change the call to `MsgWaitForMultipleObjects` to a call to `MsgWaitForMultipleObjectsEx` with the `MWMO_INPUTAVAILABLE` parameter.

```
dwRet = ::MsgWaitForMultipleObjectsEx(m_dwCount,
             m_arrThreadHandles, INFINITE, QS_ALLINPUT,
             MWMO_INPUTAVAILABLE);
```

It is unlikely that you will change the code in this way, but you have been warned!

```
DWORD AddThread(LPTSTR lpstrCmdLine, int nCmdShow){
      if(m_dwCount == (MAXIMUM_WAIT_OBJECTS - 1)){
        ::MessageBox(NULL,
          _T("ERROR: Cannot create ANY MORE threads!!!"),
           _T("HelloWorldMT"), MB_OK);
             return 0;
      }
      _RunData* pData = new _RunData;
      pData->lpstrCmdLine = lpstrCmdLine;
      pData->nCmdShow = nCmdShow;
      DWORD dwThreadID;
      HANDLE hThread = ::CreateThread(NULL, 0, RunThread,
       pData, 0, &dwThreadID);
      if(hThread == NULL){
       ::MessageBox(NULL,
       _T("ERROR: Cannot create thread!!!"),
       _T("HelloWorldMT"), MB_OK);
            return 0;
      }
      m_arrThreadHandles[m_dwCount] = hThread;
      m_dwCount++;
      return dwThreadID;
}
```


Multiple Threads SDI Architecture
Secondary Threads controlling SDI Window
If user selects "New Window" from file menu, post a WM_USER message on primary thread's message queue
If user selects "Exit" from file menu, post a WM_CLOSE message on this thread's message queue This message is not handled directly by WTL or application code, so is passed to DefWindowProc, which responds by posting a WM_DESTROY message - This message is handled inside WTL's CFrameWindowImplBase::OnDestroy which posts a WM_QUIT message, which terminates the thread
Windows on Screen
The primary thread has no window
Each secondary thread controls one visible SDI window
Primary Thread
Calls WinMain, which calls C<Projectname>ThreadMgr::Run
C<Projectname>ThreadMgr::Run calls
AddThread – for first SDI Window
Calls MsgWaitForMultipleObjects in a loop
If WM_USER window message received, call AddThread to create a new SDI thread which in turn creates a new SDI window
If a thread handle becomes signaled, call RemoveThread
If no more thread handles are available, return, and then exit WinMain
Message Queue for Primary Thread
Only ever contains WM_USER messages
Messages are appended by seconday threads
Messages are extracted in primary thread's message loop, which calls AddThread
Array of Thread Handles
Primary thread waits until handles to secondary threads become signaled (i.e. secondary thread has exited)
To control this is needs to manage an array of handles to the secondary threads - handles are appended at the end of the array in a call to AddThread
Handles are not actually removed from the array – rather they are overwritten. When the primary thread detects that a secondary thread has exited, it calls RemoveThread which copies the uppermost entry in the array to overwrite the value in the index occurred by the exited thread
Primary thread itself exits when no secondary threads exist

The AddThread method is responsible for creating a new SDI thread that in turn causes a new SDI window to be displayed. It just calls the Win32 CreateThread API and pops the thread handle at the end of the handle array and increments the m_dwCount variable that stores the number of entries.

There is no need to switch the call to m_dwCount++ with a call to InterlockedIncrement(&m_dwCount) because this code will only every be used from the primary thread – it is never used from the SDI threads.

```
void RemoveThread(DWORD dwIndex){
      ::CloseHandle(m_arrThreadHandles[dwIndex]);
      if(dwIndex != (m_dwCount - 1))
            m_arrThreadHandles[dwIndex] =
             m_arrThreadHandles[m_dwCount - 1];
      m_dwCount--;
}
```

The RemoveThread method closes the handle of the thread that has just exited and removes its entry from the array of handles. Note that this array must be kept contiguous, so the technique used is to copy the top-most entry to the index of the thread to remove.

Modeless Dialog-Based Application

The last application type to discuss is a modeless dialog-based application. The output from the WTL AppWizard for this is a mixture of the modal dialog and the SDI Application approaches.

In the Modeless Dialog code, CMainDlg is derived from CDialogImpl, just like the Modal Dialog sample, but also from CUpdateUI, CMessageFilter and CIdleHandler, as in the SDI sample.

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

There is a empty update UI map.

The closing of the dialog in the modal dialog sample was through a call to ATL's CDialogImpl::EndDialog, which in turn called the Win32 API, EndDialog. In the modeless dialog sample, the closing of the dialog results in the following generated code being called:

```
void CloseDialog(int nVal){
      DestroyWindow();
      ::PostQuitMessage(nVal);
}
```

There is also a call to CUpdateUIBase::UIAddChildWindowContainer added to OnInitDialog for updating of UI elements (CUpdateUIBase is the parent of CUpdateUI).

In the modal dialog sample, the <projectname>.cpp file contained the WinMain implementation, which handles the message loop in a call to CDialogImpl::DoModal, which in turn calls the Win32 API DialogBoxParam.

In the modeless dialog sample, the <projectname>.cpp file is exactly as with the SDI Application. It contains the WinMain and does the message loop through an instantiation of WTL's `CMessageLoop`.

Application Features - ActiveX Control Hosting

Now we turn our attention to the WTL AppWizard "Application Features" in step 1, *"ActiveX Control Hosting"* and *"Act as a COM Server"*.

When *"ActiveX Control Hosting"* is selected, two changes are made to the code.

Stdafx.h has some extra headers included. Instead of this line at the bottom of the file:

```
#include <atlwin.h>
```

Now there are these lines

```
#include <atlcom.h>
#include <atlhost.h>
#include <atlwin.h>
#include <atlctl.h>
```

Also, `WinMain` is changed to initialize COM and a call to `AtlAxInit` is added. `AtlAxInit` registers the "AtlAxWin" window, which is used for ActiveX control hosting.

```
int WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE
       /*hPrevInstance*/, LPTSTR lpstrCmdLine, int nCmdShow){
      HRESULT hRes = ::CoInitialize(NULL);
// initialize common controls
. . .
      hRes = _Module.Init(NULL, hInstance);
      ATLASSERT(SUCCEEDED(hRes));
      AtlAxWinInit();
      int nRet = Run(lpstrCmdLine, nCmdShow);
      _Module.Term();
      ::CoUninitialize();
      return nRet;
}
```

It is important to note that selecting *"ActiveX Control Hosting"* is only adding header file support, registering the ATLAxWin window class and initializing COM for the primary thread, and that is all it does. It does not itself actually add any ActiveX control. This can be done later as needed.

"ActiveX Control Hosting" and Modal Dialog Applications

A special case is when the WTL AppWizard is asked to generate a modal or modeless dialog application and ActiveX Control Hosting is selected. The generated dialog code derives from `CAxDialogImpl` instead of `CDialogImpl`.

"ActiveX Control Hosting" and SDI/MDI apps with Form view type

The second step within the WTL AppWizard allows you to select the basis for the view, which we will discuss shortly. Normally it is set to a (blank) generic window. It can

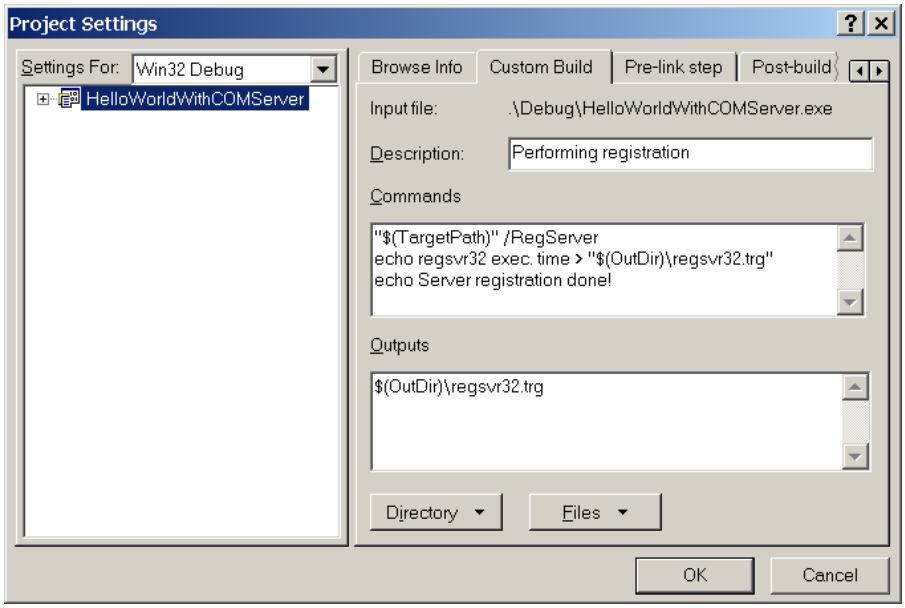
also be based on listview, treeview, etc. One additional option is to base it on a dialog template (a form). When the view is based on a form, the generated view class derives from `CDialogImpl`, regardless of the setting for *"ActiveX Control Hosting"* in step 1. However, when *"ActiveX Control Hosting"* is set, then an additional checkbox in set 2, *Host ActiveX Controls*, is enabled, and when set the form dialog derives from `CAxDialogImpl` and hence can host ActiveX controls.

Application Features - Act as a COM Server

All the samples produced up to now have concentrated exclusively on windowing. The ATL AppWizard can produce code that supports COM component creation. What happens if we wish to support both windowing and COM components within the one project? There is where the *"Act As a COM Server"* feature comes in.

When this feature is selected a number of changes are made to the project settings and the generated code.

A custom build step is added to register the components in the project (if any).



One point of interest here is that the rgs file for each component has the LocalServer entry set to `"LocalServer32 = s '%MODULE%'"` and we will need to return to this issue when examining how startup happens via Automation.

In stdafx.h, there is an external definition of `_Module` instantiating `CServerAppModule` and not `CAppModule` as in the previous sample. Note that `CServerAppModule` itself derives from `CAppModule` and adds methods very similar to the ATL AppWizard generated code when you select the *"Executable"* option. It has familiar methods such as `StartMonitor`, `MonitorShutdown`, `FindOneOf` and new methods such as `ParseCommandLine` and `Register/UnregisterAppId`. Also in stdafx.h, the header file atlcom.h is included, thus providing the application code with access to ATL's COM definitions.

Stdafx.cpp includes these additional lines, to do with registry management code:

```
#ifdef _ATL_STATIC_REGISTRY
#include <statreg.h>
#include <statreg.cpp>
#endif
```

An IDL file called <Projectname>.idl is generated and it contains a default type library definition (just as would be generated had we run the ATL AppWizard).

The main changes are in <projectname>.cpp. Essentially what it needs to do is merge the code from the WTL AppWizard and the ATL AppWizard auto-generated generated <projectname>.cpp files.

It starts by adding includes for the GUIDs and interfaces:

```
#include "initguid.h"
#include "HelloWorldWithCOMServer.h"
#include "HelloWorldWithCOMServer_i.c"
```

It instantiates a variable called `_Module` of type `CServerAppModule`. Note it MUST be called `_Module` as this name is used in various ATL macros.

```
CServerAppModule _Module;
```

It has an empty object map. Later, the ATL ObjectWizard may be used to populate this map, as in standard ATL programming.

```
BEGIN_OBJECT_MAP(ObjectMap)
END_OBJECT_MAP()
```

The Message Loop

The Run function is the similar to that of the SDI Application sample: It creates a `CMessageLoop`, instantiates `CMainFrame` and calls `CMessageLoop::Run`. The one change is the addition of a call to `_Module.Lock()`. This increments the lock count with the ATL COM code uses to decide whether the COM server can be shutdown. In essence, what is happening is that the code is pretending that the `CMainFrame` is actually a COM component, and while displayed, the process running this EXE should not shut down. We will have to track how shutdown actually happens later.

```
int Run(LPTSTR /*lpstrCmdLine*/ = NULL,
              int nCmdShow = SW_SHOWDEFAULT){
      CMessageLoop theLoop;
      _Module.AddMessageLoop(&theLoop);
      CMainFrame wndMain;
      if(wndMain.CreateEx() == NULL){
            ATLTRACE(_T("Main window creation failed!\n"));
            return 0;
      }
      _Module.Lock();
      wndMain.ShowWindow(nCmdShow);
      int nRet = theLoop.Run();
      _Module.RemoveMessageLoop();
      return nRet;
}
```

The WinMain code has called to CoInitialize/_Module.Init() and _Module.Term()/CoUninitialize at the beginning and end. It initializes the common controls. The command line is parsed for *UnregServer* and *RegServer*, and if detected the appropriate calls in _Module are made and bRun set to false. The command line is also checked for *"Automation"* and if there a bAutomation flag is set.

```
// check for UnregServer and RegServer
. . .
 if(lstrcmpi(lpszToken, _T("Automation")) == 0){
            bAutomation = true;
            break;
      }
```

The core of WinMain is as follows:

```
if(bRun){
      _Module.StartMonitor();
      hRes = _Module.RegisterClassObjects(CLSCTX_LOCAL_SERVER,
        REGCLS_MULTIPLEUSE | REGCLS_SUSPENDED);
      ATLASSERT(SUCCEEDED(hRes));
      hRes = ::CoResumeClassObjects();
      ATLASSERT(SUCCEEDED(hRes));
      if(bAutomation){
            CMessageLoop theLoop;
            nRet = theLoop.Run();
      } else {
            nRet = Run(lpstrCmdLine, nCmdShow);
      }
      _Module.RevokeClassObjects();
      ::Sleep(_Module.m_dwPause);
}
```

The Automation Command-line Parameter

Note that when the Automation flag is passed in on the command line then no application window is displayed. This is keeping with the convention of application starting through Automation not displaying any UI unless specifically asked to via an Automation API.

However, there is a problem here. When the COM run-time is asked to launch a server through a client's call to CoCreateInstance, the COM run-time retrieves the *"LocalServer"* value from the registry, appends the string *"-Embedding"* to create a command-line, and calls CreateProcess to launch a new process using this command line. As noted earlier, the WTL AppWizard (and the ATL AppWizard), generate the *LocalServer* string in the component's rgs file to be just %MODULE%, which gets expanded to the installed path to the EXE. The COM libraries will append *"-Embedding"* to this, but no one is appending *"/Automation"* and therefore by default the WTL-generated application, even when started via CoCreateInstance in a client, never even sees "/Automation" on the command-line. This is not a problem with ATL projects, because they are expected to store COM components and are designed to be always started this way. It is a problem with WTL projects, because they sometimes will be started directly by users - in which case they will immediately

display the UI, and sometimes via COM by client applications - in which case by default they do not wish to display the UI, and will need to provide a component with an Automation method for displaying the UI. This is in keeping with the convention of Automation, which allows a client to decide whether the server's UI should be displayed. Note that the application developer needs to write a component to display the UI – the WTL AppWizard generated code does not provide this.

Note that whether '-' or '/' is used to start a new command-line parameter is irrelevant. Most applications (including all ATL and WTL ones) search for both. The *"-Embedding"* command-line parameter is used to signify that the application should work as an ActiveX Document Server and is being started by an ActiveX Document Container. The *"/Automation"* command-line parameter is used to signify that the application has been launched by a client wishing programmatic access to the Automation components inside the applications. What effect those command-line parameters have on an application are of course entirely an internal matter for the application. ATL applications just ignore them, as they always assume there are run as components. WTL applications wish to display the user interface when run directly by the end-user, and not when programmatically started via Automation.

The solution is easy. Firstly one could check for Embedding rather than Automation, by changing the line:

```
else if(lstrcmpi(lpszToken, _T("Automation")) == 0)
```

for the line:

```
else if(lstrcmpi(lpszToken, _T("Embedding")) == 0)
```

Alternatively, one could add *"/Automation"* to the *LocalServer* entry in the .rgs file. With a DLL, there is no "command-line", and therefore for component servers housed inside DLLs, the InprocServer32 or InprocServer64 entry merely states the pathname to the DLL. With EXEs, there are command-lines, and when a process is started by `CoCreateInstance` it takes the FULL string in the LocalServer32 or LocalServer64 field, appends *"–Embedding"* to it, and calls the Win32/64 API `CreateProcess` and passes the command-line string as the second parameter. Specifically, LocalServer need not be just a path, it can be a path and command-line parameter, such as *"/Automation"*. So, a solution to the problem would be to change the .rgs file as follows:

```
LocalServer32 = s '%MODULE% /Automation'
```

However, this introduces another problem. The `FindOneOf` function is the WTL AppWizard generated code only checks for one parameter, and returns a pointer to the beginning of that parameter (first character after the '/' or'-') until the end of the entire command-line. (It does not copy any data). As "-Embedding" is automatically appended to the command-line, what `FindOneOf` returns is actually *"Automation –Embedding"* and therefore the line:

```
else if(lstrcmpi(lpszToken, _T("Automation")) == 0)
```

fails to find a match. One solution would be to change this to:

```
else if(lstrcmpi(lpszToken, _T("Automation -Embedding")) == 0)
```

and another (probably better) solution would be to change the `FindOneOf` function to change the next whitespace to a `NULL` character.

One final point before we leave the issue of the command line is what happens if an application only wishes to expose components with custom interfaces. There are no conventions regarding how the command-line should be set for this scenario, as normally the programmatic exposure of desktop applications is done purely via Automation. Think of the likes of Word 2000, which can be programming from virtually every Windows programming environment, precisely because it exposes an Automation API. However, if you really do need to have custom interfaces, and want to avoid the use of the *"/Automation"* command-line option, then you are free to pick you own, or simply rely on *"–Embedding"*.

Application Termination

The next item to consider is how the application (whether started via Automation or not!) gets shut down.

With normal ATL projects, the `_Module` instance maintains a variable called `m_nLockCnt`, which is incremented when components are instantiated and decremented when the instances terminates. When it hits zero, a win32 event is set, so that the main application code can shutdown.

With WTL applications, we might have one or more instances of components running, or windows only running, or a mixture of both. When should we shut down? The answer is when all windows and all component instances are terminated. When the application was started by the end-user, it could be that while running a client application programmatically created instances of components within the process, and so even when the end-user shuts down the last window (and thinks the entire application has disappeared), we should still keep the process running until the client released the component instance(s). When the application was started first by a client application via Automation, it could happen that a UI of that component is visible to end-users, and they might use a command such as "New Window" to edit another document within the same process. Alternatively, it could be that they try to launch the application again, and through certain tweaking with some code, you have it using the existing process if available. Later, when the programmatic client releases all its components, the process should not shutdown until the end-user windows are also shut directly by the end-user.

To manage all this WTL uses the `m_nLockCnt` count for both top-level windows AND component instances, and when the value hits zero this means the process can exit. `_Module` has methods `Lock` and `Unlock` to increment and decrement this value, and `GetLockCount` to returns its current value. As components are instances and terminate the normal ATL code with increment and decrement `m_nLockCnt` as needed.

What about WTL's windows? We saw in the implementation of the `Run` function in <Projectname>.cpp that there is call to `Lock` when we are displaying a window:

```
_Module.Lock();
```

When a window is closed (e.g. by selecting Exit from the File menu), then the `WM_DESTROY` message is detected and mapped to a call to a new method `OnDestroy` in the frame window class in mainfrm.h.

```
LRESULT OnDestroy(UINT /*uMsg*/, WPARAM /*wParam*/,
                  LPARAM /*lParam*/, BOOL& /*bHandled*/){
      // if UI is the last thread, no need to wait
      if(_Module.GetLockCount() == 1){
            _Module.m_dwTimeOut = 0L;
            _Module.m_dwPause = 0L;
      }
      _Module.Unlock();
      return 0;
}
```

This decrements the count. Note also it sets pauses to zero- the pauses are there for component code so that a process with no component instances will run for a short time longer, just in case a request comes in from a client to create a new instance – it is much faster doing so in an existing process, as opposed to launching a completely new process.

There is a slight bug in the WTL method, `CServerAppModule::Term`. It calls the Win32 function `CloseHandle` for `m_hEventShutdown`. However, this has already been closed in the `MonitorShutdown` function, and therefore the repeated call in `CServerAppModule::Term` results in an error - a *"First chance exception - Invalid Handle"* exception is thrown.

```
Void term(){
            if (m_hEventShutdown != NULL)
                  CloseHandle(m_hEventShutdown);
            CAppModule::Term();
      }
```

The solution is to either remove the `CloseHandle` call in `CServerAppModule::Term` or alternatively in `MonitorShutdown`, after calling `CloseHandle`, then set `m_hEventShutdown` to `NULL`.

Sample Using COM Server

To create a sample using the COM Server support in WTL, create a new project called HelloWorldSDI_WithCOMServer, and in the AppWizard in Step 1 select *"SDI Application"* and *"Act As A COM Server",* and select the defaults in step 2.

Now from DevStudio's Insert menu run the ATL Object Wizard, and insert a component called `CConversation` – with a dual-interface called `IConversation` (select "dual" in the Attributes property page). In DevStudio's ClassView, select `IConversation`, right click and select *"Add A Method"*, and in the displayed dialog add a method to `IConversation` called `Chat`. There is no need for parameters. Now in ClassView select `CConversation`, and directly beneath it, select `IConversation` and the `Chat` method – select it and the Conversation.cpp file should be displayed, showing a default implementation of `CConversation::Beep` (note: do not select the top-level `IConversation`, which refers to the IDL entry). Add this line to `CConversation::Beep`:

```
::Beep(200,200);
```

Fix the problem regarding the command-line Automation parameter, as discussed earlier. Then build.

Issues regarding registration of Universal Marshaler

Our WTL COM Server exposes an Automation Interface. So we need to build a simple WTL dialog-based client that calls the `IConversation` Interface. Run the WTL AppWizard again and in the dialog box add a push button.

#import the typelibrary which is embedded in the server's EXE.

```
#import "..\debug\HelloWorldSDI_WithComServer.exe"
rename_namespace("HelloLib")
using namespace HelloLib;
```

An aside here: Some developer dislike hard-coding a pathname into a source file. It can also be an annoyance if you are building the release version and first you have to build the debug version. You can avoid this by placing the typelibrary in a file on the OS search path or extending DevStudio's Options menu/Directories list. Alternatively, you could consider that its use as here is merely for test purposes, and that you might only be building the debug version of the test project. (Note that the type library that is produced is the same regardless whether it is the debug or release builds).

Add a command handler to be called when the pushbutton is selected.

```
      LRESULT OnChat(WORD /*wNotifyCode*/, WORD wID,
             HWND /*hWndCtl*/, BOOL& /*bHandled*/){
      IConversationPtr m_ConversationPtr;
      HRESULT hr =
       m_ConversationPtr.CreateInstance(__uuidof(Conversation));
   if (FAILED(hr))
            _com_issue_error(hr);
            if (m_ConversationPtr){
                  m_ConversationPtr->Chat();
            }
            m_ConversationPtr.Release();
            m_ConversationPtr = NULL;
            return 0;
      }
```

When we run our client app we note that unfortunately when it runs it reports an error when trying to connect to the WTL-generated application.

If we try creating the client in VB, we would start by displaying the VB References dialog, and trying to select our typelib from the list, which is generated from the registry's typelibrary entries. However, the typelibrary for our project does not appear in this list. Even when we add it manually using the browse button, when we later code up a call to the interface, it fails.

```
Private Sub Command1_Click()
Dim SH As New Conversation
SH.Chat
End Sub
```

What is going on here? What are the clients not working? The debugger will tell us the problem is when instantiating the server. Let's work our way back from the client to the server. The client and the server are in separate processes, so we need marshaling between them. We are using Automation, so we want to use type-library marshaling.

For this to happen, two important sets of registry entries are needed. Under the registry HKCR\Interfaces key, there should be an entry for the IID of the `IConversation` interface, and it should have a sub-key of *ProxyStubClsid32*. As we are using Automation and the OS provides use with a suitable proxy stub DLL known as the Universal Marshaler (with a CLSID of 00020424-0000-0000-C000-000000000046), its value should be used. Also, the typelib id should be registered, as this is needed for type-library marshaling. However, when we examine the registry neither of these sets of keys exist.

The reason the VB References dialog did not show the typelibrary for HelloWorldSDI_WithCOMServer was because its typelibrary was not registered.

Now the ATL experts will argue here that surely this cannot be. When we generate ATL out-of-proc servers based on Automation, these registry entries are correctly made. Yes, that is correct with ATL – and surely WTL is tied in closely with ATL? Hmmm…

Let us first examine what happens with an ATL EXE when "/register" is passed in on the command-line. For executables, the ATL AppWizard generates code will call `FindOneOf` and then has this code when the *RegServer* command-line parameter is found:

```
if (lstrcmpi(lpszToken, _T("RegServer"))==0) {
    _Module.UpdateRegistryFromResource(
                IDR_SimpleExeProject, TRUE);
    nRet = _Module.RegisterServer(TRUE);
    bRun = FALSE;
    break;
}
```

Note it calls the `CComModule::RegisterServer` method, with a `TRUE` parameter. The result of this is that the typelibrary and all interfaces are registered as needed, and clients can attach without further ado.

Now, when we examine the same code in the WTL AppWizard generated code, there is a slight difference.

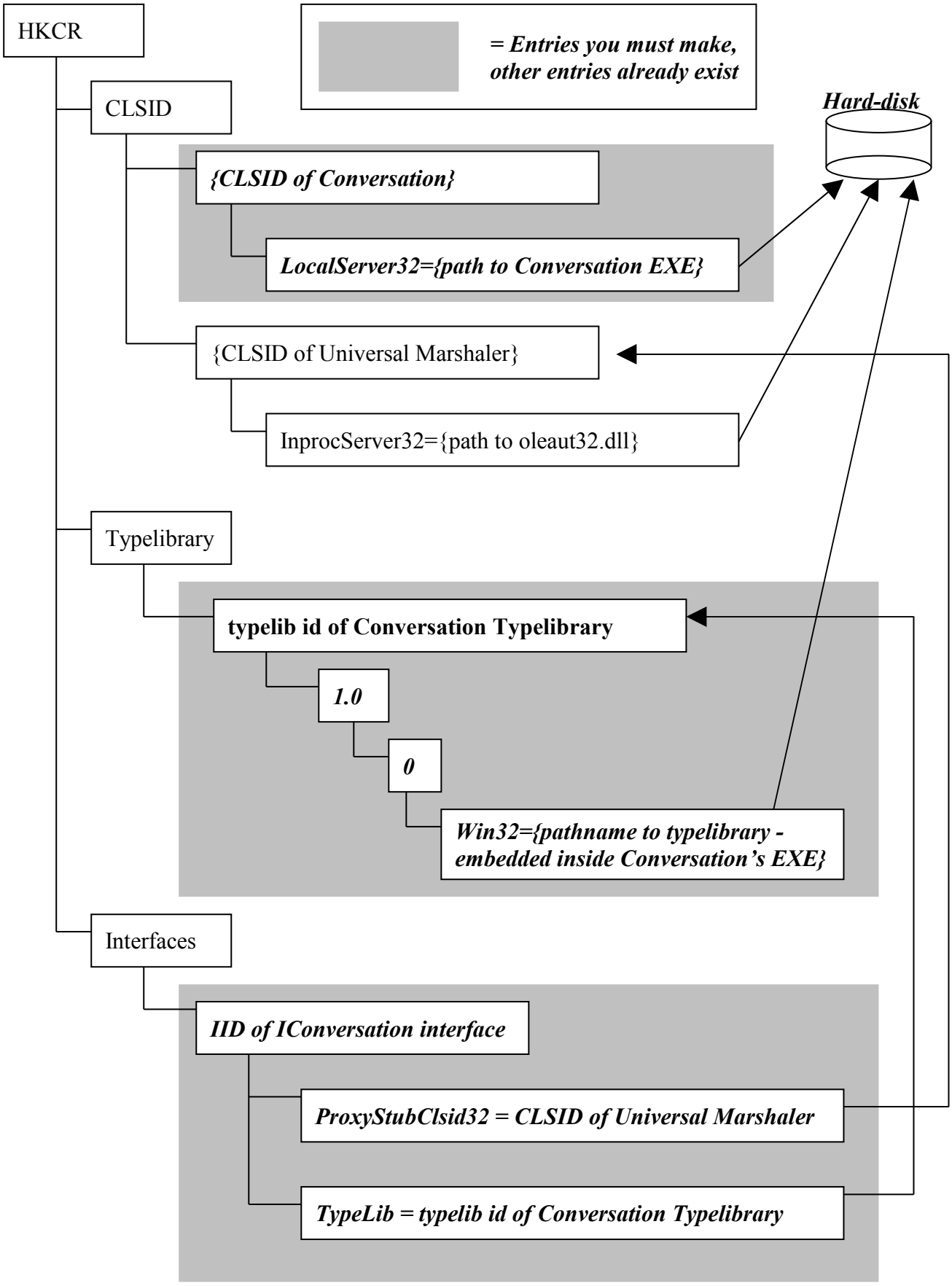
The RegServer handling code calls:

```
nRet = _Module.RegisterServer();
```

When we examine `CComModule::RegisterServer` we see it defaults the first parameter, `bRegTypeLib`, to `FALSE`.

```
HRESULT RegisterServer(BOOL bRegTypeLib = FALSE,
                       const CLSID* pCLSID = NULL)
```

Registry Layout for Local Servers supporting Automation



So now we see the difference between the ATL- and WTL-AppWizard generated code. What does the `bRegTypeLib` flag control? When set to true it results in a call to `::RegisterTypeLib`, which registers the interfaces and typelib entries necessary for Automation. When set to false, which is the case for WTL, these entries are not made. Note that regardless of the `bRegTypeLib`, the CLSID entries are made due to the entries in the .rgs file.

Assuming you wish to make the Interfaces and Typelib entries, all you have to do is change the call to *RegServer* in your WTL-AppWizard generated code as follows:

```
nRet = _Module.RegisterServer(TRUE);
```

Step 2 of the WTL AppWizard

Now we will move on to examine step 2 of the WTL AppWizard.

Note that if Dialog Based is selected in step (with or without the Modal Dialog check-box selected), all the options in step 2 are disabled. The following discussion assumes you have selected a project type other than dialog-based.

The default settings for Step 2 are as follows:

The available options can be subdivided into a number of sections.

Firstly, we can select whether we wish to have a toolbar, rebar, command bar or status bar in place.

Status Bar

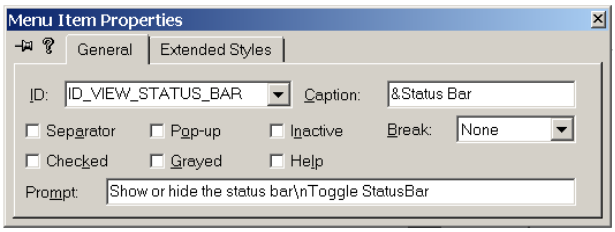
The status bar checkbox can be selected independently of the others. If selected, it causes a number of additions to the generated application.

A call to `CreateSimpleStatusBar` (with no parameter) is inserted in the Mainframe's `Create` method. Your Mainframe class is derived from WTL's `CFrameWindowImpl`, which in turn is derived from `CFrameWindowImplBase`.

`CreateSimpleStatusBar` is implemented in `CFrameWindowImplBase`. When called with no parameters it loads a string resource identified by `ATL_IDS_IDLEMESSAGE` (which is set to "Ready" in the generated resources) and calls an overloaded method, which in turn calls Win32's `CreateStatusWindow`, which create a status window as a child of the mainframe window, and records the status bar's `HWND` in a data member called `m_hWndStatusBar`.

```
      BOOL CreateSimpleStatusBar(UINT nTextID = ATL_IDS_IDLEMESSAGE,
DWORD dwStyle = WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN |
WS_CLIPSIBLINGS|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);
      }
      BOOL CreateSimpleStatusBar(LPCTSTR lpstrText, DWORD dwStyle =
WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN | WS_CLIPSIBLINGS |
SBARS_SIZEGRIP, UINT nID = ATL_IDW_STATUS_BAR)  {
            ATLASSERT(!::IsWindow(m_hWndStatusBar));
            m_hWndStatusBar = ::CreateStatusWindow(dwStyle, lpstrText,
m_hWnd, nID);
            return (m_hWndStatusBar != NULL);
      }
```

A menu item titled *Status Bar* is added to the View menu with these properties:



The mainframe message map is extended with this entry:

```
      COMMAND_ID_HANDLER(ID_VIEW_STATUS_BAR, OnViewStatusBar)
```

The `OnViewStatusBar` handler function is added.

```
      LRESULT OnViewStatusBar(WORD /*wNotifyCode*/,
          WORD /*wID*/, HWND /*hWndCtl*/, BOOL& /*bHandled*/){
            BOOL bNew = !::IsWindowVisible(m_hWndStatusBar);
            ::ShowWindow(m_hWndStatusBar, bNew ?
                        SW_SHOWNOACTIVATE : SW_HIDE);
            UISetCheck(ID_VIEW_STATUS_BAR, bNew);
            UpdateLayout();
            return 0;
      }
```

This line is also added to the `CMainframe::Create` method.

```
            UISetCheck(ID_VIEW_STATUS_BAR, 1);
```

The menu item has a check box beside it, and when selected, the status bar should be displayed. This line is initializing the checkbox to be set, as the status bar is initially displayed.

Finally it add this entry to the UPDATE_UI_MAP for the mainframe:

```
UPDATE_ELEMENT(ID_VIEW_STATUS_BAR, UPDUI_MENUPOPUP)
```

The UPDATE_UI_MAP is used by WTL to keep track of what it needs to update dynamically. Here it updates what is displayed in the status bar, but we can also use it to update the rendering of toolbar icons and menu items.

WTL also offers a sophisticated status bar class called CMultiPaneStatusBarCtrl. It supports a status bar that has sub-divisions. The WTL AppWizard does not provide extra support for its use, but it can easily be added manually. We will examine it is a later chapter.

Command Bars, Rebars and Toolbars

If the toolbar checkbox is un-selected, then the *Rebar* and *Command Bar* checkboxes are automatically disabled (hence to use a rebar or command bar you must have a toolbar). If the *Rebar* checkbox is un-selected, then the *Command Bar* checkbox is automatically disabled (hence to use a command bar you must have a rebar).

Think of a command bar as a super menubar that displays menu item text and alongside icons that share the same command id. It lives within a band inside a rebar control. The command bar takes those icons from a toolbar (the toolbar itself is also displayed in another band in the rebar). A command bar is not a Win32 control – it is something specific to WTL and involves quite an amount of code.

If you don't have a rebar, then it does not make sense to have a command-bar. If you don't have a toolbar, then there would be no icons for a command bar to display, and if you only have a single menubar then it does not make sense to have rebar or command bars.

If the *Command Bar* checkbox is selected, then the following occurs.

The CMainFrame class has this extra data member:

```
CCommandBarCtrl m_CmdBar;
```

These lines are added to CMainFrame::OnCreate:

```
// create command bar window
HWND hWndCmdBar = m_CmdBar.Create(m_hWnd, rcDefault,
       NULL, ATL_SIMPLE_CMDBAR_PANE_STYLE);
// attach menu
m_CmdBar.AttachMenu(GetMenu());
// load command bar images
m_CmdBar.LoadImages(IDR_MAINFRAME);
// remove old menu
SetMenu(NULL);
. . .
CreateSimpleReBar(ATL_SIMPLE_REBAR_NOBORDER_STYLE);
AddSimpleReBarBand(hWndCmdBar);
```

When the mainframe window is created it automatically loads up a menubar with the IDR_MAINFRAME menubar resource. What is happening in this command bar calling code is that it calls Win32's GetMenu API to get a HMENU of the loaded menubar.

Within the call to the command bar's `AppendMenu`, the contents of the existing menubar are copied to the command bar, and toolbar icons with the same command IDs are also added. When that is finished, we now have two copies of the menubar – one that was loaded by default and is the traditional menubar for the window, and the other which is in the commandbar. We only need one – hence the call to `SetMenu(NULL)`, which eliminates the existing menubar. The call to `AddSimpleReBarBand` adds the command-bar as a band to the rebar control. (To prove that two menubars do exist for a short time, simply comment out the `SetMenu(NULL)` call – when you run the application you will see the old menubar and the new commandbar).

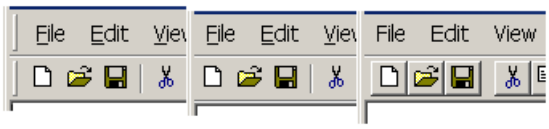
Note that the call to `CreateSimpleReBar` when command bars are supported has the parameter `ATL_SIMPLE_REBAR_NOBORDER_STYLE`. If the command bar checkbox is not selected in the WTL AppWizard, then the call to `CreateSimpleReBar` is passed no parameter, and it defaults to `ATL_SIMPLE_REBAR_STYLE`.

These two defines are:

```
#define ATL_SIMPLE_REBAR_STYLE \
      (WS_CHILD | WS_VISIBLE | WS_BORDER | WS_CLIPCHILDREN |
WS_CLIPSIBLINGS | RBS_VARHEIGHT | RBS_BANDBORDERS | RBS_AUTOSIZE)
#define ATL_SIMPLE_REBAR_NOBORDER_STYLE \
      (WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN | WS_CLIPSIBLINGS |
RBS_VARHEIGHT | RBS_BANDBORDERS |
RBS_AUTOSIZE | CCS_NODIVIDER)
```

The difference between them is one has `WS_BORDER` and the other has `CCS_NODIVIDER`.

The following three screen dumps shows an app with rebar and command-bar options selected (left), with the rebar option selected and command-bar option not selected (center) and with neither the rebar nor the command bar options selected (right).



When the command-bar is selected, then the rebar shows band dividers (those vertical-line grip marks at the front of each band) and shows no border. It is noted that the command-bar is a couple of pixels taller than the menubar, and that when only the rebar is selected that a border is displayed (beneath the toolbar). When neither the rebar nor the command bar options are selected, then a normal toolbar is displayed. Note that its icons have a border around each of them.

It is recommended for most applications to choose both the rebar and command bar options (this is the WTL AppWizard defaults settings). The extra command bar functionality is powerful and involves no extra work on the part of the application developer. End users will appreciate the extra functionality the command-bar provides over traditional menubars. However, command bars require the rebar control, and this is available in version 4.70 and later of Comctl32.dll, which is supported natively in Windows 2000/Windows 98 or later, but for Windows 95 and Windows NT 4 then Internet Explorer 3 or later needs to be installed. The vast majority (but not all) end-

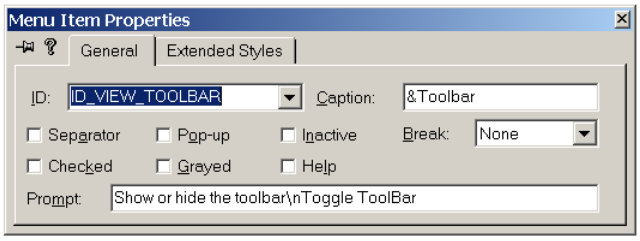
users with these older OSes also have IE3 installed. If your application must work on machines which do not IE3 installed, or if you wish to completely emulate the user interface of an old application, then you should un-select rebar and command bar support. The in-between choice, of selecting rebar but un-selecting command bar support, does not seem to be sensible in most situations – if you have rebars you might as well have command bars, which most end-users will appreciate – this choice would probably mean that the application developer intends to manually change some of the generated code for his/her specific purposes.

Now will explore what additions are made to the generate source code when these options are selected.

If you select toolbar support (by default it is selected), then the following additions are made.

The `OnIdle` method has a call to `UIUpdateToolBar` added.

The menubar resource has a toolbar menu item added beneath the View menu.



An entry is added to the `CMainFrame`'s message map:

```
COMMAND_ID_HANDLER(ID_VIEW_TOOLBAR, OnViewToolBar)
```

And a command handler is supplied:

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

This line is also added to the `CMainframe::Create` method.

```
      UISetCheck(ID_VIEW_TOOLBAR, 1);
```

The menu item has a check box beside it, and when selected, the status bar should be displayed. This line is initializing the checkbox to be set, as the toolbar in initially displayed.

Finally it add this entry to the `UPDATE_UI_MAP` for the mainframe:

```
      UPDATE_ELEMENT(ID_VIEW_TOOLBAR, UPDUI_MENUPOPUP)
```

If toolbar support is selected, but rebar support is not, then the toolbar creation code inside Mainframe's `Create` method is simply a call to:

```
CreateSimpleToolBar ();
```

If toolbar and rebar support is selected, but command bar support is not, then the code is:

```
      HWND hWndToolBar = CreateSimpleToolBarCtrl(m_hWnd,
       IDR_MAINFRAME, FALSE, ATL_SIMPLE_TOOLBAR_PANE_STYLE);
CreateSimpleReBar();
AddSimpleReBarBand(hWndToolBar);
```

If toolbar, rebar support and command-bar support are selected, then the code is:

```
      // create command bar window
HWND hWndCmdBar = m_CmdBar.Create(m_hWnd, rcDefault,
                  NULL, ATL_SIMPLE_CMDBAR_PANE_STYLE);
      // attach menu
      m_CmdBar.AttachMenu(GetMenu());
      // load command bar images
      m_CmdBar.LoadImages(IDR_MAINFRAME);
      // remove old menu
      SetMenu(NULL);
      HWND hWndToolBar = CreateSimpleToolBarCtrl(m_hWnd,
       IDR_MAINFRAME, FALSE, ATL_SIMPLE_TOOLBAR_PANE_STYLE);
      CreateSimpleReBar(ATL_SIMPLE_REBAR_NOBORDER_STYLE);
      AddSimpleReBarBand(hWndCmdBar);
      AddSimpleReBarBand(hWndToolBar, NULL, TRUE);
```

Use a View Window

Step 2 of the WTL AppWizard contains a checkbox with the title *Use a view window*. By default this is selected.

When selected, the generated code contains a view class, and the mainframe class has a data member which is an instance of the view class, and in `CMainframe::Create` a call is made to the view create member function.

The view is created as a child window of the mainframe window, with the `WS_EX_CLIENTEDGE`. The view occurs the client area in the mainframe – covering all the space beneath the toolbar and above the statusbar. The framewindow has a `m_hWndClient` data member, and this records the `HWND` of the view. When the framewindow is resized, WTL detects the `WM_SIZE` and resizes the view window as appropriately.

When the *Use a view window* checkbox is un-selected, the view type listbox in the AppWizard is disabled. Applications generated with or without a view look almost identical. The background of the view and the background of the frame's client area are displayed in the default background color (usually white). The only distinction is that with the view, one can see the edge generated by the `WS_EX_CLIENTEDGE` attribute, which is absent when the view is not there.

View Type

The view type combo-box specifies how the view should be created. It may be set to one of the following:

- Generic window (blank window – the default)
- Form (Dialog based)
- Listbox
- Edit
- Listview
- Treeview
- Rich-Edit
- HTML Page

Using the WTL AppWizard, you may not create a view based on a control not in this list and you may not create multiple views (but we will later examine how to do both of these manually). The mainframe class generated by the WTL AppWizard is not changed in any way when different view types are selected (with the exception of HTML Page).

When generic window view type is selected, then the view code is generated as follows:

```
class CWithEverythingView : public CWindowImpl<CWithEverythingView>{
public:
      DECLARE_WND_CLASS(NULL)
      BOOL PreTranslateMessage(MSG* pMsg){
            pMsg;
            return FALSE;
      }
      BEGIN_MSG_MAP(CWithEverythingView)
            MESSAGE_HANDLER(WM_PAINT, OnPaint)
      END_MSG_MAP()

      LRESULT OnPaint(UINT /*uMsg*/, WPARAM /*wParam*/,
      LPARAM /*lParam*/, BOOL& /*bHandled*/)    {
            CPaintDC dc(m_hWnd);
            //TODO: Add your drawing code here
            return 0;
      }
};
```

When a view based on a control is selected, then the code generated is:

```
class CViewBasedOnListViewView
 : public CWindowImpl<CViewBasedOnListViewView, CListViewCtrl>{
public:
      DECLARE_WND_SUPERCLASS(NULL, CListViewCtrl::GetWndClassName())
      BOOL PreTranslateMessage(MSG* pMsg){
```

```
            pMsg;
            return FALSE;
      }
      BEGIN_MSG_MAP(CViewBasedOnListViewView)
      END_MSG_MAP()
};
```

No `WM_PAINT` handler is required, as the control will draw itself when needed. The view window derives from `CWindowImpl` based on an appropriate control class. The `DECLARE_WND_SUPERCLASS` macro is passed the control's name. The listview, treeview, edit, listbox and rich-edit all work this way. If you wished to create a view based on a different control, you could simply make the appropriate change to one of the support control view types. The list that is supported contains the controls that are most frequently used as the basis of a view. Other controls might not be that suitable. For example, a pushbutton should not be the basis for a view.

Note that the version of rich edit used depends on the `_RICHEDIT_VER` macro setting, which is defaulted to `0x0100` in stdafx.h. Also note that the WTL distribution contains a good examine of RichEdit called MTPad.

When the view type is selected as *Form (dialog based),* then a dialog template is generated in the ResourceView with an id of `IDD_<projectname>_FORM` and the view class is generated as follows:

```
class CViewBasedOnFormView
       : public CDialogImpl<CViewBasedOnFormView>{
public:
      enum { IDD = IDD_VIEWBASEDONFORM_FORM };
      BOOL PreTranslateMessage(MSG* pMsg){
            return IsDialogMessage(pMsg);
      }
      BEGIN_MSG_MAP(CViewBasedOnFormView)
      END_MSG_MAP()
};
```

Note that the view is based on `CDialogImpl` and not `CAxDialogImpl`.

The difference between selecting an SDI/MDI app with its view type set to form, as here, and selecting a dialog box (in step 1 of the WTL AppWizard), is that the SDI/MDI app has a mainframe with command bars and statusbar.

When the view type is set to HTML Page, then the view will be based on the web browser. The view class will be generated as:

```
class CViewbasedOnHtmlPageView
   : public CWindowImpl<CViewbasedOnHtmlPageView, CAxWindow>{
public:
      DECLARE_WND_SUPERCLASS(NULL, CAxWindow::GetWndClassName())
      BOOL PreTranslateMessage(MSG* pMsg) {
            if((pMsg->message < WM_KEYFIRST || pMsg->message >
             WM_KEYLAST) && (pMsg->message < WM_MOUSEFIRST ||
             pMsg->message > WM_MOUSELAST))
            return FALSE;
            // give HTML page a chance to translate this message
```

```
            return (BOOL)SendMessage(WM_FORWARDMSG, 0, (LPARAM)pMsg);
      }
      BEGIN_MSG_MAP(CViewbasedOnHtmlPageView)
      END_MSG_MAP()
};
```

The view is based on `CAxWindow`. You will recall from our discussion of WTL Windowing, that `CAxWindow` will instantiate an ActiveX control to completely cover its window. Which ActiveX control depends on the window title – it could be a CLSID, a ProgID, in-line HTML (with the MSHTML prefix) or any string that CLSID_WebBrowser can understand, such as a URL, a pathname to a HTML or text file, or a pathname to a document from an Active Document Server (e.g. a Word 2000 document). The view window title is never displayed and normally when creating the view window, the WTL AppWizard uses a `NULL` window title. However, with Html Page, the title is used to indicate to the web browser what to render. The generated code is:

```
            //TODO: Replace with a URL of your choice
            m_hWndClient = m_view.Create(m_hWnd, rcDefault,
          _T("http://www.microsoft.com"), // ← change as needed
        WS_CHILD | WS_VISIBLE |
          WS_CLIPSIBLINGS | WS_CLIPCHILDREN, WS_EX_CLIENTEDGE);
```

ActiveX Control

The final part of step 2 of the WTL AppWizard is the *Host ActiveX Controls* checkbox. This is only enabled if <u>all</u> of the following conditions are true:

- The project type in step 1 is <u>not</u> set to *Dialog Based*
- The *Enable ActiveX Control Hosting* check-box in step 1 is selected
- The view type in step 2 is set to *Form (dialog-based)*

If you select *Host ActiveX Controls* then the generated view is based on `CAxDialogImpl` (otherwise it is based on `CDialogImpl`) and it is implemented as follows:

```
class CViewBasedOnFormWithActiveXControlView : public
CAxDialogImpl<CViewBasedOnFormWithActiveXControlView>{
public:
      enum { IDD = IDD_VIEWBASEDONFORMWITHACTIVEXCONTROL_FORM };
      BOOL PreTranslateMessage(MSG* pMsg) {
            if((pMsg->message < WM_KEYFIRST || pMsg->message >
             WM_KEYLAST) && (pMsg->message < WM_MOUSEFIRST ||
             pMsg->message > WM_MOUSELAST))
                  return FALSE;
            HWND hWndCtl = ::GetFocus();
            if(IsChild(hWndCtl))
            {
                  // find a direct child of the dialog from the
             // window that has focus
                  while(::GetParent(hWndCtl) != m_hWnd)
                        hWndCtl = ::GetParent(hWndCtl);
```

```
                // give control a chance to translate this message
                if(::SendMessage(hWndCtl, WM_FORWARDMSG, 0,
                        (LPARAM)pMsg) != 0)
                    return TRUE;
            }
            return IsDialogMessage(pMsg);
        }
        BEGIN_MSG_MAP(CViewBasedOnFormWithActiveXControlView)
        END_MSG_MAP()
};
```

Threads with COM and Windowing

Technically, there should be no problems using both the Win32 windowing APIs and COM together in a multithreaded application, as both can support calls from multiple threads provided suitable precautions are taken.

How COM reacts in a multi-threaded environment is guided by whether `CoInitialize` or `CoInitializeEx` is called, and in the case of `CoInitializeEx`, what parameter is used. The two interesting parameters to `CoInitializeEx` are `COINIT_APARTMENTTHREADED` and `COINIT_MULTITHREADED`. The first indicates that for COM components instantiated on one thread, all method calls to an instance of a COM component will be on the same thread in which instantiation occurred. This effectively means that data members of that instance do not need synchronization protection, as they can only be used from one thread. Separate instances of the same component may be instantiated on separate threads, so global variables and static (class) members do need protection. `COINIT_MULTITHREADED` means that a component instance may be used simultaneously from different threads. Such components should provide synchronization protection for all writeable data members. COM should only be used in a thread if `CoInitialize[Ex]` has been called on that thread. It is perfectly acceptable to call `CoInitialize[Ex]` from a subset of the available threads (even only one) and use other threads for non-COM functionality.

Calling `CoInitialize` is equivalent to calling `CoInitializeEx` with `COINIT_APARTMENTTHREADED`. `CoInitializeEx` is not available for the original ("classic") Windows 95 distribution, but is available when the DCOM add-on is installed with Windows 95. `CoInitializeEx` is available for all later Windows OSes – 98, 98SE, Me, NT 4, 2000, CE and Whistler. `CoInitialize` is not available for Windows CE – you must use `CoInitializeEx`.

Threads in which `CoInitializeEx(COINIT_APARTMENTTHREADED)` are called need a message loop, because COM uses a hidden window to ensure that incoming method calls are handled on the right thread at an appropriate time. Threads in which `CoInitializeEx(COINIT_MULTITHREADED)` are called do not need a message loop (for COM), because the incoming method calls are made on one of a queue of threads which the RPC run-time maintains. It is quite possible that the same or different clients make method calls to the same instance of a components, and these method call execute at the same time on different threads – the component needs to protect against synchronization problems. When dealing with WTL, we will have

windows on screen, so we will also need a message loop for that, but still the RPC threads are used for the methods calls.

The WTL AppWizard generated code is meant to run on all platforms, including the original Windows 95, and therefore the code contains a call to `CoInitialize`. There is also a comment containing a call to `CoInitializeEx`.

```
// If you are running on NT 4.0 or higher you can use the
// following call instead to make the EXE free threaded. This
// means that calls come in on a random RPC thread.
//    HRESULT hRes = ::CoInitializeEx(NULL,
//                             COINIT_MULTITHREADED);
```

It is left as a design decision to the application developer to decide, which is a sensible strategy.

With regarding to windowing, windows may be created on any thread. There is a message queue per thread and messages for a particular window are queued to the message queue of the thread in which the window was created. Calls to `SendMessage` may be make from a thread different to the thread upon which the window was created, but the thread that calls `SendMessage` will block until the message is actually processed by the thread that handles the message queue for the window. This can lead to deadlock situations in poorly designed applications. Calls to `PostThreadMessage` post the message on the message queue of the specified thread and returns immediately.

Sample using Multiple Threads SDI with COM Server

Now that we have examined the treading issues with windows and COM, and confirmed that one can have a multiple threads SDI application AND support COM, we will create a sample project.

Even though the WTL AppWizard does not directly create a multi-threaded SDI app that supports COM Server, we can still use it to help us. Essentially what we have to do is use it to create two separate projects – one which is based on *Multiple Threads SDI* without the *COM Server* option, and the other based on the (single-threaded) *SDI Application* with the *COM Server* option, and merge the code to complete our project.

We will start by creating a project called HelloWorldMT_WithCOMServer and in the WTL AppWizard select *SDI Application* with the *COM Server* option. We start with this as it contains most of the code we need. Next we create a dummy project, which we will discard, based on *Multiple Threads SDI.* It will help to have both of these projects open in separate instances of DevStudio, to facilitate copying and pasting.

The previous discussion of Multiple Threads SDI contained a detailed description of the differences between the (single-threaded) *SDI Application* and the *Multiple Threads SDI Application*. We need to apply all these changes manually to the HelloWorldMT_WithCOMServer project source code and make appropriate changes.

Firstly delete the generated Run function in HelloWorldMT_WithCOMServer.

Copy the `RunData` structure and the `RunThread` and `AddThread` member functions from the `CHelloWorldMTThreadManager` class in the dummy *Multiple Threads SDI Application* to HelloWorldMT_WithCOMServer.cpp.

Within `WinMain`, change the message loop code to the following:

```
        if(bRun) {
            _Module.StartMonitor();
            hRes = _Module.RegisterClassObjects(CLSCTX_LOCAL_SERVER,
                   REGCLS_MULTIPLEUSE | REGCLS_SUSPENDED);
            ATLASSERT(SUCCEEDED(hRes));
            hRes = ::CoResumeClassObjects();
            ATLASSERT(SUCCEEDED(hRes));
            CMessageLoop theLoop;
            if (!bAutomation)
                  AddThread(lpstrCmdLine, nCmdShow);
            nRet = theLoop.Run();
            _Module.RevokeClassObjects();
            ::Sleep(_Module.m_dwPause);
    }
```

In mainfrm.h, add a message map entry:

```
      COMMAND_ID_HANDLER(ID_FILE_NEW_WINDOW,
                                    OnFileNewWindow)
```

In the ResourceView, add a "New Window" entry to the file menu.

Add the command handler for `CMainFrame::OnFileNewWindow`, which calls `AddThread`.

```
      LRESULT OnFileNewWindow(WORD /*wNotifyCode*/, WORD /*wID*/,
       HWND /*hWndCtl*/, BOOL& /*bHandled*/){
                  AddThread(0, SW_SHOWNORMAL);
            return 0;
```

That's it – now we have a multi-threaded SDI app which can work as a COM server.

Chapter 6
Dialog Boxes and Controls

Objectives

The aims of this chapter are to:

- Cover how WTL supports dialog boxes
- Describe dynamic data exchange and validation functionality
- Introduce the WTL wrappers for each type of window control

Introduction

Virtually all applications that expose a user interface contain one or more dialog boxes, each showing multiple controls. A much smaller proportion of applications need to display windows with a client area in which graphical primitives can be rendered. We will start our in-depth look at WTL by examining how it handles dialog boxes and controls. In later chapters we will cover client-area graphics.

In WTL, the top-level window can be a dialog box itself, or can be a mainframe containing a child window known as a view, based on a dialog (resource template) or a single control. When the top-level window is not a dialog, it usually contains a command bar (menubar) and toolbar, whose entries when selected often result in dialog boxes being displayed.

Dialogs use an assortment of standard controls, common controls and ActiveX controls. Data must be copied between these controls and data members of C++ classes representing the dialogs. The resource information containing the layout for the dialog may be created using a resource editor. The dialog must be displayed on screen and when appropriate removed. Dialogs may be modal or non-modal. Property sheets and wizards may be provided. There are numerous system dialogs that the OS provides to automate and standardize how common UI tasks should work. Message loops are needed to process messages.

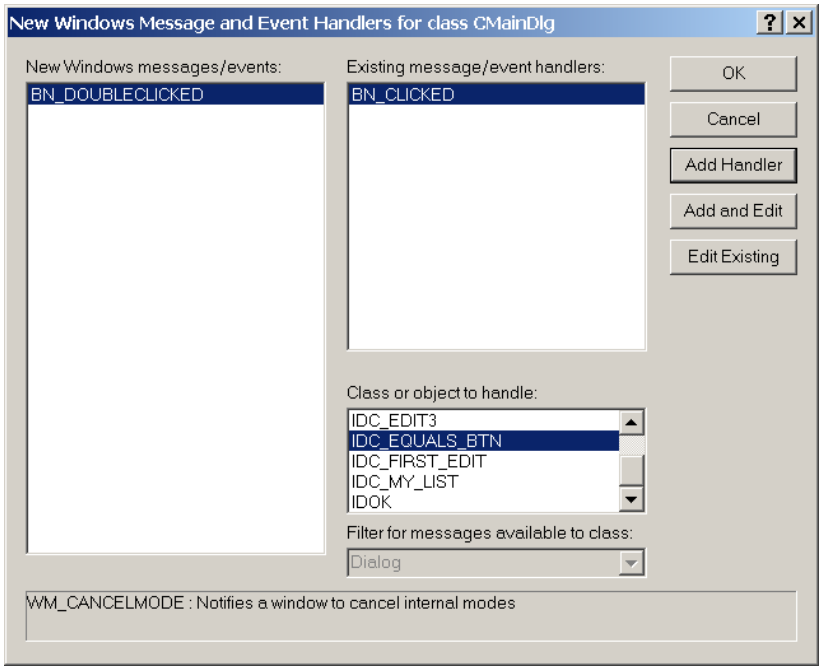
In our exploration of WTL dialogs and controls we will need to understand all these issues.

What WTL Supports

WTL provides comprehensive support for the complete range of dialog box functionality. It supports modal dialogs, modeless dialogs, and property sheets and provides easy-to-use wrappers for the system dialogs, such as `OpenFile`. WTL provides wrappers for each Win32 standard control, such as buttons, listboxes, edit boxes, and for each common control, such as listview, treeview, imagelist and

monthcalendar. WTL supports more advanced functionality, which is not directly based on existing controls, including bitmap buttons, hyperlinks and a wait cursor. WTL supports the hosting of ActiveX controls through the existing ATL mechanism. In short, every aspect of dialog boxes development is very well catered for by WTL.

One shortcoming is the integration between WTL and the development environment within Visual C++ v6. There is no WTL ClassWizard (the ClassWizard that is there is for MFC). What is currently provided is a *New Windows Message and Event Handler* wizard, which allows messages for `CWindowImpl`-derived classes to be mapped to function handlers.



This was designed for ATL windowing and can also be used with WTL, as WTL is based on ATL windowing. It supports mapping all the `WM_XXX` messages for the dialog itself to member function, and for each control id, mapping a subset of notifications that they send to their parent (the dialog). The handler functions it creates are generic (they all have the same prototype). WTL offers message crackers, (e.g. when a `lParam` contains a `POINT` data, a WTL `CPoint` is used in the prototype). These are not supported by the current version of the wizard. It is likely with the next version of Visual Studio that the development environment will be extended with more Wizard support for WTL.

The WTL Message crackers refer to the `WM_XXX` messages only. They do not provide any special functionality for the notification of a `WM_COMMAND` message. We will examine them in a later chapter.

The concept of mapping UI controls to C++ classes (similar to the *Member Variables* tab of the MFC ClassWizard) is currently not supported in WTL. At the moment this has be done manually in source code.

How WTL works with Dialog boxes and Controls

To prepare to learn about dialog development with WTL, you first need a good understanding of traditional Win32 UI programming (from the "Petzoldian" era of Windows programming) and you need to know about ATL windowing. This might be a good point to review the earlier chapters on these topics. WTL uses both of these extensively, and most of WTL's support is a set of lightweight wrappers and more functionality in some specific areas.

What WTL adds is a set of wrappers for each Windows standard control and each Windows common control. WTL provides a member function for absolutely every message that can be sent to these controls. The naming of the member functions of these wrappers is an exact match for the messages that each control accepts (just one reason for the usefulness of a clear understanding of Win32 UI programming). In contrast, MFC sometimes renamed certain features or amalgamated them. For the dialog box template itself and message mapping, WTL uses ATL windowing. MFC provided elaborate message routing, which provided additional functionality but which often caused as many problems as it solved. In contrast, WTL/ATL windowing message routing is very simple and easy to understand. It is trivial to follow it in a debugger.

Varying Degrees of Interaction with Controls

Application developers need different levels of interaction with controls.

It could be applications only need to determine when a specific message is received. For example, when a pushbutton is pressed its parent window (usually the dialog window), is sent a `WM_COMMAND` message with the `BN_CLICKED` notification. To detect these WTL provides the concept of message maps, which maps incoming messages to handlers within the dialog class.

Alternatively, it could be that applications need to set and get data stored in the control. For example, when displaying a dialog, developers might wish to set the text to appear in an edit box, and when the dialog disappears to retrieve (the possibly updated) text. For this, WTL provides DDX functionality.

Finally, it could be that applications wish to manage the control in detail. For this, WTL provides wrapper classes, such as `CEdit`, which provides the necessary fine-grained access to the control's properties.

WTL provides good support for all these levels of interaction with controls. It is a design decision developers will need to take as to which technique to use. Most applications will use all of them at some stage.

Dialog Boxes in WTL

Dialogs within WTL are based directly on ATL's `CDialogImpl` or `CAxDialogImpl`. The "`Ax`" term means that the dialog provides the necessary support to host ActiveX controls. The following discussion of working with dialogs applies equally well to `CDialogImpl` and `CAxDialogImpl`-based dialogs. In the chapter on ATL Windowing we looked at the specifics of ActiveX controls hosting within `CAxDialogImpl`.

WTL is a thin layer above ATL and Win32 SDK

As with other parts of WTL, its dialog box support is a very thin layer above what is provided by ATL and the Win32 SDK. If you read through the WTL source code you will very quickly hit the Win32 SDK.

WTL AppWizard-Generated Dialogs

When you run the WTL AppWizard and as the *Project Type* select the *Dialog* option, the generated code contains one dialog that acts as the main window of the application. By default, the generated dialog is based on `CDialogImpl`. If you select the *ActiveX Control Hosting* option in the step 1, the dialog is based on `CAxDialogImpl`. Note that when you select *Dialog* as the *Project Type* in step 1 of the WTL AppWizard, then all options in Step 2 are disabled.

If you select either a *SDI* or *MDI* application for *Project Type*, and in step 2 of the WTL AppWizard, select *Form (Dialog-based)* as the *View Type*, then by default the generated view is based on `CDialogImpl`. However, if the *Host ActiveX Controls* checkbox in step 2 of the WTL AppWizard is selected, then the view is based on `CAxDialogImpl`. Note that *Host ActiveX Controls* checkbox is normally disabled. It becomes enabled only when all three of these conditions are true: *Project Type* in step 1 is not set to *Dialog*, *ActiveX Control Hosting* is selected in step 1 and the *View Type* in step 2 is set to *Form (Dialog-Based).*

Creating Additional Dialog Boxes Manually

Apart from the WTL AppWizard generated dialog, you must create additional dialogs based on `CDialogImpl` manually. There is no wizard support. With Visual Studio 7 it is expected that wizard support will be provided. The current ATL Object Wizard does support the creation of `CAxDialogImpl`-based dialogs but not `CDialogImpl`.

To add a `CDialogImpl`-based dialog, you need to create the dialog resource and a C++ class to manage the dialog. To create the resource, display ResourceView in DevStudio's workspace, select *Dialog,* and in the context menu select *Insert Dialog*. A dialog template will be added to the resource. [The term "template" used for resources refers to the layout of the controls – it is of course a totally distinct concept from a C++ template]. Use the ResourceView's control palette to add controls onto the dialog and then set their various properties. Specify an IDD identifier for the dialog. To construct a C++ class, you will need to manually create a class with code similar to this:

```
class CMyFirstManualDlg:public CDialogImpl<CMyFirstManualDlg >{
public:
      enum { IDD = IDD_MY_FIRST_MANUAL_DIALOG };
      BEGIN_MSG_MAP(CMyFirstManualDlg)
            MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
            COMMAND_ID_HANDLER(IDOK, OnOK)
            COMMAND_ID_HANDLER(IDCANCEL, OnCancel)
      END_MSG_MAP()
      LRESULT OnInitDialog(UINT /*uMsg*/, WPARAM /*wParam*/,
                  LPARAM /*lParam*/, BOOL& /*bHandled*/){
            // center the dialog on the screen
            CenterWindow();
```

```
            return TRUE;
      }
LRESULT OnOK(WORD /*wNotifyCode*/, WORD wID, HWND /*hWndCtl*/,
             BOOL& /*bHandled*/){
            // TODO: Add validation code
            EndDialog(wID);
            return 0;
      }
LRESULT OnCancel(WORD /*wNotifyCode*/, WORD wID,
       HWND /*hWndCtl*/, BOOL& /*bHandled*/){
            // TODO: Add validation code
            EndDialog(wID);
            return 0;
      }
};
```

The enum IDD must be set to that you used in ResourceView. Also note that you must have a message map. This will implement a virtual function `ProcessWindowMessage`, which is needed to get everything to compile.

These are tedious manual steps; after you have completed them for one dialog, the use of CTRL-C and CTRL-V mean it does not take too long, and hopefully VS7 will automate this.

Instantiating and Displaying Dialogs

Just as with MFC, one instantiates and displays a WTL dialog using this code:

```
CMyFirstManualDialog dlg;
int res = Dlg.DoModal();
```

To dismiss a dialog (usually after the user selecting *OK* or *CANCEL*), the dialog class itself will call `EndDialog`, and the parameter passed to this will become the return value from `DoModal`.

Dialog Initialization

When a dialog class is instantiated, the windows for the dialog and its control do not exist. A common mistake for new MFC developers and it applies equally to WTL is to have code such as:

```
CMyFirstManualDialog dlg;
Dlg.m_edit.SetWindowText(TEXT("My initial text")); // WRONG!
int res = Dlg.DoModal();
```

The problem here is that the edit box does not exist in the hierarchy of windows until during the `DoModal` call and therefore there is no valid destination from the window message resulting from the call to `SetWindowText`.

The proper way to handle this situation is to execute the code after the window has been created but before it has been displayed on screen. The OS send a `WM_INITDIALOG` message to the dialog when at exactly this point in time, and we can detect it with a message map entry:

```
      MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
```

So the solution is as follows:

```
CMyFirstManualDialog dlg;
int res = Dlg.DoModal();
. . .
class CMyFirstManualDialog : CDialogImpl<CMyFirstManualDialog >{
     BEGIN_MSG_MAP(CMyFirstManualDlg)
          MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
     END_MSG_MAP()
     LRESULT OnInitDialog(. . .){
          m_edit.SetWindowText(TEXT("My initial text"));
          return TRUE;
     }
. . .
};
```

If the data to use when initializing the controls is known where we instantiate the dialog, and not inside the dialog class, then we could pass it in as a constructor parameter or add an extra method which gets called before `DoModal`.

```
CMyFirstManualDialog dlg;
Dlg.m_strName = TEXT("My name"):
int res = Dlg.DoModal();
. . .
class CMyFirstManualDialog : CDialogImpl<CMyFirstManualDialog >{
CString m_strName;
. . .
     LRESULT OnInitDialog(. . .){
          m_edit.SetWindowText(m_strName);
          return TRUE;
     }
```

Dialog Termination

Dialogs terminate when a call is made to `CDialogImpl<T>::EndDialog`. It is up to you code to decide when to call this method. There is no default handling of `WM_COMMAND` messages such as `IDCANCEL`, which is sent to the dialog when the close button ('X' symbol) in the dialog's top corner is pressed, or (by default) when the ESC key is pressed. This contrasts to MFC's implementation, which in its default message map maps the `IDCANCEL` message to a call to MFC's `CDialog::OnCancel`, which results in `EndDialog` being called.

With WTL/ATL, if you do not handle the `IDCANCEL` message then nothing will happen – specifically the dialog box will not disappear, even when you pressed the *Close* button.

It is the responsibility of your code to call `EndDialog`, and therefore you will usually add *OK* and *Cancel* pushbuttons, with ids of `IDOK` and `IDCANCEL` and map these to appropriate message handlers that will call `EndDialog`. Even if you do not have a *Cancel* button, you should still add a command handler for `IDCANCEL`, as the *Close* Button and the *ESC* key cause `IDCANCEL` to be sent to the dialog.

Handling Dialog Messages

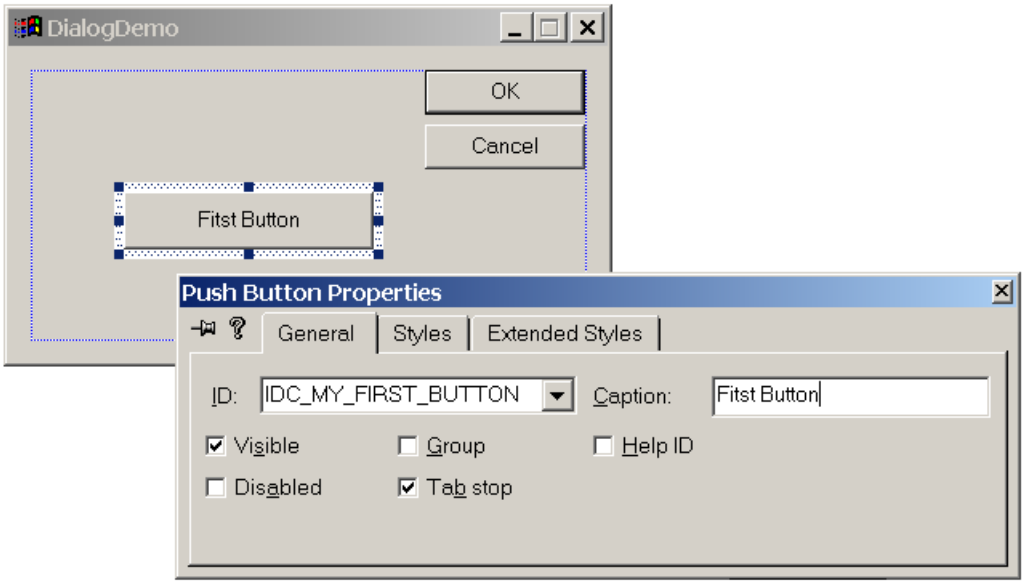
DevStudio's *New Window Message and Event Handler* wizard enables you to attach command handlers to a variety of messages for the dialog itself and all the controls it contains. This is similar to the message map feature in MFC.

Per selected window/control, the list of possible messages is displayed, and you can select the *Add Handler* button to create a new command handler for that message.

Sample : DialogDemo

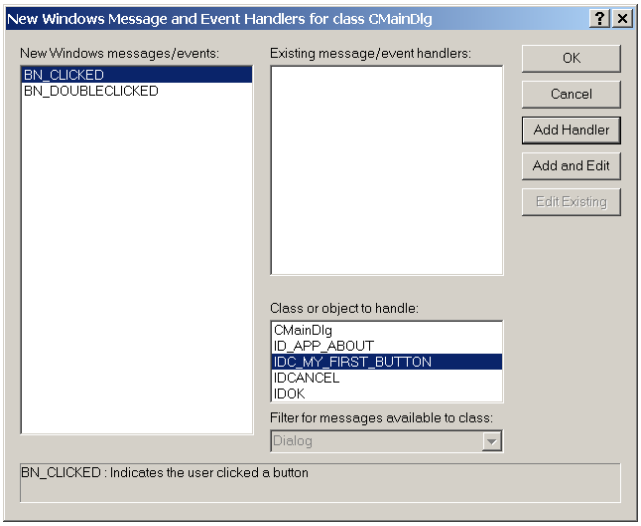
To explore dialogs, we will now create an example. Start up the WTL AppWizard and create a project called Dialogdemo. In Step 1, select modal dialog, and select the defaults in Step 2.

In the dialog node in ResourceView, select the DialogDemo. Using the Controls palette, add a pushbutton to the dialog. Select the pushbutton and from the context menu bring up its properties. Set the id to `IDC_MY_FIRST_BUTTON` and give it a caption of "First Button".



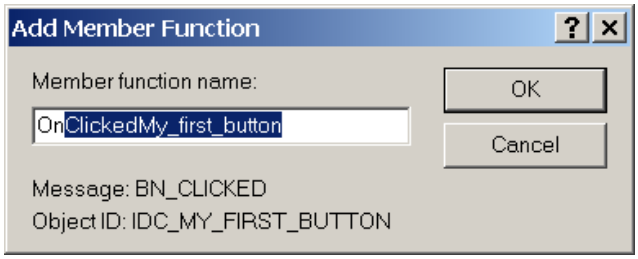
We are now finished adding the button to the resource, and can move on to creating a handler to be activated when the button is pressed.

In DevStudio's ClassView, select the `CMainDlg` class which manages the dialog. In the context menu for it select *Add Windows Message Handler*. The following dialog will be displayed.



In the Class or object to handle list, select `IDC_MY_FIRST_BUTTON`. The messages that are most commonly needed for a window of the selected type are displayed. In this case, a button is selected, and the messages are `BN_CLICKED` and `BN_DOUBLECLICKED`. Select *BN_CLICKED* and then select *Add Handler*.

You are given an opportunity to enter a name for the handler function, but normally you should select the default.



The message map will automatically have this line added:

```
COMMAND_HANDLER(IDC_MY_FIRST_BUTTON, BN_CLICKED,
                OnClickedMy_first_button)
```

A default implementation of the handler will also be added, and you can add a call to Win32's `Beep` function.

```
LRESULT OnClickedMy_first_button(WORD wNotifyCode, WORD wID,
             HWND hWndCtl, BOOL& bHandled){
            ::Beep(200,200);
            return 0;
      }
```

DDX

Passing data in and out of dialog controls is very frequently needed. It can be done using low-level window messages, but that involves many lines of code if done manually. Naturally developers wished to minimize the effort involved, and out of this grew the concept of dynamic data exchange (DDX). MFC developers will be quite familiar with DDX, as it has been used there for many years. WTL's DDX support is

modeled on MFC's support. These are some differences, but in general, if you understand DDX in MFC you will very quickly understand WTL's also.

To get started with DDX in a WTL application you will need to carry out a few manual steps. Firstly you will have to include atlddx.h in your project. Adding it to stdafx.h is the recommended location. Then you will have to derive your dialog class from WTL's `CWinDataExchange`, set up a DDX Map, call `DoDataExchange` when needed and finally think about error handling. It is likely that in future the wizards will be updated to automate the necessary support.

CWinDataExchange

The heart of WTL's DDX support is a WTL template called `CWinDataExchange`. It contains a method called `DoDataExchange` that the application has to over-ride (usually using a DDX Map). This contains the list of dialog-specific exchanges needed. `CWinDataExchange` also contains methods called `OnDataExchangeError` and `OnDataValidateError` for error handling, and the application may over-ride these if desired. The bulk of the methods inside `CWinDataExchange` are methods to transfer data in and out of controls. These include:

- `DDX_Text` – text based on `LPTSTR`, `BSTR`, `CComBSTR` & WTL's `CString`
- `DDX_Int` – integer
- `DDX_Float` – float and double
- `DDX_Control` – control sub-classing (nothing to do with ActiveX controls)
- `DDX_Check` – check box
- `DDX_Radio` – group of radio buttons

You derive your dialog box's C++ class from the `CWinDataExchange` template and this gives you access to these methods. Each DDX method in `CWinDataExchange` takes in a Boolean parameter, which specifies whether a save or a load operation is to be carried out.

Note that WTL's `CWinDataExchange` does contain a default implementation of `DoDataExchange` – when called it simply asserts and throws you into the debugger. If you omit the DDX map then this will happen. To solve the problem you will simply have to add an appropriate DDX map.

DDX Maps

Instead of calling the data transfer methods, such as `DDX_Int`, directly, WTL provides the concept of DDX Maps. One of these maps should exist for each dialog class that uses DDX. The DDX Map implements the `DoDataExchange` function and its essential role is to map the control IDs of dialog controls to C++ data members. Data is transferred from the controls to the C++ data members when `DoDataExchange` is called with a `TRUE` parameter, and in the reverse direction when called with `FALSE`.

Calling DoDataExchange

The application developer needs to decide when to call `DoDataExchange`. Unlike MFC, it is not done automatically for you.

The controls must actually exist when you call `DoDataExchange`. Typically you will add a call to `DoDataExchange(FALSE)` to `OnInitDialog`, and a call to `DoDataExchange(TRUE)` to `OnOK`.

Sample : DataExchangeDemo

To show a simple example of how to use WTL's data exchange, create a project called `DoDataExchange`. Add this line at the bottom of stdafx.h:

```
#include <atlddx.h>
```

In the dialog resource, add a text box with an id of `IDC_MYFIRST_INT`.

Change the definition of `CMainDlg` so it derives from `CWinDataExchange`:

```
class CMainDlg : public CDialogImpl<CMainDlg>,
                 public CWinDataExchange<CMainDlg>
```

Add an integer data member to the `CMainDlg` class:

```
int m_num;
```

Add a data map with an entry mapping m_num data member to the text box in the dialog:

```
BEGIN_DDX_MAP(CMainDlg)
      DDX_INT(IDC_MYFIRST_INT, m_num)
END_DDX_MAP()
```

Put calls to `DoDataExchange` in `OnInitDialog` and `OnOK`.

```
LRESULT OnInitDialog(. . .){
      . . .
      m_num = 43; // initialize m_num here
      DoDataExchange(FALSE); // FALSE means copy TO the dialog
      return TRUE;
}
LRESULT OnOK(){
      DoDataExchange(TRUE); // TRUE means copy FROM the dialog
      // use m_num here
      EndDialog(wID);
      return 0;
}
```

In `OnInitDialog`, the data is transferred from the C++ data member to the dialog controls using the call to `DoDataExchange`. This means that the C++ data member values may be set before the call as needed. As they are only C++ data members, then may also be set where the dialog class is instantiated.

```
CMainDlg dlg;
Dlg.m_num = 20;
Dlg.DoModal();
```

The actual window for the control does not need to exist at this point. It does need to exist when `DoDataExchange` is called.

If you wish to carry out data exchange for a particular control, you can pass its id as the second parameter to `DoDataExchange`. If our `DDX_MAP` had many entries and for some reason we only wished to perform data exchange for the `IDC_MYFIRST_INT` control, we could call `DoDataExchange` as follows:

```
DoDataExchange(TRUE, IDC_MYFIRST_INT);
```

DDX With Integers

The member function `CWinDataExchange::DDX_Int` is responsible for handling integer exchange. Its prototype is:

```
template <class Type> BOOL DDX_Int(UINT nID, Type& nVal,
       BOOL bSigned, BOOL bSave, BOOL bValidate = FALSE,
       Type nMin = 0, Type nMax = 0);
```

At a minimum, its needs a control's id and variable as parameters.

The macros to put inside the DDX map which using `DDX_Int` are:

- `DDX_INT(nID, var)`: exchanges signed integer values, no validation
- `DDX_INT_RANGE(nID, var, min, max)`: exchanges signed integer values and for validation ensures it is between min and max (the max and min values themselves are valid in the range)
- `DDX_UINT(nID, var)`: exchanges unsigned integer values, no validation
- `DDX_UINT_RANGE(nID, var, min, max)`: exchanges unsigned integer values and for validation ensures it is between min and max (the max and min values themselves are valid in the range)

For each transfer method in `CWinDataExchange`, there are multiple DDX macros, which call it using various combinations of parameters.

If validation is turned on (i.e. using the `DDX_INT_RANGE` or `DDX_UINT_RANGE` macros), then when loading `var` must be between `nMin` and `nMax` (inclusive) – this is enforced by an assertion; when saving if `var` is not in the `nMin/nMax` range then the `OnDataValidateError` method is called. Note the difference between loading and saving – with loading the condition must be true – if false (and running in debug mode) an assertion is activated, whereas with saving the condition may be false and if so causes a call to the error handler but the application continues running.

DDX with Checkboxes and Radio-Buttons

The DDX member functions for check boxes and radio buttons are:

```
void DDX_Check(UINT nID, int& nValue, BOOL bSave)
void DDX_Radio(UINT nID, int& nValue, BOOL bSave)
```

For the checkbox, the valid values are 0, 1 and 2, which correspond to unchecked, checked and indeterminate (for tri-state buttons). When transferring data to the check

box control, if the value is not 0,1 or 2, then `ATLTRACE2` is called to print out an error message in the output window – note that `OnDataExchangeError` is not called. When loading, the `nValue` must be `0`, `1` or `2` – this is enforced by an assertion.

For a radio button group, the members of the group are delimited by the control id of the first radio button in the group, and the next control with the `WS_GROUP` attribute set (this is one past the end). You can arrange the ordering of controls by selecting tab order from the layout menu in resource view. You will need to set the `WS_GROUP` property for the first radio button in the group and for the control one past the last radio button in tab order.

If there are members of the group that are not radio-buttons (this should not be the case), then this message is sent to the output window

```
ATL: Warning - skipping non-radio button in group.
```

Only one radio button in the group can be selected and the control id parameter must be that of a radio button control (both of these should be the case) – this is enforced by assertions.

The DDX macros are:

- `DDX_CHECK(nID, var)` : exchanges between an unsigned integer and a check box
- `DDX_RADIO(nID, var)` : exchanges data between an unsigned integer and radio button group

DDX with Floating Point Values

The DDX member functions for floating point values are:

```
BOOL DDX_Float(UINT nID, float& nVal, BOOL bSave,
   BOOL bValidate = FALSE, float nMin = 0.F, float nMax = 0.F)

BOOL DDX_Float(UINT nID, double& nVal, BOOL bSave,
   BOOL bValidate = FALSE, double nMin = 0., double nMax = 0.)
```

Both floats and doubles are supported. At a minimum a control id and a data member are needed as parameters.

The macros for floats/doubles are:

- `DDX_FLOAT(nID, var)` – data exchange only (no validation)
- `DDX_FLOAT_RANGE(nID, var, min, max)` : data exchange and validation

The floating point functionality requires the C-Run-Time library. If you wish to use floating point DDX you will need to add a definition to the preprocessor of `_ATL_USE_DDX_FLOAT` to both the debug and release build settings, and remove the definition of `_ATL_MIN_CRT` from the release build settings (it was put these automatically by the WTL AppWizard). You cannot have both of these defined in the preprocessor – as one is saying *include the CRT* and the other is saying *exclude the CRT*. If you forget, these lines in atlddx.h will remind you:

```
#if defined(_ATL_USE_DDX_FLOAT) && defined(_ATL_MIN_CRT)
      #error Cannot use floating point DDX with _ATL_MIN_CRT defined
#endif //defined(_ATL_USE_DDX_FLOAT) && defined(_ATL_MIN_CRT)
```

If validation is turned on (i.e. using the `DDX_FLOAT_RANGE` macro), then when loading the following must be true (this is enforced by an assertion):

```
nVal >= nMin && nVal <= nMax
```

When saving if

```
nVal < nMin || nVal > nMax
```

is true, then the `OnDataValidateError` method is called.

Similarly to integer processing, if validation is turned on (i.e. using the `DDX_FLOAT_RANGE` macro), then when loading `var` must be between `nMin` and `nMax` (inclusive) – this is enforced by an assertion; when saving if `var` is not in the `nMin/nMax` range then the `OnDataValidateError` method is called. Also an assertion ensures that `nMin` is not equal to `nMax`.

DDX with Strings

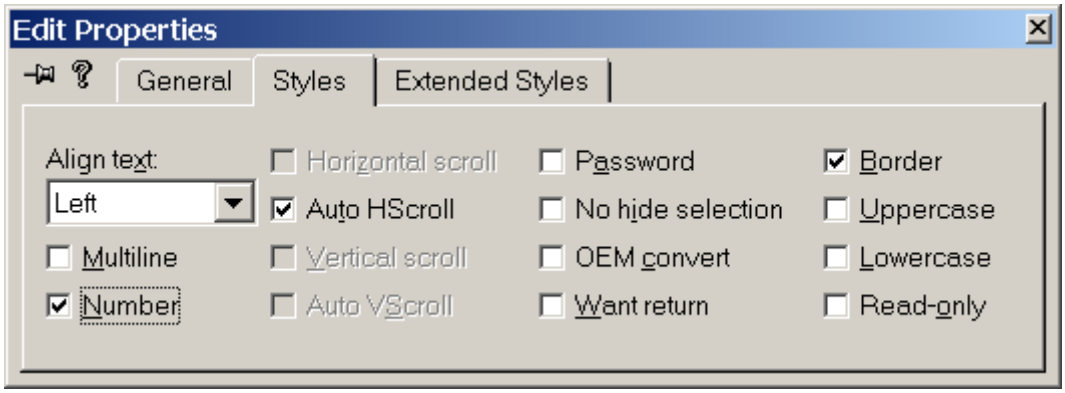
`CWinDataExchange` offers a number of member functions dealing with data exchange for text. They are:

```
BOOL DDX_Text(UINT nID, LPTSTR lpstrText, int nSize, BOOL bSave,
       BOOL bValidate = FALSE, int nLength = 0);
BOOL DDX_Text(UINT nID, BSTR& bstrText, int /*nSize*/,
       BOOL bSave, BOOL bValidate = FALSE, int nLength = 0);
BOOL DDX_Text(UINT nID, CComBSTR& bstrText, int /*nSize*/,
       BOOL bSave, BOOL bValidate = FALSE, int nLength = 0);
BOOL DDX_Text(UINT nID, CString& strText, int /*nSize*/,
       BOOL bSave, BOOL bValidate = FALSE, int nLength = 0);
```

The string data types they support are `LPTSTR`, `BSTR`, `CComBSTR` and `WTL::CString`.

Note that `WTL::CString` is only supported if `_ATL_TMP_NO_CSTRING` is NOT defined (and usually it is not), and if the header file atlmisc.h is included.

Each of these exchange functions transfer string data to a textbox control. The optional validation is the length of the string. If the string is to accept numeric characters only, then the *NUMBER* property should be set in the resource editor for the text-box.



The DDX macros for text are:

- `DDX_TEXT(nID,var)` - data exchange between text box and string data member (no validation)
- `DDX_TEXT_LEN(nID, var, len)` - data exchange between text box and string data member, and also validation of string length

The var field can be one of `LPTSTR`, `BSTR`, `CComBSTR` or `CString` and depending on its data type the appropriate `DDX_Text` member function will be called.

If saving and validation is turned on, then the length field must be greater than zero (this is enforced by an assertion).

If loading, and validation is turned on, and the data or type is one of `LPTSTR`, `BSTR` or `CComBSTR` (but not `CString`), then the number of characters in the string must be greater than the `nLength` parameter (this is enforced by an assertion).

DDX For Control Sub-Classing

To support subclassing of controls, `CWinDataExchange` offers the `DDX_Control` method. It simply calls the `CWindowImplBaseT< TBase, TWinTraits >::SubclassWindow`, which we discussed in the coverage of ATL Windowing.

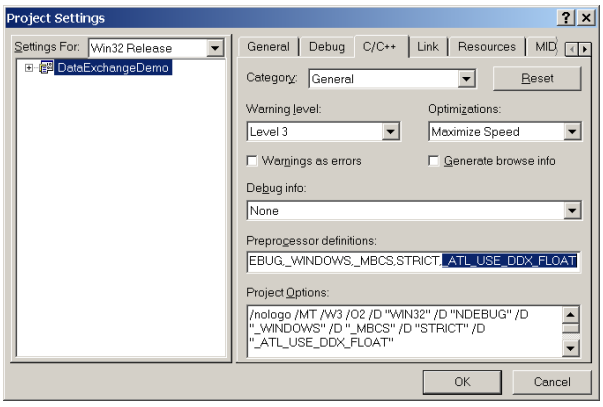
One macro is provided to support subclassing:

- `DDX_CONTROL(nID, Control)` – the control class subclasses the control window in the dialog box

Sample: CallMacrosDlg in DataExchangeDemo

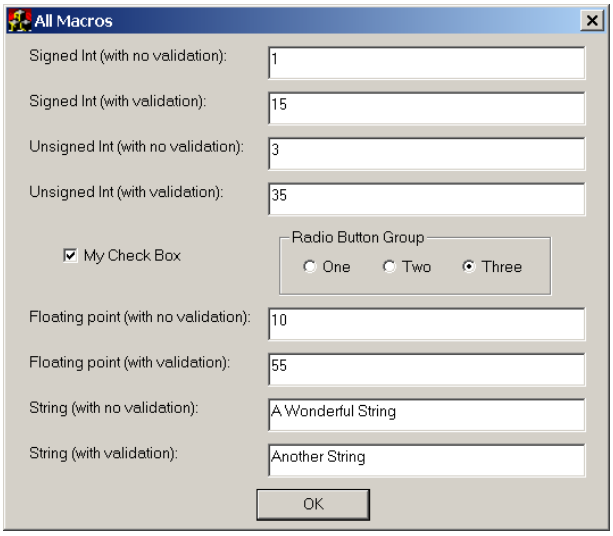
We will now extend the DataExchangeDemo with a dialog box that shows all these macros in use.

As we wish to use floating-point, we first add `_ATL_USE_DDX_FLOAT` as a preprocessor definition in both the debug and release build settings. We also remove `_ATL_MIN_CRT` from the release build preprocessor definitions.



As we wish to use the WTL's `CString` functionality and the DDX functionality, we add atlmisc.h and atlddx.h to stdafx.h.

In the ResourceView, create a new dialog template, with the id of `IDD_ALL_MACROS_DLG` and the title of All Macros. Lay out a series of controls as follows:



Create a new dialog class called `CAllMacrosDlg` (say, in a file called allmacrosdlg.h).

Add variables of each type we need:

```
int m_nSignedIntNoValidation;
int m_nSignedIntValidation;
unsigned int m_nUnsignedIntNoValidation;
unsigned int m_nUnsignedIntValidation;
int m_nCheckBoxValue;
int m_nRadioButtonValue;
float m_nFloatNoValidation;
float m_nFloatValidation;
CString m_nStringNoValidation;
CString m_nStringValidation;
```

Now manually construct a DDX map with all the entries:

```
BEGIN_DDX_MAP(CAllMacrosDlg)
      DDX_INT(IDC_SIGNED_INT_NO_VAL, m_nSignedIntNoValidation)
      DDX_INT_RANGE(IDC_SIGNED_INT_VAL,
             m_nSignedIntValidation, 10,20)
  DDX_UINT(IDC_UNSIGNED_INT_NO_VAL,
        m_nUnsignedIntNoValidation)
      DDX_UINT_RANGE(IDC_UNSIGNED_INT_VAL,
            m_nUnsignedIntValidation, (unsigned int) 30,
            (unsigned int) 40)
      DDX_CHECK(IDC_CHECK_BOX, m_nCheckBoxValue)
      DDX_RADIO(IDC_RADIOBUTTON_GROUP, m_nRadioButtonValue)
      DDX_FLOAT(IDC_FP_NO_VAL, m_nFloatNoValidation)
      DDX_FLOAT_RANGE(IDC_FP_VAL, m_nFloatValidation,
            50.0, 60.0)
      DDX_TEXT(IDC_STRING_NO_VAL, m_nStringNoValidation)
```

```
        DDX_TEXT_LEN(IDC_STRING_VAL, m_nStringValidation, 20)
END_DDX_MAP()
```

Note that the `DDX_MAP` is entirely separate from the message map and each control can appear in neither, or both or any one of the two maps.

In `OnInitDialog`, initialize the variables, and then call `DoDataExchange`.

```
m_nSignedIntNoValidation = 1;
m_nSignedIntValidation = 15;
m_nUnsignedIntNoValidation = 3;
m_nUnsignedIntValidation = 35;
m_nCheckBoxValue = 1; // set check box
m_nRadioButtonValue = 2; // third entry - zero'd
                     // based index;
m_nFloatNoValidation = 10.0;
m_nFloatValidation = 55.0;
m_nStringNoValidation = TEXT("A Wonderful String");
m_nStringValidation = TEXT("Another String");
DoDataExchange(FALSE);
```

Validation occurs when transferring data from C++ data members to dialog controls. So it is where we call `DoDataExchange(FALSE)`, usually inside `OnInitDialog`, where validation can fail.

In `OnOK`, call `DoDataExchange(TRUE)`, which results in the updated values being copied from the dialog box controls into the member variables.

Sample : CStringWithFloat

While on the subject of support for floating point, we will now return to the `CString` class. WTL's `CString::Format` provides `printf`-like formatting functions. Currently it does not support floating point values (e.g. "`%f`"). WTL's DDX optionally does supports floating-point (based on a `#define`), but this requires the inclusion of the C Run Time (CRT) library.

What happens if we wish to use floats with WTL's `CString::Format`? It is likely that this support will be built into `CString` in future, but for now we have to do some work.

The CStringWithFloat sample explores how to add this support. It has a class `CStringWithFloat` that derives from WTL's `CString`. It has two member functions, `Format` and `FormatV`. `Format` is a wrapper function for `FormatV`. `CStringWithFloat::Format` is the same implementation as `CString::Format`. It is there to ensure that it calls `CStringWithFloat::FormatV` and not `CString::FormatV`.

If the macro `_ATL_USE_CSTRING_FLOAT` is defined, we add support for floats (this is similar to how floats are supported in WTL's DDX code). The implementation of `CStringWithFloat::FormatV` is very similar to that of `CString::FormatV`, with two modifications. Firstly, when walking the argument list to determine the length for the `CString` buffer, we add this code:

```
#ifdef _ATL_USE_CSTRING_FLOAT
```

```
            va_arg(argList, float);
            nItemLen = 32;
            nItemLen = max(nItemLen, nWidth+nPrecision);
#else
            ATLASSERT(!"Floating point (%%e, %%f, %%g, and %%G) is "
            "not supported by the WTL::CString class.");
#endif // !_ATL_USE_CSTRING_FLOAT
```

Secondly, when actually formatting, we use the CRT function `_vstprintf` (which can handle floats and is TCHAR-friendly) rather than `wvsprintf` from the Win32 API (which does not support floats):

```
#ifdef _ATL_USE_CSTRING_FLOAT
      int nRet = _vstprintf(m_pchData, lpszFormat, argListSave);
#else
      int nRet = wvsprintf(m_pchData, lpszFormat, argListSave);
#endif
```

We can use this as follows:

```
            CStringWithFloat str;
            str.Format("This is a float = %f\n", 4354.56);
```

The DDX Map Macros

The aim of the DDX map macros is to simplify the construction of the `DoDataExchange` function. The begin and end macros start and finish the function, and the various data transfer macros call the transfer function when appropriate.

The `BEGIN_DDX_MAP` macro is defined as:

```
#define BEGIN_DDX_MAP(thisClass) \
      BOOL DoDataExchange(BOOL bSaveAndValidate = FALSE, UINT nCtlID =
(UINT)-1) \
      { \
```

The data transfer macros are all defined similarly. Here is the one for `DDX_INT`:

```
#define DDX_INT(nID, var) \
            if(nCtlID == (UINT)-1 || nCtlID == nID) \
            { \
                  if(!DDX_Int(nID, var, TRUE, bSaveAndValidate)) \
                        return FALSE; \
            }
```

The `END_DDX_MAP` macro is defined as:

```
define END_DDX_MAP() \
            return TRUE; \
      }
```

So the DDX map . . .

```
      BEGIN_DDX_MAP(CMainDlg)
            DDX_INT(IDC_MYFIRST_INT, m_num)
      END_DDX_MAP()
```

. . . gets converted to this code:

```
      BOOL DoDataExchange(BOOL bSaveAndValidate = FALSE,
```

```
                                    UINT nCtlID = (UINT)-1){
            if(nCtlID == (UINT)-1 || nCtlID == nID) {
            if(!DDX_Int(nID, var, TRUE, bSaveAndValidate))
                        return FALSE;
            }
            return TRUE;
      }
```

The vast majority of times, `DoDataExchange` will be called without the second control id parameter. This means it defaults to `-1`, which results in all the DDX transfer functions being called. For specialist needs you can call `DoDataExchange` and pass it in a control id, when you need to carry out data transfer for a single control.

The first parameter, `bSaveAndValidate`, specifies if a save or a load is occurring. If saving is occurring, then validation is an optional extra – it is called using some macros and not with some others. In our previous listing of all the available DDX macros, we noted those that provided validation.

Detailed Look at one DDX Transfer Method

To gain a deeper understanding of how data transfer works, we will look in more detail at one sample from the WTL source code.

`DDX_Text` is called when any of the `DDX_TEXT` macros are used. The last two parameters, `bValidate` and `nLength`, are only used from DDX macros which require validation. The save parameters states whether we are copying data to the control or from the control to the string.

The control id specific that control in the dialog we wish to exchange data with. Its actual `HWND` is retrieved in a call to Win32's `GetDlgItem`. The `lpstrText` parameter is a buffer, filled with text if loading, and allocated with `nSize` characters if saving.

Here is the implementation (we added comments inline):

```
BOOL                            // RETURN: Did operation succeed?
DDX_Text
(
       UINT nID,                // IN: The control id used in
                                //     the transfer
       LPTSTR lpstrText,        // IN/OUT: the text string
                                // (buffer must be already allocated)
       int nSize,                    // IN: the length of the buffer
       BOOL bSave,              // IN: Saving or loading
       BOOL bValidate = FALSE, // IN: If saving, should we validate
       int nLength = 0             // IN: If validating, the length
)
{
            T* pT = static_cast<T*>(this);
            BOOL bSuccess = TRUE;
            if(bSave){
              // ------------------------------------------
              // COPY data from the control to the string buffer
```

```
            // --------------------------------------------

            // calls Win32's ::GetDlgItem
            HWND hWndCtrl = pT->GetDlgItem(nID);
                int nRetLen = ::GetWindowText(hWndCtrl,
                 lpstrText, nSize);

                // If buffer was not large enough, flag an error
            if(nRetLen < ::GetWindowTextLength(hWndCtrl))
                        bSuccess = FALSE;
           } else {
            // --------------------------------------------
            // COPY data from the string buffer to the control
            // --------------------------------------------
                ATLASSERT(!bValidate || lstrlen(lpstrText) <=
nLength);
                bSuccess = pT->SetDlgItemText(nID, lpstrText);
           }
       // --------------------------------------------
       // If transfer was unsuccessful, call the data
       // exchange error handler and return FALSE
       // --------------------------------------------
            if(!bSuccess){
                pT->OnDataExchangeError(nID, bSave);
            }
       else if(bSave && bValidate)
       {
            // ----------------------------------------
            // Perform validation if required - note that
            // many DDX macros do not mandate validation
            // ----------------------------------------
                ATLASSERT(nLength > 0);
                if(lstrlen(lpstrText) > nLength){
                    //
-----------------------------------------------
                    // If validation failed, fill in the _XData
                 // structure and call the validation error handler
                    //
-----------------------------------------------
                    _XData data;
                    data.nDataType = ddxDataText;
                    data.textData.nLength = lstrlen(lpstrText);
                    data.textData.nMaxLength = nLength;
                    pT->OnDataValidateError(nID, bSave, data);
                    bSuccess = FALSE;
                }
            }
            return bSuccess;
      }
```

Error Handling

Errors can occur during data transfer. Four scenarios are catered for:

- General data exchange problems
- Validate upon loading
- Data exchange error
- Validate upon saving

The initial two are covered by assertions in the WTL code. There are coding errors and ATLASSERTs are sprinkled around the DDX code to protect against these. These could include trying to validate that a string has a length of zero, setting the check box to a value outside the range 0-1-2, etc. In our discussion of the DDX macros/DDX data exchange functions we noted all the rules to do with each data type.

To handle the latter two situations, CWinDataExchange provides two overrideable error handlers, OnDataExchangeError and OnDataValidateError. A data exchange error will occur if a window with the specific control id does not exist. This is common if creating a dialog resource with a control which has a particular control id, and you use this in a DDX map; and later you edit the dialog resource and change/delete the control, but do not change the DDX map. The DDX map is left with an entry for a non-existent control id. When a data exchange error occurs, this error is detected and OnDataExchangeError is called. The problem can easily be fixed by ensuring that all entries in the DDX map correspond to entries in the dialog. When validating upon saving, then a validation error occurs (e.g. when a string is longer than permitted, or a numeric value is outside the permitted range), then OnDataValidateError is called.

The reason there is a difference in strategy between validating upon load (use asserts if an error occurs) and validate upon save (call OnDataValidateError if an error occurs), is that for loading, the data is coming from the application code and should be correct, whereas when saving the data is coming from the end-user and could well be incorrect.

An alternative for text validation is to set the text limit for an edit box. WTL's CEdit provides the SetLimitText method, which internally sends an EM_SETLIMITTEXT message. Note that this affect what the user types into the edit box, but not what might be added through WM_SETTEXT messages or what is already in the edit box.

Error Handlers

CWinDataExchange has two data exchange handlers, OnDataExchangeError and OnDataValidateError. Their default implementations are:

```
// Overrideables
      void OnDataExchangeError(UINT nCtrlID, BOOL /*bSave*/){
            // Override to display an error message
            ::MessageBeep((UINT)-1);
            T* pT = static_cast<T*>(this);
            ::SetFocus(pT->GetDlgItem(nCtrlID));
      }
      void OnDataValidateError(UINT nCtrlID, BOOL /*bSave*/,
```

```
            _XData& /*data*/){
        // Override to display an error message
        ::MessageBeep((UINT)-1);
        T* pT = static_cast<T*>(this);
        ::SetFocus(pT->GetDlgItem(nCtrlID));
}
```

They simply beep and set the focus to the offending control. Note that if the control id is incorrect (a likely possibility for calls to `OnDataExchangeError`), then the call to `SetFocus` will fail (but the app will not crash). Applications can provide custom implementation of these error handlers.

It could be argued that these are coding errors and not end-user errors causes calls to `OnDataExchangeError`, and therefore an `ATLASSERT` statement would be appropriate. These are serious errors, and should be highlighted during debugging. Alternatively, a message box could be displayed stating that an error occurred. The end-user cannot do anything to fix the problem. Maybe the problem will still let the user work with other parts of the application, but to fix it a new build is required from the developer.

Calls to `OnDataValidateError` occur due to end-user action. These are to be expected, and the application should robustly handle all of the validation errors and definitely continue running. End-users often do not understand what has gone wrong from a simple beep and the focus. They will probably need more descriptive error description. You will have to provide this by overriding `OnDataValidateError`.

Per dialog, the same validation error handler is called for all validation errors. WTL provides the handler with detailed identification of what went wrong, you're your implementation can display it to the end-user.

The handler is passed in a `_XData` variable. The `nDataType` field identifies the data type that is in error and the union contains additional information.

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

The `nDataType` field will be set to one of:

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

Note that `ddxDataNull` and `ddxDataDouble` are not used in the current implementation (even `DDX_Float` when called with doubles uses `ddxDataFloat` when a validation error occurs).

The additional information structures identify the permissible and the actual values:

```
struct _XTextData {
      int nLength;
      int nMaxLength;
};
struct _XIntData {
      long nVal;
      long nMin;
      long nMax;
};
struct _XFloatData {
      double nVal;
      double nMin;
      double nMax;
};
```

The `_XData` is only used for validation errors – it is not used for data exchange errors.

Custom Validation Error Handler

It is recommended that you provide a custom validation error handler for your dialogs if you are using the DDX validation features.

First you will need to add strings to the application's string table in the resources. In our implementation we will use the following:

```
IDS_VALIDATE_TEXTERR
You entered a text string of length %d, whereas the maximum allowed
is %d
IDS_VALIDATE_INTERR
Error detected: You entered integer of %d, whereas it must be between
%d and %d inclusive
IDS_VALIDATE_FLOATERR
Error detected: You entered floating-point number of %f, whereas it
must be between %f and %f
IDS_VALIDATE_UNEXPECTEDERR
Unexpected error during validation
```

Then we will implement the error handler. First we set the focus to the errant control and we select all its text. Validation errors occur with edit boxes and the particular one with the problem is highlighted when its contents are selected.

Then we switch on the data type, and display a message box with as much information describing the problem. Even when the data type is float, we wish to display the current and permissible range. However, WTL's `CString` does not permit floating point in its `Format` method. Our specialization of `CString`, call `CStringWithFloat`, does, so we use it here. (An alternative would be to directly call the CRT's `sprintf`).

When we have formatted the error message we display it in a message box.

```
      void OnDataValidateError(UINT nCtrlID, BOOL /*bSave*/, _XData&
data){
            CString strFormat;
            CStringWithFloat strError;
            ::SetFocus(GetDlgItem(nCtrlID));
            ::SendMessage(GetDlgItem(nCtrlID), EM_SETSEL, 0, -1);
            switch (data.nDataType){
            case ddxDataText:
                  strFormat.LoadString(IDS_VALIDATE_TEXTERR);
                  strError.Format(strFormat, data.textData.nLength,
                   data.textData.nMaxLength);
                  break;
            case ddxDataInt:
                  strFormat.LoadString(IDS_VALIDATE_INTERR);
                  strError.Format(strFormat, data.intData.nVal,
                   data.intData.nMin, data.intData.nMax);
                  break;
            case ddxDataFloat:
                  strFormat.LoadString(IDS_VALIDATE_FLOATERR);
                  strError.Format(strFormat, data.floatData.nVal,
                   data.floatData.nMin, data.floatData.nMax);
                  break;
            case ddxDataDouble: // fall through to default
            case ddxDataNull: // fall through to default
            default:
                  strError.LoadString(IDS_VALIDATE_UNEXPECTEDERR);
            }
            MessageBox(strError);
      }
```

Typically the same implementation will work for all your dialogs. You could define it as one big macro, such as

```
#define ON_MY_DATA_VALIDATE_ERROR \
            CString strFormat;      \
       . . .                  \
            MessageBox(strError);   \
      }
```

and then add the macro to each of your dialogs:

```
class CMyFirstManualDialog : CDialogImpl<CMyDialog >{
. . .
ON_MY_DATA_VALIDATE_ERROR
BEGIN_DDX_MAP
. . .
};
```

Alternatively, you could define it in a template and derive your dialog class from it.

Processing the return value from DoDataExchange

We have seen how to create a custom implementation of the `DoDataExchange` method using DDX maps populated with DDX macros. When data transfer errors occur in one of the transfer method, then the DDX macros immediately return false from

DoDataExchange. The subsequent DDX macros are not called at all. The DoDataExchange method returns a Boolean specifying if the data exchange succeeded. It should be checked, and if false action taken.

Imagine we have a dialog box with two edit controls, identified by IDC_FIRST_VAL and IDC_LAST_VAL. Imagine we used this DDX map to conduct data exchange with it.

```
BEGIN_DDX_MAP(CMyDlg)
      DDX_INT(IDC_FIRST_VAL, m_nFirstVal)
      DDX_INT(IDC_MIDDLE_VAL, m_nMiddleVal)
      DDX_INT(IDC_LAST_VAL, m_nLastVal)
END_DDX_MAP()
```

As no control with the id IDC_MIDDLE_VAL exists, data exchange will fail. We have two problems, the DDX_Int call with IDC_MIDDLE_VAL failed, and the DDX call IDC_LAST_VAL is never actually made. Here the problem will occur when initializing the data. The OnDataExchangeError method will have been called from inside DDX_Int, and then it returns false, which causes DoDataExchange to return false. Such errors are very serious as they usually indicate a coding error. If an error occurred, then it usually is better to display an error message and not displayed the half-initialized dialog box. Some developer make the mistake of thinking that returning false from OnInitDialog will accomplish this, which is incorrect. The Platform SDK description for WM_INITDIALOG states: *"The dialog box procedure should return TRUE to direct the system to set the keyboard focus to the control specified by wParam. Otherwise, it should return FALSE to prevent the system from setting the default keyboard focus."* If you do not want the dialog to be displayed, call EndDialog.

In this snippet we assume we have declared an error string resource IDS_DATA_TRANSFER_ERR with a suitable error message.

```
LRESULT OnInitDialog(. . .){
      . . .
      if (!DoDataExchange(FALSE)){
            strFormat.LoadString(IDS_DATA_TRANSFER_ERR);
      MessageBox(strError);
            EndDialog(wID);
 }
      return TRUE;
}
```

If we manage to transfer data to the controls by successfully calling DoDataExchange(FALSE) in OnInitDialog, then that probably means we will not have data exchange problems when calling DoDataExchange(TRUE), assuming no one have dynamically destroyed some of the controls! However, there can very easily be validation errors when saving. These validation errors are to be expected, and are not a programming error – that there are merely end-user mistakes. We need to be able to robustly handle them. We have seen how our validation handler will be called, and the offending control highlighted and focus set upon it. However, typically we call DoDataExchange(TRUE) in our OnOk handler – and afterwards calls EndDialog to dismiss the dialog.

An interesting problem can occur here. If a validation error arises, we do set the focus etc., but if we then call `EndDialog`, the dialog will be immediately dismissed. Instead, what we need to do is check the return from `DoDataExchange`, and only if it is successful, should we call `EndDialog`. If it is not successful, then the dialog will remained displayed, with the offending control highlighted.

```
LRESULT OnOK(WORD /*wNotifyCode*/, WORD wID, HWND /*hWndCtl*/,
                  BOOL& /*bHandled*/){
            if (DoDataExchange(TRUE))
                  EndDialog(wID);
            return 0;
      }
```

As you certainly do not wish to trap users in the dialog until then get the right entries in all the fields, it is essentially from a usability viewpoint to provide them with a *Cancel* button, so that if they are encountering validation errors, and do not know/want to fix them, then at least they can select *Cancel* and dismiss the dialog.

So to recap, when loading the dialog with values, we only display it if no problems occur. When dismissing the dialog, we only hide it if no errors occur.

What WTL's DDX Does Not Support

There are a few desirable features missing from the current implementation of WTL's DDX. It does not support DDX for list-boxes or combo-boxes. Neither does it support DDX with an ActiveX control – we examined in the ATL chapter how this has to be done. It does not have a masked edit-box functionality, which would allow on-the-fly validation (after each entered character).

Differences between MFC DDX and WTL DDX

The following table summarizes the differences between DDX within WTL and MFC.

Feature	In MFC	In WTL
Creation of DDX maps	Automatic, when inserting dialog	Manual
Populating DDX maps	Automatic, via MFC's ClassWizard	Manual (Wizard support expected in VC++ 7)
Class support	Built into `CDialog`	Need to manually derive from `CWinDataExchange`
Calling `DoDataExchange`	Built in (which can sometimes be a problem)	You decide when to call (a little extra work, but more flexible)
Validation Location	Separate to DDX (DDV)	Part of DDX
Validation Direction	Occurs when retrieving data from dialogs	Occurs when transferring data to and from dialog controls

Range of DDX macros	Widespread	Text, ints, floats, checkboxes and radio-buttons – no support for listboxes and combo-boxes

WTL Wrappers For The Standard Controls

WTL provides a range of wrapper classes for the standard Windows controls. These simplify the sending of messages to the control to manage its behavior. It should be noted that for most cases, the use of message maps to detect user interface actions (e.g. `BN_CLOCKED`) and DDX maps to exchange data are all that is needed. The use of these wrapper controls is only necessary when more advanced management of controls is needed.

The wrapper classes are:

- `CStatic`
- `CButton`
- `CListBox`
- `CComboBox`
- `CEdit`
- `CScrollBar`

Each wrapper consists of a constructor, a function member called `GetWndClassName` that returns the window class name of the control, and a method for each valid type of message that can be sent to that particular control type.

The wrappers are in fact templates and have `T` appended to their names. The templates have a single template parameter, which is usually `CWindow`. For each control there is a typedef of the control using `CWindow`, and this is named without the '`T`'.

```
typedef CStaticT<CWindow>     CStatic;
```

The simplest wrapper is `CStatic`, and it exposes methods for each message it accepts. The method implementation simply calls `SendMessage` with the appropriate message. A sample implementation is as follows:

```
HICON GetIcon() const {
      ATLASSERT(::IsWindow(m_hWnd));
      return (HICON)::SendMessage(m_hWnd, STM_GETICON, 0, 0L);
}
```

The wrappers set up the `wParam` and `lParam` fields correctly and also provide type safety.

Using the Wrappers

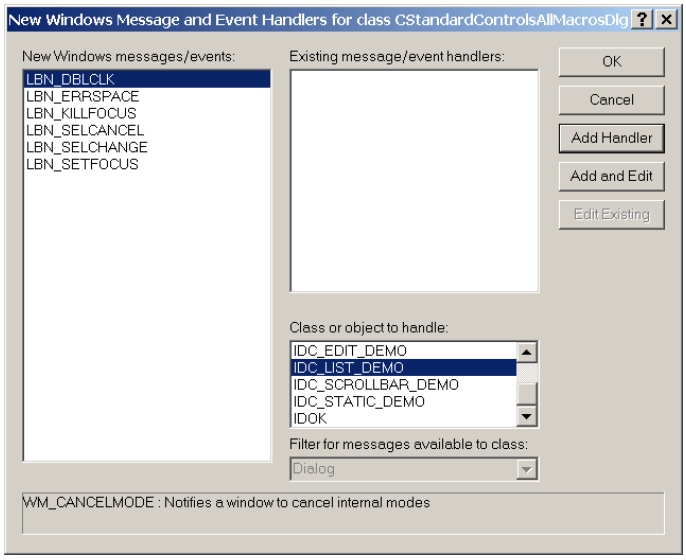
To use the wrapper templates you will first have to include atlctrls.h, usually in stdafx.h. Then in the dialog box in which you wish to use the wrappers, you will need to add data members:

```
CEdit m_Edit;
```

Then you will need to attach to the windows:

```
M_Edit.Attach(GetDlgItem(IDC_ANY_EDITBOX));
```

After that you can use the wrapper as needed to send message to the control. Note that you would normally not use the wrapper to react to incoming messages for a particular control. These are normally sent to the control's parent (the dialog box itself), and as discussed in the previous section on dialog box message maps, the *Add Message Wizard* understands the messages individual controls will receive, and support their detection.



Sample: Standard Controls Dialog to DataExchangeDemo

Now we will add a dialog called StandardControls to our demo project.

In ResourceView add a static based on a picture, a button based on a checkbox, a listbox, a combobox, an editbox and a scrollbar.

Add member variables for each of these to the dialog's definition.

```
CStatic m_Static;
CButton m_Button;
CListBox m_ListBox;
CComboBox m_ComboBox;
CEdit m_Edit;
CScrollBar m_ScrollBar;
```

In the dialog's `OnInitDialog`, attach the wrapper instances to the dialog controls. Also initialize the contents of the listboxes and combobox, and set the range for the scroll-bar.

```
m_Static.Attach(GetDlgItem(IDC_STATIC_DEMO));
```

```
            m_Button.Attach(GetDlgItem(IDC_BUTTON_DEMO));
            m_ListBox.Attach(GetDlgItem(IDC_LIST_DEMO));
            m_ListBox.AddString(TEXT("One"));
            m_ListBox.AddString(TEXT("Two"));
            m_ListBox.AddString(TEXT("Three"));

            m_ComboBox.Attach(GetDlgItem(IDC_COMBO_DEMO));
            m_ComboBox.AddString(TEXT("One"));
            m_ComboBox.AddString(TEXT("Two"));
            m_ComboBox.AddString(TEXT("Three"));
            m_ComboBox.SetCurSel(2);

            m_Edit.Attach(GetDlgItem(IDC_EDIT_DEMO));
            m_Edit.SetWindowText(TEXT("Some text"));
            m_ScrollBar.Attach(GetDlgItem(IDC_SCROLLBAR_DEMO));
            m_ScrollBar.SetScrollRange(0,100);
```

Finally add a push-button called *ChangeAttr* and a corresponding member function which exercises the wrappers.

```
      LRESULT OnClickedChangeAttr(WORD wNotifyCode, WORD wID, HWND
hWndCtl, BOOL& bHandled){
            HICON MyIcon;
        MyIcon = (HICON) LoadImage(_Module.GetModuleInstance(),
              MAKEINTRESOURCE(ID_MY_ICON), IMAGE_ICON, 0, 0,
              LR_DEFAULTCOLOR);
            m_Static.SetIcon(MyIcon);
            m_Button.SetCheck(1);
            m_ListBox.SetCurSel(1);
            m_ComboBox.SetEditSel(0,1);
            m_Edit.AppendText(TEXT("- Extra text"));
            m_ScrollBar.SetScrollPos(50);
            return 0;
      }
```

Initializing a Combo-Box

The ResourceView in VC++'s DevStudio allows entries in a combobox to be added to the resource template at design time. When you add a combobox, and bring up its properties page, then the "Data" tab is used to add entries.

When you run an MFC app and display dialogs with combo-boxes who have had such entries defined, they are displayed with the combo-boxes populated as expected. With WTL, the combo-boxes get displayed, but are empty. What is happening?

Information about the combobox (such as its control id and location on the dialog, but excluding its data entries) is stored as part of a resource, of type `DIALOG`, in the resource file. The Win32 dialog APIs understand this and load it directly before rendering the dialog. The information about the entries in a combobox is stored as a separate resource, of type `DLGINIT`. The Win32 dialog APIs do not understand `DLGINIT`. Other code in higher-level libraries or the application itself has to process `DLGINIT` resources.

MFC provides functionality as part of its dialog classes to cater for DLGINIT. WTL does not. The reuse header file (rtl.h) that we have written contains a WTL-friendly function called `RUiDlgInit`, which loads the `DLGINIT` data. It is modeled on the equivalent MFC functionality.

```
RUiDlgInit(IDD, m_hWnd);
```

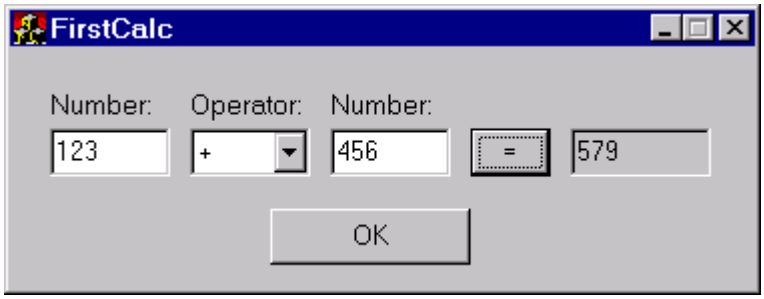
You need to pass it in the IDD of the dialog and the `HWND` of the existing dialog window. Typically it is called from `OnInitDialog`. `RUiDlgInit` just runs through the fields in the `DLGINIT` and sends a `CB_ADDSTRING` to the combo-box for each entry.

Sample : FirstCalc

The FirstCalc sample shows the use of the combo-box data entries.

It is a calculator, which accepts two numbers and lets the user select an operator ('+', '-', '/', '*') from a combo-box.

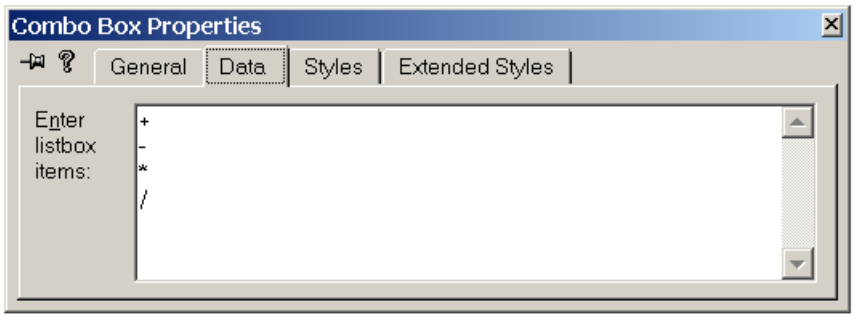
Its dialog is laid out as:



In the resource file this is stored as:

```
IDD_MAINDLG DIALOG DISCARDABLE  0, 0, 179, 62
STYLE WS_MINIMIZEBOX | WS_CAPTION | WS_SYSMENU
CAPTION "FirstCalc"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "OK",IDOK,65,40,50,14
    PUSHBUTTON      "=",IDC_EQUALS_BTN,110,15,25,15
    EDITTEXT        IDC_RESULT_EDIT,140,15,35,15,ES_AUTOHSCROLL |
                    ES_READONLY
    EDITTEXT        IDC_FIRST_EDIT,5,15,30,15,ES_AUTOHSCROLL
                                        | ES_NUMBER
    COMBOBOX        IDC_OP_COMBO,40,15,30,36,CBS_DROPDOWNLIST
                                        | WS_VSCROLL | WS_TABSTOP
    EDITTEXT        IDC_SECOND_EDIT,75,15,30,15,ES_AUTOHSCROLL
                                        | ES_NUMBER
    LTEXT           "Number:",IDC_STATIC,5,5,30,10
    LTEXT           "Operator:",IDC_STATIC,40,5,35,10
    LTEXT           "Number:",IDC_STATIC,75,5,30,10
END
```

The combobox entries are set using the ResourceView (for this demo they are just one character each in length, but of course they can be any length you like):

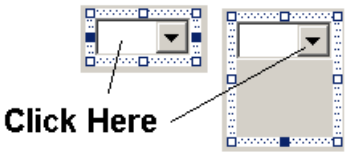


At the end of each entry do not press <CR> - this will dismiss the properties dialog – rather press <CTRL> and <CR> simultaneously, which will put the caret on the next line.

This information is stored in the resource file in a DLGINIT:

```
IDD_MAINDLG DLGINIT
BEGIN
    IDC_OP_COMBO, 0x403, 2, 0
0x002b,
    IDC_OP_COMBO, 0x403, 2, 0
0x002d,
    IDC_OP_COMBO, 0x403, 2, 0
0x002a,
    IDC_OP_COMBO, 0x403, 2, 0
0x002f,
    0
END
```

Developers new to DevStudio sometimes have problems sizing a combo-box. There are two sizes which need to be set: the (horizontal) size of the combo box when it is not in a drop-down state, and the (vertical) size when it is. To set the first, click the mouse in the combo-box outside of the drop-down arrow – to second the second click inside the arrow.



To load this data, add a call to RUiDlgInit to the OnInitDialog method – also set the current selection to 0 (the first entry in the combo-box list):

```
LRESULT OnInitDialog(UINT /*uMsg*/, WPARAM /*wParam*/,
      LPARAM /*lParam*/, BOOL& /*bHandled*/){
     . . .
      RUiDlgInit(IDD, m_hWnd);
           m_ctlOpCombo.Attach(GetDlgItem(IDC_OP_COMBO));
           m_ctlOpCombo.SetCurSel(0);
           return TRUE;
     }
```

The handler carries out the calculation is as follows:

```
      LRESULT OnClickedEquals_btn(WORD wNotifyCode, WORD wID, HWND
hWndCtl,
  BOOL& bHandled){
      CString strError;
            // get current values
            if ((!DoDataExchange(TRUE, IDC_FIRST_EDIT)) ||
              (!DoDataExchange(TRUE, IDC_SECOND_EDIT))){
                  strError.LoadString(IDS_ERR_EXCHANGE);
                  MessageBox(strError);
                  return 0;
            }
            int sel = m_ctlOpCombo.GetCurSel();
            switch (sel){
                  case 0:
                  m_nResultNum = m_nFirstNum + m_nSecondNum; break;
            case 1:
                  m_nResultNum = m_nFirstNum - m_nSecondNum; break;
                  case 2:
                  m_nResultNum = m_nFirstNum * m_nSecondNum; break;
            case 3:
                  m_nResultNum = m_nFirstNum / m_nSecondNum; break;
                  default:
                  ATLASSERT(FALSE); // should not get here!
            }
            // Put result in result window
            if (!DoDataExchange(FALSE, IDC_RESULT_EDIT)){
                  strError.LoadString(IDS_ERR_EXCHANGE);
                  MessageBox(strError);
            }
            return 0;
      }
```

WTL Wrappers for Common Controls

WTL provides the following classes for controls and helper classes:

CListViewCtrl	CTreeViewCtrl	CTreeItem
CHeaderCtrl	CImageList	CTreeViewCtrlEx
CToolBarCtrl	CToolInfo	CProgressBarCtrl
CStatusBarCtrl	CToolTip	CHotKeyCtrl
CTabCtrl	CTrackBarCtrl	CAnimate
CFlatScrollBar	CUpDownCtrl	CRichEdit
CDragListBox	CComboBoxEx	CRichEditComamnds
CDragListNotifyImpl	CDataTime	CIPAddressCtrl
CReBarCtrl	CMonthCalendar	CPagerCtrl

	CFlatScrollBarImpl	CCustomDraw

Chapter 7
Graphical Primitives

Objectives

The objectives of this chapter are to:

- Explain when we need graphical primitives
- Describe the `CSize/CPoint/CRect` helper classes
- Explain device contexts
- Review graphical settings, such as pens, brushes and fonts

Overview

All modern operating systems provide windowed graphics, which operate in a similar manner. The screen is divided into a hierarchical set of windows, which can be controlled by multiple applications. An application can set up the co-ordinates and mapping mode for its windows, specific drawing attributes and then use a variety of primitives to render into the window. On Windows family of operating systems, the graphics technology is known as GDI.

The most important type of GDI object is a device context. One can apply methods such as `LineTo` or `Rectangle` or `TextOut` to a device context to effect rendering. The output from device context rendering methods can be configured using other types of GDI objects such as pens, brushes, fonts and bitmaps.

WTL provides a full range of classes and templates for graphical primitives that cover all aspects of the Win32 graphics facilities. The WTL support is a very thin layer above the Win32 GDI API, so it is highly efficient and also it is easy to mix WTL and direct Win32 GDI calls.

Rendering Graphics In Windows

Win32 graphics is based on the concept of a device context, which provides access to drawing primitives and stores drawing attributes and modes. Primitives consists of lines, rectangles, text, bitmaps and anything else which is to appear as pixels in a window. How these primitives are rendered is controlled by a set of attributes, some of which are simple values, such as drawing mode (e.g. `R2_COPYPEN` for plain or `R2_NOT` for inverting), and some of which are other objects, such as pens, brushes and fonts. These attributes can be assigned to the device context, and when rendering occurs the graphical primitives use the current attributes.

Windows GDI provides a number of object types. Each object is identified by a handle, has construction and destruction functions and offers a variety of other graphical output

and settings functions. The idea of providing wrappers as C++ templates for each GDI object types is appealing – and this is exactly what WTL does for all of them. The Windows GDI object types are:

- Device Context (DC) – Rendering and current attribute selections
- Font – Font to use for text
- Pen – How to render edge of primitives
- Brush – How to fill internals of primitives
- Region – Collection of one or more rectangles, polygons, or ellipses, which together describes an area of the window (need not be contiguous)
- Palette – A collection of color indices and values
- Bitmap – A two-dimensional collection of pixels

A DC is used for rendering inside windows. A DC is used to setup rendering attributes, either using discrete attribute settings (e.g. current-pen position) or attribute object types, such as a pen object, which color, width and dashing styles). A DC is used to store graphical modes, such as whether XOR is used for drawing or the co-ordinate management. Hence the DC is the main object and one can consider the others as helper objects

Where to Render?

Display device contexts can either cover the client area of window (the normal case), or the entire window including the window frame& title bar (rarely used).

Typically, Win32 applications render graphics in the client-area of a window. By "client-area", we mean the area inside the window frame, below the window title bar and above the statusbar, and inside scrollbars and command-bars if present. A display device context used for client-area rendering is known as a client DC. A Win32 function called `GetDC` will return a client DC.

Normally all these other parts of a window, such as the frame and title bar, are rendered by the operating system itself, and applications rarely needed to render here. It is possible to retrieve a DC, known as a window DC, capable of rendering in the entire window. A Win32 function called `GetWindowDC` will return a window DC. However, note that the Microsoft Platform SDK documentation for `GetWindowDC` has this line: *"Painting in non-client areas of any window is not recommended."*

The Windows standard and common controls are normally responsible for rendering themselves. One exception is the concept of owner-draw controls, in which case your code is responsible for refreshing controls as needed. It is possible to use the Win32 graphical primitives to render anywhere in any window, but this is not frequently done. Two WTL templates that could help are `CBitmapButton` and `COwnerDraw`.

Types of Device Context

Think of device context as providing access to the destination for drawing. Very often, device contexts represent on-screen windows (display device context), but the can also represent printers, metafiles and off-screen memory (memory device contexts).

Hence the types of device contexts are:

- `ClientDC`
- `WindowDC`
- `PaintDC`
- `EnhancedMetafileDC`
- `PrintDC`

A memory device context (for off-screen bitmaps) operate like the generic device context, so WTL need not provide anything special in this case. A memory device context is constructed by calling the Win32 function `CreateCompatibleDC`.

During refreshing, the OS sending a `WM_PAINT` message to the window. To signify that the update has actually occurred, the application code must call `BeginPaint` and `EndPaint`. To facilitate this, WTL provides which is known as a `PaintDC`, which is a normal DC, but in addition calls `BeginPaint` in its constructor and `EndPaint` in its destructor.

What to Render?

The application needs to keep track of everything that appears in the client area of each window. When a window is hidden and later displayed, it is the application's duty to redraw everything. WTL does not provide any support in this area. Typically the application maintains a hierarchy of one or more collection classes, such as arrays or linked lists, and during refreshes, it cycles through the data (in a painter's algorithm, back to front) and rendering each data item. Each item has a bounding box, which identifies its perimeter, and when an update occurs often only a subsection of the window needs to be refreshed. Only items whose bounding box intersects the refresh area need be redrawn.

When the application is about to shutdown, there is a need to persist such data to the hard disk. Again, WTL does not provide assistance in this area. Modern applications normally use XML here.

When to Render?

There are two schools of thought about when to do rendering. One opinion is that it should all be centralized inside the handling of `WM_PAINT`. This has the advantage of doing it in one place, but the disadvantage of redrawing everything within the region, which needs to be refreshed. The function will be needed anyway to refresh the screen after it has been de-iconized or brought in front of another window which was obscuring it, or re-sized or scrolled. However, in this first approach it can also be used even if the window is already the foreground window. The idea is when adding an item

to appear on screen, it is appended to the internal hierarchical data structures, and then the area on screen it will cover is invalidated. This will result in a `WM_PAINT` message being sent to the window. Your windowing code will respond as usual by erasing the background and walking through the hierarchy redrawing everything that appears in the window. As the new object will be last in the hierarchy, it will be last to be drawn, so will appear on top of everything else. You might think this is an awful lot of work just to draw one object, but for very many applications it happens so fast the end-user will not notice. However, for applications will very complex output, yes, it can slow things down and there can be screen flicker. For such applications, the use of an off-screen bitmap can be helpful. All the graphical primitives can be rendered off-screen, and when finished, the off-screen bitmap can be bitblt'ed on-screen.

The second school of thought on how to do rendering is to permit rendering as needed, and still have a complete `WM_PAINT` implementation. When adding an object, it can be done inside the handler for mouse button up. Some people disapprove of this technique, as it is spreading drawing code in multiple locations. However, most applications manage items, which appear on screen as objects, and each has a `draw` method, so the bulk of the drawing code in inside such methods. What appears inside your handler for `WM_PAINT` or `WM_LBUTTONUP` is just calls to such methods. Also note that for rubber banding, this will normally have to be done in mouse move handling, so this second approach will still need to be catered for.

Whether you choose the first or the second approach usually depends on how complex the scene will be and whether the appearance of the new item will change existing items. For 3D graphics with shadowing etc., adding an item definitely will change other items, so it should be done in `WM_PAINT`. For an application drawing an item onto a window with 500,000 existing items, and if the item will just appear on top of these, then it would make sense to render it directly inside your `WM_LBUTTONUP` handler.

`WM_PAINT` messages are considered of a low-priority. The reason for this is that if additional messages are in the message queue, it is likely that they will change what needs to be rendered, and hence it is more efficient to leave the painting until all the other messages are handled, and then update once. When the OS is ready to present a `WM_PAINT` message, it gathers all the outstanding messages and amalgamates their invalid regions, and presents a single `WM_PAINT` with the amalgamated region to the application code.

Device Context Types

There are two popular approaches to DC management. One is to use the DC pool and the other is to maintain a private DC per window.

The OS maintains a pool of DCs and lends one to an application when it calls `GetDC`, and it is returned when `ReleaseDC` is called. What is provided is known as a common DC, as it is used by multiple windows (at different times). Using DCs from the pool is efficient and very often the `Get/Release` occurs inside the same function. In the distant past (e.g. Win3.1) there was a tiny limit to the number of DCs in the pool, but now it is only limited by system memory availability, and if memory is full you are going to have a lot more worries than simply refreshing the screen.

It is possible to maintain a separate DC for a particular window. Passing `CS_OWNDC` among the styles to the Win32 API `RegisterClass` does this.

A third approach, known as a class DC, is possible but not recommended. This approach uses one DC per class. Every window of that class will share the same DC. This can cause serious problems in a multi-threaded application.

DCs and Threads

A DC may only be used in one thread at a time. If an attempt is made to use it in multiple threads in parallel, one of the calls will fail. Another problem is that even though a DC might not be used in different threads at exactly the same time, it could happen that one thread has set up the DC with pens and brushes etc. and renders with it, but another thread sets it up with different attributes, and thus the first thread will render un-expectedly.

If you wish to share DCs across threads, then you are responsible for ensuring each DC is used in only one thread at a time and you must use synchronization constructs such as mutexes or critical sections. GDI itself does not do this for you. Most applications do not share DCs across threads because of this. Hence most applications do not use class DCs. Rather they use common device DCs, or if they can guarantee that all rendering to a particular window will be from the same thread, a private DC.

It is noted that with the Windows Template Library, there is an option to use multiple threads with the SDI approach, and critically these use one thread per window, so using private device contexts will work. However, the default setting is that a common device context will be used.

To use a private DC, add `CD_OWNDC` to the `WNDCLASS` passed to `RegisterClass`. In WTL/ATL, adding a custom implementation of `GetWndClassInfo` as follows can do this:

```
static CWndClassInfo& GetWndClassInfo() {
      static CWndClassInfo wc = {
            { sizeof(WNDCLASSEX), CS_HREDRAW | CS_VREDRAW|\
                  CS_DBLCLKS | CS_OWNDC, StartWindowProc,
             0, 0, NULL,
                  NULL, NULL, (HBRUSH)(COLOR_WINDOW + 1),
                  NULL, TEXT("myWindow"), NULL },
                  NULL, NULL, IDC_ARROW, TRUE, 0, _T("")
            };
      return wc;
}
```

Selecting Attribute Objects

GDI attribute objects are selected into a device context using `SelectObject`. When finished, the object has to be de-selected by calling `SelectObject` again with a default value or any other valid object handle. It is deleted by calling `DeleteObject`. Note that an object that is selected into a DC may not be deleted.

If you are using a common DC, every time you retrieve it its attributes will be set to default values. You will have to set it up with settings suitable for your application.

With a private DC, the first time you retrieve a DC it will have default values, but all subsequent retrieves will return the exact same DC and the attributes.

Minimize DC Alterations

For high-performance graphics applications, it is important to minimize the amount of changing of DC attributes. If you are drawing ten elements on-screen it makes no difference. If you are drawing a million elements, it certainly does.

For each element, it is ill advised to select a new pen, new brush, new font etc., as this is a waste of CPU resources. Instead, only as needed should these be selected. Hence we need to minimize the amount of DC alterations. One idea is to use a private DC. It is likely that application default attributes can be different from system default, and with a private DC we could set the default once. Also, certain attributes are likely not to change, and they only need to be set in the private DC once.

The use of the concept of a style is another way of encouraging users to share attributes among multiple elements. One idea is to have a bitmask identifying desired attributes – and default attributes – only where bits are different, is there a need to select objects into DC.

One final possibility is that you could use multiple DCs per window. If you were managing a small number of graphical styles, you could create a DC for each style, and never release it until the application was shut down. Therefore your redraw code would never have to change a DC, thus speeding it up considerably. The number of times a style is usually changed is small compare to the number of times it is used (e.g. one per each element based on it per refresh).

Getting Started with WTL Graphics

Now we cam move on to WTL graphics.

WTL Header Files

When using WTL's graphics features you will have to include the WTL header file atlgdi.h.

If you plan to also use the helper classes such as `CSize`, `CRect` and `CPoint`, then you will have to include atlmisc.h. The entries in atlgdi.h do not depend on the entries in atlmisc.h, and vice versa, so it is quite possible to include one but not the other header file.

Atlgdi.h is included in atlapp.h, which in turn is included in every WTL project by the WTL AppWizard, so unless you change something you can be sure it is always included.

Sample : First Line

To draw your first line in WTL, generate a sample project, say of type SDI Application. In the <projectname>view.h file, there will be an implementation of the class `C<Projectname>View`. It will have a default `OnPaint` method with an

instantiation of a device context based on the view window. Add a call to `CPaint::LineTo` method, and build and run the application.

```
LRESULT OnPaint(UINT /*uMsg*/, WPARAM /*wParam*/,
       LPARAM /*lParam*/, BOOL& /*bHandled*/){
      CPaintDC dc(m_hWnd);
      dc.LineTo(200, 200); // Add this line only
      return 0;
}
```

The view window will contain a line, from the upper-left corner, down to the dimensions you have specified as parameters to `LineTo`.

As you see, using WTL for graphics is very simple, especially if you have a prior knowledge of Win32 graphics programming. All WTL provides is an efficient set of wrappers for Win32 graphical constructs, and in some cases adds functionality, which can be quite useful and timesaving. Using WTL for graphics also makes your code more easily tie in with other parts of a WTL application.

WTL Helper Classes – CSize, CPoint and CRect

WTL provides three helper classes to manage size, rectangle and point information. These classes have no data members. Instead then inherit from Win32 data types, which is where the data is stored.

`CSize/CPoint/CRect` are not used in any WTL GDI templates – so for example, to draw a rectangle, you may either pass in four integer parameters representing the boundary, or a variable of a Win32 data type `LPCRECT`, which points to the Win32 `RECT` structure.

```
typedef struct tagRECT { // Win32
    LONG    left;
    LONG    top;
    LONG    right;
    LONG    bottom;
} RECT;
```

The result of this is that as the WTL GDI functionality is based on the same Win32 data types, there is no direct dependency between the WTL helper classes and the WTL GDI templates/classes. As the WTL helper classes derive from the Win32 structures, then are interchangeable. You may either continue to use the Win32 structures, or the WTL helper classes.

CSize

`CSize` manages rectangular dimensions. It stores the width and height values as `LONG`s. WTL's `CSize` derives from Win32's `SIZE` structure.

`CSize` provides the following constructors:

```
CSize();
CSize(int initCX, int initCY);
CSize(SIZE initSize);
CSize(POINT initPt);
```

```
CSize(DWORD dwSize);
```

`CSize` provides these operations:

```
BOOL operator==(SIZE size) const;
BOOL operator!=(SIZE size) const;
void operator+=(SIZE size);
void operator-=(SIZE size);
void SetSize(int CX, int CY);
```

`CSize` has these operators which return `CSize` values:

```
CSize operator+(SIZE size) const;
CSize operator-(SIZE size) const;
CSize operator-() const;
```

`CSize` has these operators which return `CPoint` values:

```
CPoint operator+(POINT point) const;
CPoint operator-(POINT point) const;
```

Finally, `CSize` has these operators which return `CRect` values:

```
CRect operator+(const RECT* lpRect) const;
CRect operator-(const RECT* lpRect) const;
```

CPoint

`CPoint` manages a point. It stores the x and y co-ordinates as `LONG`s. WTL's `CPoint` derives from Win32's `POINT` structure.

`CPoint` provides the following constructors:

```
CPoint();
CPoint(int initX, int initY);
CPoint(POINT initPt);
CPoint(SIZE initSize);
CPoint(DWORD dwPoint);
```

`CPoint` provides these operations:

```
void Offset(int xOffset, int yOffset);
void Offset(POINT point);
void Offset(SIZE size);
BOOL operator==(POINT point) const;
BOOL operator!=(POINT point) const;
void operator+=(SIZE size);
void operator-=(SIZE size);
void operator+=(POINT point);
void operator-=(POINT point);
void SetPoint(int X, int Y);
```

`CPoint` has these operators which return `CPoint` values:

```
CPoint operator+(SIZE size) const;
CPoint operator-(SIZE size) const;
CPoint operator-() const;
CPoint operator+(POINT point) const;
```

`CPoint` has this operator which return a `CSize` value:

```
CSize operator-(POINT point) const;
```

`CPoint` has these operators which return `CRect` values:

```
CRect operator+(const RECT* lpRect) const;
CRect operator-(const RECT* lpRect) const;
```

CRect

`CRect` manages a rectangle. It stores the upper-left and lower-right co-ordinates as `LONG`s WTL's `CRect` derives from Win32's `RECT` structure.

`CRect` provides the following constructors:

```
CRect();
CRect(int l, int t, int r, int b);
CRect(const RECT& srcRect);
CRect(LPCRECT lpSrcRect);
CRect(POINT point, SIZE size);
CRect(POINT topLeft, POINT bottomRight);
```

Note that the `CRect` constructor, which takes four ints, expects left, top, right and bottom parameters, and not left, top, width and height. It one assumed the latter and create a `CRect` with say `(10,20,10,200)`, it would occupy no area of screen.

`CRect` provides these additional methods to access the stored data:

```
int Width() const;
int Height() const;
CSize Size() const;
CPoint& TopLeft();
CPoint& BottomRight();
const CPoint& TopLeft() const;
const CPoint& BottomRight() const;
CPoint CenterPoint() const;
```

`CRect` provides these conversion methods:

```
operator LPRECT();
operator LPCRECT() const;
```

`CRect` provides these querying methods:

```
BOOL IsRectEmpty() const;
BOOL IsRectNull() const;
BOOL PtInRect(POINT point) const;
BOOL EqualRect(LPCRECT lpRect) const;
```

`CRect` provides these editing methods:

```
void SetRect(int x1, int y1, int x2, int y2);
void SetRect(POINT topLeft, POINT bottomRight);
void SetRectEmpty();
void CopyRect(LPCRECT lpSrcRect);
```

`CRect` provides these methods to inflate, deflate and offset the rectangle:

```
void InflateRect(int x, int y);
void InflateRect(SIZE size);
void InflateRect(LPCRECT lpRect);
```

```
void InflateRect(int l, int t, int r, int b);
void DeflateRect(int x, int y);
void DeflateRect(SIZE size);
void DeflateRect(LPCRECT lpRect);
void DeflateRect(int l, int t, int r, int b);
void OffsetRect(int x, int y);
void OffsetRect(SIZE size);
void OffsetRect(POINT point);
void NormalizeRect();
```

`CRect` provides these rectangle positioning methods:

```
void MoveToY(int y);
void MoveToX(int x);
void MoveToXY(int x, int y);
void MoveToXY(POINT point);
```

`CRect` provides these Boolean operator methods:

```
BOOL IntersectRect(LPCRECT lpRect1, LPCRECT lpRect2);
BOOL UnionRect(LPCRECT lpRect1, LPCRECT lpRect2);
BOOL SubtractRect(LPCRECT lpRectSrc1, LPCRECT lpRectSrc2);
```

`CRect` provides these overloaded operators:

```
void operator=(const RECT& srcRect);
BOOL operator==(const RECT& rect) const;
BOOL operator!=(const RECT& rect) const;
void operator+=(POINT point);
void operator+=(SIZE size);
void operator+=(LPCRECT lpRect);
void operator-=(POINT point);
void operator-=(SIZE size);
void operator-=(LPCRECT lpRect);
void operator&=(const RECT& rect);
void operator|=(const RECT& rect);
```

`CRect` provides these overloaded operators which return a `CRect`**:**

```
CRect operator+(POINT point) const;
CRect operator-(POINT point) const;
CRect operator+(LPCRECT lpRect) const;
CRect operator+(SIZE size) const;
CRect operator-(SIZE size) const;
CRect operator-(LPCRECT lpRect) const;
CRect operator&(const RECT& rect2) const;
CRect operator|(const RECT& rect2) const;
CRect MulDiv(int nMultiplier, int nDivisor) const;
```

Sample : HelperClasses

The HelperClasses sample shows a simple use of these helper classes.

It is a simple SDI project. The header file "atlmisc.h" is added to atdafx.h. This method is added to show `CPoint`, `CSize` and `CRect` in action.

```
void HelperDemo(){
      CPoint pt1(100,100);
```

```
        CPoint pt2(300,300);
        CRect rect(pt1, pt2);
        CSize sz;
        sz.SetSize(100,100);
        pt1 += sz;
        BOOL b = rect.PtInRect(pt2);
}
```

GDI Objects and Handle Management

GDI objects are a totally separate concept to Win32 kernel objects, C++ objects and COM objects. They are internal OS-managed pieces of memory, which are used to render graphics. Applications refer to them via handles. The OS frees GDI objects automatically when the process, which uses them, dies. However, it is considered good practice to release them when no longer needed.

A GDI object gets created with a call to Win32 APIs `::Create<ObjectType>` and get freed with a call to the Win32 API `DeleteObject`. Consider a GDI handle as an opaque pointer to the memory. WTL provides templates to all the GDI objects to manage their lifetime and to expose type-safe wrappers to access all their functionality.

Who owns the handle

One issue that needs to be considered is how the GDI object handle is managed. Is the GDI object created inside the WTL template, or outside? Who looks after object destruction? How is access to the handle provided through the WTL template, so that direct Win32 calls can be made if needed?

Each WTL template for a GDI object works through this set of issues in the same way. Here we will take the example of a WTL's `CPenT` template. The same applies to `CBrushT`, `CFontT`, `CPaletteT`, `CBitmapT`, `CPaletteT`, `CRgnT`, `CEnhMetaFileT` and `CDCT` and all its derivatives (`CPaintDCT`, `CClientDC`, `CWindowDC` and `CEnhMetaFileDC`).

There is one Boolean template parameter, which specifies how handle management is to be done. Each template has two typedefs, which specify that the class will be used to manage its own handle, or will use an existing handle created elsewhere.

```
template <bool t_bManaged>
class CPenT {
. . .
};
typedef CPenT<false>    CPenHandle;
typedef CPenT<true>           CPen;
```

The constructor can take in a handle as an optional parameter (it defaults to `NULL`).

```
CPenT(HPEN hPen = NULL) : m_hPen(hPen)
{ }
```

The destructor will destroy the object represented by the handle only WTL is in change of handle management.

```
~CPenT(){
      if(t_bManaged && m_hPen != NULL)
```

```
            DeleteObject();
}
```

An assignment operator is provided.

```
CPenT<t_bManaged>& operator=(HPEN hPen){
      m_hPen = hPen;
      return *this;
}
```

Methods are provided to attach and detach the handle.

```
void Attach(HPEN hPen){m_hPen = hPen;}
HPEN Detach(){
      HPEN hPen = m_hPen;
      m_hPen = NULL;
      return hPen;
}
```

Methods are provided to access the internal handles and determine if it is `NULL`.

```
operator HPEN() const { return m_hPen; }
bool IsNull() const { return (m_hPen == NULL); }
```

A method is provided to delete the object. This is called from the destructor and also may be called directly.

```
BOOL DeleteObject(){
      ATLASSERT(m_hPen != NULL);
      BOOL bRet = ::DeleteObject(m_hPen);
      if(bRet)
            m_hPen = NULL;
      return bRet;
}
```

Parameters for WTL GDI Member Functions

WTL's GDI is a thin wrapper around Win32 GDI functionality. The design decision was taken to use standard Win32 data types for parameters into most WTL template methods. Hence one will see a `HPEN` rather that a `&CPen` being used. One exception is `CString`, which is supported in some templates for string inputs.

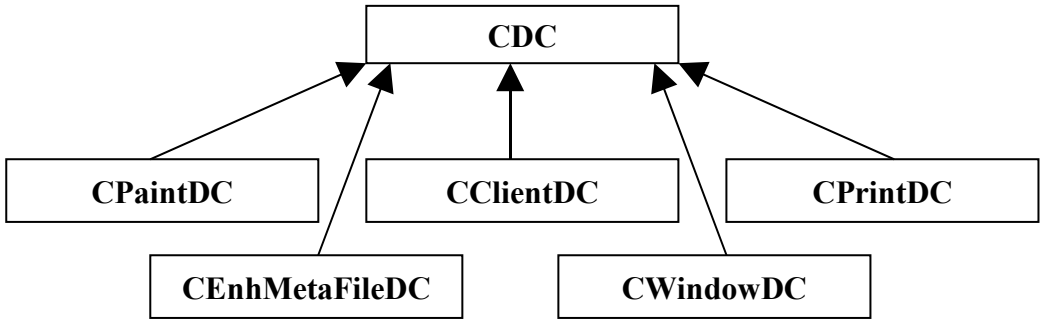
Device Context Templates

WTL provides a number of templates for device contexts. `CDC` is the main one and the other inherit from it. `CDC` is a big template – it is by far the biggest within WTL. It offers about 240 methods, which mostly wrap the familiar Win32 drawing functions such as `LineTo` and `TextOut`. Their implementations simply check that the DC handle is valid and then execute the Win32 function. For example, WTL's `CDC::LineTo` is implemented as:

```
BOOL LineTo(int x, int y){
      ATLASSERT(m_hDC != NULL);
      return ::LineTo(m_hDC, x, y);
}
```

There are a small number of methods within CDC, which contain more substantial functionality. We will shortly examine all of CDC's methods.

The other WTL templates for device contexts are small – they essentially perform a few extra duties and use CDC for the entire graphics rendering.



CPaintDC

CPaintDC is a class that derives from CDC. It is aimed at provided refresh functionality – so it manages a Win32 PAINTSTRUCT. In its constructor it calls Win32's BeginPaint and in its destructor it calls Win32's EndPaint. Apart from that, it is exactly like a CDC.

It should be used inside an application's handling of WM_PAINT messages. It should not be used elsewhere.

CClientDC

CClientDC derives from CDC and it is a DC based a client area of a window. Therefore it is ideal when rendering into a window – especially the client-area. Hence the name – CClientDC. In its constructor it calls Win32's GetDC and in its destructor it calls Win32's ReleaseDC. Apart from that, it is exactly like a CDC.

CClientDC is used when an application wishes to render outside of refresh handling. Examples would be inside button down, mouse move and button up handlers.

CWindowDC

CWindowDC derives from CDC and in its constructor it calls Win32's GetWindowDC and in its destructor calls ReleaseDC. Apart from that, it is exactly like a CDC. The CWindowDC template can be used to render in all parts of a window – including the non-client areas such as the title bar. It is rarely used, as most applications render only into the client-area (using CClientDC) do not need to render in the non-client area of a window. The DC retuned is a common device context – regardless of whether CS_OWNDC is set or not.

CEnhMetaFileDC

CEnhMetaFileDC is used when creating enhanced meta files.

We will discuss this template further when covering enhanced metafiles.

CPrintDC

`CPrintDC` is used when printing.

We will discuss this template further in the chapter on printing.

Managing Attributes Objects

`CDC` provides a number of methods to work with attribute objects.

To retrieve the current attribute value, use:

```
CPenHandle GetCurrentPen() const
CBrushHandle GetCurrentBrush() const
CPaletteHandle GetCurrentPalette() const
CFontHandle GetCurrentFont() const
CBitmapHandle GetCurrentBitmap() const
```

To set and get the origin of the brush in use:

```
BOOL GetBrushOrg(LPPOINT lpPoint) const
BOOL SetBrushOrg(int x, int y, LPPOINT lpPoint = NULL)
BOOL SetBrushOrg(POINT point, LPPOINT lpPointRet = NULL)
```

To list the pens and brushes available use:

```
int EnumObjects(int nObjectType, int (CALLBACK* lpfn)(LPVOID,
LPARAM), LPARAM lpData)
```

When selecting attribute objects, use:

```
HPEN SelectPen(HPEN hPen)
HBRUSH SelectBrush(HBRUSH hBrush)
HFONT SelectFont(HFONT hFont)
HBITMAP SelectBitmap(HBITMAP hBitmap)
int SelectRgn(HRGN hRgn)
```

The OS provides a number of standard object types, known as stock objects. When selecting stock objects use:

```
HPEN SelectStockPen(int nPen)
HBRUSH SelectStockBrush(int nBrush)
HFONT SelectStockFont(int nFont)
HPALETTE SelectStockPalette(int nPalette,
                            BOOL bForceBackground)
```

Stock pens may be one of: `WHITE_PEN`, `BLACK_PEN` or (Win98/2000 only) `DC_PEN`. Stock brushes may be `WHITE_BRUSH`, `HOLLOW_BRUSH` or (Win98/2000 only) `DC_BRUSH`. Stock fonts may be one of the range from `OEM_FIXED_FONT` to `SYSTEM_FIXED_FONT` or `DEFAULT_GUI_FONT`. The stock palette must be `DEFAULT_PALETTE`.

Lines and Pens

WTL's `CDC` offers these member functions to render lines, arcs, polylines and bezier curves.

```
BOOL GetCurrentPosition(LPPOINT lpPoint) const;
```

```
BOOL MoveTo(int x, int y, LPPOINT lpPoint = NULL);
BOOL MoveTo(POINT point, LPPOINT lpPointRet = NULL);
BOOL LineTo(int x, int y);
BOOL LineTo(POINT point);
BOOL Arc(int x1, int y1, int x2, int y2, int x3, int y3,
                        int x4, int y4);
BOOL Arc(LPCRECT lpRect, POINT ptStart, POINT ptEnd);
BOOL Polyline(LPPOINT lpPoints, int nCount);
BOOL AngleArc(int x, int y, int nRadius, float fStartAngle,
                        float fSweepAngle);
BOOL ArcTo(int x1, int y1, int x2, int y2, int x3, int y3,
                        int x4, int y4);
BOOL ArcTo(LPCRECT lpRect, POINT ptStart, POINT ptEnd);
int GetArcDirection() const;
int SetArcDirection(int nArcDirection);
BOOL PolyDraw(const POINT* lpPoints, const BYTE* lpTypes,
                        int nCount);
BOOL PolylineTo(const POINT* lpPoints, int nCount);
BOOL PolyPolyline(const POINT* lpPoints,
                        const DWORD* lpPolyPoints, int nCount);
BOOL PolyBezier(const POINT* lpPoints, int nCount);
BOOL PolyBezierTo(const POINT* lpPoints, int nCount);
```

Some functions use and update the current position and some don't.

For some drawing primitives, two versions of the function are provided – one ending in "`To`" and one not. The "`To`" versions (e.g. `ArcTo`, `PolylineTo`, `PolyBezierTo`) use the current point as the starting point of the rendering and when finished update the current position. The alternatives (`Arc`, `Polyline`, `PolyBezier`) ignore the current point, by starting the rendering at the first point in the point list parameter and not updating the current point when finished.

Some pairs of member functions provide access to the same underlying Win32 API, but support multiple data types – e.g. there is a `MoveTo` which takes integer `x` and `y`, and another `MoveTo` which takes `POINT`s.

The `PolyDraw` member function is not supported on Windows 9x. Check out Microsoft's KB article id: Q135059 to see how to provide a custom implementation for Win9x.

CPenT

A pen determines the line-rendering attributes, such as line color, dimensions and dashing style.

The `CPenT` template maintains a single data member, of type `HPEN`, which is a GDI handle to a pen object.

There are two types of pens: cosmetic and geometric. Cosmetic means that the pen's width is fixed and does not change with scaling, whereas with geometric pens the width can be scaled.

Three `Create` methods are provided:

```
      // creates a cosmetic pen
 HPEN CreatePen(int nPenStyle, int nWidth, COLORREF crColor);
 //creates a geometric pen
      HPEN CreatePen(int nPenStyle, int nWidth, const
            LOGBRUSH* pLogBrush, int nStyleCount = 0,
            const DWORD* lpStyle = NULL);
 // Creates cosmetic pen
      HPEN CreatePenIndirect(LPLOGPEN lpLogPen);
```

`CPenT` also provides these methods to access pen attributes:

```
      int GetLogPen(LOGPEN* pLogPen) const;
      bool GetLogPen(LOGPEN& LogPen) const;
      int GetExtLogPen(EXTLOGPEN* pLogPen) const;
      bool GetExtLogPen(EXTLOGPEN& ExtLogPen) const;
```

These are wrappers for the Win32 API `GetObject`. They return information about the currently selected pen.

Note that the `Width` field of the `LOGPEN` structure is actually a `POINT`, and only the `x` field is used.

Sample : LinesAndPens

The LinesAndPens sample shows how to use the `CDCT` and `CPenT` templates. It is an SDI application. Menu-items have been added for each important line member function of `CDC` and each `CPen` member.

We wish to use a private DC, so that DC settings will be retained across multiple `GetDC` calls. To enable this, in the view class, one line was commented out:

```
      //DECLARE_WND_CLASS(NULL)
```

and this implementation of `CWndClassInfo` added:

```
            static CWndClassInfo& GetWndClassInfo() {
            static CWndClassInfo wc = {
                  { sizeof(WNDCLASSEX), CS_HREDRAW | CS_VREDRAW|\
                        CS_DBLCLKS | CS_OWNDC, StartWindowProc,
                        0,0,NULL,NULL,NULL,(HBRUSH)(COLOR_WINDOW+1),
                        NULL, TEXT("myWindow"), NULL },
                        NULL, NULL, IDC_ARROW, TRUE, 0, _T("")
                  };
            return wc;
      }
```

The menu handlers are in the `CMainFrame` class, but the rendering is to be done in the view class, which is identified inside the `CMainFrame` class by `m_hWndClient`. Hence, when accessing the DC we use the line:

```
            CClientDC dc(m_hWndClient);
```

There are three types of handlers. One type calls rendering methods, such as `LineTo`, `Arc`, or `PolyBezier`. Here is one sample handler for `PolyDraw`:

```
      LRESULT OnPloyDraw(WORD /*wNotifyCode*/, WORD /*wID*/,
```

```
HWND /*hWndCtl*/, BOOL& /*bHandled*/)     {
      #define NUM_BEZIER_POINTS 4
      POINT ptList1[NUM_BEZIER_POINTS]={{20,20}, {250, 900},
                                 {450,145}, {600,320}};
      BYTE typeList1[NUM_BEZIER_POINTS]={PT_MOVETO,
             PT_BEZIERTO, PT_BEZIERTO, PT_BEZIERTO};
      dc.PolyDraw(ptList1, typeList1, NUM_BEZIER_POINTS);
      return 0;
}
```

Another type of handler is used to retrieve information from the device context. In this project, the data is simply sent to the output window.

```
LRESULT OnGetLogPen(WORD /*wNotifyCode*/, WORD /*wID*/,
       HWND /*hWndCtl*/, BOOL& /*bHandled*/){
      CClientDC dc(m_hWndClient);
      CPenHandle penhandle = dc.GetCurrentPen();
      LOGPEN logpen;
      penhandle.GetLogPen(logpen);
      CString str;
      str.Format("Output from GetLogPen: style = %d, \
       width = %d, color = (%d, %d, %d)\n",
       logpen.lopnStyle, logpen.lopnWidth.x,
       GetRValue(logpen.lopnColor),
       GetGValue(logpen.lopnColor),
       GetBValue(logpen.lopnColor));
      OutputDebugString(str);
      return 0;
}
```

The final type of handler is used to create a new pen.

```
LRESULT OnCreatePen(WORD /*wNotifyCode*/, WORD /*wID*/,
       HWND /*hWndCtl*/, BOOL& /*bHandled*/){
      CClientDC dc(m_hWndClient);
      // check if old pen existed
      if (!m_pen.IsNull()){
            // Before deleting pen, should de-select it from DC
            dc.SelectStockPen(BLACK_PEN);
            m_pen.DeleteObject();
      }
      // Generate logical pen
      m_pen.CreatePen(PS_DASHDOT, 5, RGB(150,50,100));
      dc.SelectPen(m_pen);
      return 0;
}
```

Note that the `m_Pen` parameter is a member variable of the `CMainFrame` class. If it were a local variable inside `OnCreatePen`, upon exiting the function the destructor for `CPen` would be called, and it calls `DeleteObject`, which is definitely not what is wanted in this situation.

Filled Shapes and Brushes

WTL's CDC offers these methods to draw filled rectangles.

```
      BOOL Rectangle(int x1, int y1, int x2, int y2);
      BOOL Rectangle(LPCRECT lpRect);
 BOOL FillRect(LPCRECT lpRect, HBRUSH hBrush);
      BOOL FrameRect(LPCRECT lpRect, HBRUSH hBrush);
      BOOL InvertRect(LPCRECT lpRect);
      BOOL RoundRect(int x1, int y1, int x2, int y2, int x3, int y3);
      BOOL RoundRect(LPCRECT lpRect, POINT point);
      BOOL DrawEdge(LPRECT lpRect, UINT nEdge, UINT nFlags);
      BOOL DrawFrameControl(LPRECT lpRect, UINT nType, UINT nState);
```

The Rectangle method draws the outline with the current pen and fills with the current brush. FillRect draws using the brush parameter, does not use the pen, and draw the top and left borders, but does not touch the bottom and right borders. FrameRect uses the brush to draw the outline of the rectangle. InvertRect performs a logical NOT operation on each pixel inside the rectangle. RoundRect draws a rounded rectangle. DrawEdge renders the edges of a rectangle according to a variety of styles, such as sunken or raised, specified in the flags parameter. DrawFrameControl renders the face of a standard control, such as a pushbutton or a radiobutton.

CDC offers these methods to render chords, pies and ellipses.

```
      BOOL Chord(int x1, int y1, int x2, int y2, int x3, int y3,
                                                 int x4, int y4);
      BOOL Chord(LPCRECT lpRect, POINT ptStart, POINT ptEnd);
      void DrawFocusRect(LPCRECT lpRect);
      BOOL Ellipse(int x1, int y1, int x2, int y2);
      BOOL Ellipse(LPCRECT lpRect);
      BOOL Pie(int x1, int y1, int x2, int y2, int x3, int y3,
                                                 int x4, int y4);
      BOOL Pie(LPCRECT lpRect, POINT ptStart, POINT ptEnd);
```

The Pie method draws a pie shape. Chord draws part of an ellipse bounded by a line. Ellipse draws the entire ellipse. DrawFocusRect draws a rectangle in XOR mode.

CDC offers these methods to draw polygons:

```
      BOOL Polygon(LPPOINT lpPoints, int nCount);
      BOOL PolyPolygon(LPPOINT lpPoints, LPINT lpPolyCounts,
                                                  int nCount);
```

The Polygon method draws a single polygon, whereas the PolyPolygon method draws multiple polygons.

CDC offers these methods to rendered already loaded icons:

```
      BOOL DrawIcon(int x, int y, HICON hIcon);
      BOOL DrawIcon(POINT point, HICON hIcon);
```

CDC offers these methods to render an image with an effect.

```
BOOL DrawState(POINT pt, SIZE size, HBITMAP hBitmap,
       UINT nFlags, HBRUSH hBrush = NULL);
BOOL DrawState(POINT pt, SIZE size, HICON hIcon, UINT nFlags,
       HBRUSH hBrush = NULL);
BOOL DrawState(POINT pt, SIZE size, LPCTSTR lpszText,
       UINT nFlags, BOOL bPrefixText = TRUE,
       int nTextLen = 0, HBRUSH hBrush = NULL);
BOOL DrawState(POINT pt, SIZE size, DRAWSTATEPROC lpDrawProc,
       LPARAM lData, UINT nFlags, HBRUSH hBrush = NULL);
```

The image can be a bitmap, an icon, some text with or without a mnemonic, or what is known as complex, which is a callback function you provide to draw it. The effects can be hidden, hide-prefix, mono, normal (none), prefix only, right aligned or union (dithering).

CBrushT

A brush is used to specify how filling is to be done in filled shapes.

The WTL template `CBrushT` manages one data member: an `HBRUSH` instance.

`CBrushT` offers the following `Create` methods:

```
HBRUSH CreateSolidBrush(COLORREF crColor);
HBRUSH CreateHatchBrush(int nIndex, COLORREF crColor);
HBRUSH CreateBrushIndirect(const LOGBRUSH* lpLogBrush);
HBRUSH CreatePatternBrush(HBITMAP hBitmap);
HBRUSH CreateDIBPatternBrush(HGLOBAL hPackedDIB, UINT nUsage);
HBRUSH CreateDIBPatternBrush(const void* lpPackedDIB,
                                        UINT nUsage);
HBRUSH CreateSysColorBrush(int nIndex);
```

`CBrushT` provides these methods to access brush attributes:

```
int GetLogBrush(LOGBRUSH* pLogBrush) const
bool GetLogBrush(LOGBRUSH& LogBrush) const
```

Sample : FilledShapesAndBrushes

The FilledShapesAndBrushes sample exercises the `CDC` methods for filled shapes and the `CBrush` template. It is a modal-dialog based application. It contains a series of pushbuttons, which have been mapped to handlers that called the various methods. It has a static with an id of `IDC_OUTPUT_WINDOW` which is used as a destination for rendering. We wish to use a common DC, so we do nothing special when constructing the static. A member variable of the dialog class identifies this.

```
     CStatic m_OutputWindow;
LRESULT OnInitDialog(UINT /*uMsg*/, WPARAM /*wParam*/,
            LPARAM /*lParam*/, BOOL& /*bHandled*/){
      . . .
           m_OutputWindow.Attach(GetDlgItem(IDC_OUTPUT_WINDOW));
}
```

One sample handler when creating a brush is implemented as follows:

```
LRESULT OnCreateBrushIndirect(WORD /*wNotifyCode*/, WORD wID,
        HWND /*hWndCtl*/, BOOL& /*bHandled*/){
```

```
            LOGBRUSH logbrush;
            logbrush.lbColor = RGB(190,90,70);
            logbrush.lbHatch = HS_CROSS;
            logbrush.lbStyle = BS_HATCHED;
            CBrush br;
            br.CreateBrushIndirect(&logbrush);
            CClientDC dc(m_OutputWindow);
            dc.SelectBrush(br);
            dc.Rectangle(190,190,130,130);
            return 0;
      }
```

Here is one sample of the handlers for a drawing method:

```
      LRESULT OnDrawState(WORD /*wNotifyCode*/, WORD wID,
             HWND /*hWndCtl*/, BOOL& /*bHandled*/){
            CClientDC dc(m_OutputWindow);
            CPoint pt(150,150);
            CSize size(0,0);
            dc.DrawState(pt, size, TEXT("Hello there!"),DST_TEXT);
            return 0;
      }
```

Text and Fonts

WTL's `CDC` offers these methods to render text:

```
      BOOL TextOut(int x, int y, LPCTSTR lpszString, int nCount =-1);
      BOOL ExtTextOut(int x, int y, UINT nOptions, LPCRECT lpRect,
   LPCTSTR lpszString, UINT nCount = -1, LPINT lpDxWidths=NULL);
      SIZE TabbedTextOut(int x, int y, LPCTSTR lpszString,
       int nCount = -1, int nTabPositions = 0,
       LPINT lpnTabStopPositions = NULL, int nTabOrigin = 0);
      int DrawText(LPCTSTR lpszString, int nCount, LPRECT lpRect,
       UINT nFormat);
```

The `TextOut` method renders the string at the specified location using attributes of the DC. `ExtTextOut` is similar to `TextOut`, but also provides for clipping and opaquing. `TabbedTextOut` is also similar to `TextOut`, but expands tabs using the dimensions in the tab positions array. `DrawText` renders formats (e.g. justifies) and renders text. `CDC` offers these methods to get and set text-related information:

```
BOOL GetTextExtent(LPCTSTR lpszString, int nCount,
       LPSIZE lpSize) const;
      BOOL GetTabbedTextExtent(LPCTSTR lpszString, int nCount,
       int nTabPositions, LPINT lpnTabStopPositions) const;
      BOOL GrayString(HBRUSH hBrush,
       BOOL (CALLBACK* lpfnOutput)(HDC, LPARAM, int),
       LPARAM lpData, int nCount, int x, int y,
       int nWidth, int nHeight);
      UINT GetTextAlign() const;
      UINT SetTextAlign(UINT nFlags);
      int GetTextFace(LPTSTR lpszFacename, int nCount) const;
      int GetTextFaceLen() const;
```

```
BOOL GetTextMetrics(LPTEXTMETRIC lpMetrics) const;
int SetTextJustification(int nBreakExtra, int nBreakCount);
int GetTextCharacterExtra() const;
int SetTextCharacterExtra(int nCharExtra);
```

`GetTextExtent` will return the space occupied by a string when rendered in this DC. `GetTabbedTextExtent` does the same and takes tabbing into consideration. `GrayString` renders a string (or a bitmap) with a grayed background. `GetTextMetrics` returns a Win32 `TEXTMETRIC` structure filled in with lots of detailed information about the DC's current font. `Get/SetTextAlign` manages the alignment of the text relative to the bounding rectangle of the text. `GetTextFace` returns the typeface name of the currently selected font. There are versions of `GetTextFace` that return a `BSTR` (if `_ATL_NO_COM` is not defined) and a `WTL::CString` (if `__ATLSTR_H__` is defined). `GetTextFaceLen` returns the length of the name. Often applications will call this first, allocate enough memory for the name, and then call `GetTypeFace`. There is no Win32 function called `GetTypeFaceLen` – the way `WTL::CDC::GetTypeFaceLen` is implemented is by calling Win32's `GetTypeFace` with a `NULL` buffer. Note that there are some problems with the implementation of `CDC::GetTextFace(CString &str);` See the later "Bugs and Suggestions" section for details.

`SetTextJustification` sets the spacing to be added for break characters when a string is later rendered with `TextOut` etc. `Get/SetTextCharacterExtra` manages the inter-character spacing. This is the space used between each character when rendering.

`CDC` offers these methods to manage fonts:

```
BOOL GetCharWidth(UINT nFirstChar, UINT nLastChar,
 LPINT lpBuffer) const;
DWORD SetMapperFlags(DWORD dwFlag);
BOOL GetAspectRatioFilter(LPSIZE lpSize) const;
BOOL GetCharABCWidths(UINT nFirstChar, UINT nLastChar,
 LPABC lpabc) const;
BOOL GetCharABCWidths(UINT nFirstChar, UINT nLastChar,
 LPABCFLOAT lpABCF) const;
BOOL GetCharWidth(UINT nFirstChar, UINT nLastChar,
 float* lpFloatBuffer) const;
DWORD GetFontData(DWORD dwTable, DWORD dwOffset,
 LPVOID lpData, DWORD cbData) const;
int GetKerningPairs(int nPairs, LPKERNINGPAIR lpkrnpair) const;
UINT GetOutlineTextMetrics(UINT cbData,
 LPOUTLINETEXTMETRIC lpotm) const;
DWORD GetGlyphOutline(UINT nChar, UINT nFormat,
 LPGLYPHMETRICS lpgm, DWORD cbBuffer, LPVOID lpBuffer,
 const MAT2* lpmat2) const;
```

`GetCharWidth` returns the character widths of characters from a font. `SetMapperFlags` is used to get the DC to match the aspect ratio when mapping logical to physical fonts. `GetAspectRatioFilter` returns the aspect-ratio filter, which is a `CSize` structure, which specifies the desirable font size in pixels, when selecting fonts. `GetCharABCWidths` returns the ABC width of characters in a

Truetype font. A stands for the space before the character glyph, B the width of what is drawn in the glyph and C the space after the drawn glyph. A+B+C is the space the current position is advanced when a character is drawn. `GetFontData` returns information about the metrics of a Truetype font. `GetKerningPairs` returns the kerning pairs for the current font. When certain pairs of characters are to be rendered together, then it is sometimes the case that the inter-character distance should be different than the standard. This is known as kerning. Usually the distance is less. `GetOutlineTextMetrics` returns TrueType font metrics. `GetGlyphOutline` returns the outline for a character, e.g. as bezier curve data.

The following methods are only available when `_WIN32_WINNT >= 0x0500` (i.e. Windows 2000 or later):

```
DWORD GetFontUnicodeRanges(LPGLYPHSET lpgs) const;
DWORD GetGlyphIndices(LPCTSTR lpstr, int cch, LPWORD pgi,
 DWORD dwFlags) const;
BOOL GetTextExtentPointI(LPWORD pgiIn, int cgi,
 LPSIZE lpSize) const;
BOOL GetTextExtentExPointI(LPWORD pgiIn, int cgi,
 int nMaxExtent, LPINT lpnFit, LPINT alpDx,
 LPSIZE lpSize) const;
BOOL GetCharWidthI(UINT giFirst, UINT cgi, LPWORD pgi,
 LPINT lpBuffer) const;
BOOL GetCharABCWidthsI(UINT giFirst, UINT cgi, LPWORD pgi,
 LPABC lpabc) const;
```

`GetFontUnicodeRanges` returns data about the UNICODE characters in a font. `GetGlyphIndices` converts a string into indices for glyphs. `GetTextExtentPointI` determines the dimensions of an array of glyph indices. `GetCharABCWidthsI` returns the ABC widths of glyph indices.

CFontT

A font is used when rendering text.

The WTL template `CFontT` manages one data member, a `HFONT`.

`CFontT` provides these `Create` methods:

```
HFONT CreateFontIndirect(const LOGFONT* lpLogFont);
HFONT CreateFont(int nHeight, int nWidth, int nEscapement,
      int nOrientation, int nWeight, BYTE bItalic,
      BYTE bUnderline, BYTE cStrikeOut, BYTE nCharSet,
      BYTE nOutPrecision, BYTE nClipPrecision, BYTE nQuality,
      BYTE nPitchAndFamily, LPCTSTR lpszFacename);
HFONT CreatePointFont(int nPointSize, LPCTSTR lpszFaceName,
      HDC hDC = NULL);
HFONT CreatePointFontIndirect(const LOGFONT* lpLogFont,
      HDC hDC = NULL);
```

When running on Windows 2000 or later, it also offers:

```
HFONT CreateFontIndirectEx(CONST ENUMLOGFONTEXDV* penumlfex);
```

`CFontT` provides these methods to access attrobites:

```
int GetLogFont(LOGFONT* pLogFont) const
```

```
bool GetLogFont(LOGFONT& LogFont) const
```

Sample : TextandFonts

The TextAndFonts sample explores the text methods in WTL's `CDC` and the `CFont` template. It uses treeview to present the names of the methods, and when one item in the treeview is selected, the relevant methods are called.

A sample handling of one API is as follows:

```
m_pDC->TabbedTextOut(40,40, TEXT("Tabbed\tText\tOut"),
                                    -1, TAB_COUNT, tabList, 20 );
```

A sample creation of a font is:

```
  m_Font.CreateFont( 16, 0, 900, 900,
FW_DONTCARE, 0, 0, 0,DEFAULT_CHARSET,
OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
DEFAULT_PITCH|FF_DONTCARE, "Helvetica" );
  m_pDC->SelectFont(m_Font);
```

Chapter 8
Internals of WTL

Objectives

- Identify the templates/classes in WTL's source code

Overview

As it is a template library, WTL consists of a set of header files only (atl*.h). It has no .cpp files. It includes some header files from ATL but in general you can use WTL independently of using ATL.

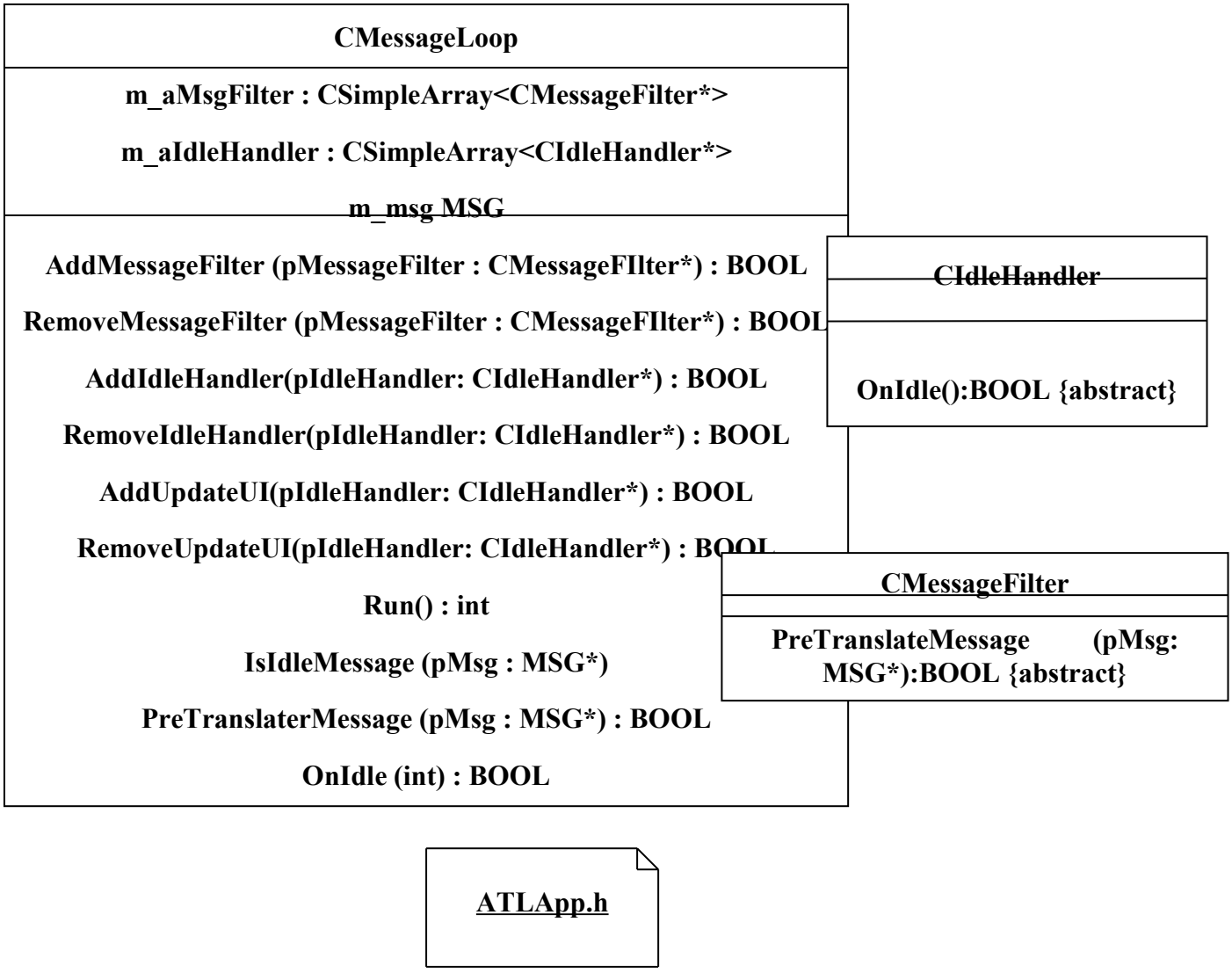
Header Files

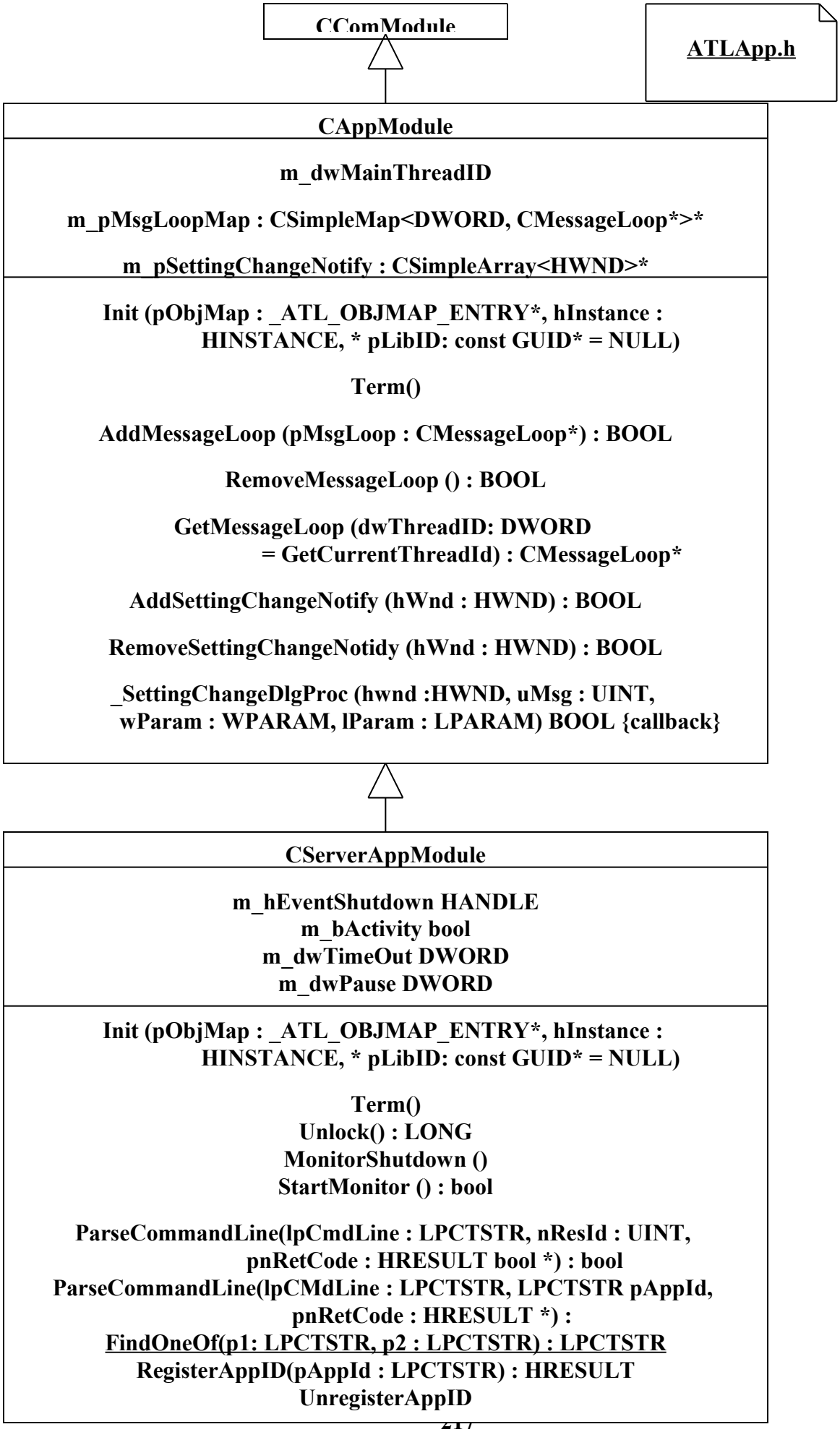
WTL consists of the following files:

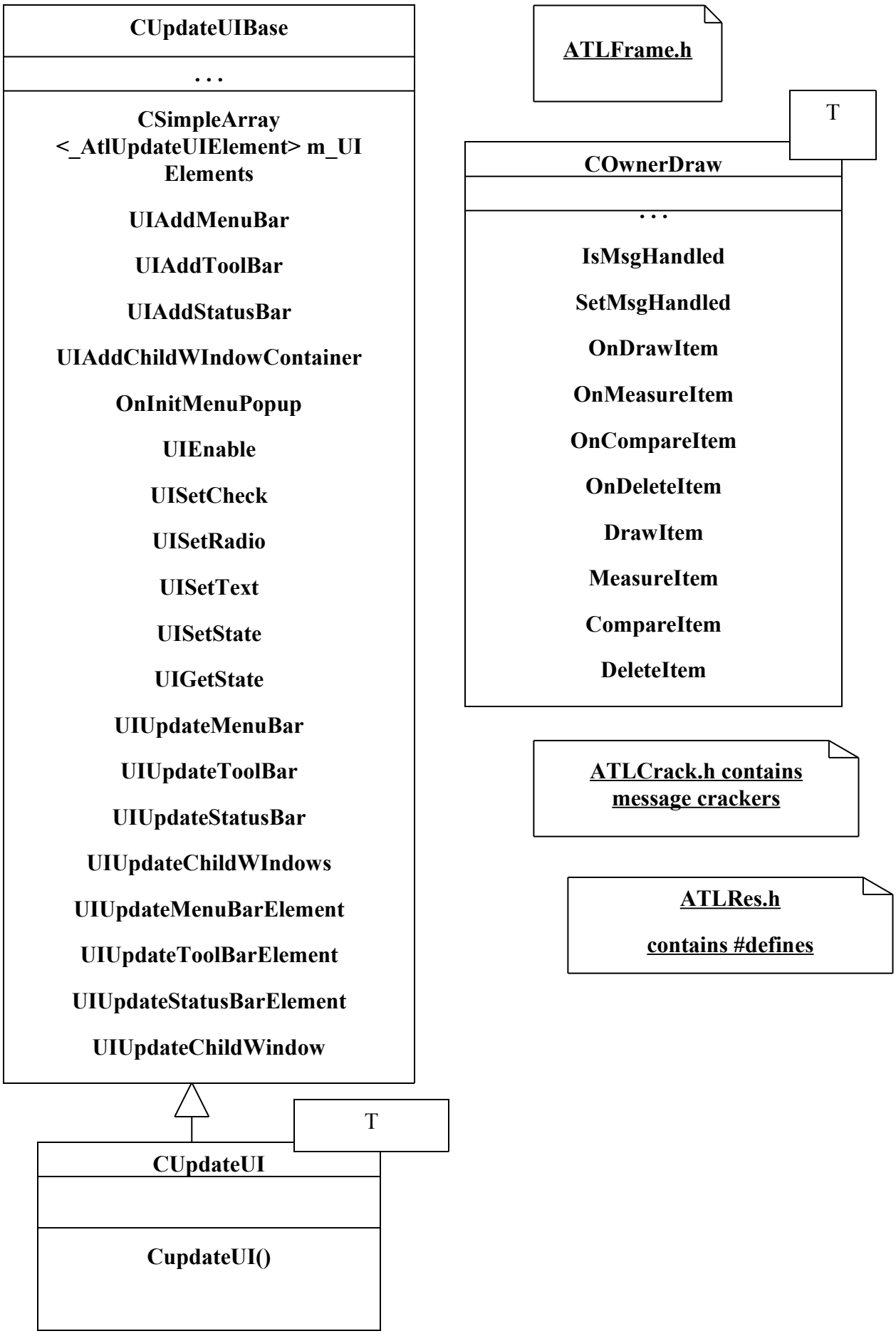
File Name	*Comments*	*Length (text lines)*	*Length (count of ';')*
AtlApp.h	AppModel, messagepump, reflectioning	872	244
AtlCrack.h	Windows message crackers	1835	748
AtlCtrls.h	Wrapper code for each of the Windows controls (The old code from ATLCON have evolved into this file)	7134	2732
AtlCtrlw.h	Command Bars	2218	886
AtlCtrlx.h	Bitmap buttons, checklistview, hyperlink, wait cursor and multipane status bar	1427	523
AtlDdx.h	DDX Support	563	191
AtlDlgs.h	Wrappers for common selection dialogs for File, folder, font, wrappers for common find/replace, printer setup, and property pages	2382	813
AtlFrame.h	MDI/SDI frame windows, MDI child & UpdateUI	2259	823
AtlGdi.h	Wrappers for pens, brushes, fonts, bitmaps, palettes and device contexts; OpenGL pixel formats and wgl wrappers	2847	1095
AtlMisc.h	Wrappers for common windows structures such	3026	1211

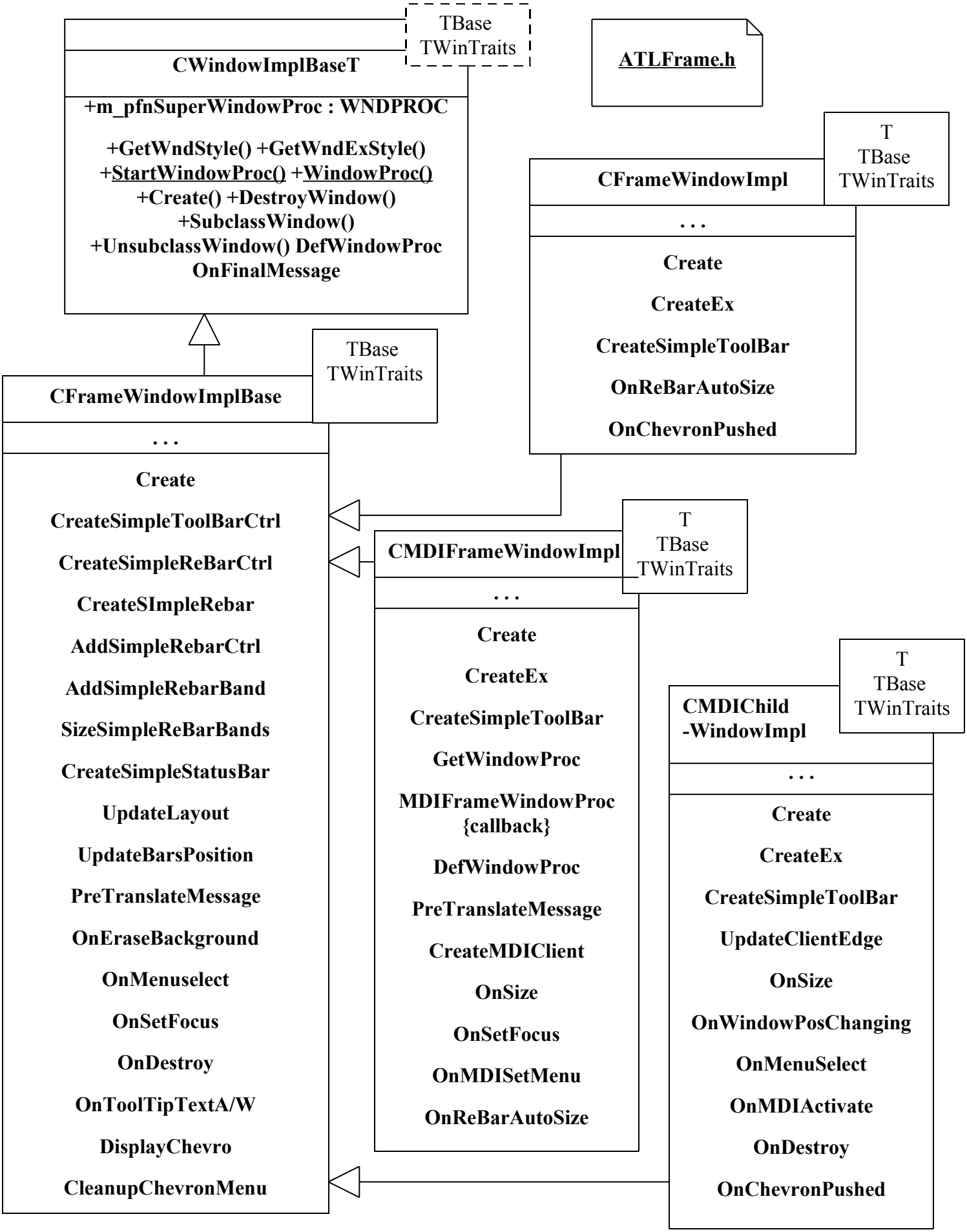
	as `SIZE, POINT, RECT`, a string class called `CString` and helpers for Find File, recent documents, etc.		
AtlPrint.h	Printing support	831	363
AtlRes.h	List of #defines of Resource IDs	245	0
AtlScrl.h	Scrolling	1020	364
AtlSplit.h	Splitter windows	724	246
AtlUser.h	Menu implementation	291	99
Total		27,674	10,338

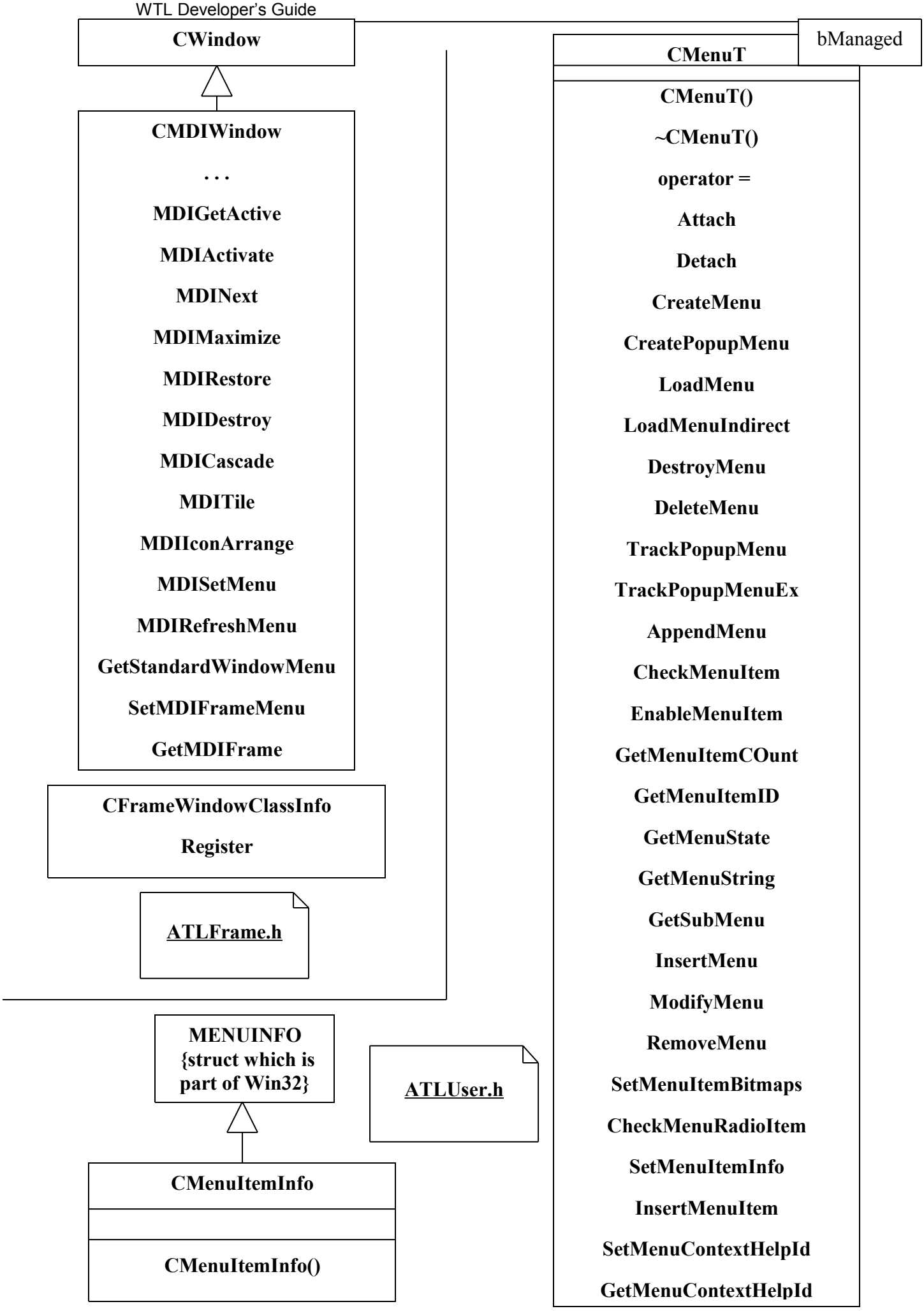
Contents of Each Header File

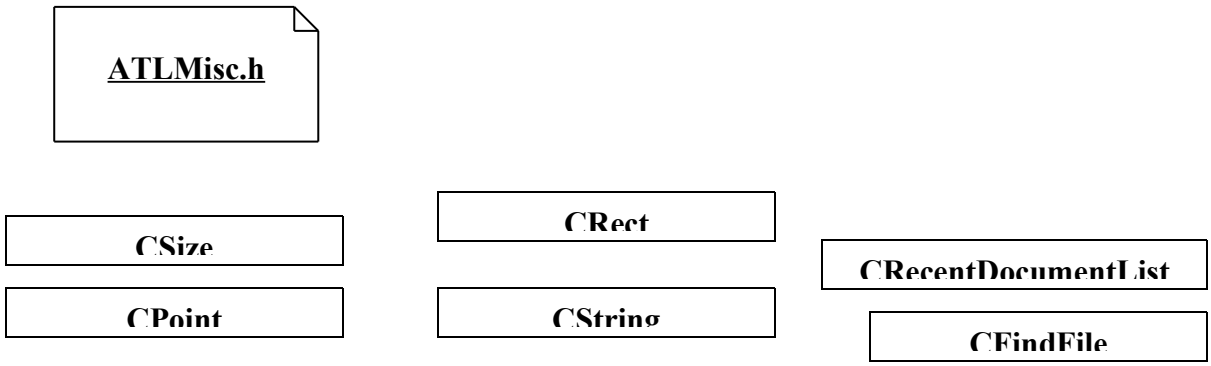


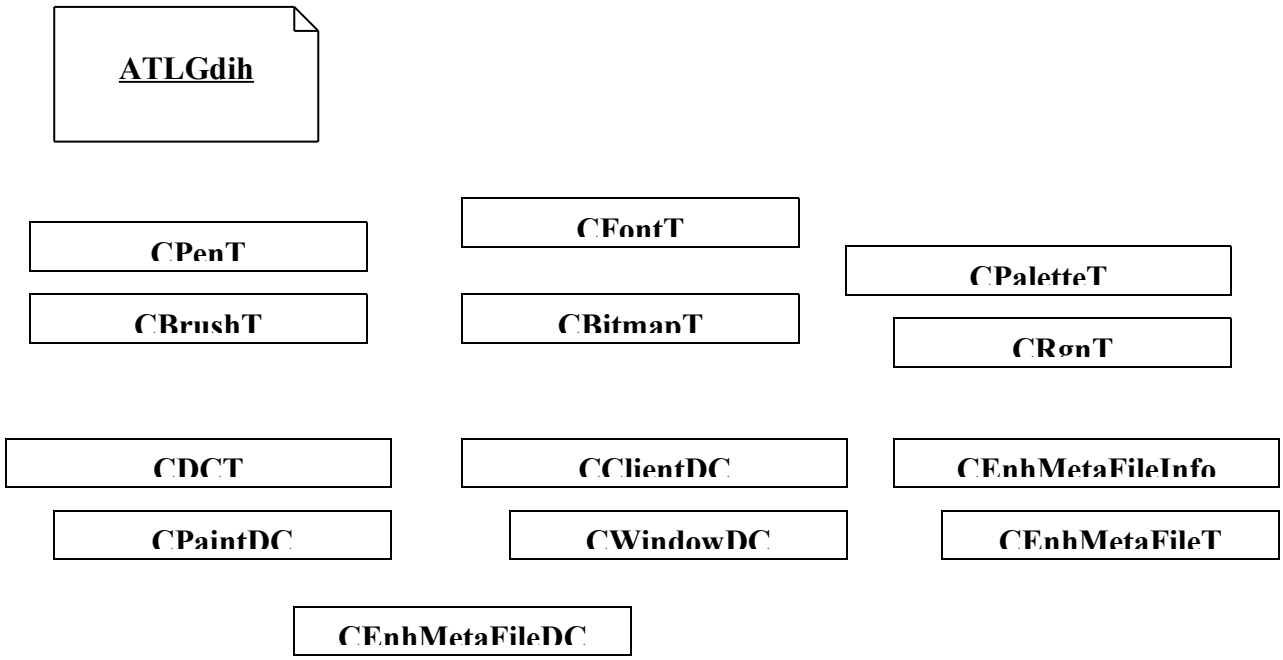

CComModule
ATLApp.h
CAppModule
m_dwMainThreadID
m_pMsgLoopMap : CSimpleMap<DWORD, CMessageLoop*>*
m_pSettingChangeNotify : CSimpleArray<HWND>*
Init (pObjMap : _ATL_OBJMAP_ENTRY*, hInstance :
HINSTANCE, * pLibID: const GUID* = NULL)
Term()
AddMessageLoop (pMsgLoop : CMessageLoop*) : BOOL
RemoveMessageLoop () : BOOL
GetMessageLoop (dwThreadID: DWORD
= GetCurrentThreadId) : CMessageLoop*
AddSettingChangeNotify (hWnd : HWND) : BOOL
RemoveSettingChangeNotidy (hWnd : HWND) : BOOL
_SettingChangeDlgProc (hwnd :HWND, uMsg : UINT,
wParam : WPARAM, lParam : LPARAM) BOOL {callback}
CServerAppModule
m_hEventShutdown HANDLE
m_bActivity bool
m_dwTimeOut DWORD
m_dwPause DWORD
Init (pObjMap : _ATL_OBJMAP_ENTRY*, hInstance :
HINSTANCE, * pLibID: const GUID* = NULL)
Term()
Unlock() : LONG
MonitorShutdown ()
StartMonitor () : bool
ParseCommandLine(lpCmdLine : LPCTSTR, nResId : UINT,
pnRetCode : HRESULT bool *) : bool
ParseCommandLine(lpCMdLine : LPCTSTR, LPCTSTR pAppId,
pnRetCode : HRESULT *) :
FindOneOf(p1: LPCTSTR, p2 : LPCTSTR) : LPCTSTR
RegisterAppID(pAppId : LPCTSTR) : HRESULT
UnregisterAppID

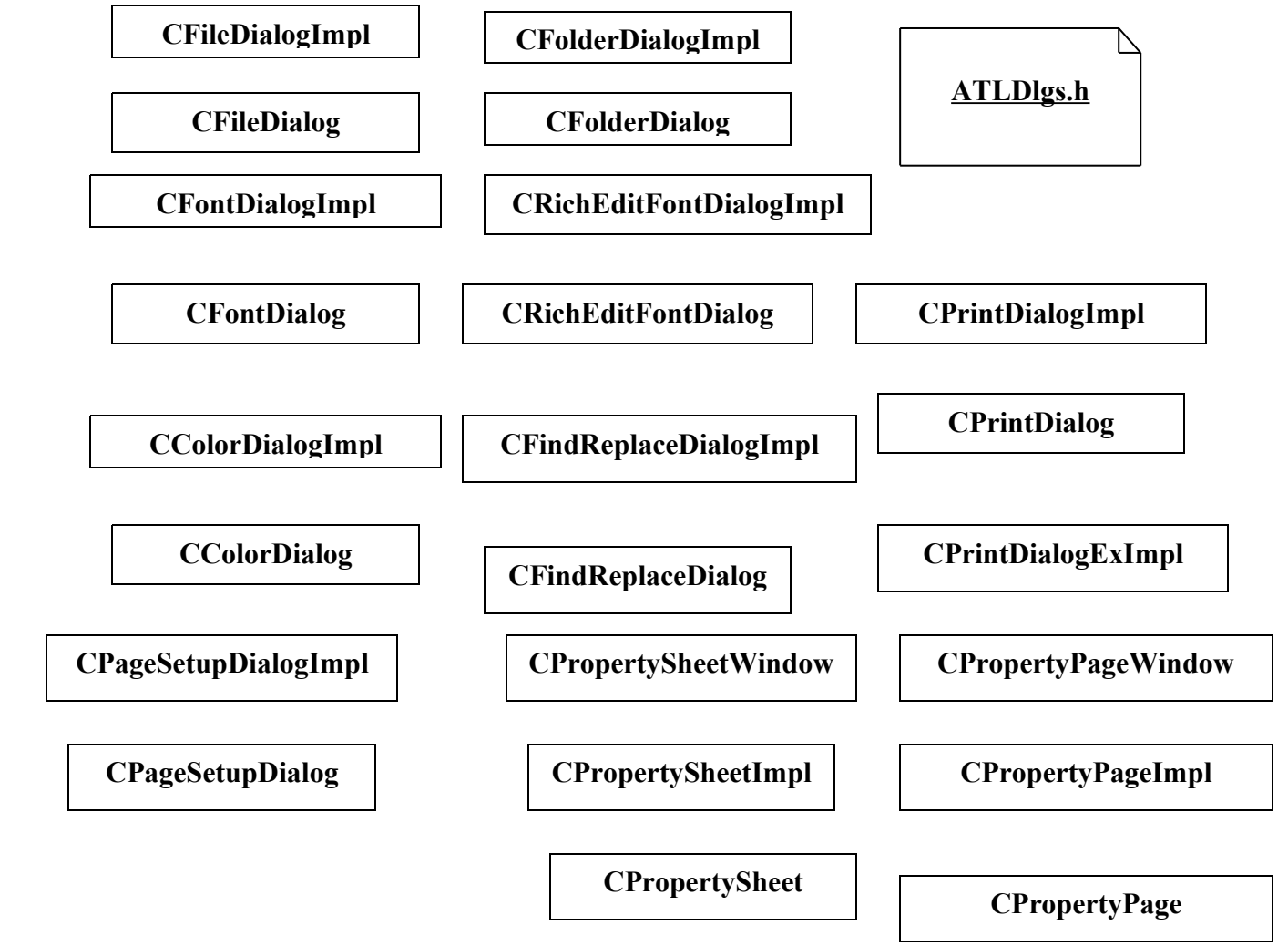

CUpdateUIBase
. . .
CSimpleArray <_AtlUpdateUIElement> m_UI Elements
UIAddMenuBar
UIAddToolBar
UIAddStatusBar
UIAddChildWIndowContainer
OnInitMenuPopup
UIEnable
UISetCheck
UISetRadio
UISetText
UISetState
UIGetState
UIUpdateMenuBar
UIUpdateToolBar
UIUpdateStatusBar
UIUpdateChildWIndows
UIUpdateMenuBarElement
UIUpdateToolBarElement
UIUpdateStatusBarElement
UIUpdateChildWindow
T
CUpdateUI
CupdateUI()
ATLFrame.h
T
COwnerDraw
. . .
IsMsgHandled
SetMsgHandled
OnDrawItem
OnMeasureItem
OnCompareItem
OnDeleteItem
DrawItem
MeasureItem
CompareItem
DeleteItem
ATLCrack.h contains message crackers
ATLRes.h
contains #defines

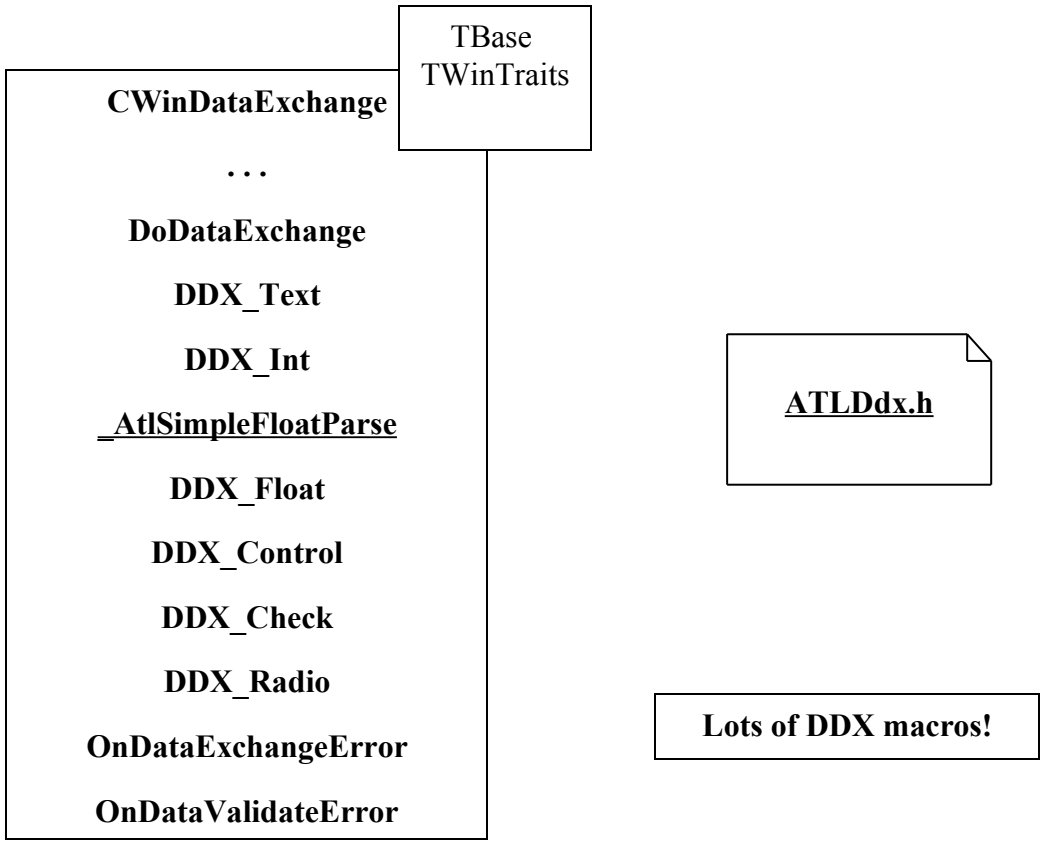

TBase
TWinTraits
CWindowImplBaseT
+m_pfnSuperWindowProc : WNDPROC
+GetWndStyle() +GetWndExStyle()
+StartWindowProc() +WindowProc()
+Create() +DestroyWindow()
+SubclassWindow()
+UnsubclassWindow() DefWindowProc
OnFinalMessage
ATLFrame.h
T
TBase
TWinTraits
CFrameWindowImpl
. . .
Create
CreateEx
CreateSimpleToolBar
OnReBarAutoSize
OnChevronPushed
TBase
TWinTraits
CFrameWindowImplBase
. . .
Create
CreateSimpleToolBarCtrl
CreateSimpleReBarCtrl
CreateSImpleRebar
AddSimpleRebarCtrl
AddSimpleRebarBand
SizeSimpleReBarBands
CreateSimpleStatusBar
UpdateLayout
UpdateBarsPosition
PreTranslateMessage
OnEraseBackground
OnMenuselect
OnSetFocus
OnDestroy
OnToolTipTextA/W
DisplayChevro
CleanupChevronMenu
T
TBase
TWinTraits
CMDIFrameWindowImpl
. . .
Create
CreateEx
CreateSimpleToolBar
GetWindowProc
MDIFrameWindowProc
{callback}
DefWindowProc
PreTranslateMessage
CreateMDIClient
OnSize
OnSetFocus
OnMDISetMenu
OnReBarAutoSize
T
TBase
TWinTraits
CMDIChild
-WindowImpl
. . .
Create
CreateEx
CreateSimpleToolBar
UpdateClientEdge
OnSize
OnWindowPosChanging
OnMenuSelect
OnMDIActivate
OnDestroy
OnChevronPushed

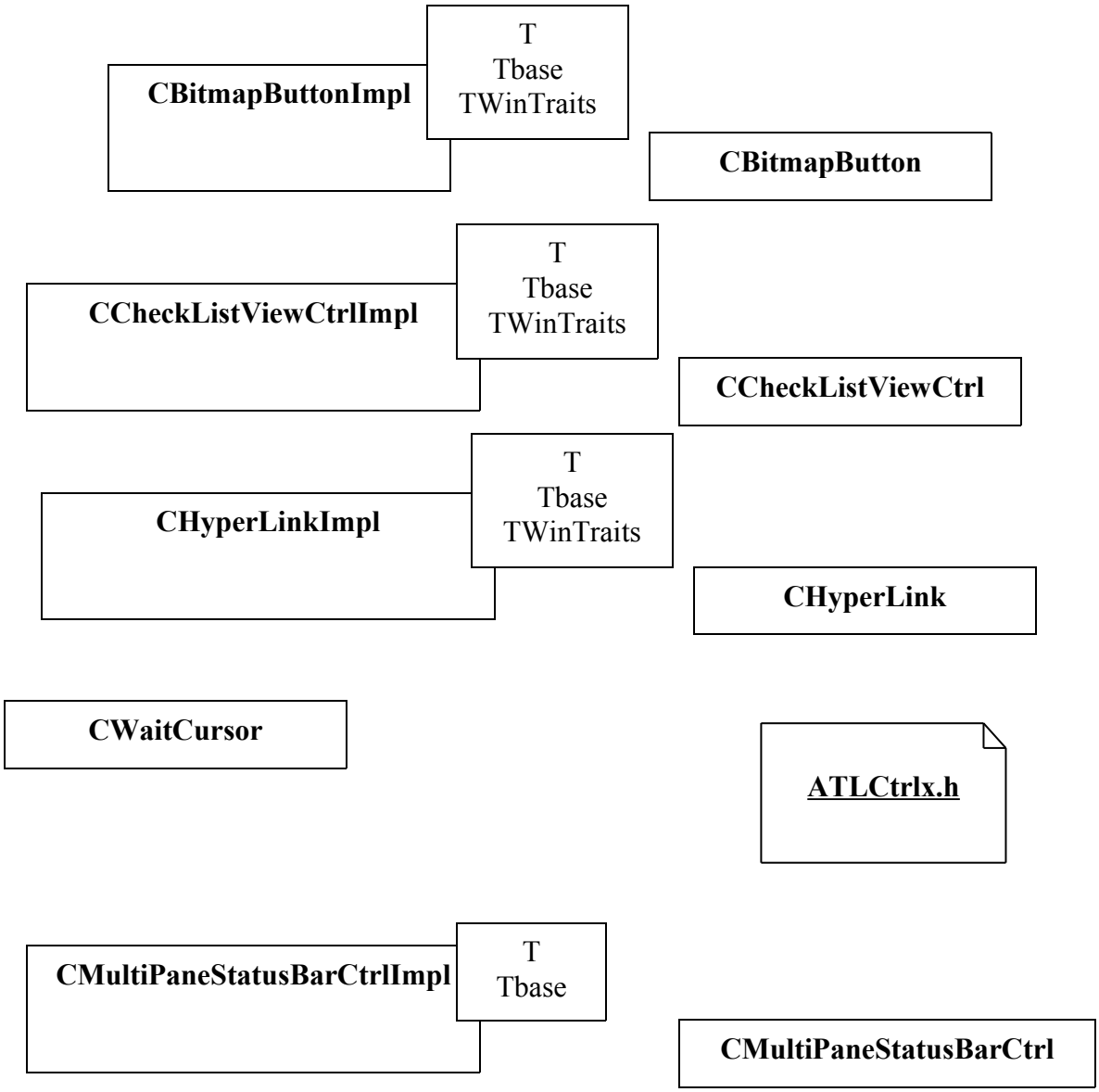

CWindow
CMDIWindow
. . .
MDIGetActive
MDIActivate
MDINext
MDIMaximize
MDIRestore
MDIDestroy
MDICascade
MDITile
MDIIconArrange
MDISetMenu
MDIRefreshMenu
GetStandardWindowMenu
SetMDIFrameMenu
GetMDIFrame
CFrameWindowClassInfo
Register
ATLFrame.h
MENUINFO
{struct which is
part of Win32}
CMenuItemInfo
CMenuItemInfo()
ATLUser.h
CMenuT
bManaged
CMenuT()
~CMenuT()
operator =
Attach
Detach
CreateMenu
CreatePopupMenu
LoadMenu
LoadMenuIndirect
DestroyMenu
DeleteMenu
TrackPopupMenu
TrackPopupMenuEx
AppendMenu
CheckMenuItem
EnableMenuItem
GetMenuItemCOunt
GetMenuItemID
GetMenuState
GetMenuString
GetSubMenu
InsertMenu
ModifyMenu
RemoveMenu
SetMenuItemBitmaps
CheckMenuRadioItem
SetMenuItemInfo
InsertMenuItem
SetMenuContextHelpId
GetMenuContextHelpId

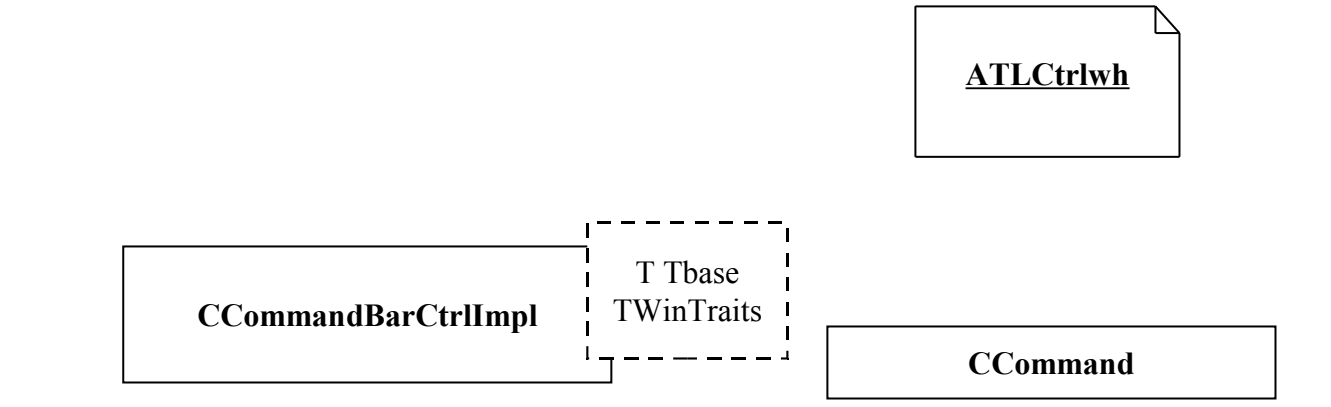

ATLMisc.h
CSize
CPoint
CRect
CString
CRecentDocumentList
CFindFile

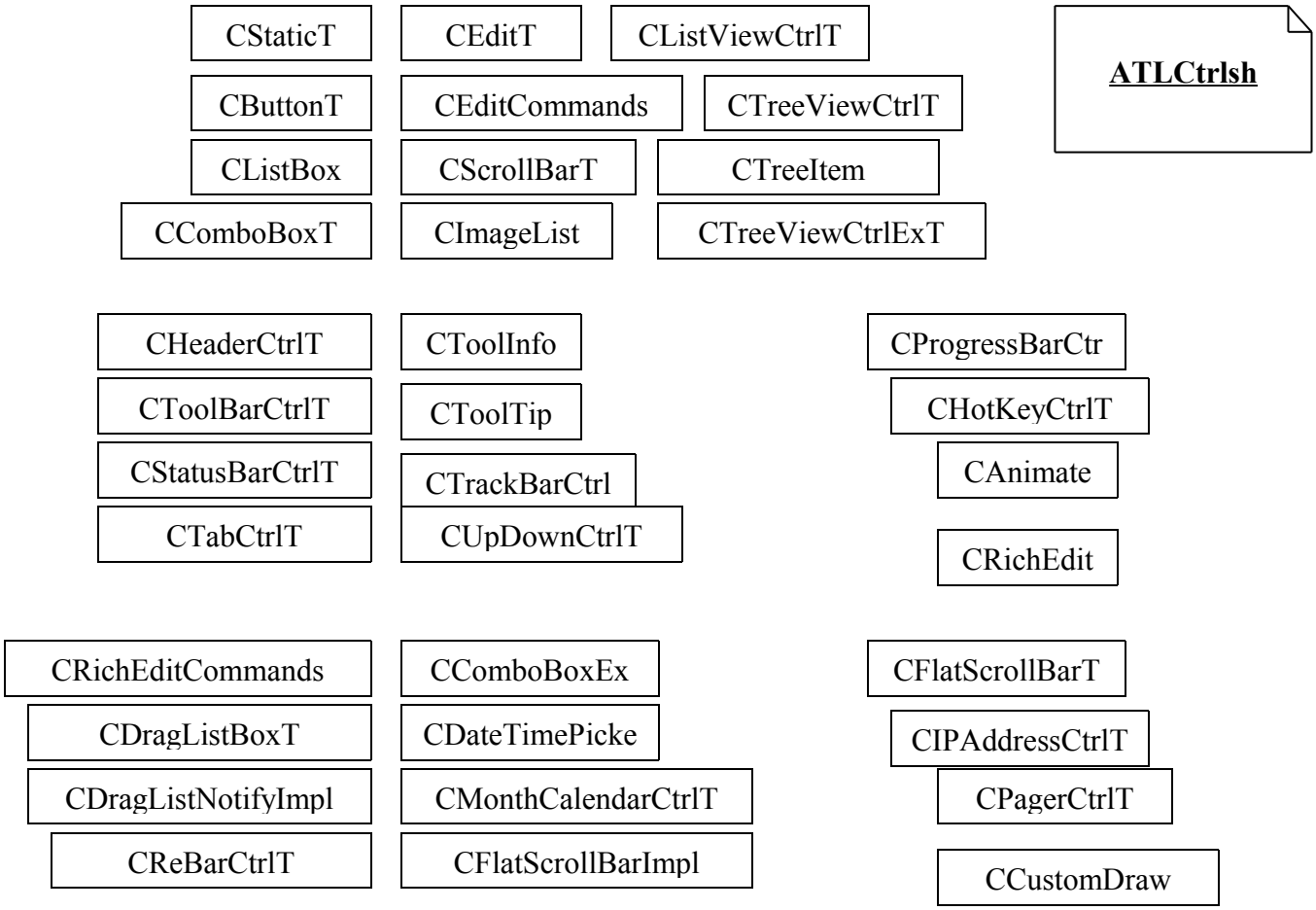

ATLGdih
CPenT
CBrushT
CFontT
CBitmapT
CPaletteT
CRgnT
CDCT
CPaintDC
CClientDC
CWindowDC
CEnhMetaFileDC
CEnhMetaFileInfo
CEnhMetaFileT

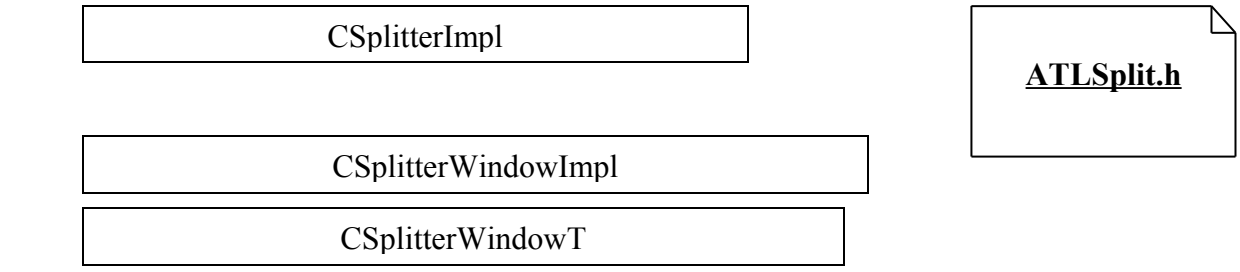

CFileDialogImpl
CFolderDialogImpl
ATLDlgs.h
CFileDialog
CFolderDialog
CFontDialogImpl
CRichEditFontDialogImpl
CFontDialog
CRichEditFontDialog
CPrintDialogImpl
CColorDialogImpl
CFindReplaceDialogImpl
CPrintDialog
CColorDialog
CFindReplaceDialog
CPrintDialogExImpl
CPageSetupDialogImpl
CPropertySheetWindow
CPropertyPageWindow
CPageSetupDialog
CPropertySheetImpl
CPropertyPageImpl
CPropertySheet
CPropertyPage

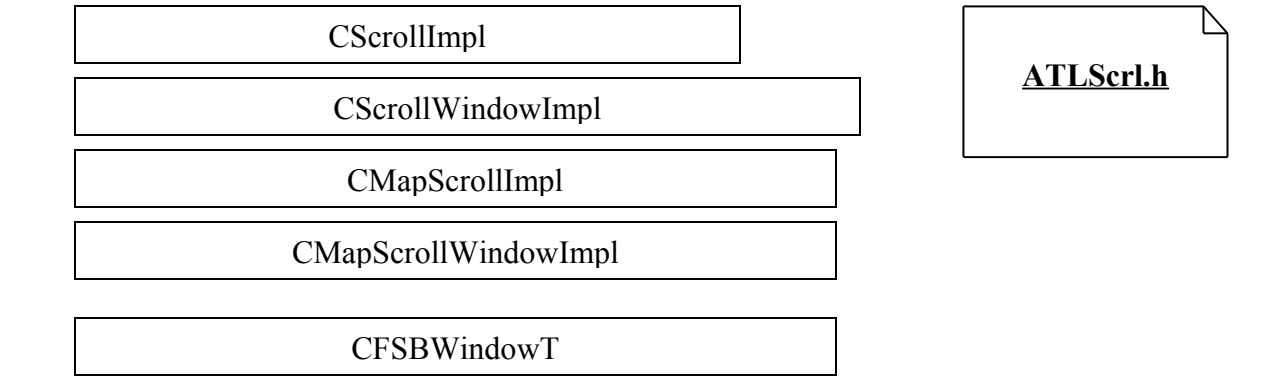

TBase
TWinTraits
CWinDataExchange
. . .
DoDataExchange
DDX_Text
DDX_Int
_AtlSimpleFloatParse
DDX_Float
DDX_Control
DDX_Check
DDX_Radio
OnDataExchangeError
OnDataValidateError
ATLDdx.h
Lots of DDX macros!

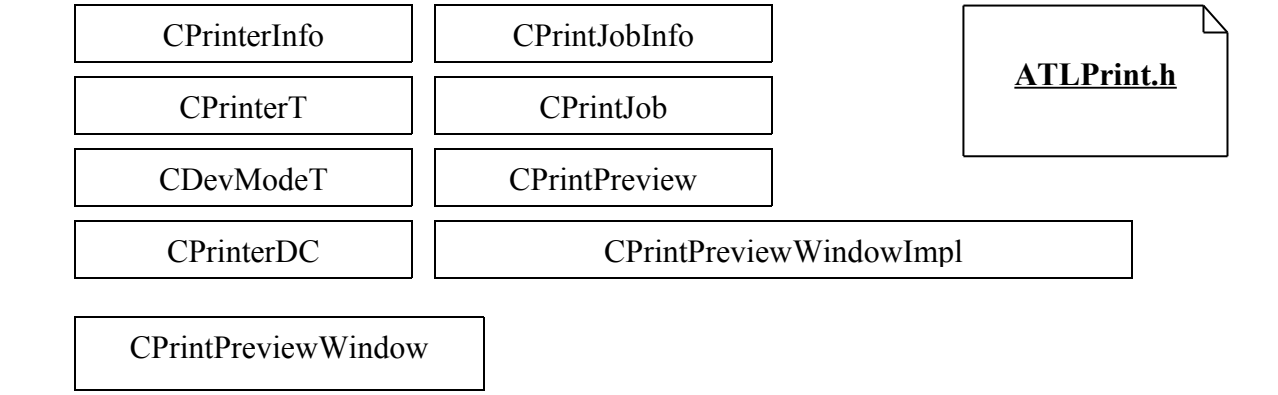

CBitmapButtonImpl
T
Tbase
TWinTraits
CBitmapButton
CCheckListViewCtrlImpl
T
Tbase
TWinTraits
CCheckListViewCtrl
CHyperLinkImpl
T
Tbase
TWinTraits
CHyperLink
CWaitCursor
ATLCtrlx.h
CMultiPaneStatusBarCtrlImpl
T
Tbase
CMultiPaneStatusBarCtrl


ATLCtrlwh
CCommandBarCtrlImpl
T Tbase
TWinTraits
CCommand


CStaticT
CEditT
CListViewCtrlT
ATLCtrlsh
CButtonT
CEditCommands
CTreeViewCtrlT
CListBox
CScrollBarT
CTreeItem
CComboBoxT
CImageList
CTreeViewCtrlExT
CHeaderCtrlT
CToolInfo
CProgressBarCtr
CToolBarCtrlT
CToolTip
CHotKeyCtrlT
CStatusBarCtrlT
CTrackBarCtrl
CAnimate
CTabCtrlT
CUpDownCtrlT
CRichEdit
CRichEditCommands
CComboBoxEx
CFlatScrollBarT
CDragListBoxT
CDateTimePicke
CIPAddressCtrlT
CDragListNotifyImpl
CMonthCalendarCtrlT
CPagerCtrlT
CReBarCtrlT
CFlatScrollBarImpl
CCustomDraw


CSplitterImpl
ATLSplit.h
CSplitterWindowImpl
CSplitterWindowT


CScrollImpl
ATLScrl.h
CScrollWindowImpl
CMapScrollImpl
CMapScrollWindowImpl
CFSBWindowT


CPrinterInfo
CPrintJobInfo
ATLPrint.h
CPrinterT
CPrintJob
CDevModeT
CPrintPreview
CPrinterDC
CPrintPreviewWindowImpl
CPrintPreviewWindow