



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

**SEARCH**
ITKNOWLEDGE[Brief](#) [Full](#)

- [Advanced Search](#)
- [Search Tips](#)

**BROWSE**
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

Introduction

What's on the CD-ROM

Acknowledgments

Chapter 1—The Architecture

The Players

But My Program Doesn't Work Like That!

The Army Officer's Aptitude Test

Frame Windows

About Message Maps

Message Routing

Document Templates

The Details

CWinApp

CView

CDocument

CFrameWnd And Related Classes

CDocTemplate

Navigating Objects At Runtime

Supporting Objects

The CWnd Object

[CObject Support](#)
[About Collections](#)
[Using Templates](#)
[Collection Details](#)
[Summary](#)

Practical Guide To The Architecture

[Handling User Messages](#)
[Creating A New Document Type](#)
[Creating A Private Document](#)
[Attaching Multiple Views To A Document](#)
[Making Separate File/New Menus](#)
[Preventing A New Document On Startup](#)
[Parsing Command Line Parameters](#)
[Calculating A View Size](#)
[Using Typedef With Templates](#)
[A 2D CArray](#)

Chapter 2—Serialization

[Persistence Vs. Storage](#)
[A Quick Look At CArchive](#)
[Inside File Open And Save](#)
[Providing A Custom Dialog](#)
[Another Example](#)
[Inside CDib](#)
[The Sample Application](#)
[Serializing Objects](#)
[Handling Multiple Versions](#)
[Custom Serialization](#)
[Simple Customizations](#)
[Portability Issues](#)
[Summary](#)

Practical Guide To Serialization

[Making A Class Serializable](#)

[Customizing File Prompting](#)
[Using Existing Or Custom File Code](#)
[Creating Archives On Nonstandard Streams](#)
[Reading Old File Versions](#)

Chapter 3—Printing

[MFC Printing—The Big Lie?](#)
[The Dilemma](#)
[A Complete Printing Example](#)
[Customizing Print Preview](#)
[Stripping Down Print Preview](#)
[A Custom Print Preview Example](#)
[Advanced Customizations](#)
[Deriving The Class](#)
[Preview Internals](#)
[Creating An Editable Print Preview](#)
[Summary](#)

Practical Guide To Printing

[Controlling The Print Dialog](#)
[Scaling Printing](#)
[Printing Something Different](#)
[Printing Headers And Footers](#)
[Customizing Print Preview's Tool Bar](#)
[Customizing Print Preview](#)

Chapter 4—Windows, Views, And Controls

[An Improved CListCtrl](#)
[Altering The Control](#)
[Showing The Selection](#)
[Using The Modified List](#)
[Dialog Controls](#)
[General Window Operations](#)
[Setting Styles And Initial Conditions](#)
[Custom Window Classes](#)

- [Restricting Window Size](#)
- [Setting The Title](#)
- [Using UpdateCMDUI](#)
- [About CScrollView](#)
 - [Adding Keyboard Scrolling](#)
 - [Optimizing Scrolling](#)
 - [Scrolling More Than 32K Units](#)
- [About CEditView](#)
 - [Fixing CEditView](#)
 - [CEditView And Splitters](#)
- [About CRichEditView](#)
- [Working With Owner-Draw Controls](#)
 - [The MFC Solution: Self-Draw](#)
 - [Other Solutions](#)
 - [Using Self-Draw Controls](#)
 - [Self-Draw List And Combo Boxes](#)
 - [Self-Draw Menus](#)
- [Editing Tree Or List View Items In Dialogs](#)
- [Splitter Windows](#)
 - [What The User Sees](#)
 - [Programming Splitters](#)
 - [Nesting Splitters](#)
 - [Why Not Use CSplitterWnd?](#)
- [Summary](#)

Practical Guide To Windows, Views And Controls

- [Setting Window Styles](#)
- [Removing The Document Title](#)
- [Setting A Custom Icon, Cursor, Or Background](#)
- [Setting A View To A Specific Size](#)
- [Making List Controls Select In All Columns](#)
- [Scroll Using The Keyboard](#)
- [Scrolling Many Items In Windows 95](#)
- [Using Multiple CEditViews With The Same Document](#)
- [Setting Formatting For CRichEditView](#)
- [Using Owner-Draw \(Or Self-Draw\) Controls](#)

Effectively Using Label Editing For List And Tree Controls
In Dialog Boxes

Nesting Splitter Windows

Chapter 5—Dialogs

MFC And Dialogs

Implementing Modeless Dialogs

Using DDX/DDV

About Data Validation

Live Data Validation

Other Data Map Tricks

Adding Custom DDX/DDV

Integrating With Class Wizard

Using Dialog Bars

Customizing Tool Bars

Customizing Common Dialogs

Customizing Step By Step

An Example Color Dialog

Customizing File Open

Summary

Practical Guide To Dialogs

Creating Modeless Dialogs

Updating DDX Variables On Changes

Live Data Validation

Writing Custom DDX/DDV Routines

Integrating Custom DDX/DDV With Class Wizard

Dialog Bars Vs. Tool Bars

Customizing Common Dialogs

Chapter 6—Property Sheets And Wizards

Property Sheet Overview

Using A Single Template

Wizard Mode

Modeless Property Sheets

Custom App Wizards

Creating A Wizard

Customizing The Customizer

Creating The Project

Other Options

Pressing On

Debugging Wizards

More Ideas For Wizards

Summary

Practical Guide To—Property Sheets And Wizards

Creating A Property Sheet

Creating A Wizard

Using A Single Template

Modeless Property Sheets

Making Custom App Wizards

Chapter 7—DLLs And MFC

The Link Process

Language Considerations

Using An Ordinary DLL

Creating An Ordinary DLL

The Main File

Exporting Functions

Private And Shared Variables

MFC DLLs

What About OLE (Or ActiveX) DLLs?

Summary

Practical Guide To DLLs And MFC

Determining What DLLs A Program Uses Or Functions A

DLL Exports

Linking At Build Time

Linking At Runtime

Creating A DLL

Exporting Functions And Data
Creating An MFC Extension DLL
Optimizing DLL Load Addresses

Chapter 8—ActiveX

What Is An ActiveX Object?

ActiveX And OOP

ActiveX Encapsulation

ActiveX Reuse

ActiveX Polymorphism

Fun With Interfaces

Properties

Methods

Events

Names Vs. Numbers

ActiveX And MFC

MFC And ActiveX Controls

Using Control Wizard

Code You Add

Adding Properties

Using Ambient Properties

Adding Methods

Adding Events

Adding Property Sheets

Examining The Generated Files

Testing And Using The Control

A Simple Control

Using ActiveX Controls

Summary

Practical Guide To ActiveX

Making An MFC Object With An IDispatch Interface

Interpreting CLSIDs, PROGIDs, And The Registry

Creating ActiveX Controls

Debugging ActiveX Controls

Allowing VB Or Web Developers To Initialize Your ActiveX Control

What Is ATL?

Adding Property Sheets

Using ActiveX Controls

Chapter 9—MFC And The Internet

An Internet Primer

TCP/IP

Sockets

Protocols

Inside HTTP And URLs

ISAPI

ActiveX And Java

MFC Sockets

Using Archives With CSocket

Going Deeper: CAsyncSocket

Blocking Calls

The Example

The Basic Framework

Adding A Custom Socket

Other Considerations

Socket Wrap Up

Higher-Level Protocols

The Link Checker

Other Ideas

ActiveX Internet Support

The Transfer Control

ISAPI Support

The Plan

A May-December Marriage

A Quick Look At ISAPI

Writing The HILO.DLL Server

Inside The C++ DLL

Installation And Distribution

Future Directions

[Traditional MFC ISAPI](#)

[Summary](#)

[Practical Guide To MFC And The Internet](#)

[Using Sockets](#)

[Using Sockets As Streams](#)

[Using WinInet With MFC](#)

[The Internet Transfer Control](#)

[Writing ISAPI Extensions And Filters With MFC](#)

[When Not To Use ISAPI](#)

[CBISAPI—An Object-Oriented Approach To ISAPI](#)

[Chapter 10—MFC And Databases](#)

[Database In Detail](#)

[Adding More Features](#)

[Adding And Deleting Records](#)

[Not Using A View](#)

[An Example Program](#)

[Examining The Example](#)

[Summary](#)

[Practical Guide to MFC And Databases](#)

[Starting A Database Application](#)

[Selecting ODBC Or DAO](#)

[Setting Up A Data Source](#)

[Binding Database Fields To Recordset Variables](#)

[Binding Recordset Variables To Controls](#)

[Deleting Records](#)

[Adding And Updating Records](#)

[Working With Computed Fields](#)

[Chapter 11—Multithreading](#)

[Threads Vs. Processes](#)

[Problems With Threads](#)

[Threads And MFC](#)

[Creating An MFC Worker Thread](#)

[Creating An MFC User-Interface Thread](#)

[Manipulating Threads](#)

[Learning The Return Value](#)

[Synchronizing Threads](#)

[Types Of Synchronization Objects](#)

[Alternatives To Threading](#)

[An Example Program](#)

[Summary](#)

[Practical Guide to Multithreading](#)

[Creating A Worker Thread](#)

[Creating A User-Interface Thread](#)

[Terminating A Thread](#)

[Making Windows Appear On Top](#)

[Making Message Boxes Appear On Top](#)

[Preventing Autodestruction Of Threads](#)

[Creating A Suspended Thread](#)

[Learning The Return Value](#)

[Types Of Synchronization Objects](#)

[Waiting For A Synchronization Object](#)

[Waiting For Multiple Synchronization Objects](#)

[Using OnIdle](#)

[Chapter 12—The End Of The Road](#)

[The End Of The Road?](#)

[Things To Come](#)

[ICs Vs. Core Memory](#)

[Other Resources](#)

[Appendix A](#)

[Appendix B](#)

[Index](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Table of Contents](#)

Introduction

Are you an MFC programmer? Good. There are two types of MFC programmers. What kind are you? The first kind are the *good* programmers who write programs that conform to the way MFC wants you to do things. The second bunch are wild-eyed anarchists who insist on getting things done their way. Me, I'm in the second group. If you are in the same boat (or would like to be) this book is for you.

This book won't teach you MFC—not in the traditional sense. You should pick it up with a good understanding of basic MFC programming and a desire to do things differently. This isn't a Scribble tutorial (although I will review some fundamentals in the first chapter). You will learn how to wring every drop from your MFC programs. You'll discover how to use, abuse, and abandon the document/view architecture. If you've ever wanted custom archives, you'll find that, too.

Why This Book?

You have to have a license to practice medicine. Or do you? I was reading an editorial the other day that proposed that specific employees at HMOs and insurance companies were, in effect, practicing medicine without a license. Here's how it works: Your doctor suggests an expensive procedure for you, but your insurance company won't cover it because it isn't (in their opinion) necessary. So you don't have the procedure.

The insurance company would argue that they aren't practicing medicine. You

are free to get the procedure done at your own expense. However, with the price of medicine today, an insurance company's refusal to pay is tantamount to refusing you the treatment.

Windows programming has some interesting parallels to this situation. OLE is difficult to do, so you use MFC. That makes it easy. Sure, you are free to implement OLE on your own (perhaps a man-year of development). But if you want to do it in three days, you'd better stick with MFC.

Generally, MFC is a good thing. But if you use it for something, you essentially have to buy into everything it provides. You can't just use the OLE part (or the print preview part, or splitter windows). When you use MFC, you are agreeing to do things the MFC way.

Even inside MFC, you'll find the same phenomenon. Did you know that you can use dynamic data exchange (DDX) with any window that child controls? However, Class Wizard only helps you work with certain kinds of windows (like dialog boxes). Because using DDX without Class Wizard is poorly documented, this—in effect—limits how you use DDX.

One of the great things about being a programmer is that you get to create. I often think that software, if it's done right, is one of the purest forms of creation available to man. Think about it. You dream up an idea for new software, mumble over your keyboard for a few hours, and voilà, a new creation comes to life. If you type well enough, it is almost as though the program pours from your mind to the computer. Tools like MFC should help you realize your creations, not constrain them.

Ye Olde Days

The first computers I did any serious programming on were embedded microcontrollers that I designed. Here, creativity was king. Anything I could dream up, I could write (subject to my 4K of ROM). The problem was that you had to dream up every detail, no matter how small. In those days, I spent time writing code to pick apart bytes and send them over a software-driven RS232 port. I had to count interrupts to figure out the time of day. Clearly, too much creativity can be a bad thing.

Today, you don't have to deal with these minute details. If you want to send something to a serial port, you open it and dump bytes out of it. Better still, if you want to talk to a mouse, a modem, or a printer, there are higher-level ways of accessing them so that you don't even have to care about the serial port that might connect them to the PC. But what you gain in productivity, you lose in creativity. Whereas at one time you could control every detail, now you must suffer the whims of whatever operating system you use.

This isn't necessarily a bad thing. Who wants to write their own routines to read and write to the hard disk? Will the routine work with every hard disk on the market? Do you really want to write code for each type of printer and display card? Usually, the answer is no. But these are low-level details. What about the higher-level details, such as user interface style? Windows constrains you somewhat even at this level. But here's the catch: The higher-level the tool, the more it constrains you.

The ultimate example is languages like Visual Basic. VB is great at doing certain things. If your program does the things that VB is good at doing, you'd be crazy not to use it. Point, click, and ship. But what happens when your program does things that VB isn't good at? That's different. You can often tweak VB to do something that it isn't supposed to, but that takes work. At times, it can take lots of work. Often, you are better off using a more flexible language. Therefore, VB, like all tools, sets constraints on the kinds of solutions you design.

MFC Constraints

What constraints do you suffer as an MFC programmer? Plenty, even if you don't realize it. MFC constrains you in several ways:

- MFC has a distinct architecture that many of its pieces rely on.
- When MFC offers a component or a class, it is often easier to use it as-is than it is to develop a new one (or enhance the old one).
- MFC's tools (App Wizard and Class Wizard) only work with certain kinds of programs; it is difficult to create other kinds of programs because they won't help you.

This book is all about breaking free of those constraints. Sometimes that means taking on a lot of work. Sometimes it'll be easy, if you know how. In every case, you'll achieve your goals by working with MFC. You don't have time for hacks that might not work with later versions of MFC.

Of course, being different for its own sake isn't a good idea either. For example, putting your File menu in the middle of the menu bar violates the famous Shock Minimization Principle (SMP). This is the software engineering axiom that states: *Software shall operate in such as way as to minimize shock to the most users.*

On the other hand, there are plenty of cases where you'd like something to work a bit differently. You'd like to exercise your creativity and develop something that no one has ever seen before, right? That's the goal of this book: To help you to realize *your* programming vision.

Using This Book

To get the most from this book, you should have a copy of Microsoft's Visual C++ 5.0 or later. You can probably use most of the techniques with other versions of C++ from Microsoft or other vendors, but the code in the listings and the CD-ROM use VC++ 5.0.

Each chapter covers a particular topic. The first part of the chapter contains detailed information about the chapter's topic. The second part shows practical problems and cookbook-style solutions to them. This might be references to the MFC technical notes, Web pages, or magazine articles.

Each chapter can stand apart from the rest. If you have a specific problem, you might want to thumb through the practical guide sections at the end of each chapter. Of course, you should only use the solutions as a guide. After all, you

want to exercise your own creativity.

Perhaps you have a vision for your software. Maybe your customers, your boss, or your competition are pushing you to produce your software a certain way. In either case, pick a chapter and learn how to do things your own way.

Table of Contents

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Table of Contents](#)

What's on the CD-ROM

*The companion CD-ROM for **MFC Black Book** contains the following elements, specially selected to enhance the use of this book:*

Reusable MFC classes ready to use in your programs

Demos of several developer's tools from Sax Software

A demo of Elsinore's Visual Intercept bug-tracking software

The complete set of source code listings from the book

See the *readme* files in each folder for acknowledgments, descriptions, copyrights, installation instructions, limitations, and other important information.

Requirements

Software: A windows C++

compiler with MFC; for
example, Visual C++
5.0 or better

Hardware:

486DX/66 or higher processor
(Pentium 90 recommended)

[Table of Contents](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Table of Contents](#)

Acknowledgments

When you go to the bookstore and look around, you see plenty of books. Most of them only have one name on the cover. A few have two or three names on their glossy jackets. But the truth is, it is amazing how many people have to work on these books. We should have credits, like at the end of a movie, perhaps.

Unfortunately, books don't have ending credits, so this is my one chance to say thanks to everyone who helped make this book a reality. Sure, you want to dig right into the book, but take a minute and find out who was behind the scenes. Besides, I am bound to forget someone, and if it is you, you can always send me an email and complain.

Of course, all the fine folks at Coriolis are partially to blame for this book. Jeff Duntemann, Keith Weiskamp, and Michelle Stroup are, of course, the ringleaders, but there are many others who deserve a tip of your hat for a job well done.

I couldn't write books without the support of my family. My wife Pat puts up with so much so that I can do all the things that I love to do. Our three kids, Jerid, Amy, and Patrick, are used to seeing me sitting in my office as deadlines draw near. Even our dachshund Madison keeps me company quietly while I am working much of the time.

And then there are the people who didn't have to help, but did anyway. Particularly, I want to mention Mark Uland (the author of *Visual Intercept* and the only person I know who has a sense of humor remotely like mine). Mark is

an MFC guru (among other things), and when one of us calls the other with a problem, it is sure to be an interesting (and probably frustrating) one. Many of our discussions resulted in code that wound up in this book. Jim Miller, an old friend that I go to with my X Windows questions, made the transition to MFC recently, and you can hear echoes of many of our conversations in this book, as well. Jim was also kind enough to share his comments on several of the chapters.

Finally, I should thank my good friend (and fellow ham radio operator) Curt Tallman for his timely loan of a ZIP drive. Without it, I'd still be sending the listing files for this book over my modem. Thanks Curt!

I sincerely believe you will enjoy this book. If you don't, blame me. But if you do, remember all the other people who have had a hand in it. Without them, this book would never have seen the light of day.

For Billy Monti, a friend I can't remember being without, and one who has helped my family and me so much in this troubling year.

And, as always, for my wonderful wife Pat, who I am lucky to have.

Table of Contents

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Chapter 1

The Architecture

Before you can tackle advanced MFC programming techniques, you need to have a solid grasp of the basics. Use this chapter to review your understanding of fundamental MFC architecture, including the document/view architecture, message maps, command line parsing, sub-classing windows, and using collections.

Programmer's Notes...

Ancient Romans were bad at math. Of course, the fact that they neglected to discover the number zero didn't help things any. In the ancient western world, the Arabs were the great mathematicians. They learned about it from the people of India who had discovered it around 200 B.C. On the other side of the pond, the Mayans also had zero and were very mathematically sophisticated. Their calendar was more accurate than the one we use today.

Imagine if ancient man had started counting in binary instead of decimal. Then we could all count to 1,023 on our fingers. That isn't as farfetched as you might think. Both the Teutonic and Celtic peoples counted in base 12. That's why we still count in dozens and there are 12 hours on the clock (not to mention 12 inches in a foot). That is also why we have special words for eleven and twelve (instead of oneteen and twoteen).

The point is, the tools you use can shape your solutions to problems. Look at

pictographic languages like Chinese and some forms of Japanese. These languages don't lend themselves very well to computers. This has presented major obstacles to computing in the Far East. For example, a user typing on a laptop might enter a word phonetically using the Kanji alphabet. Then the laptop displays several pictograms (all homonyms) and the user selects the proper one. This isn't very efficient. On the other hand, pictographic languages are much easier for handwriting recognition systems. Many Japanese palmtop computers allow you to simply write input on the screen. Only a few of the English-speaking palmtops do, and those usually don't do a very good job. Our written letters aren't distinctive enough for computers to recognize easily.

Every aspect of your development environment influences your programs. The fact that you develop for Windows, for example, is sure to alter how you write programs. MFC, C++, and the tools you use to build code all influence your programs, as well.

Although one of the purposes of this book is to discover how to do things differently, you need to understand how your tools (in this case, MFC and Visual C++) work. There are two reasons you want to do this. First, you need to know how MFC works so that you can make it do what you want. Second, you often don't know exactly how MFC does things because the tools that it uses (App Wizard and Class Wizard) do the work for you. The Wizards are like a VCR that uses the VCR Plus codes. As long as you have the codes, no problem. But if you want to tape something that doesn't have a VCR Plus code, you are in trouble. Not only do you have to program the VCR, but you probably don't know how to do it.

The first goal of this chapter is to discuss each of the major portions of MFC and how they relate to the overall architecture. Second, you'll dissect an App Wizard-generated program to see *why* it does the things that it does.

If you've written a great deal of MFC code, you may think you don't need to read this section. You may not. See if you can answer the following questions. If you can, you can probably skip this section safely. You'll still want to skim the practical section later in this chapter. Here goes:

- What is the only class an MFC program must have?
- How can you manually construct a message map?
- How can you attach extra views to a document?
- How can you create new document types?
- When does the document *not* handle I/O?
- When does the view *not* perform a program's drawing?
- Can you put a menu handler in a document class?
- Why can't **CRect** derive from **CObject**?
- Do document templates have to reside in the application object?
- Why is it difficult to make a two dimensional **CArray** with templates?

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

The Players

There are many classes that make up MFC. However, only a few of them are core components that influence your program's architecture. The other classes don't contribute directly to the way you write programs.

The core classes are: **CWinApp**, **CView**, **CDocument**, **CFrameWnd**, and **CDocTemplate**. Although these five classes form the core of most MFC programs, they are not always required. In fact, the only class absolutely necessary for an MFC program to use is **CWinApp**. However, to get the most from MFC, you'll use the other classes (or classes derived from them).

The power of MFC is in a technique sometimes referred to as "design by difference." The idea is that MFC provides classes that form a model Windows program. However, this prototypical program doesn't do anything interesting. Your job as an MFC programmer is to write the code that makes your application different. MFC takes care of all the default behavior, while allowing you to selectively override that behavior as you see fit.

CWinApp represents your program running in memory. The user never sees anything that relates to **CWinApp** directly. This is the key place to retrieve data related to the application (for example, the command line, resources, instance handles, and so on).

You'll find key **CWinApp** members in Table 1.1. The most important of these are the two overrideable functions **InitInstance** and **ExitInstance**. Normally, App Wizard places code in **InitInstance** to create your main window and perform other initialization tasks. However, you can conceivably do all the

work right here and then return **FALSE** to terminate your program. If you ask App Wizard to create a dialog-based program, that is exactly what it does (see Listing 1.1). In this case, the program brings up a dialog box, but it could do anything you like (as long as it doesn't require an event loop).

Table 1.1 Key CWinApp members.

Member	Description
m_pszAppName	Application's name
m_lpCmdLine	Command line
m_hInstance	Application's handle
m_hPrevInstance	Previous instance handle
m_pMainWnd	Main application window
m_bHelpMode	TRUE if in context help mode
m_pszExeName	Name of EXE file
m_pszProfileName	Name of INI file
m_pszRegistryKey	Name of registry key to use instead of INI file
LoadCursor	Loads an application cursor
LoadStandardCursor	Loads system cursor (IDC_*)
LoadOemCursor	Loads OEM cursor (OCR_*)
LoadIcon	Loads an application icon
LoadStandardIcon	Loads system icon (IDI_*)
LoadOemIcon	Loads OEM icon (OIC_*)
ParseCommandLine	Initialize a CCommandLineInfo object based on the command line
ProcessShellCommand	Process shell commands in a CCommandLineInfo object
GetProfileInt	Get integer from INI file
WriteProfileInt	Write integer to INI file
GetProfileString	Get string from INI file
WriteProfileString	Write string to INI file
AddDocTemplate	Add a document template to application
GetFirstDocTemplatePosition	Find POSITION preceding the first document template in the list
GetNextDocTemplate	Retrieve next document template from the list
OpenDocumentFile	Open document file by name
AddToRecentFileList	Add name to recently opened file menu
CreatePrinterDC	Fetch a printer DC based on the selected printer
GetPrinterDeviceDefaults	Gets printer defaults for printer from WIN.INI or last print setup
InitInstance	Per instance initialization
Run	Event loop
OnIdle	Idle time processing

ExitInstance	Program cleanup
PreTranslateMessage	Filter message
SaveAllModified	Prompts to save document if needed
DoMessageBox	Override to modify AfxMessageBox()
ProcessMessageFilter	Process messages for filter hook
ProcessWndProcException	Default handler for exceptions
DoWaitCursor	Turns hourglass on and off
WinHelp	Opens the help file
LoadStdProfileSettings	Loads standard INI file settings and MRU file list
SetDialogBkColor	Sets color of dialog backgrounds
SetRegistryKey	Inform MFC that it is to use the registry instead of an INI file for persistent storage
EnableShellOpen	Allows drag and drop document open
RegisterShellFileTypes	Registers document types
OnFileNew	Default menu handler
OnFileOpen	Default menu handler
OnFilePrintSetup	Default menu handler
OnContextHelp	Default menu handler
OnHelp	Default menu handler
OnHelpIndex	Default menu handler
OnHelpFinder	Default menu handler
OnHelpUsing	Default menu handler

Overriding **InitInstance** and **ExitInstance** is a prime example of the design by difference philosophy. The default **InitInstance** routine doesn't do anything. It is a good bet that your program needs to create a main window and perform other start-up tasks. Therefore, you nearly always override **InitInstance**. On the other hand, you may not need to do anything when your program exits. If you don't, don't write an **ExitInstance** function. Of course, you are free to override it if you need exit processing.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

BROWSE
BY TOPIC

A great deal of the power you get from MFC, such as print preview, splitter windows, and more, only appears when you use the document/view architecture. This is the main architectural feature of most MFC programs.

Listing 1.1 Excerpt from an App Wizard Dialog application.

```

BOOL CDlgdemoApp::InitInstance()
{
    AfxEnableControlContainer ();

    // Standard initialization
    // If you are not using these features and wish to reduce the size
    // of your final executable, you should remove from the following
    // the specific initialization routines you do not need.

#ifdef _AFXDLL
    Enable3dControls();          // Call this when using MFC in a shared DLL
#else
    Enable3dControlsStatic();    // Call this when linking to MFC statically
#endif

    CDlgdemoDlg dlg;
    m_pMainWnd = &dlg;
    int nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
        // TODO: Place code here to handle when the dialog is
        // dismissed with OK
    }
    else if (nResponse == IDCANCEL)
    {
        // TODO: Place code here to handle when the dialog is
        // dismissed with Cancel
    }
}

```

```

    // Since the dialog has been closed, return FALSE so that we exit the
    // application rather than start the application's message pump.
    return FALSE;
}

```

To best understand the document/view architecture, consider a case in which you don't use it. Suppose I wrote a simple spreadsheet program that draws data from atmospheric sensors. It isn't very fancy; just an ordinary grid that handles numbers and formulas. Since it was a simple job, I didn't bother using a document and a view. I simply drew the grid as an integral part of the processing. A few months later, you inherit my spreadsheet and the boss asks you to add a bar graph display to the spreadsheet.

Now things aren't so simple. You'll have to change all of my drawing code to decide if it is sketching a chart or the original grid. What happens if you want to show both while using the same data? What if you want to show two separate pieces of data in two different ways? You may run into big trouble.

Life is easier, in this case, if you use the document/view architecture. First you define a document object (derived from **CDocument**). This object is responsible for maintaining the data in the program (in this case, the numbers, formulas, and sensor data). It should load and save data (probably to a file) and recalculate each formula. However, it has no responsibility for drawing anything to the screen. It also has no user interface responsibility.

Drawing and handling the user interface is the purview of the view class (derived from **CView**). Each view refers to one document. The view's job is to:

- Draw a representation of the data in the associated document.
- Accept user manipulation of the data (for example, mouse clicks) and update the document accordingly.

There are many advantages to this scheme. Reconsider the spreadsheet example. This time, imagine that I used the document/view model. Now, adding a bar chart is trivial. Making changes to the document (for instance, reading different sensors) is easy, too.

It is also simple to have one document that drives multiple views at the same time. You'll see examples of this in the practical section of this chapter. Users never see or manipulate a document object directly. The user only sees the view.

But My Program Doesn't Work Like That!

Often, people think their programs don't fit the document/view architecture. Sometimes they are correct, but more often, their program will fit the model, given a little thought. I blame Microsoft for this problem because of its poor choice of names.

The issue is the word document. You usually think of a document as a file on disk. In fact, the document object often (but not always) represents such a file. If you are writing a spreadsheet, for example, your document probably does represent a file. But what if you are writing a program that displays real-time weather data from a cluster of instruments? In that case, the document is just the data from the instruments. If you are writing a network monitor program, your document contains the performance data for the network.

Smalltalk uses the same type of design for its windowed applications. They call it the "Model View Controller" architecture (an MFC view corresponds to a Smalltalk view and a controller combined). A Smalltalk model is the same as an MFC document. I prefer the word "model" because that helps you remember it doesn't have to be a file.

Tip: What's A Document?

Remember: A document is just an abstract representation of your program's data. It doesn't matter where the data comes from.

Sometimes it takes a bit of thought to decide what goes in your document (or file, for that matter). A while back, I wrote a program that was sort of a dumb terminal (the program appeared in the December/January 1996 issue of *PC Techniques*, now *Visual Developer Magazine*). This is a classic case where the document/view scheme doesn't appear to work (especially because I used **CEditView** for the view class).

What do you save to a file in a program like this? The answer turns out to be configuration information. When you load a file into this program, you are not interested in seeing the last 200 lines of output from the remote computer. What you want is that the program remembers you are on COM2 at 9600 baud, with no parity.

Tip: Saving Document Data

It is perfectly all right to store data in the document object that isn't permanent. You only need to save things that you want restored later.

Practically every user-driven program can be split into these two parts: abstract data (the document) and a representation of the data for the user to view and manipulate (the view). Your job is to figure out how to make that split in your particular program.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

The Army Officer's Aptitude Test

Have you ever taken the Army officer's aptitude test? It only takes a minute. Here's the setup: You are a Lieutenant. You have a Sergeant, two Privates, two 9' poles, an 11' pole, three 6' lengths of rope, a post hole digger, and a U.S. flag. In 30 seconds or less, tell me how you will erect a 14 1/2' flagpole.

The correct answer is: "Sergeant, get that flagpole up!"

This has a lot to do with MFC. You might have noticed that in the preceding sections I never said what a document or a view *does*. Instead, I talked about the responsibilities of each piece. Although MFC insists that each object take responsibility for certain actions, it doesn't require that that object actually perform the task.

Most of the time, the objects in question actually do perform these tasks. However, it is possible for the objects to delegate the task to another object. For example, you might have non-MFC objects that know how to draw your data. That's not a problem; use only those objects inside the view.

Another example is **CEditView** (a view class that MFC provides to do text editing). This class knows how to read and write ASCII text files. Still, the document is ultimately responsible for reading and writing files. If the document chooses to find the view class and delegate the actual operation to it, MFC doesn't care.

CEditView is a perfect example of why you might want to delegate an operation. Suppose you are writing a class that handles GIF files. You'd like to share your class with other programmers on a variety of projects. Sure, you

could write **CGIFView** and **CGIFDoc**, but that isn't very reusable. Some programmers might want to put a GIF file in their program with other things. Sure, they could derive new classes, but what if they also wanted to use **CPCXView** and **CPCXDoc** (to handle PCX files, too)? As MFC doesn't care for multiple inheritance, this would be a problem.

Using delegation, there are at least two possible answers. First, you could create a **CGIFDoc** that knows how to read and write GIF files. It would also know how to draw GIF files. The view would be responsible for calling the document's drawing routine. If the program called for multiple documents, you'd construct a super document that managed the various documents required.

Another solution might be to create an independent class (perhaps **CGIF**) that has functions that the document and view object can call to do the necessary work. This class would not need to derive from an MFC class (although you might derive it from **CObject**, a class you'll find out more about later in this chapter).

Tip: Using MFC Classes

You don't have to put all of your code in an MFC class. Feel free to derive new classes to represent your program. This can help you reuse existing code or use features MFC doesn't support well (like multiple inheritance).

Frame Windows

Views can't simply appear in the middle of the user's screen. Users expect the view to have all the usual accoutrements: resizable borders, a caption bar, a system menu, and so on. Although it isn't usually clear to users, those things actually reside in the frame window. What users think of as one window is usually two: the frame and the view.

Frames come in two main flavors. SDI (Single Document Interface) programs have only one window. A common example of an SDI program is the standard Notepad editor (see Figure 1.1). It only has one window and it works on one file at a time. If Notepad was an MFC program (it isn't, as far as I know), its interior would be a view and the border, menu bar, and system menu would belong to a class derived from **CFrameWnd**.

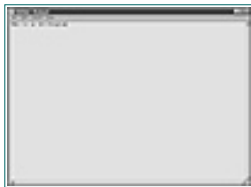


Figure 1.1 Notepad, a typical SDI program.

MDI (Multiple Document Interface) programs use the **CMDIFrameWnd** class for their main frame window. These programs (like Microsoft Word) place each open document in their own child window (see Figure 1.2). The child window consists of a **CView**-derived class and a **CMDIChildWnd**.

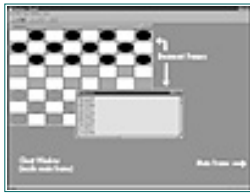


Figure 1.2 An MDI program (annotated).

Views and frames are both derived from the generic window class (**CWnd**). You can do anything to a window that you can do to a view or a frame. An SDI program, then, contains two different windows (the view and the frame). MDI programs contain one frame window, and a frame and view for each open document. MDI programs also have a client window that MFC largely ignores (see Figure 1.2). This is the window that is inside the main frame window. Although you can't really see it directly, it is a window and can be seen with a spy program. It is the parent window for the document frames. Without it, the document frames could overlap the tool bar and status bar. In both cases, the main frame window owns the menu, as well as tool bars, status bars, and the similar items.

This leads to an interesting question: Does all menu handling code belong in the frame? The answer is no. To find out exactly why, you need to find out how MFC programs process messages.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

About Message Maps

If you only use Class Wizard to handle messages, you might not know how it does the magic that it does. Somehow it causes incoming Windows messages to call specific functions in your class.

If you were to design a class library, you might be tempted to use virtual functions to handle messages. **CWnd** could have a virtual function named **OnSize**, for example. Then your standard event loop could arrange to call **OnSize** in response to **WM_SIZE** messages.

That would work in many cases. However, it has two major drawbacks:

- Most windows only process a small number of messages, yet each window would require a giant virtual function table with entries for each message.
- Virtual functions don't easily handle user-defined messages, registered messages, and other custom cases.

To avoid these problems, Class Wizard simulates virtual functions using a message map. A message map is simply an array of entries that MFC consults when deciding how to dispatch a message. The array stores several pieces of critical information:

- The message to handle
- The control ID the message applies to, if any (explained in the following paragraph)
- The arguments the message passes
- The return type the message expects

The second item is especially important. Certain messages (like **WM_COMMAND** and **WM_NOTIFY**) are further divided to apply to certain IDs. So you rarely write a handler for **WM_COMMAND**. Instead, you write a handler for **WM_COMMAND** with an argument of **ID_MENU_FILE_OPEN** (or whatever).

This scheme has several important ramifications. First, it parses the traditional **wParam** and **lParam** arguments so that your function receives unpacked arguments of the correct type. Second, the message map array is very flexible and allows you to put handlers in for any message you want.

This is especially important as the message map handles specific **WM_COMMAND** messages. I could have hundreds of unique command IDs in a program. No one could guess which IDs I might choose. By

using the message map, the choice of ID is unimportant as long as I properly construct the map.

When MFC receives most messages, it determines the target window and the corresponding MFC class instance. Then it searches that window's message map for a match. If the window does not have a handler for that message, MFC searches the window's base class recursively. If there are no more base classes, and MFC found no handler, MFC forwards the message to the original window procedure for the window (in other words, the non-MFC message handling code).

Of course, manually constructing the message map's data would be a tedious and error prone task. At a high level, MFC provides Class Wizard to do this job. But even at a lower level, there are macros that Class Wizard uses to simplify the process.

There are several macros that Class Wizard uses, and a few that you can use but Class Wizard doesn't. You shouldn't have any problem adding your own macros to an existing message map. Make sure you put your additions outside of Class Wizard's special comments. Consider this message map from an App Wizard-generated program:

```
BEGIN_MESSAGE_MAP(CLinkckView, CFormView)
    //{AFX_MSG_MAP(CLinkckView)
    ON_BN_CLICKED(IDC_SCAN, OnScan)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, OnFilePrintPreview)
    ON_COMMAND(ID_FILE_SCAN, OnScan)
    ON_LBN_DBLCLK(IDC_LB, OnScan)
    ON_UPDATE_COMMAND_UI(ID_FILE_PRINT, OnUpdateFilePrint)
    ON_UPDATE_COMMAND_UI(ID_FILE_PRINT_PREVIEW, OnUpdateFilePrint)
    //}}AFX_MSG_MAP
// Put your extra macros here
// Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CFormView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CFormView::OnFilePrint)
END_MESSAGE_MAP()
```

The macros between the **AFX_MSG_MAP** lines are from Class Wizard. After the second **AFX_MSG_MAP** comment line, you can put anything you want and Class Wizard won't know about it. The two printing macros already there are from App Wizard. It places them there so they don't show up in Class Wizard.

As you can tell from the previous message map example, the two most basic macros are **BEGIN_MESSAGE_MAP** and **END_MESSAGE_MAP**. These simple macros just start the message map table, make a few basic entries, and close the table again. In between are the important macros (see Table 1.2). Of course, there are also macros for each specific message (for example, **ON_WM_CLOSE** for **WM_CLOSE** or **ON_WM_PAINT** for **WM_PAINT**).

Table 1.2 Useful message map macros.

Name/Arguments	Description
ON_COMMAND ID func	Handles WM_COMMAND messages Control ID associated with the WM_COMMAND message Member function to call [void func(void)]>
ON_COMMAND_RANGE ID IDLast func	Handles a range of IDs for a WM_COMMAND message First ID of range Last ID in range Member function to call [void func(WORD id)]
ON_COMMAND_EX	Like ON_COMMAND, but handler function receives ID as an argument and returns BOOL

ID	Control ID
func	Member function to call [BOOL func(WORD id)]
ON_COMMAND_EX_RANGE	Like ON_COMMAND_EX, but for a range of IDs
ID	First ID of range
IDLast	Last ID in range
func	Member function to call [BOOL func(UINT id)]
ON_UPDATE_COMMAND_UI	Handles MFC's request for state of this user interface item
ID	Control ID
func	Member function to call [void func(CCmdUI *pCmdUI)]
ON_UPDATE_COMMAND_UI_RANGE	Like ON_UPDATE_COMMAND_UI, but for a range of IDs
ID	First ID
IDLast	Last ID
func	Function to call [void func(CCmdUI *pCmdUI)]
ON_NOTIFY	Handles WM_NOTIFY messages from new-style controls (for example, the common controls)
code	Notification code
ID	Control ID
func	Function to call [void func(NMHDR *hdr,LRESULT *result)]
ON_NOTIFY_RANGE	Similar to ON_NOTIFY, but for a range
code	Notification code
ID	Control ID
IDLast	Last ID
func	Function to call [void func(UINT id,NMHDR *hdr,LRESULT *result)]
ON_NOTIFY_EX	Similar to ON_NOTIFY, but calls a function that returns BOOL
code	Notification code
ID	Control ID
func	Function [BOOL func(NMHDR *hdr,LRESULT *result)]
ON_NOTIFY_EX_RANGE	Similar to ON_NOTIFY_EX, but for a range of IDs
code	Notification code
ID	Control ID
IDLast	Last ID
func	Function [BOOL func(UINT id,NMHDR *hdr,LRESULT *result)]
ON_CONTROL	Handles WM_COMMAND messages that are control notifications (for example, EN_ and BN_ messages)
code	Notification code
ID	Control ID
func	Function [void func(void)]
ON_CONTROL_RANGE	Similar to ON_CONTROL, but handles a range of IDs
code	Notification code
ID	Control ID
IDLast	Last control ID

func	Function to call [void func(UINT id)]
ON_MESSAGE	Handles any arbitrary message (including user-defined messages) without argument parsing
msg	Message to handle
func	Function to call [LRESULT func(WPARAM wParam, LPARAM lParam)]
ON_REGISTERED_MESSAGE	Processes registered messages (created with RegisterWindowMessage)
msgv	Variable that contains registered message ID
func	Function to call [LRESULT func(WPARAM wParam, LPARAM lParam)]

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Message Routing

You saw earlier that the main frame window owns the menu bar. Does this mean that all menu handling code resides in the frame window? Not at all. MFC automatically routes **WM_COMMAND** (and certain other messages) to other parts of your program. If you don't handle the message, MFC keeps searching for a handler. Finally, if no one else will handle the message, the main frame gets a chance. You can see the exact search order in Tables 1.3 and 1.4.

Table 1.3 Search order for MDI programs.

Search order	Object
1	Active View (if any)
2	Active Document (if any)
3	Document Template that created Document (if any)
4	Active Child Frame Window (if any)
5	Application Object
6	Main Frame Window

Table 1.4 Search order for SDI programs.

Search order	Object
1	Active View

2	Active Document
3	Document Template
4	Main Frame Window
5	Application Object

Tip: Where To Put Handlers?

Although Tables 1.3 and 1.4 seem confusing, here is a simple rule:
Command messages go where it is easiest for them to get the data they need.

This is one of the most powerful features in MFC. Imagine you have written a multipurpose program that contains a word processor and a spreadsheet. At any time, the user could have any number of windows open of both types. There might be times when there are no active windows at all (just an empty main MDI frame window).

Suppose you want to implement a “view status” command. If the current window is a spreadsheet, you want to display the number of nonblank cells, rows, and columns. If a word processing document is active, the command should display the number of words and lines in the document. If there is no active document, the command shows the free disk space available on the current disk.

Suppose MFC didn’t do command routing. Then you’d put the command handler for the view status command in the main frame. You’d have to make calls to learn the active window, determine its type, and do the correct work. Also, if you tried to move the spreadsheet code to another project, you’d need to separate out the right code from the main frame. Not a pretty solution.

With command routing, the answer is trivial. You’d simply write three distinct handlers. You’d write one handler in the word processing document class, another in the spreadsheet’s document class, and finally, a handler for the application class. The active document will get the first chance at the message. If no document is active, the application object will eventually get a chance to handle the message. None of the handlers need to know about the others. This makes for a clean, easy to maintain answer.

You might wonder how a document or application object can handle messages. After all, these classes are not windows and they don’t derive from **CWnd**. MFC makes special provisions for this. Any class that derives from **CCommandTarget** can have a message map. **CWnd**, **CWinApp**, **CDocument**, and other classes derive from **CCommandTarget**. Of course, the classes that don’t derive from **CWnd** can only handle command (and related) messages that a **CWnd**-derived class routes to them.

Document Templates

The final piece to the puzzle is how to put all of these things together. When you open a file, for example, you have to select a document class, create an instance of it, and create a corresponding view and frame. You also need to tie all of these together somehow so they work together. That’s what a document template does: *It defines the relationship between documents, views, and*

frames.

Most people don't get involved with document templates because App Wizard creates your first one for you. However, if you want to attach multiple views to a document or add a second document type, you'll really need to learn about document templates.

You'll usually create document templates during **CWinApp::InitInstance** and add them to the application object's list of document templates (using **AddDocTemplate**). If there is more than one entry in the list, MFC does some special things. First, when MFC tries to make a new file (for example, because the user clicked the File|New command), the application object will display a list of document templates. Also, if you open a file, MFC searches the document templates in the list to find one that matches the file extension.

However, sometimes you want to use the logic embodied in the document template to create a document, view, and frame under program control. In this case, you don't need to add the document template to the application's list.

The Details

So much for theory. Let's look at the details for each major class. Again, focus on how these objects work even though App Wizard and Class Wizard hide the details from you. Understanding these details is key when you want to do something the tools won't help you do.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

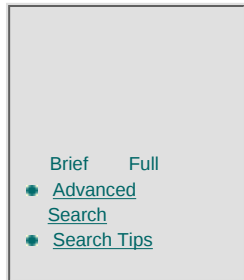
[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE



BROWSE BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

CWinApp

Every MFC program has exactly one object derived from **CWinApp**. App Wizard creates it as a global variable in your project's main CPP file. Although this variable is global (it is named **theApp**), you can't readily access it from other source files. App Wizard doesn't place a declaration for **theApp** in any of the header files.

The official way to obtain a pointer to the application object is to call the global function **AfxGetApp**. However, this is not ideal because it returns a pointer to a **CWinApp** object. If you want to access members of your custom application object, you'll have to cast the return value to the appropriate type.

Tip: Retrieving The Application Object

If you need to access members in your custom application class, you have several choices. One possibility is to add a line declaring **theApp** into a header file:

```
extern CCustomApp theApp;
```

Another idea is to define a macro named **APP**:

```
#define APP (CCustomApp *)AfxGetApp
```

The application object has several important members (refer back to Table 1.1). One of these members (**m_lpCmdLine**) contains the command line passed to your program. However, it is often more useful to use MFC's built-in command line parsing system.

App Wizard writes code to parse your program's command line. This code is often difficult to understand and modify. Here's the relevant code:

```
// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);

// Dispatch commands specified on the command line
if (!ProcessShellCommand(cmdInfo))
    return FALSE;
```

The key to understanding this snippet of code is the **CCommandLineInfo** class (see Table 1.5).

ParseCommandLine scans each argument on the command line and decides if it is a file name or option

switch (starting with / or -). Depending on what it finds, it fills in the fields of the **CCommandLineInfo** class accordingly. However, it still doesn't take any action. That comes when you pass the class to the **ProcessShellCommand** function. Exactly what this function does depends on the **m_nShellCommand** member of the **CCommandLineInfo** class (see Table 1.6).

Table 1.5 Key members of CCommandLineInfo.

Member	Description
ParseParam	Override to provide custom parameter parsing
m_bShowSplash	Indicates if a splash screen should be shown
m_bRunEmbedded	Indicates the command-line /Embedding option was found
m_bRunAutomated	Indicates the command-line /Automation option was found
m_nShellCommand	Indicates the shell command to be processed
m_strFileName	Indicates the file name to be opened or printed; empty if the shell command is New or DDE
m_strPrinterName	Indicates the printer name if the shell command is Print To; otherwise empty
m_strDriverName	Indicates the driver name if the shell command is Print To; otherwise empty
m_strPortName	Indicates the port name if the shell command is Print To; otherwise empty

Table 1.6 Possible values for m_nShellCommand.

Value
FileNew
FileOpen
FilePrint
FilePrintTo
FileDDE
FileNothing

Of course, you can neglect to call these functions if you don't want any command line processing. You can also modify **m_nShellCommand** to achieve the desired result.

For example, if you don't want a new document when the program starts, you can insert this line between the calls to **ParseCommandLine** and **ProcessShellCommand**:

```
if (cmdInfo.m_nShellCommand==FileNew) cmdInfo.m_nShellCommand=FileNothing;
```

This allows your code to still process file and option arguments.

You can also derive your own class from **CCommandLineInfo** and override the **ParseParam** function to handle custom command line switches and arguments. **ParseCommandLine** calls **ParseParam** for each command line argument it finds. You can process these arguments anyway you like.

Where is the traditional **WinMain** function? It is embedded deep in the MFC source code. However, you still have full control over everything. The key is that the **CWinApp** constructor stores a pointer to your **CWinApp**-derived object in a global variable. It isn't a problem to use a global variable in this case as there is only one application object.

Because the application object is also global, the constructor runs before any other code. By the time MFC's internal **WinMain** function starts, the global variable is already set. The **WinMain** function calls the **InitApplication** and **InitInstance** functions in the application object. These functions are virtual, so you can override them and your version will execute. If you don't override them, you'll use the default versions built into **CWinApp**. In old Windows 3.x programs, there was a difference between **InitApplication** and **InitInstance**. However, under Windows NT and 95, **WinMain** always calls both of these functions.

Once the initialization is complete, **WinMain** calls the application's **Run** method. Usually, you won't override **Run** because it implements the default message loop. The default loop is usually perfectly adequate.

However, in true MFC fashion, you could override **Run** if you wanted to do so. The default **Run** also provides several overrideable functions to allow you to customize its processing (for example, **OnIdle** and **PreTranslateMessage**).

When the event loop terminates, the internal **WinMain** calls **TerminateInstance**. You can use this function to provide any final processing you might require. Although **WinMain** is buried inside MFC, you can still customize your processing as much as you desire. For most cases, you can simply use the default with no effort.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

CView

The view is the class that the user is most likely to identify with your application. In addition to its own functions and member variables, you can also use the members of **CWnd** (the base class).

One of the most important functions you'll usually override in **CView** is the **OnDraw** function. This is where you do your drawing. This is not the same as handling the **WM_PAINT** message (using **OnPaint**). Windows sends a **WM_PAINT** message whenever your window requires redrawing. However, **CView** handles this message internally, and eventually calls **OnDraw**.

You can find the entire process outlined in Table 1.7. Why use **OnDraw** instead of **OnPaint**? **OnDraw** also handles output for printing and print preview. If you handle **OnDraw** correctly, you get printing and print preview for free (see Chapter 3 for more about printing and print preview).

Table 1.7 The drawing process.

MFC calls...	Why?	Override when...
CView::OnUpdate	You called CDocument::UpdateAllViews	You want to handle a hint on optimizing the call to InvalidateRect

CWnd::OnPaint	A WM_PAINT message occurred	You want to directly handle WM_PAINT
CView::OnPrepareDC	To set up the DC	You want to allocate GDI resources or meddle with the device context
CView::OnDraw	To update the screen	You want to draw (nearly always)

The entire drawing process begins when the framework calls **OnUpdate**. This usually occurs in response to the document's **UpdateAllViews** call. The default **OnUpdate** simply invalidates the entire client area of the view. This causes Windows to generate a **WM_PAINT** message with the entire client area invalidated. Often, you'd like to optimize this to only redraw a specific portion of the client area. That's what the hints that you pass to **OnUpdate** do.

To use hints, you must override **OnUpdate**. There are two 32-bit parameters passed to **OnUpdate**. Technically, the hints are an **LPARAM** and a **CObject ***, but in practice, you always have to cast them to whatever you want anyway. Your job is to convert the hints into a rectangle to pass to **InvalidateRectangle**. What you pass as hints is strictly up to you, with some limitations.

What should you pass as a hint? Something that allows you to figure out what part of your view requires redrawing. Some views (like **CScrollView**) call **OnUpdate** internally. Then the hints will be zero. That means your hints must never legitimately be zero. It also means your **OnUpdate** routine must correctly handle the case where both hints are zero. You can either call the base class **OnUpdate** or simply call **InvalidateRect** with a **NULL** rectangle. One thing you don't want to do is handle a hint, call **InvalidateRect**, and then call the base class **OnUpdate**, too. That will simply invalidate the entire window regardless of the hints.

Suppose you are writing a spreadsheet program (again). To update the grid view of the spreadsheet, you might pass a cell number as a hint (as long as it is never zero). You don't want to pass an absolute pixel location. Why not? Because you might have multiple views and some of them could be scrolled to different positions. A cell might be at one position in one view, another position in a second view, and scrolled completely out of sight in a third view. Also, what if you have a pie chart view open? Certainly, the location of the cell in the pie chart won't be the same as in the grid view.

You can selectively ignore **OnUpdate** hints if you choose. For example, perhaps you will handle the hints in the spreadsheet grid view but ignore them in the pie chart view. That means that your pie chart will flash each time a cell changes, but you might decide that is acceptable if it is difficult to calculate the actual rectangle to use.

In a similar vein, you are always free to be sloppy with your update rectangle as long as it is always the minimum required size *or larger*. For example, you might decide it is easier to determine which quadrant of the pie chart requires redrawing and just invalidate it. That makes the drawing less efficient, but it might still be better than redrawing the entire chart. Another possibility is to decide if the cell even appears in the pie chart. If it doesn't, you can forgo the **InvalidateRectangle** call.

There are many variations of **CView** for specific purposes (see Table 1.8). Using these special views can greatly simplify your programs. Of course, each of these classes derives from **CView**, so anything you can do with a **CView**, you can do with these special view classes.

Table 1.8 CView-derived classes.

Class	Purpose
CEditView	Simple text editor based on an ordinary Windows edit control
CListView	View that manages a list of items
CTreeView	Hierarchical list view
CRichEditView	Sophisticated editor view capable of multiple fonts, OLE, and RTF
CScrollView	Scrolling view
CFormView	Form-based view that uses a dialog template
CRecordView	View that connects to an ODBC database
CDaoRecordView	View that connects to a DAO database

App Wizard creates one document and one view for your program. App Wizard knows these two classes go together—because of this knowledge, it makes special changes to the view class. MFC associates each view with a document. To find out which document a view belongs to, you call the **GetDocument** member function. Usually, this function returns a **CDocument ***. However, App Wizard overrides **GetDocument** and casts the return value to the correct document type.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

What this means is that if you have customized members in your document class, you can easily access them from the view. But this only works for the document and view that you define using App Wizard. If you create new classes either using Class Wizard or from scratch, the **GetDocument** function will return a **CDocument ***. Class Wizard has no idea what document you intend to use with the view. This causes confusion when you can't access your custom members in the document class. The answer? You can cast the return from **GetDocument** to the correct type as needed. A better answer is to override **GetDocument** just like App Wizard does. Of course, that assumes you won't use the same view with multiple document types.

CDocument

Don't forget: The document object is the place to store an abstract representation of your program's data. You'll define most of the interesting document members based on the needs of your application. Only a few of the built-in functions are useful in most cases (see Table 1.9).

Table 1.9 Key document members.

Member	Description
AddView	Add a view to the view list
RemoveView	Remove a view from the view list
GetDocTemplate	Get document template that created this document
GetFirstViewPosition	Get POSITION that precedes the first view in the attached view list

GetNextView	Get next view from list of attached views
GetPathName	Get path name
GetTitle	Get title
SetPathName	Set path name
SetTitle	Set title
IsModified	Tests modified flag
SetModifiedFlag	Sets modified flag
UpdateAllViews	Calls OnUpdate in all attached views
OnNewDocument	Create new document
OnOpenDocument	Called to open document (usually use Serialize)
OnSaveDocument	Called to save document (usually use Serialize)
Serialize	Used to load or store document using an archive
ReportSaveLoadException	Override to catch exceptions during serialization
OnFileSendMail	Handles MAPI File Send command
OnUpdateFileSendMail	Handles update command UI message for MAPI File Send command

The primary built-in functionality for **CDocument** supports the following:

- Saving and loading documents (serialization; see Chapter 2)
- Maintaining a list of views attached to the document
- Storing the path and title for the document
- Maintaining a modification flag to determine when the document is changed
- Sending the document via electronic mail

Most programs will only use a few of these features. The document loads and saves itself (see Chapter 2). When you modify the document, it is your job to call **SetModifiedFlag** to mark the document as changed. Then, if the user closes the last view attached to the document, the internal **CDocument** code examines the modified flag. If it is set, the document prompts the user to save the file. About the only thing you have to do to get this to work is call **SetModifiedFlag**. The rest is automatic.

The document maintains a list of all the views attached to it. If you need to, you can walk the list by calling **GetFirstViewPosition** and **GetNextView**. The most common reason you'll want to find all of the views is to update them (by calling **OnUpdate**) when the document changes. However, **CDocument** provides **UpdateAllViews** to cause the document to call **OnUpdate** for the views. The call takes a pointer to a view to exclude. This is important because sometimes when a view modifies the document, it invalidates the correct portion for itself. By passing its **this** pointer to **UpdateAllViews**, the view can exclude itself from the update process. If you want all views updated, simply pass **NULL** as the parameter. **UpdateAllViews** also takes optional hints. The document passes these hints directly to the view's **OnUpdate** routine.

The really interesting part about using documents is serializing the data.

Usually, this causes the document to save and load to a file. However, in Chapter 2, you'll see that it doesn't have to be a file. You might serialize data to a database record, part of another file, or a network connection.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

CFrameWnd And Related Classes

Frames are the unsung heroes of MFC applications. The users often don't distinguish between the frame and the view, but as a programmer, you must understand the differences. **CFrameWnd** (see Table 1.10) is the base class for **CMDIFrameWnd** and **CMDIChildWnd**, which are only useful for MDI programs.

Table 1.10 Key CFrameWnd members.

Member	Purpose
m_bAutoMenuEnable	When TRUE (the default), the frame disables menu items that have no handler
rectDefault	Static member used to set default size rectangle during Create
Create	Creates the actual window corresponding to this frame
LoadFrame	Creates the actual window for this frame using information in the application's resources
LoadAccelTable	Load the frame's accelerator table
LoadBarState	Load information about the bar status (docked, visible, horizontal, etc.) from the program's INI file or the registry
SaveBarState	Save bar status for a later restoration with LoadBarState
ShowControlBar	Show or hide an attached bar

SetDockState	Set bar status (see LoadBarState); useful if you want to serialize the bar status with a document
GetDockState	Get bar status (see LoadBarState); useful if you want to serialize the bar status
EnableDocking	Enable docking of bars on any or all edges
DockControlBar	Dock a control bar
FloatControlBar	Float a control bar
GetControlBar	Get control bar
RecalcLayout	Recalculate layout based on current view and control bars
ActivateFrame	Make frame active
InitialUpdateFrame	Call OnInitialUpdate in all views belonging to this frame
GetActiveFrame	Get active frame (useful in MDI programs)
SetActiveView	Select a view in this frame to be active
GetActiveView	Learn which view in this frame is active
CreateView	Create a new view or window using a CreateContext structure
GetActiveDocument	Get active document object
GetMessageString	Get message associated with a command ID
SetMessageText	Display a message in the status bar
GetMessageBar	Get a pointer to the status bar

Much of the important functionality in **CFrameWnd** happens behind the scenes. Remember that the main frame window owns the menu, tool bar, and status bar. Therefore, the main frame is responsible for routing the messages from these items. Of course, the actual class that manages all message maps is **CCmdTarget** (an eventual base class for **CFrameWnd**). If you want to change the message routing, you can override **OnCmdMsg** and change the routing for command messages. However, you should rarely need to do this in practice.

As a byproduct of searching the message map for command handlers, the frame window knows if you have a menu item that doesn't have a corresponding handler. When it detects this, it normally grays the menu item and disables it. If you don't want this behavior, you'll need to set the frame's **m_bAutoMenuEnable** member to **FALSE**.

You should also know that **WM_COMMAND** messages are not the only ones subject to command message routine. **WM_COMMAND** usually occurs when the user manipulates a user interface element (such as a button or a menu item). However, MFC also sends a custom message, **WM_IDLEUPDATECMDUI**, for each command ID. This message routes exactly like a **WM_COMMAND** message, but the macro in the message map is **ON_UPDATE_COMMAND_UI**. This maps the message to a handler that receives a **CCmdUI** object (see Table 1.11). This object is an abstract representation of the user interface element in question. Using this object, you

can enable or disable the element. You can also set other attributes. You only need to provide this handler if you want to explicitly control the item.

Table 1.11 CCmdUI members.

Member	Purpose
m_nID	ID of item (if applicable)
m_nIndex	Index of item (if applicable)
m_pMenu	Handle to main menu (if applicable)
m_pSubMenu	Handle to sub menu (if applicable)
m_pOther	Window handle to other type of item
Enable	Enable (or disable) the item
SetCheck	Set (or clear) the item's check
SetRadio	Check the item and uncheck the others in its group
SetText	Set the item's text
ContinueRouting	Defer this to a handler further down the routing chain

Why does MFC use the **CCmdUI** object? Why not just pass a pointer to the menu item, button, or other user interface item in question? Suppose you have a menu item that also has a matching tool bar button. Then, you use the same command update handler for both items. Your code need not worry about the differences between the two. Presumably, if the next version of Windows offers a virtual reality skeet shoot user interface item, **CCmdUI** will control it, too.

One thing that makes frames different from other windows is that you can load the frame's menu, icon, cursor, and title directly from your resources. Simply give each item the same ID (for example, **IDR_MAINFRAME**). Then call **LoadFrame** passing **IDR_MAINFRAME** as an argument. MFC will create the frame and automatically load the resources and associate them with the frame window. SDI frames use the first portion of the resource string to set their title. The string has more than one purpose (the document template uses it also). That's why you have to use \n characters to divide it into multiple fields. If you create a new frame through a document template, it uses **LoadFrame**. It also uses other fields in the document string.

You can manually create views inside a frame using **CreateView**. However, that is rarely worth the trouble, as **CreateView** really doesn't do much of the work for you. Instead, you should allow document template objects to do the work that they do well: creating or associating documents, views, and frames.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

CDocTemplate

The item that ties documents, views, and frames together is the **CDocTemplate**. SDI programs actually use the **CSingleDocTemplate** class and MDI programs use **CMultiDocTemplate**; both classes derive from **CDocTemplate**.

There aren't many interesting members in **CDocTemplate** (see Table 1.12). Most of the functions that **CDocTemplate** performs take place because of user actions (File|New and File|Open menus, for example).

Table 1.12 Key CDocTemplate members.

Member	Purpose
GetFirstDocPosition	Get POSITION that precedes the first document in the list
GetNextDoc	Gets the next document from the list
GetDocString	Returns a field from the document's resource string
CreateNewDocument	Create a new document only
CreateNewFrame	Create a new frame window with an associated view and document
InitialUpdateFrame	Causes an OnInitialUpdate in a frame's views
SaveAllModified	Save all associated documents
CloseAllDocuments	Close all associated documents
OpenDocumentFile	Opens a file or creates a new empty document
SetDefaultTitle	Allows you to change the default title

The document template uses the document string in the resources (the same string that the frame window uses). The string is actually seven substrings separated by \n characters (see Table 1.13). You provide the ID of the string when you create the document template. You also provide the class names of the document, view, and frame classes you want to use for this type of document.

Table 1.13 Document string fields.

Field	Name	Purpose
0	CDocTemplate::windowTitle	Title used for SDI programs only
1	CDocTemplate::docName	Name used for new documents (MFC appends a serial number to this name)
2	CDocTemplate::fileNewName	Name MFC displays in a dialog box in response to File New if there is more than one template
3	CDocTemplate::filterName	Name of file extension for this document type (example: Black Book Files)
4	CDocTemplate::filterExt	File extension for this document type (example: .BLK)
5	CDocTemplate::regFileTypeId	Internal registry name for this document type
6	CDocTemplate::regFileName	User-visible name for this document type placed in registry

Once again, this is an unfortunate choice of terminology. “Document,” in the context of document template, means a document in the way users think of a document. It does not mean your document object (the program’s model). For example, you might have a document template for a spreadsheet, another for a word processing document, and yet another for a communications terminal. Don’t confuse this with your document object. The document template creates a relationship between a document, view, and frame. That relationship defines what users think of as a document.

App Wizard sets things up so that your program creates an initial document template on the heap during your application’s **InitInstance** function. It then uses **AddDocTemplate** to place this template in the application’s list of document templates. For an ordinary program, there is only this one template in the list. Therefore, it is easy for MFC to select that template for every operation.

However, it is possible for you to define more templates. If you add your new template to the application’s list, things aren’t so clear anymore. If you try to open a new file, MFC attempts to match the file’s extension with the extension specified in the resource string. If the user asks to create a new file, MFC

creates a dialog box that contains a list of the document template names (this is the second field in Table 1.13). The user then selects the document type they desire.

Tip: Secondary Document Templates

There is no reason you have to add document templates to the application's list unless you want the application to consider the template during File|New and File|Open.

There are a few subtleties to this. First, you don't have to add a document template to the application's list. Why would you want to do that? Suppose you've written a checker game. You'd probably make the main document template create a checkerboard. You might also want a secondary view that shows a list of all the moves in the game. That's not something you want to show the user in response to the File|New command. Instead, you can make an independent document template and manipulate it.

Another thing that may not be obvious is how you can use a document template to create a new document without asking the application object to do it. For example, suppose you don't like the default behavior of the File|New command. You might prefer having a cascading New command that has a submenu for each different type of document. To achieve this, you can call the correct document template's **OpenDocumentFile** with a **NULL** file name. As odd as that sounds, this will cause the template to create a new document. You'll find an example in the practical section of this chapter.

One non-traditional use for a document template is to create new views attached to the same document. Just call the document template's **CreateNewFrame** followed by the **InitialUpdateFrame** function. You'll find an example of this in the practical section.

Tip: Name Alert

Be sure you call the document template's **InitialUpdateFrame**. **CFrameWnd** also has an **InitialUpdateFrame** function, but that's not the one you want to use in this case.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Navigating Objects At Runtime

How often have you started a chore around the house, grabbed a screwdriver, and wound up unable to finish it because you had the wrong screwdriver for the job? The same thing can happen in MFC programs. Ideally, you should handle messages in the class where it makes most sense. For example, suppose you are writing a word processor program. Where should you handle the Edit|Paste command? It probably makes sense to handle this menu command in the document object. Think about it. You are going to take text from the clipboard and place it in the document. Then you'll call **UpdateAllViews** to redraw the view or views.

That's a simple case. But consider Edit|Copy. Where do you put the handler for that? If you put it in the document, you have a problem. You don't know what the current selection in the current view is. Okay, so put the handler in the view, right? Well, the view doesn't have the data it needs to place on the clipboard. There is no right answer for this problem. No matter what you do, you will have to handle the message in one class and access another class to do the work. This is not at all uncommon.

Luckily, MFC provides many ways to navigate from an object you know to an object you'd like to find. You can find the MFC treasure map in Table 1.14. To use the map, find the class you know in the left-hand column. Then locate the class you'd like to learn in the middle column. The right-hand column tells you what calls you need to make. If you can't find a direct path, you might have to make several conversions to get the class you want.

Table 1.14 MFC treasure map.

If you have...	And you want...	Use...
Document	View	GetFirstViewPosition and GetNextView
Document	Template	GetDocTemplate
View	Document	GetDocument
View	Frame	GetParentFrame
Frame	View	GetActiveView
Frame	Document	GetActiveDocument
MDI Main Frame	MDI Child Frame	MDIGetActive
MDI Child Frame	MDI Main Frame	GetParentFrame

Most of the time, the call required to locate another class is simple. For example, the view only needs to call **GetDocument** to locate the associated document. However, there are a few odd situations you should know about.

Finding views is frequently a problem. If the view is active, that is, it has the focus, you won't have any problems. But if you want to find a view from a document, you have a problem. One document may have many views, so there is no single call that just returns a view. You can get all the views by calling **GetFirstViewPosition** followed by calls to **GetNextView** to retrieve a pointer to each view. Of course, if you know your document only has a single view, you can safely make the **GetNextView** call once and assume that is the sole view.

Another interesting conundrum occurs when you work with MDI programs. If you call **GetActiveView** on the MDI main frame, the result is **NULL**. The same thing occurs if you call **GetActiveDocument**. This is not surprising, as it's merely a thin disguise for calling **GetActiveView** and **GetDocument** together. At first, this behavior seems puzzling, but if you think about it, it makes sense. MDI main frames don't have an active view. They have an active MDI child frame window. The child frame window has an active view. Mystery solved. You can learn the active child frame by calling **MDIGetActive**.

Another puzzle can occur when you work with dialogs (the **CDialog** class). The dialog's constructor takes an argument that allows you to specify a parent window for the dialog. Suppose you use a view object as the parent. You might reasonably expect to be able to call **GetParent** from the dialog to learn the **CView *** for the current view. That doesn't work. The dialog automatically determines if its parent window is a top-level window. If it isn't, the dialog goes to the next higher window and tries again. This continues until the dialog finds a top-level window that then becomes the dialog's parent. Of course, you could then use the treasure map to determine the view from the top-level window (presumably a frame window).

Supporting Objects

There are many supporting objects in MFC that don't directly contribute to the

architecture. Classes like **CString**, **CFile**, and many others help you write your MFC program. Of course, you usually don't have to use any of these classes if you don't want to do so. After all, you can continue using ordinary character arrays for strings or regular C files, or C++ iostreams, if you prefer. However, once you get used to using the MFC classes, you'll probably want to use them.

One of MFC's greatest strengths is that it recognizes its own limitations. MFC knows it doesn't do everything, and that sometimes you'll need to call the Windows API or some add-on DLL that it doesn't know about. That's why many classes can easily convert back and forth between their MFC and old-fashioned identities.

CString is a good example of this. The **CString** class automatically knows how to convert itself to a constant character pointer. That means you can use a **CString** anywhere you can use an ordinary C-language string that won't be modified. You can't modify the string because if you do, the **CString** code can no longer track the size of the string. As an aside, if you need a character pointer that you can modify, you'll need to call **GetBuffer** and specify how many characters to reserve.

Another example of this effortless conversion is in the **CRect**, **CPoint**, and **CSize** classes. These classes derive from the corresponding Windows structures (**RECT**, **POINT**, and **SIZE**). It is perfectly legal for a class to use a structure as a base class. The trick here is that MFC doesn't add any extra member variables or virtual functions to these classes. This ensures that the C++ instance data will be exactly like the old C structure. That means you can treat, for example, a **RECT** as a **CRect** and vice versa. Very handy. However, since these classes can't have virtual functions, they don't act like other MFC classes in certain respects. For example, you can't use MFC type identification on these classes (see the *CObject Support* section later in this chapter).

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

The CWnd Object

Of all the supporting objects, perhaps none is as important as **CWnd**. **CWnd** is the MFC class that represents all windows (including views and frame windows that derive from **CWnd**).

There are a few things you should realize about working with **CWnd**s. First, don't think of a **CWnd** as an actual window; it isn't. It's a C++ class that *contains* a window. I like to think of a **CWnd** as a bottle. When you create the bottle, it is empty. You can fill the bottle with a new window by calling certain functions (for example, **Create**, **DoModal**, **LoadFrame**). You can also attach an existing window to a **CWnd** in a variety of ways.

Suppose you have a window handle and you'd like to treat it as a **CWnd**. If you just want to manipulate it briefly, you can use the static **CWnd::FromHandle** to obtain a **CWnd** pointer. If the window already has a corresponding **CWnd** object, **FromHandle** returns a pointer to it. If the window isn't an MFC window, the function creates a temporary **CWnd** object that refers to the window. Be careful, though—this **CWnd** doesn't really process messages. Also, MFC will automatically delete it sometime after processing the current message. You can't store the pointer for later use.

If you want to make a more permanent attachment to a **CWnd**, you can use **SubclassWindow** (or **SubclassDlgItem**). This attaches the window to the **CWnd**, activates the message map, and allows you to work with the window just as if MFC created it.

It is easy to transform a window handle into a **CWnd**. It is even easier to go in the other direction. You can always read the window handle back out of a **CWnd** by accessing the **m_hWnd** member variable. This is another example of how MFC allows you to interoperate with the Windows API.

Creating a **CWnd** is not the same as creating a window. This is known as two-phase construction. You construct the C++ object and then call **Create** (for example) to create a window attached to the **CWnd**. This allows maximum flexibility for a **CWnd**. You can create new windows or attach to existing ones. Also, the error-prone window creation process is in a function that can return an error. C++ constructors can't return error information very easily.

CWnd is not the only class that uses two-phase construction. Many other classes do also. For

example, **CPen**, **CBrush**, and other GDI objects work the same way. Many of the simpler objects also allow you to create them with a special constructor (single-phase construction). These classes also allow you to make temporary associations with the real GDI object. For example, you can use **CPen::FromHandle** to convert a Windows **HPEN** to a **CPen**. To convert a GDI object to a handle, access the **m_hObject** member (part of the base class, **CGDIObject**).

CObject Support

Most important objects in MFC derive from **CObject**. This object provides several important features for MFC programs:

- You can use a **CObject *** to represent most objects (via C++ polymorphism).
- **CObject** provides support for identifying and creating types at runtime.
- Objects derived from **CObject** can use MFC's facilities for storing and loading objects to persistent storage (like files).

The polymorphism support was particularly important before Visual C++ had template support. Because everything interesting derives from **CObject**, MFC could supply generic data structures (like lists and arrays) that hold **CObject** pointers. You could put any **CObject**-derived class pointers in these data structures. Today, MFC provides similar data structures that use templates (see the *About Collections* section later in this chapter). However, the old-style data structures are still present, mainly for compatibility.

Although there is now a standard way for C++ programs to use runtime type identification, MFC predates such luxuries. Therefore, it implements its own form of runtime type identification. Each class has a special value that uniquely identifies the class. You can convert a class name to this special value by using the **RUNTIME_CLASS** macro. You can also find the unique number from a class instance by calling the **GetRuntimeClass** member function. To build complete support into an object, you must add the **DECLARE_DYNAMIC** macro to the class definition (usually in a header file). You also have to add the **IMPLEMENT_DYNAMIC** macro to the CPP file (at global scope).

Suppose you have a class named **Base1** that derives from **CObject** and uses the dynamic macros. Further imagine that you derive another class, **Class2**, from **Base1**. This class also uses the dynamic macros. Then you might have the following code:

```
Base1 b1;
Class2 c2;
CObject *o1, *o2;
o1=&b1;
o2=&c2;
if (o1->IsKindOf(RUNTIME_CLASS(Base1))) /* true */ ;
if (o1->IsKindOf(RUNTIME_CLASS(Class2))) /* false */ ;
if (o2->IsKindOf(RUNTIME_CLASS(Class2))) /* true */;
// watch this...
if (o2->IsKindOf(RUNTIME_CLASS(Base1))) /* true! */;
// o2 is actually a Base1 (and a class 2)
// if you want to tell if it is exactly a Base1 object, try this...
if (o2->GetRuntimeClass()==RUNTIME_CLASS(Base1)) /* false */;
```

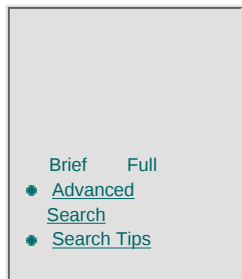
[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE



BROWSE BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97



Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

IsKindOf's behavior makes sense if you think about it. Consider a **CView** object. If you ask the object if it is a kind of **CWnd**, the answer is yes. Even though it is a **CView**, that is just a specific type of **CWnd**. You can still make any legal **CWnd** call to a **CView**.

CObject also contributes to making persistent objects. That is, a program can write a **CObject**-derived class instance out to persistent storage, like a disk file or database record, and later recreate the class. You'll learn more about this facility (called serialization) in Chapter 2.

Another level of runtime support that MFC offers is dynamic creations support. If you use the **DECLARE_DYNCREATE** and **IMPLEMENT_DYNCREATE** macros instead of the **DYNAMIC** macros, you'll still get all the same benefits. However, you'll also enable MFC to create instances of the class by passing its runtime type identifier to **CreateObject**.

MFC makes extensive use of this facility in document templates and when creating duplicate views (for example, in the Window|New Window menu command). Classes that are dynamically createable must have a default constructor (yet another reason for two-phase construction). Many classes make this constructor protected so that you can only use dynamic creation to instantiate the class.

About Collections

MFC provides several interesting collection classes that allow you to create arrays, lists, and associate arrays (called maps). By using templates, you can create these data structures, which can contain any data type. In addition, these collection classes allow you to write helper functions that customize the collection's behavior.

Using Templates

Templates are a relatively new addition to the C++ language. They allow you to write functions and classes that are easy to reuse. They do this by transforming code at compile time so that you can ignore data types. Think about a function that is supposed to return the maximum of two values. Here's a version for integers:

```
int Max(int v1,int v2)
{
    return v1>v2?v1:v2;
}
```

The logic for this code only depends on the greater-than operator. There is no reason this code shouldn't work for any type that has such an operator. Of course, it won't work because the type **int** is hard coded in the function. The integers are the function's data, but the greater-than operation is the function's logic or algorithm. Templates solve this type of problem by allowing you to specify the function's algorithm independently of the

data involved. Consider this code:

```
template <class TYPE> TYPE Max(TYPE v1, TYPE v2)
{
    return v1>v2?v1:v2;
}
```

Now, when you call function **Max** with two integer arguments, the compiler will automatically create a function just like the first example. However, if you later call **Max** with two floating point arguments, the compiler will generate another function for floating point. Remember, C++ allows you to have multiple functions with the same name provided the arguments are different (function overloading). Notice that even though the syntax uses the class keyword, that doesn't imply a user-defined class. You can use user-defined classes or built-in types (like **int**, **char ***, or **float**, to name a few).

Templates are especially useful for encapsulating algorithms in classes. Suppose you wanted a class that accepted two values and always returned the largest one. Here's some template code that accomplishes that for any data type:

```
template <class TYPE> class Selector
{
private:
    TYPE v1;
    TYPE v2;
public:
    Selector(TYPE xv1, TYPE xv2) { v1=xv1; v2=xv2; };
    void Set1(TYPE v) { v1=v; };
    void Set2(TYPE v) { v2=v; };
    TYPE GetMax(void) { return v1>v2?v1:v2; };
};
```

Again, the class algorithm is separate from the data types. To create a **Selector** class for integers, you can write:

```
Selector<int> sel;
Selector<int> *selptr;
```

This can lead to some unwieldy type casts. You might prefer to simplify things with a **typedef**:

```
typedef Selector<int> IntSelector;
IntSelector sel, *selptr;
```

Notice that everywhere the parameter **TYPE** appears in the template, **int** appears in the final code. Of course, you can create as many kinds of **Selector** classes as you need (one for **float** and another for **CString**, for example). The compiler only generates functions for those you use. You don't have to write the functions inline, by the way. If you wanted to write the **GetMax** function in the conventional way, it would look like this:

```
template <class TYPE>
    TYPE Selector<TYPE>::GetMax(TYPE v1, TYPE v2)
    {
        return v1>v2?v1:v2;
    }
```

Although the function does not need to be inline, source files that use the template must have the entire function visible. Therefore, you'll almost always put the functions (inline or not) in a header file. The only exception would be where you place a template in the same CPP file that uses it and no other source file needs the same template.

The above examples were straightforward because none of the code changed for specific types. Frequently, however, you'll want to have some specialized code depending on the type. Then you can create a generic base class with virtual functions. For example, suppose you want to extend the above class so that it can print the type formatted in a way specific to each type. Further suppose that the default stream I/O formatting is not adequate. You can still use templates:

```

template <class TYPE> class SelectorBase
{
private:
TYPE v1;
TYPE v2;
public:
SelectorBase(TYPE xv1, TYPE xv2) { v1=xv1; v2=xv2; };
void Set1(TYPE v) { v1=v;};
void Set2(TYPE v) { v2=v;};
TYPE GetMax(void) { return v1>v2?v1:v2; };
// pure virtual function ñ sub class must override
virtual void PrintMax(void) =0;
};

class IntSelector : public SelectorBase<int>
{
void PrintMax(void) { printf("Largest integer=%d (0x%x)\n",GetMax()); };
};

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Although type parameters are useful, you can also specify other constant arguments to a template. Suppose you wanted a **Selector** class that could handle more than two entries. Here's some code to consider:

```
template <class T, int ct> class ArySelect
{
    T * ary;
public:
    ArySelect() { ary=new T[ct]; };
    ~ArySelect() { delete [] ary; };
    void Set(int n,T v) { ary[n]=v; };
    T GetMax(void);
};

template <class T,int ct>
    T ArySelect<T,ct>::GetMax(void)
    {
        T maxv=ary[0];
        for (int n=1;n<ct;n++)
            if (maxv<ary[n]) maxv=ary[n];
        return maxv;
    };
};
```

This code passes an integer counter to define the size of the array. It is easy to imagine a simple template that didn't do any type manipulation at all. For example:

```
template <int sz> struct dataholder
```

```

{
char *name[sz];
int score[sz];
struct dataholder<sz> *links[2*sz];
};

```

There is one other special case you'll find for template arguments. Suppose you have a class that you want to work one way for nearly every type. However, you want some special processing for character pointers and **CString** types. You could write something like this:

```

template <class T>
    class Special
    {
    // general code
    };
template <char *>
    class Special
    {
    // char * only code
    };
template <CString>
    class Special
    {
    // CString only code
    };

```

Collection Details

MFC provides three types of template-based collections:

- **CList**—A collection that stores items you usually want to access in a certain order. **CList** collections are good for implementing stacks, queues, and similar data structures. See Table 1.15 for a list of **CList** members.

Table 1.15 Key CList members.

Member	Description
GetHead	Get first element (the head of the list)
GetTail	Get last element (the tail of the list)
RemoveHead	Remove item at head
RemoveTail	Remove item at tail
AddHead	Add item to the head of the list
AddTail	Add item to the tail of the list
RemoveAll	Remove all elements
GetHeadPosition	Get POSITION of beginning (head) of list
GetTailPosition	Get POSITION of end (tail) of list
GetNext	Get next element

GetPrev	Get previous element
GetAt	Get arbitrary element; this inefficiently treats the list as an array
SetAt	Set an arbitrary element; this inefficiently treats the list as an array
RemoveAt	Removes an arbitrary element
InsertBefore	Inserts an element before a specified item
InsertAfter	Inserts an element after a specified item
Find	Finds an item and returns a POSITION
FindIndex	Returns a POSITION corresponding to a particular index
GetCount	Get count of items
IsEmpty	Tests for an empty list

- **CArray**—A one-dimensional array of data (see Table 1.16). **CArrays** can grow to contain additional data, and you may easily access data in any order.

Table 1.16 Key CArray members.

Member	Description
GetSize	Get size of array
GetUpperBound	Get maximum bound of array
SetSize	Preset the array's size
FreeExtra	Free extra memory that doesn't contain elements
RemoveAll	Remove all elements
GetAt	Get an element
SetAt	Set an element
ElementAt	Retrieve a reference to an element
GetData	Retrieve a pointer to the raw elements in the array
SetAtGrow	Set an element, expanding the array if necessary
Add	Adds an element to the end of the array, expanding it if required
Append	Appends another array to this one, expanding the destination array if required
Copy	Copies one array to another
InsertAt	Inserts an element or array at a specified location
RemoveAt	Remove a specific element
operator[]	Sets or gets an element (usually used instead of GetAt/SetAt)

- **CMap**—The **CMap** collection is like an array that can take any data type as an index. For example, you could use a **CMap** to translate a name (in a **CString**) into a **CWnd**. Table 1.17 shows the member functions available in the **CMap** class.

Table 1.17 Key CMap members.

Member	Description
Lookup	Find a value by key
SetAt	Set a value/key pair
operator[]	Treat the map as an array
RemoveKey	Remove element
RemoveAll	Remove all elements
GetStartPosition	Get POSITION that precedes the first pair
GetNextAssoc	Get next pair
GetHashTableSize	Find size of hash table
InitHashTable	Set up a hash table with a specific size (useful for optimizing)
GetCount	Get count of elements
IsEmpty	Test for empty map

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Why use the collection classes? You certainly aren't obligated to use them. In many cases, using a classic C language array is as good as using a **CArray**. Still, the collection classes do have advantages in many cases. One nice thing about the collection classes is that they can grow to accept more data. Another thing is that because they are ordinary MFC classes, you can use MFC type identification and serialization mechanisms with a collection.

You can customize how the collection classes work by writing global helper functions (see Table 1.18). These functions are only necessary if you want to alter the collection's operation. The compiler decides what helper function to use based on its argument types. For example, suppose you write a **SerializeElements** helper that takes a **CBlackBook** class argument. That function will control serialization of all collections that contain **CBlackBook** classes.

Table 1.18 Collection helpers.

Helper Function	Description
CompareElements	Compare two elements
ConstructElements	Construct elements when created
CopyElements	Move elements from one collection to another
DestructElements	Destroy elements when removed
DumpElements	Dump elements for debugging purposes
HashKey	Computes a hash value for the given key
SerializeElements	Read/write elements from/to an archive

The helper functions are nearly always optional. For example, if you serialize a collection that doesn't have a **SerializeElements** helper function, it will simply dump the bits of its contents to the archive. If your collection contains integers, that's fine. If it contains **CWnd** pointers, that's not a good idea. You'll read more about serialization in general in Chapter 2.

Obviously, not all helpers apply to each type of collection. The **HashKey** helper, for example, is only required for **CMaps**. Even then, you only need to supply the helper if you want your own hashing algorithm or if your key isn't castable to a **DWORD**.

Summary

If you are going to bend MFC to your own purposes, you need to understand *why* things work in MFC. If you rely completely on the tools, you will become a slave to them. Ideally, you should know how MFC works and why the design is the way that it is.

Many pundits complain that MFC is poorly designed, that it is not a good example of object-oriented programming, and that there are better ways to accomplish the same things. Perhaps some of their complaints are true. Pragmatically, however, MFC is the way that it is. Complaining really isn't going to help. Certainly, complaining to me doesn't do any good—I'm just another MFC user, just like you are. Somehow, I suspect complaining to Microsoft isn't any better. MFC, Windows, C++—none of these are perfect, but they are the tools we use to write our programs.

Practical Guide To The Architecture

- Handling User Messages
- Creating A New Document Type
- Creating A Private Document
- Attaching Multiple Views To A Document
- Making Separate File|New Menus
- Preventing A New Document On Startup
- Parsing Command Line Parameters
- Calculating A View Size
- Using Typedef With Templates
- A 2D CArray

Sometimes, knowing the internals to MFC can come in handy. No one wants to dig around in message maps and manually create views, but sometimes you don't have a choice. The following practical tips will help you when you have no choice but to dig into the architecture.

Handling User Messages

If you define your own messages (messages greater than or equal to

WM_USER), you can't rely on Class Wizard to handle them. You have several options. You can use the **ON_MESSAGE** macro to handle the message. You supply the message ID and the name of the function you want MFC to call when it detects the message.

The message handling function must return a **LONG** and must take a **WPARAM** and **LPARAM** as arguments—even if you don't care about the arguments or return values. Of course, you can always omit the parameter names (but not the types) if you don't plan to use them.

Technically, you can use this technique with any message, but you are better off using the built-in macros where available. The built-in macros know how to parse the message arguments and handle the correct return type. If your message ID is in a variable, as it would be if you were using a registered message, you can use **ON_REGISTERED_MESSAGE**. Here's an example using **ON_MESSAGE**:

```
#define WM_BLACKBOOK  (WM_USER+5)
BEGIN_MESSAGE_MAP
//{{AFX_MSG_MAP(CMainFrame)
    // Whatever Macros Class Wizard puts in here...
//}}AFX_MSG_MAP
ON_MESSAGE(WM_BLACKBOOK, OnBlackBook)
END_MESSAGE_MAP
```

Class Wizard won't help you place most of the special message macros (such as **ON_MESSAGE**; see Table 1.19), so you have to put the entries in yourself. It is good practice to put your entries outside the Class Wizard comments (the `//{{` and `//}}` characters) so you don't confuse the Wizard.

Table 1.19 Selected special message map macros.

Macro	Meaning
ON_MESSAGE	Handles user-defined messages
ON_REGISTERED_MESSAGE	Handles registered messages
ON_THREAD_MESSAGE	Used to handle user-defined messages in a thread (CWinThread)
ON_REGISTERED_THREAD_MESSAGE	Used to handle registered messages in a thread (CWinThread)
ON_COMMAND_RANGE	Handles a range of WM_COMMAND messages
ON_CONTROL_RANGE	Handles a range of WM_COMMAND notifications
ON_UPDATE_COMMAND_UI_RANGE	Like ON_UPDATE_COMMAND_UI but for a range

ON_NOTIFY_RANGE

Handles a range of
WM_NOTIFY notifications

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Another way to handle custom messages is to define your own message map macro. Although this isn't officially sanctioned by Microsoft, you can figure out most of the details by examining AFXMSG_.H. The idea is to make entries in the message map (which is just an array of structures). The structure in question is **AFX_MSGMAP_ENTRY** (see Table 1.20).

Table 1.20 The AFX_MSGMAP_ENTRY structure.

Member	Type	Definition
nMessage	UINT	Message to process
nCode	UINT	Control or WM_NOTIFY code
nID	UINT	Control ID or 0 for regular messages
nSig	UINT	Code specifying function signature (see Table 1.21)
pfn	AFX_PMSG	Pointer to handler function

The only field that deserves special mention is **nSig**. This is a special code (taken from the **AfxSig** enumeration) that tells MFC what the signature of **pfn** is. If your function signature doesn't appear in the enumeration (see Table 1.21), then you can't use it directly to handle a message.

Table 1.21 Message map function signatures.

Code	Signature
AfxSig_end	N/A (marks end of message map)

AfxSig_bD	BOOL (CDC*)
AfxSig_bb	BOOL (BOOL)
AfxSig_bWww	BOOL (CWnd*, UINT, UINT)
AfxSig_hDWw	HBRUSH (CDC*, CWnd*, UINT)
AfxSig_hDw	HBRUSH (CDC*, UINT)
AfxSig_iwWw	int (UINT, CWnd*, UINT)
AfxSig_iww	int (UINT, UINT)
AfxSig_iWww	int (CWnd*, UINT, UINT)
AfxSig_is	int (LPTSTR)
AfxSig_lwl	LRESULT (WPARAM, LPARAM)
AfxSig_lwwM	LRESULT (UINT, UINT, CMenu*)
AfxSig_vv	void (void)
AfxSig_vw	void (UINT)
AfxSig_vww	void (UINT, UINT)
AfxSig_vvii	void (int, int) (wParam is ignored)
AfxSig_vwww	void (UINT, UINT, UINT)
AfxSig_vwii	void (UINT, int, int)
AfxSig_vwl	void (UINT, LPARAM)
AfxSig_vbWW	void (BOOL, CWnd*, CWnd*)
AfxSig_vD	void (CDC*)
AfxSig_vM	void (CMenu*)
AfxSig_	void (CMenu*, UINT, BOOL)
AfxSig_vW	void (CWnd*)
AfxSig_vWww	void (CWnd*, UINT, UINT)
AfxSig_vWp	void (CWnd*, CPoint)
AfxSig_vWh	void (CWnd*, HANDLE)
AfxSig_vwW	void (UINT, CWnd*)
AfxSig_vwWb	void (UINT, CWnd*, BOOL)
AfxSig_vwwW	void (UINT, UINT, CWnd*)
AfxSig_vwwx	void (UINT, UINT)
AfxSig_vs	void (LPTSTR)
AfxSig_vOWNER	void (int, LPTSTR) (force return TRUE)
AfxSig_iis	int (int, LPTSTR)
AfxSig_wp	UINT (CPoint)
AfxSig_wv	UINT (void)
AfxSig_vPOS	void (WINDOWPOS*)
AfxSig_vCALC	void (BOOL, NCCALCSIZE_PARAMS*)
AfxSig_vNMHDRpl	void (NMHDR*, LRESULT*)
AfxSig_bNMHDRpl	BOOL (NMHDR*, LRESULT*)
AfxSig_vwNMHDRpl	void (UINT, NMHDR*, LRESULT*)
AfxSig_bwNMHDRpl	BOOL (UINT, NMHDR*, LRESULT*)
AfxSig_bHELPINFO	BOOL (HELPINFO*)
AfxSig_vwSIZING	void (UINT, LPRECT) (force return to TRUE)

AfxSig_cmdui	void (CCmdUI*)
AfxSig_cmduiw	void (CCmdUI*, UINT)
AfxSig_vpv	void (void*)
AfxSig_bpv	BOOL (void*)
AfxSig_vvwh	void (UINT, UINT, HANDLE)
AfxSig_vwp	void (UINT, CPoint)
AfxSig_bw	BOOL (UINT)
AfxSig_bh	BOOL (HANDLE)
AfxSig_iw	int (UINT)
AfxSig_ww	UINT (UINT)
AfxSig_bv	BOOL (void)
AfxSig_hv	HANDLE (void)
AfxSig_vb	void (BOOL)
AfxSig_vbh	void (BOOL, HANDLE)
AfxSig_vbw	void (BOOL, UINT)
AfxSig_vhh	void (HANDLE, HANDLE)
AfxSig_vh	void (HANDLE)
AfxSig_viSS	void (int, STYLESTRUCT*)
AfxSig_bwl	LRESULT (WPARAM, LPARAM)
AfxSig_vwMOVING	void (UINT, LPRECT) (force return TRUE)
AfxSig_vW2	void (CWnd*) (CWnd* comes from lParam)
AfxSig_bWCDS	BOOL (CWnd*, COPYDATASTRUCT*)
AfxSig_bwsp	BOOL (UINT, short, CPoint)
AfxSig_vws	void (UINT, LPCTSTR)

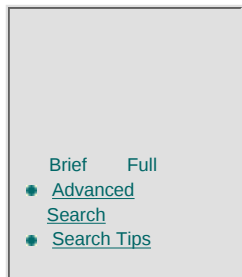
The **BEGIN_MESSAGE_MAP** macro begins the declaration of the message map array, so a custom macro must fill in the structure as a static initializer. Look at this excerpt from AFXMSG_.H:

```
#define ON_WM_CREATE()    { WM_CREATE, 0, 0, 0, AfxSig_is, \
    (AFX_PMSG)(AFX_PMSGW)(int (AFX_MSG_CALL CWnd::*)\
    (LPCREATESTRUCT) )&OnCreate },
```

Here, the message is **WM_CREATE** and the signature is **AfxSig_is** (although the pointer is a **CREATESTRUCT**, the signature treats it as a **LPTSTR**). The final argument specifies the member function to call (**OnCreate**, of course).

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97



Search this book:



Creating A New Document Type

When you use App Wizard, it creates a default document class and a default view class for your program. What happens if you want to have an extra document? App Wizard isn't very helpful. Here are the steps:

1. Use Class Wizard to derive a new class from **CDocument**.
2. Use Class Wizard to derive a new class from **CView** (or a related class).
3. (Optional) Override **GetDocument** in the new view class so that it returns a pointer to the class created in Step 1.
4. Add a string to the string table with the ID you will use for the new document type. See Table 1.13 for the format of the string. Here's a typical string for an SDI document that isn't interested in files:

"Extra Document\n\nExtra\n\n\nXtra.Document\nExtra Document"

5. Add other resources (menu, icon, etc.) using the same ID as in Step 4.
6. Create a document template object using **new** in the application's **InitInstance** member function. Suppose your resource ID is **ID_NEWDOC**, the document class is **CNewDoc**, and the view class is **CNewView**. Also, assume you want to use an ordinary **CMDIChildWnd** as a frame window. The document template code should look like this:

```
CMultiDocTemplate *dtp=new CMultiDocTemplate(ID_NEWDOC,
    RUNTIME_CLASS(CNewDoc),
    RUNTIME_CLASS(CMDIFrameWnd),
    RUNTIME_CLASS(CNewView);
```

7. Call **AddDocTemplate** to add the new document template to the application's list.

Once you have completed these steps, MFC will prompt you with a dialog box that lists each document template every time it creates a new file. It will also select an appropriate template (based on the file extension in the string resource) when you open a file.

If you are writing an MDI program, you should use **CMultiDocTemplate** (as in the preceding example). SDI programs use **CSingleDocTemplate** instead. If you want to handle menu messages in the document template (a rare case, but certainly possible), you'll need to derive a class from the appropriate document template and use it.

Creating A Private Document

Sometimes you want to create a document type that doesn't show up on the New dialog box. Perhaps you have a debugging document that you want to use in certain cases or a document you want to create yourself when your program starts.

You can easily do this by creating a document template as usual. However, don't add the template to the document's list. When you want to create an instance of the document (that is, a document class, a view class, and a frame), call the document template's **OpenDocumentFile** with a **NULL** file name.

Attaching Multiple Views To A Document

One of the biggest advantages to the document/view architecture is that you should be able to easily attach more than one view to a single document. For example, if you are writing a spreadsheet program, you could have two grids and a pie chart reflecting the data in the same spreadsheet. The user might scroll the grids to view different portions of the data.

As important as this is, the documentation is largely mute when it comes to how you might actually do this. If you tried to do all the work by hand, you'd find it a daunting task. You have to create a frame window, attach a view to it, and inform the document that it has a new view. However, document templates already know how to do all of these things, so why not ask the document template to do the work?

Suppose you open an MFC program that presents a view of a checkerboard. It creates a **CCheckerDoc** and a **CCheckerView**. Later, you'd like to add a **CDebugView** view to the same document. Here's what you do:

1. Create or obtain a document template object (for example, you might save a pointer to the document template when you create it during **InitInstance**).
2. Call the template's **CreateNewFrame** function. Pass it a pointer to the document you want to use (use a **NULL** for the second argument). This function returns a pointer to the new frame window that contains the view attached to your existing document.
3. Call the frame window's **InitialUpdateFrame** member. This causes the **OnInitialUpdate** member to fire. If you omit this step, your view might work if it doesn't depend on **OnInitialUpdate**, but don't count on it. Some views (like **CScrollView**, for example) require **OnInitialUpdate** for their own purposes.

You can find an example of this technique in Listing 1.2 and Figure 1.2. The code handles a menu item in the document class because it is easy to obtain the document pointer here (simply use **this**). You can find the views and other classes on the CD-ROM.

Listing 1.2 Creating a Debug view.

```
void CCheckerDoc::OnDebugwin()
{
    CMultiDocTemplate temptemplate(IDR_DEBUGTYPE, RUNTIME_CLASS(CCheckerDoc),
        RUNTIME_CLASS(CMDIChildWnd), RUNTIME_CLASS(DebugView));
    CFrameWnd *frame=temptemplate.CreateNewFrame(this, NULL);
    if (!frame)
        AfxMessageBox("Can't create debug window");
    else
        frame->InitialUpdateFrame(this, TRUE);
}
```

Making Separate File|New Menus

By default, the File|New menu has the ID **ID_FILE_NEW**. The default handlers for this item create a dialog box that contains a list of all the document templates in the application's template list (unless, of course, there is only one document template). Although this is ordinary MFC behavior, it is unusual for an ordinary Windows application. Frequently, a non-MFC application will use a separate menu item for each type of document. Usually, the menu items cascade (see Figure 1.3).

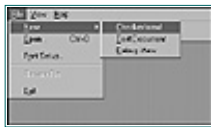


Figure 1.3 A cascading menu.

As long as you know the associated document template for each document type, this is easy. Just make a custom menu handler for each document. Each handler uses the appropriate document template and calls **OpenDocumentFile** with a **NULL** file name. Here is an example code fragment:

```
void CCheckDoc::OnNewText()  
{  
    // assume m_pTextTemplate stores the text document template  
    m_pTextTemplate->OpenDocumentFile(NULL);  
}
```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



Enter EarthWeb's Bag-A-Million Sweepstakes
for your chance to win \$1 million dollars or
tons of other great prizes. [Click here to register!](#)

This site is brought to you by
EarthWeb

EARTHWEB



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP

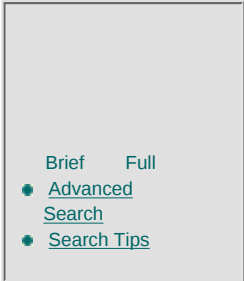


CONTACT US



SEARCH

ITKNOWLEDGE



BROWSE

BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

Preventing A New Document On Startup

In older versions of MFC, it was fairly simple to understand how MFC parsed the command line. In the **InitInstance** method, the App Wizard code examined the command line. If the command line was not empty, the program treated it as a file name. If the command line was empty, the program created a new document. However, subsequent versions of MFC found it necessary to parse command line arguments required for OLE and other features. These extra features led Microsoft to complicate the processing a great deal.

Modern MFC programs use a **CCommandLineInfo** object to control the actions the program takes when it begins. The **ParseCommandLine** object fills in the **CCommandLineInfo** object based on the command line parameters. Then, the program calls **ProcessShellCommand** to perform the action specified in the object.

Knowing how this works, it is easy to examine the **CCommandLineInfo** object after the call to **ParseCommandLine** and take appropriate action. For example, if you don't want an empty document on startup, you might use code similar to this:

```
CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);

    // Dispatch commands specified on the command line
    if (cmdInfo.m_nShellCommand != FileNew)
        if (!ProcessShellCommand(cmdInfo))
            return FALSE;
```

Parsing Command Line Parameters

If you read the previous topic, you can use the same technique to parse your own command line arguments. First, you need to derive your own class from **CCommandLineInfo**. In your derived class, you'll override **ParseParam**. The base class will call this virtual function for each command line argument. If the argument begins with a dash or a forward slash, the second argument to **ParseParam** will be **TRUE** and MFC will strip the leading character from the argument. The first argument contains the argument to process. The third argument is **TRUE** if the argument is the last one on the command line.

Listing 1.3 shows a derived **CCommandLineInfo** class. This simple class sets a few parameters in its

associated application object. You can find the associated application code in Listing 1.4 and the form view code that uses the information in Listing 1.5.

Listing 1.3 Parsing options.

```
#include "stdafx.h"
#include "string.h"
#include "params.h"
#include "customcmd.h"

void CCustomCmd::ParseParam( LPCTSTR lpszParam, BOOL bFlag, BOOL bLast)
{
    CParamsApp *app=(CParamsApp *)AfxGetApp();
    if (bFlag && !strcmp(lpszParam,"P1")) app->m_P1=TRUE;
    else if (bFlag && !strcmp(lpszParam,"P2")) app->m_P2=TRUE;
    // no match, do default things
    else CCommandLineInfo::ParseParam(lpszParam,bFlag,bLast);
}
```

Listing 1.4 Using CCustomCmd.

```
// Parse command line for standard shell commands, DDE, file open
    CCustomCmd cmdInfo;
    ParseCommandLine(cmdInfo);

    // Dispatch commands specified on the command line
    if (!ProcessShellCommand(cmdInfo))
        return FALSE;
```

Listing 1.5 Reading the options.

```
void CParamsView::OnInitialUpdate()
{
    CParamsApp *app=(CParamsApp *)AfxGetApp();
    // read values from app before showing form
    m_P1=app->m_P1;
    m_P2=app->m_P2;
    CFormView::OnInitialUpdate();
}
```

Calculating A View Size

Occasionally, you'll want to create a view with a specific size. For example, perhaps you'd like to create a checkerboard with cells 50 pixels wide (400×400 pixels). The problem is that you can't directly resize the view. Instead, you must resize the frame that contains the view. When you resize the frame (using **MoveWindow** or **SetWindowPos**), you resize the entire window, including the non-client area. Therefore, to get a 400×400 view, you need to make the frame larger than 400×400.

How much larger will it need to be? That depends on several factors. First, the size of the non-client area varies depending on the hardware in use, user preferences, and other factors. Also, the view itself contains a small non-client area. To make the view's client area 400×400, you must first calculate the total size of the view, including the non-client area in your calculation. Then you must compute how large the frame window must be to contain that size.

Windows can compute a total size based on a desired client area size using the **::AdjustWindowRectEx** call. You must supply the window's styles and a flag specifying if the window has a menu or not. You must also specify the size you'd like the window's client area to be.

Views never have menus, and MDI child frames don't either. You can use **GetStyle** and **GetStyleEx** to learn the styles of an existing window. If you need the size before you have the window, you can always learn the

styles using a tool like Spy++ and hard code them. Remember, the sizes may change on different machines, but the styles should not.

Here's some sample code you might use to size a view (**v**) to have a 400x400 pixel client area:

```
CRect vsize(0,0,400,400);
CWnd *frame=v->GetParentFrame();
::AdjustWindowRectEx(&vsize,v->GetStyle(),FALSE,v->GetExStyle ());
// vsize may not start at (0,0) now
::AdjustWindowRectEx(&vsize,frame->GetStyle(),FALSE,frame->GetExStyle());
frame->SetWindowPos(NULL,0,0,
    vsize.Width(),vsize.Height(), SWP_NOACTIVATE|SWP_NOMOVE|SWP_NOZORDER);
```

Using Typedef With Templates

MFC's template-based collections are very helpful, but the syntax required to use templates is often unpleasant. For example, suppose you want a list of **CWnd**s. You'd have to write:

```
CList<CWnd,CWnd &> winlist;
```

Better hope you don't need to cast a void pointer to a pointer to this list. That looks like this:

```
CList<CWnd,CWnd &> *lp = (CList<CWnd,CWnd &> *)vp;
```

A better answer is to use a **typedef** to make this more readable. How about:

```
typedef CList<CWnd, CWnd &> CWndList;
CWndList winlist;
CWndList *lp=(CWndList *)vp;
```

This is much better, don't you agree?

A 2D CArray

Have you ever started to use a **CArray** and then found you wanted more than one dimension? There are many possible answers (including just using an ordinary C-language array). However, one that always strikes me as nifty is to create a **CArray** of **CArrays** (this is for a 2D array; you can extend the idea for more dimensions).

The code (below) creates a type for an array of integers (**CIntArray**) and another for an array of **CIntArray**s. The array of arrays is called **CIntArray2**. Finally, it derives a class from **CIntArray2** named **CIntMatrix**. This subclass has a constructor that sets the sizes of each array to hold the specified number of rows and columns:

```
#include <afxtempl.h>
typedef CArray<int,int> CIntArray;
typedef CArray<CIntArray,CIntArray> CIntArray2;

class CIntMatrix : public CIntArray2
{
public:
    CIntMatrix(unsigned x,unsigned y)
    {
        SetSize(y);
        for (unsigned i=0;i<y;i++)
            ElementAt(i).SetSize(x);
    }
};
```

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Chapter 2 Serialization

*Archiving is one area of MFC where the framework automatically does what you want it to do—most of the time. What about the other times? Peek inside the **CArchive** class and learn how to customize it for your own purposes.*

Programmer's Notes...

My wife Pat and I watch a lot of movies. Don't tell her, but I really only enjoy one kind of movie. Like many programmers, I enjoy science fiction. However, I'm not overly enthused with all of the new high tech, glitzy, special effects extravaganzas. I prefer the old campy movies like *Plan Nine from Outer Space*, *The Queen of Outer Space*, and, of course, *Destination Saturn* (Buck Rogers). Flash Gordon suits me, too, but for some reason, I always related better to Buck (after all, Flash was a jock).

Today when you go to the theater, you feel like you need to fill out a credit application. Back when I was a kid, you could go for a quarter (or even less). But the best part about these movies is that they were serials. You'd go watch Buck try to free Wilma from Kane's hideout. Then Kane would say something like, "When I push this button, Buck Rogers will die!" As his finger stabbed at the button, the movie would end. You'd have to go back next Saturday afternoon and spend another quarter. They tell me that westerns (which never interested me) often ended with the hero hanging by a branch off a cliff, hence the name "cliffhanger." Some serials were better than others; I always liked

the ones from Republic Studios, which must have made a million serial movies.

Thanks to the magic of the VCR, you can enjoy these serials all over again (you can sometimes catch them on the American Movie Classics channel, too). But there's one big disadvantage: When you watch them back to back, you notice that Kane's finger actually pushes the button in episode 21, but at the beginning of episode 22, his finger hovers a millimeter over the button, and someone yells, "Wait!"

To keep this from being such a problem, some companies edit the serials together to form one feature-length movie on a VCR. I suppose the word for this is de-serialization. Somehow, the loss of the cliffhangers makes the movie less satisfying.

MFC uses the word "serialization" to describe how it makes objects persistent. The serial part is that you write data out to a stream. Each piece of data goes out to the stream ahead of the next piece of data. Just like at the cinema, you have to watch one episode before you can see the next one (it's no fair to fast forward).

What's a stream? Well, that depends. Most often, it is a file. In fact, the MFC tools always assume it's a file, so that's what most people do with them. However, a stream might be any kind of storage you want to put data into for retrieval later.

Persistence Vs. Storage

As usual, MFC uses one word to describe multiple concepts, confusing everyone. In one sense, serialization is how an object writes its state to a stream so that it can later be recreated. You might recreate it later in the same program. You might recreate it while running a different program, another instance of the same program, or even on another machine. You might recreate it on the other side of a network connection.

However, a more common use for serialization is to have your **CDocument** object save itself. Wait! Isn't that the same thing? Not exactly. Suppose you are writing a text editor program using MFC. What do you want to save? Not the **CDocument** object. Who wants to open your text file and see all the guts of a **CDocument**? No one. That's why MFC's **CEditView** has a **SerializeRaw** function to write just the ASCII text out to a file. So in the text editor, serializing a **CDocument** doesn't mean making the object persistent. It merely means to save the state of the program, which is not always the same thing. Another example would be a program that wants to save its screen layout in the saved document. That information is probably not in the **CDocument** object, but the object still must save it.

Making an object persistent isn't as easy as it might sound. Think about writing an array object to disk. If the array contains integers, that's easy, right? Just dump out the contents along with whatever bookkeeping information the object needs (number of elements, for example). But what if the array contains pointers to strings? That's more work. You'll have to write the strings out to

the file. But what if some of the elements point to the same string? Will you write the string out twice? That means when you read the array back in, it will be different from the original array. Ultimately, you'll have to write a string table and store an array of indices into the table.

The situation can also get worse. What if the array contains pointers to an object? Or, what if the object type is a virtual base class and the pointers actually point to various derived class objects? These objects might contain pointers to other objects, strings, and who knows what else.

MFC provides a nice mechanism for handling this problem. Any class that derives from **CObject** (the base class for most major objects) can serialize itself if the class author included the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros in the class. Each class is also responsible for reading and writing any data it needs to the stream (a **CArchive**). The base classes take care of themselves. If your object contains another object, you need only ask it to serialize itself as part of your processing.

Serializing Classes

A class doesn't need to use **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** unless you want to serialize the object itself. Just to save program data, you don't need all the extra things these macros do for you. For example, App Wizard-generated programs all use **IMPLEMENT_DYNCREATE** (see Chapter 1) in the document object even though you usually override **Serialize**. **IMPLEMENT_SERIAL** adds the >> operator, adds the class information to a linked list, and stores a version number (schema) for the object. These things facilitate serializing the entire object to an archive, as you'll see later in this chapter.

Do you have to use serialization? No. As usual, the tools don't really understand this, so if you don't want to use serialization, you are on your own. Why wouldn't you use serialization? Maybe you are writing an address book and you want to store new entries immediately. Perhaps your data is from an instrument connected via a serial port. Or, you might have to shoehorn your data into an existing file format.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

A Quick Look At CArchive

You can't understand serialization without knowing about **CArchive**. There's a lot of confusion about what exactly **CArchive** is. It is simply a class that represents *someplace* to store and retrieve data. Although this place is often a file, that isn't necessarily the case. Table 2.1 shows the members available. Most often you'll use << and >> to write or read the archive. This is just the usual C++ iostream nomenclature. For example, to write a **DWORD** named **dwvar** to an archive named **ar**, you might use the code:

```
ar<<dwvar;
```

Table 2.1 CArchive members.

Member	Description
m_pDocument	The document using this archive
Abort	Closes archive without throwing exceptions
Close	Closes archive
Flush	Writes pending data to data stream
operator<<	Writes an object or primitive type to the archive
operator>>	Reads an object or primitive type from the archive
Read	Reads bytes
Write	Writes bytes
ReadString	Reads a string
WriteString	Writes a string

GetFile	Get the underlying CFile or CFile-derived class
GetObjectSchema	Read the object version number, if known
SetObjectSchema	Sets the object version number
IsLoading	Returns TRUE if archive is for reading
IsStoring	Returns TRUE if archive is for writing
IsBufferEmpty	Detects empty buffer for socket-based archives
ReadObject	Read and create a serialized object
WriteObject	Serializes an object
MapObject	Places an object in the archive's map without actually serializing the object itself
SetStoreParams	Sets the initial size of the hash table used to track objects written to the archive
SetLoadParams	Sets the size the hash table will grow by as the archive reads objects
ReadClass	Reads class information
WriteClass	Writes class information
SerializeClass	Reads or writes class information based on the direction of the archive

If you need to know if you should read instead of write, you can ask the archive. The **IsStoring** and **IsLoading** functions tell you which direction the object wants data to flow. Trying to store data to a loading archive (or vice versa) causes an exception. So you might write:

```
if (ar.IsStoring())
    ar<<dwvar;
else
    ar>>dwvar;
```

For times when you want to write an entire object, a string, or just some raw bytes, you can use **WriteObject**, **WriteString**, or **Write** (there are corresponding **Read** methods, too).

For now, that's all you really need to know about **CArchive**. The real trick is when you want a **CArchive** to represent something odd (like an encrypted file, a database, or a network socket). That's a topic I'll return to later in this chapter.

Inside File Open And Save

When you create a program with App Wizard, it magically knows how to open and save files. Your only job is to flesh out the skeletal **Serialize** function in your document class. That's fine if you just want to use an arbitrary file format in a regular file. But, as usual, if that's not what you want, you are on your own. Before you consider the alternatives, look at how the default File|Open processing works in Table 2.2.

Table 2.2 File|Open processing.

MFC calls...	Override when...
CWinApp::OnFileOpen	You want to do all your own work
CWinApp::OpenDocumentFile	You only want MFC to get the file name
CDocTemplate::OpenDocumentFile	You want MFC to get the file name and select a template
CDocument::OnOpenDocument	You want MFC to get the file name, select a template, and create the frame, document, and view objects
CDocument::Serialize	You want MFC to do all the work, including opening the file and assigning it to a CArchive

[Previous](#)
[Table of Contents](#)
[Next](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

The standard menu ID for File|Open is **ID_FILE_OPEN**. If you don't provide a handler, the application object's default handler catches it. If you wanted to do all the work yourself, you could put your own message map handler in one of your classes. Of course, you could always just use another ID, too.

Although this will give you complete control over the file open process, this may be too much of a good thing. After all, think about what MFC has to do to open a file. Here are the rough steps:

1. Prompt for a file name.
2. Select a document template that matches the file.
3. Create a document, frame, and view object.
4. Open the file.
5. Bind the file to an archive.
6. Call `Serialize` on the new document.

Steps 2 and 3, in particular, are a big pain. Why not let MFC do at least some of the work?

The default handler for **ID_FILE_OPEN** simply prompts for a file name by calling **CWinApp::DoPromptFileName**. You should be able to override **DoPromptFileName** to provide a custom dialog box, right? Not exactly. The problem is that MFC doesn't define **DoPromptFileName** as virtual. You can override it all you want, and MFC won't call your version. The answer is to also replace the functions that call **DoPromptFileName** (**CWinApp::OnFileOpen** and **CDocument::DoSave**, which are virtual).

Once MFC has the file name, it calls **CWinApp::OpenDocumentFile**. This function searches the template list and selects one template (usually based on

the file extension). This would be a good function to override if you had more sophisticated criteria for selecting document templates.

Once MFC selects a document template, it calls the template's **OpenDocumentFile** function. You'll rarely want to override this because it creates the document, view, and frame objects. Once everything is in place, the template calls **CDocument::OnOpenDocument**.

OnOpenDocument can be a very useful place to insert your own code. At the point it is called, your MFC objects are all ready and you have a file name. If you have custom code that knows how to open a document based on a file name, this is the place to put it. The default code opens the file as a **CFile** object, binds that file to a **CArchive**, and calls the document's **Serialize** function. As usual, MFC allows you to take the defaults, making your job easy, or override things if you want to change them. You can take control at any point in the process. If you like, you can also pass control back to MFC.

For example, suppose you are converting a legacy program to MFC. The existing file format is sacred, so you dare not change it. You have three problems:

- The file extension doesn't matter, but the first 3 bytes in the file identify its type
- Because the file extension doesn't matter, you only want a file filter of *.*.
- You have existing code to read the file that uses C **FILE** pointers

This really isn't a problem. You have two choices: First, you could override **OnFileOpen**, do your own file prompt, select a template (based on the first 3 bytes in the file), and call the template's **OpenDocumentFile** function. MFC takes care of the rest. Alternatively, you could override **DoPromptFileName** (and its related functions) and **CWinApp::OpenDocumentFile**. Either way is satisfactory. Overriding **OnFileOpen** doesn't affect File|Save As (as you'll see in just a moment).

What about the third problem? That's the easiest problem of all. Just override the **CDocument::OnOpenDocument** function. There, you can open a file with **fopen** (remember **fopen**, the old C library way to open a file?) and call the old code that does the File|Open logic.

The File|Save logic is much simpler (see Table 2.3). This is expected because there's no need to select a template or create a lot of objects. Everything is already in place. If you try to save the file before it has a name, the File|Save As logic executes. However, calling **OnFileSave** never causes a call to **OnFileSaveAs**. If you want to handle these cases in a special way, you'll need to override both functions (or change the menu IDs).

Table 2.3 The File|Save and Save As process.

MFC calls...	Override when...
CDocument::OnFileSave/OnFileSaveAs You want to do all your own work	

CDocument::OnSaveDocument	You only want MFC to get the file name
CDocument::Serialize	You want MFC to do all the work, including opening the file and assigning it to a CArchive

Because these commands apply to the active document, the handlers reside in the document object. That means the handlers already know what document is active. If the name is unknown, MFC calls **DoPromptFileName**. This is the same function that File|Open uses to display a file dialog, but this time it will get the name to save. Once the name is known, the menu handler calls **OnSaveDocument**. In the legacy code example above, you'd want to override **OnSaveDocument** so you could call **fopen** and pass it along to the old-style code that saves a document.

The default **OnSaveDocument** opens a **CFile**, binds it to a **CArchive**, and calls **Serialize**. Of course, File|Open calls **Serialize**, too. You can ask the archive which case you have by calling **IsLoading** or **IsStoring** on the archive.

Why Does Serialize Save And Load?

A question that comes up frequently is why does **Serialize** do both loading and saving duty? Why not have **SerializeIn** and **SerializeOut** functions instead? The answer becomes clear when you work with a document that contains other serializable objects. App Wizard writes a default **Serialize** function that looks like this:

```
if (ar.IsStoring())
{
// TODO: add storing code here
}
else
{
// TODO: add loading code here
}
```

Suppose that your document stores everything in a **CArray** (see Chapter 1) named **ary** that knows how to serialize itself. It would be silly to write:

```
if (ar.IsStoring())
{
ary.Serialize(ar);
}
else
{
ary.Serialize(ar);
}
```

Instead, why not delete the **if** statement and write:

```
ary.Serialize(ar);
```



[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Providing A Custom Dialog

The easiest way to customize the Open and Save As dialogs is to override

CWinApp::DoPromptFileName (which is largely undocumented—only Technical Note 22 mentions it). However, this function isn't as simple as you'd think because the function searches the document templates to dynamically build the filter string for the dialog. That way, the dialog's filter (the part that allows you to select certain types of files) matches the templates available.

For your own purposes, it may be sufficient to hard code the filter. If you prefer, you can lift the code from the MFC source and use it as a starting point for your own code. The actual

CWinApp::DoPromptFileName function calls a function with the same name in the document manager object (an internal object that MFC uses to handle documents). You can find the function in the MFC source (DOCMGR.CPP). You can lift this code out, put it in your **CWinApp**-derived object, and make any changes you like. You'll have to change a few references that are specific to the document manager, but the compiler will flag all of these for you (you'll also see an example in just a moment).

Using The MFC Source

Nearly everyone will agree that changing the MFC source code isn't a good idea. Although you can, in theory, change MFC and compile your own DLLs, that path is fraught with peril. For one, your DLLs become incompatible with the standard MFC DLLs. Second, when Microsoft releases a new version of MFC, you'll have to propagate your changes to the new code (which may not resemble the old code).

Does that mean the MFC source code is useless? Not at all. You can often gain valuable insights by examining the MFC source code. Also, you can always copy code from the source into your own overrides and change that code. You might still have problems when Microsoft releases a new version, but at least the problem will be in your code.

Sometimes you might find a bug in MFC. Consider deriving a new class from the faulty class and fixing the problem there. Resist the urge to make any other changes (if you need more changes, derive a subclass from your new class). That way, if Microsoft repairs the bug later, you can simply revert to the original class.

However, that's not the whole story. The **DoPromptFileName** function calls a static function named **AppendFilterSuffix**. You'll need to copy it, too. Because **DoPromptFileName** isn't virtual, you also need to add an **OnFileOpen** message handler to the application object and override **DoSave** in the document. You can copy the code nearly verbatim from the source code (or the example below) because the only reason you replace them is to make sure your **DoPromptFileName** executes. It isn't clear why **DoPromptFileName** isn't virtual. It is virtual in the internal **CDocManager** object, but that doesn't do you much good.

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

You can find a very simple example of this in Figure 2.1 and Listings 2.1 and 2.2. Here, I put a fixed message in the file dialog. Obviously, you could do anything you wanted (including replacing the entire dialog, if you like). If you want to learn more about customizing the common file dialog, see Chapter 5.



Figure 2.1 A custom file dialog.

By overriding **DoPromptFileName**, you change both File|Open and File|Save As prompting. In a more sophisticated application, you'd probably use a custom subclass instead of **CFileDialog**.

Listing 2.1 Custom file dialog (Document Object).

```
// custdlgDoc.cpp : implementation of the CCustdlgDoc class
//

#include "stdafx.h"
#include "custdlg.h"
#include "custdlgDoc.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////
// CCustdlgDoc

IMPLEMENT_DYNCREATE(CCustdlgDoc, CDocument)

BEGIN_MESSAGE_MAP(CCustdlgDoc, CDocument)
    //{AFX_MSG_MAP(CCustdlgDoc)
    //{AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////
// CCustdlgDoc construction/destruction

CCustdlgDoc::CCustdlgDoc()
{
    // TODO: add one-time construction code here
}

CCustdlgDoc::~CCustdlgDoc()
{
}

BOOL CCustdlgDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)
```

```

        return TRUE;
    }

    //////////////////////////////////////
    // CCustdlgDoc serialization

void CCustdlgDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
}

    //////////////////////////////////////
    // CCustdlgDoc diagnostics

#ifdef _DEBUG
void CCustdlgDoc::AssertValid() const
{
    CDocument::AssertValid();
}

void CCustdlgDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}
#endif // _DEBUG

    //////////////////////////////////////
    // CCustdlgDoc commands

// This is more or less a copy of the original
// version of DoSave, except it calls our version
// of DoPromptFileName
BOOL CCustdlgDoc::DoSave(LPCTSTR lpszPathName, BOOL bReplace)
{
    CString newName = lpszPathName;
    if (newName.IsEmpty())
    {
        CDocTemplate* pTemplate = GetDocTemplate();
        ASSERT(pTemplate != NULL);

        newName = m_strPathName;
        if (bReplace && newName.IsEmpty())
        {
            newName = m_strTitle;
#ifdef _MAC
            // check for dubious filename
            int iBad = newName.FindOneOf(_T(" #%;/\\"));
#else
            int iBad = newName.FindOneOf(_T(":"));
#endif
#endif
        }
    }
}

```

```

        if (iBad != -1)
            newName.ReleaseBuffer(iBad);

#ifdef _MAC
        // append the default suffix if there is one
        CString strExt;
        if (pTemplate->GetDocString(strExt, CDocTemplate::filterExt) &&
            !strExt.IsEmpty())
        {
            ASSERT(strExt[0] == '.');
            newName += strExt;
        }
#endif
    }

    CCustdlgApp *app=(CCustdlgApp *)AfxGetApp();
    if (!app->DoPromptFileName(newName,
        bReplace ? AFX_IDS_SAVEFILE : AFX_IDS_SAVEFILECOPY,
        OFN_HIDEREADONLY | OFN_PATHMUSTEXIST, FALSE, pTemplate))
        return FALSE;        // don't even attempt to save
    }

    CWaitCursor wait;

    if (!OnSaveDocument(newName))
    {
        if (lpszPathName == NULL)
        {
            // be sure to delete the file
            TRY
            {
                CFile::Remove(newName);
            }
            CATCH_ALL(e)
            {
                TRACE0("Warning: failed to delete file after failed
                    SaveAs.\n");
            }
            END_CATCH_ALL
        }
        return FALSE;
    }

    // reset the title and change the document name
    if (bReplace)
        SetPathName(newName);

    return TRUE;        // success
}

```

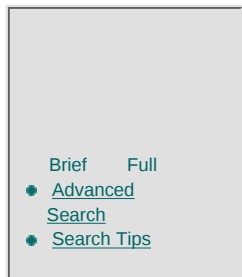
[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE



BROWSE BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97



Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Listing 2.2 Custom file dialog (Application Object).

```
// custdlg.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "custdlg.h"

#include "MainFrm.h"
#include "custdlgDoc.h"
#include "custdlgView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CCustdlgApp

BEGIN_MESSAGE_MAP(CCustdlgApp, CWinApp)
//{{AFX_MSG_MAP(CCustdlgApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
ON_COMMAND(ID_FILE_OPEN, OnFileOpen)
//}}AFX_MSG_MAP
// Standard file based document commands
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
// Standard print setup command
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

////////////////////
// CCustdlgApp construction

CCustdlgApp::CCustdlgApp()
```

```

{
    // TODO: add construction code here
    // Place all significant initialization in InitInstance
}

////////////////////////////////////
// The one and only CCustdlgApp object

CCustdlgApp theApp;

////////////////////////////////////
// CCustdlgApp initialization
BOOL CCustdlgApp::InitInstance()
{
    AfxEnableControlContainer();

    // Standard initialization
    // If you are not using these features and wish to reduce the size
    // of your final executable, you should remove from the following
    // the specific initialization routines you do not need.

#ifdef _AFXDLL
    Enable3dControls();          // Call this when using MFC in a shared DLL
#else
    Enable3dControlsStatic();    // Call this when linking to MFC statically
#endif

    // Change the registry key under which our settings are stored.
    // You should modify this string to be something appropriate
    // such as the name of your company or organization.
    SetRegistryKey(_T("Local AppWizard-Generated Applications"));

    LoadStdProfileSettings();    // Load standard INI file options
    (including MRU)

    // Register the application's document templates. Document templates
    // serve as the connection between documents, frame windows, and views.

    CSingleDocTemplate* pDocTemplate;
    pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(CCustdlgDoc),
        RUNTIME_CLASS(CMainFrame),           // main SDI frame window
        RUNTIME_CLASS(CCustdlgView));
    AddDocTemplate(pDocTemplate);

    // Parse command line for standard shell commands, DDE, file open
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);

    // Dispatch commands specified on the command line
    if (!ProcessShellCommand(cmdInfo))
        return FALSE;

    // The one and only window has been initialized, so show and update it.
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}

```

```

////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

// Dialog Data
    //{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{AFX_VIRTUAL(CAboutDlg)
protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}AFX_VIRTUAL

// Implementation
protected:
    //{AFX_MSG(CAboutDlg)
    // No message handlers
    //}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    //{AFX_DATA_INIT(CAboutDlg)
    //}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{AFX_DATA_MAP(CAboutDlg)
    //}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{AFX_MSG_MAP(CAboutDlg)
    // No message handlers
    //}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void CCustdlgApp::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}

////////////////////////////////////
// CCustdlgApp commands

// This is just a copy of an existing (static) function
// from the docmgr object
void AppendFilterSuffix(CString& filter, OPENFILENAME& ofn,
    CDocTemplate* pTemplate, CString* pstrDefaultExt)

```

```

{
    ASSERT_VALID(pTemplate);
    ASSERT_KINDOF(CDocTemplate, pTemplate);

    CString strFilterExt, strFilterName;
    if (pTemplate->GetDocString(strFilterExt, CDocTemplate::filterExt) &&
        !strFilterExt.IsEmpty() &&
        pTemplate->GetDocString(strFilterName, CDocTemplate::filterName) &&
        !strFilterName.IsEmpty())
    {
        // a file based document template - add to filter list
#ifdef _MAC
        ASSERT(strFilterExt[0] == '.');
#endif
        if (pstrDefaultExt != NULL)
        {
            // set the default extension
#ifdef _MAC
            *pstrDefaultExt = ((LPCTSTR)strFilterExt) + 1; // skip the '.'
#else
            *pstrDefaultExt = strFilterExt;
#endif
            ofn.lpstrDefExt = (LPTSTR)(LPCTSTR)(*pstrDefaultExt);
            ofn.nFilterIndex = ofn.nMaxCustFilter + 1; // 1 based number
        }

        // add to filter
        filter += strFilterName;
        ASSERT(!filter.IsEmpty()); // must have a file type name
        filter += (TCHAR)'\0'; // next string please
#ifdef _MAC
        filter += (TCHAR)'\0';
#endif
        filter += strFilterExt;
        filter += (TCHAR)'\0'; // next string please
        ofn.nMaxCustFilter++;
    }
}

// This is more or less a copy of the original
// but it uses our dialog template
BOOL CCustdlgApp::DoPromptFileName(CString & fileName, UINT nIDSTitle,
    DWORD lFlags, BOOL bOpenFileDialog, CDocTemplate * pTemplate)
{
    CFileDialog dlgFile(bOpenFileDialog);

    CString title;
    VERIFY(title.LoadString(nIDSTitle));

    dlgFile.m_ofn.Flags |= lFlags;

    CString strFilter;
    CString strDefault;
    if (pTemplate != NULL)
    {
        ASSERT_VALID(pTemplate);
        AppendFilterSuffix(strFilter, dlgFile.m_ofn, pTemplate, &strDefault);
    }
    else
    {

```

```

        // do for all doc template
        POSITION pos = GetFirstDocTemplatePosition();
        BOOL bFirst = TRUE;
        while (pos != NULL)
        {
            CDocTemplate* pTemplate = GetNextDocTemplate(pos);
            AppendFilterSuffix(strFilter, dlgFile.m_ofn, pTemplate,
                bFirst ? &strDefault : NULL);
            bFirst = FALSE;
        }
    }

    // append the "*.*" all files filter
    CString allFilter;
    VERIFY(allFilter.LoadString(AFX_IDS_ALLFILTER));
    strFilter += allFilter;
    strFilter += (TCHAR)'\0';    // next string please
#ifdef _MAC
    strFilter += _T("*.");
#else
    strFilter += _T("*****");
#endif
    strFilter += (TCHAR)'\0';    // last string
    dlgFile.m_ofn.nMaxCustFilter++;

    dlgFile.m_ofn.lpstrFilter = strFilter;
#ifdef _MAC
    dlgFile.m_ofn.lpstrTitle = title;
#else
    dlgFile.m_ofn.lpstrPrompt = title;
#endif
    dlgFile.m_ofn.lpstrFile = fileName.GetBuffer(_MAX_PATH);

    // Customize
    dlgFile.m_ofn.Flags|=OFN_ENABLETEMPLATE;
    dlgFile.m_ofn.lpTemplateName="CustomTempl";
    BOOL bResult = dlgFile.DoModal() == IDOK ? TRUE : FALSE;
    fileName.ReleaseBuffer();
    return bResult;
}

// This is a copy of the original, but it calls
// our DoPromptFileName
void CCustdlgApp::OnFileOpen()
{
    // prompt the user (with all document templates)
    CString newName;
    if (!DoPromptFileName(newName, AFX_IDS_OPENFILE,
        OFN_HIDEREADONLY | OFN_FILEMUSTEXIST, TRUE, NULL))
        return; // open cancelled

    OpenDocumentFile(newName);
    // if returns NULL, the user has already been alerted
}

```

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



Enter EarthWeb's Bag-A-Million Sweepstakes
for your chance to win \$1 million dollars or
tons of other great prizes. [Click here to register!](#)

This site is brought to you by
EarthWeb



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US



SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)



BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

Another Example

Magazine articles and books are excellent places to look for existing code. Usually, you won't even need to pay for code from these sources. Another good place to look for free code is the online services and the Internet. An excellent place to look for free Windows code is from Microsoft itself. Microsoft publishes its Developer's Network CD four times a year. A subscription isn't cheap, but it is chock full of documentation and sample code, which you are free to use in your applications.

You can also find existing code as part of commercial tool kits. Usually, the company will offer support, but it is more difficult to obtain source code with these tools. You'll often pay a hefty price or royalties to use these products, as well.

The problem with lifting existing code is that it is often incompatible with MFC. Consider a case in which I needed to write a simple bitmap viewer. A quick look in the Developer's Network CD turned up a DLL that works with bitmap files. The DIBAPI.DLL library does many common operations on BMP files, including opening and reading them from disk and drawing them. The library is part of the WINCAP sample. You can also find the WINCAP sample (which contains DIBAPI) on some SDK CDs and online. The document number is S14049 on older CDs or (lately) Q97193.

Because the CD contains the source code to DIBAPI, you could rewrite it to have a C++-style interface. I prefer, however, not to tear into someone else's code that is working. Instead, I decided to wrap the DLL in a C++ object (a **CDib**—see Chapter 7 and Table 2.4). Anyone using a **CDib** need not know that the implementation is inside a C language DLL. My original implementation of **CDib** was for 16-bit MFC. When I moved it all to 32 bit, I did have to make some changes to the DIBAPI

source (you can find it all on the companion CD-ROM).

Table 2.4 CDib members.

Member	Description
Create()	Create DIB from various parameters
ErrorMessage()	Display error message (static function)
GetPalette()	Get DIB palette
Paint()	Paint DIB
Print()	Print DIB
Save()	Save DIB to BMP file
Height()	Get DIB height
Width()	Get DIB width
NumColors()	Get number of colors in DIB
Bits()	Get pointer to DIB bits
GetSize()	Return size of DIB
GetHDIB()	Returns underlying HDIB
GetHDIBCopy()	Get copy of underlying HDIB

Inside CDib

If you want the complete story about **CDib**, look in Chapter 7. For now, you need only realize that a **CDib** represents a BMP file much as a **CWnd** represents a window.

The **CDib** object, like many other MFC objects, uses a two-phase construction scheme. Declaring a **CDib** does not associate it with an actual DIB. To initialize the **CDib**, you must call one of the **Create** functions. You can create a **CDib** from a file, another DIB, a window, or the screen. You can also create a new **CDib** if you like. The problem of interest here is that **CDib** understands file names. It doesn't want a **CFile** or a **CArchive**.

The Sample Application

To illustrate the use of **CDib**, I wrote a simple BMP viewer (BMPVIEW; see Figure 2.2 and Listing 2.3).



Figure 2.2 The Bitmap Viewer opens a file.

BMPVIEW started life as an App Wizard application. However, I removed some of App Wizard's standard features. For example, as BMPVIEW can't modify a file, there is no Save option on the menu (you can use Save As to store a bitmap in a different file). Also, there is no need to start the program with an empty document.

Listing 2.3 The Bitmap Viewer document object.

```
// bmpvidoc.cpp : implementation of the CBmpViewDoc class
//

#include "stdafx.h"
#include "bmpview.h"

#include "bmpvidoc.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

////////////////////
// CBmpViewDoc

IMPLEMENT_DYNCREATE(CBmpViewDoc, CDocument)

BEGIN_MESSAGE_MAP(CBmpViewDoc, CDocument)
   //{{AFX_MSG_MAP(CBmpViewDoc)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////
// CBmpViewDoc construction/destruction

CBmpViewDoc::CBmpViewDoc()
{
    m_dib=NULL;
}

CBmpViewDoc::~CBmpViewDoc()
{
}

BOOL CBmpViewDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    return TRUE;
}

void CBmpViewDoc::DeleteContents(void)
{
    if (m_dib) delete m_dib;
}
```

```

        m_dib=NULL;
    }

    BOOL CBmpViewDoc::OnOpenDocument(const char * fn)
    {
        DeleteContents();
        SetModifiedFlag(FALSE);

        m_dib=new CDib;
        return m_dib->Create((LPSTR)fn);
    }

    BOOL CBmpViewDoc::OnSaveDocument(const char *fn)
    {
        WORD err;
        err=m_dib->Save(fn);
        if (err) CDib::ErrorMessage(err);
        return err==0;
    }

    ////////////////////////////////////
    // CBmpViewDoc serialization

    void CBmpViewDoc::Serialize(CArchive& ar)
    {
        if (ar.IsStoring())
        {
            // TODO: add storing code here
        }
        else
        {
            // TODO: add loading code here
        }
    }

    ////////////////////////////////////
    // CBmpViewDoc diagnostics

#ifdef _DEBUG
    void CBmpViewDoc::AssertValid() const
    {
        CDocument::AssertValid();
    }

    void CBmpViewDoc::Dump(CDumpContext& dc) const
    {
        CDocument::Dump(dc);
    }
#endif // _DEBUG

```

The document object requires little change. Of course, you must add a **CDib** object

(the **m_dib** member) and the CDIB.H header file. Because **CDib** doesn't work with ordinary MFC serialization, the document object overrides **OnOpenDocument** and **OnSaveDocument**. These calls translate to a **CDib** constructor and the **CDib Save** member function, respectively. This subverts the standard serialize operations, but it preserves all the MFC preparation (such as dialog boxes, document construction, and so on). BMPVIEW would not want to use true serialization anyway. The value of this program is that it reads and writes *standard* bitmap files.

Here is a case where MFC serialization is in the way. The program doesn't need or want it, so it just goes around it. By overriding **OnOpenDocument** and **OnSaveDocument**, you can take complete control of the loading and saving processes.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



Enter EarthWeb's Bag-A-Million Sweepstakes
for your chance to win \$1 million dollars or
tons of other great prizes. [Click here to register!](#)

This site is brought to you by
EarthWeb

EARTHWEB



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US



SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)



BROWSE
BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

Serializing Objects

Serialization is a little different only if you are trying to completely dump objects out. When you are dumping an entire object, you need to call the base class implementation of **Serialize** as the first step of your **Serialize** override. This allows the base class to write whatever data it requires. In reality, **CObject** doesn't serialize any data, but you shouldn't count on that. Future versions of MFC might have persistent **CObject** data. Of course, if your class derives from another class, it very likely will want to serialize data.

The other important reason to call the base class is to make sure your class information is written out properly. MFC maintains a record of interesting information about serializable classes: their name, size, creation function, version number, and other specifics. The first time you serialize an object, MFC writes out the class information for the object. MFC is smart enough, however, not to write it out twice.

After you call the base class, you should serialize any data you need to remember. It is up to you what you want to save. After all, there is no need to save anything you don't care to remember when you reload.

You shouldn't need to manually get into the object file format that MFC uses. However, if you ever need to read the file directly, the format is documented in Technical Note 2.

The **ReadObject** and **WriteObject** methods of **CArchive** are the only way to properly serialize an object. However, you'll rarely call them directly. Instead, the **IMPLEMENT_SERIAL** and some internal MFC code conspire to provide << and >> operators that call **ReadObject** and **WriteObject** for you.

When you use the >> operator to read an object in, you just provide a **CObject** pointer. MFC creates a new instance of the appropriate class and reads it in for you. If you know the type you expect, you can cast it to the right type (use **IsKindOf** if you are not sure, or especially careful).

One interesting note about object serialization: **CDocument** objects don't readily lend themselves to true object serialization. That's because you need to call **WriteObject** to serialize an object properly. However, MFC's save and load routines directly call the document's **Serialize** method. So even if you change your document object to use **IMPLEMENT_SERIAL**, it really doesn't use the serialization file format.

However, there is one case in which you might want to use **IMPLEMENT_SERIAL** in conjunction with a **CDocument**-derived class. The problem arises when you want to serialize objects that have pointers to the **CDocument** object. You don't want to write the object itself into the archive, but you need a reference to the object to satisfy the pointers. The solution is to call **MapObject**. This places a reference to the specified object in the archive, but not an actual copy of the object. On output, this gives the archive something to reference, and on input, it informs the archive what object it should use.

Handling Multiple Versions

Look at Figures 2.3 and 2.4. The first program maintains a list of names and email addresses. You'll find that the document object uses a helper object, **EMailDB**. This object maintains the database (such that it is) and is a serializable class. You'll notice in Listing 2.4 that the **IMPLEMENT_SERIAL** macro has a special third argument: **VERSIONABLE_SCHEMA|1**.



Figure 2.3 Version 1 of the email program.

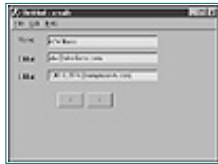


Figure 2.4 Version 2 of the email program.

Listing 2.4 The first email program document.

```
// emailsDoc.cpp : implementation of the CEmailsDoc class
//

#include "stdafx.h"
#include "emails.h"

#include "emailsDoc.h"
#include "emailsView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CEmailsDoc

IMPLEMENT_DYNCREATE(CEmailsDoc, CDocument)
IMPLEMENT_SERIAL(EMailDB, CObject, VERSIONABLE_SCHEMA|1)

void EMailDB::Serialize(CArchive& ar)
{
    int i;
    if (ar.IsLoading())
    {
        int version=ar.GetObjectSchema();
        if (version!=1)
        {
            AfxMessageBox("Unknown file message");
            return;
        }
    }
    CObject::Serialize(ar);
    ar>>m_count;
    for (i=0;i<=m_count;i++)
    {
        ar>>name[i];
    }
}
```

```

        ar>>email[i];
    }
}
else
{
    CObject::Serialize(ar);
    ar<<m_count;
    for (i=0;i<=m_count;i++)
    {
        ar<<name[i];
        ar<<email[i];
    }
}
}

BEGIN_MESSAGE_MAP(CEmailsDoc, CDocument)
//{{AFX_MSG_MAP(CEmailsDoc)
// NOTE -- the ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////
// CEmailsDoc construction/destruction

CEmailsDoc::CEmailsDoc()
{
    db=NULL;
}

CEmailsDoc::~CEmailsDoc()
{
}

BOOL CEmailsDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
    delete db;
    db=new EMailDB;
    return TRUE;
}

////////////////////////
// CEmailsDoc serialization

void CEmailsDoc::Serialize(CArchive& ar)
{
    CObject *ob;
    if (ar.IsStoring())
    {
        ar<<db;
    }
    else
    {
        ar>>ob;
        db=(EMailDB *)ob;
        CEmailsView *v;
        POSITION pos;
        pos=GetFirstViewPosition();
    }
}

```

```

        v=(CEmailsView *)GetNextView(pos);
        v->m_current=0;
        v->UpdateView();
    }
}

////////////////////////////////////
// CEmailsDoc diagnostics

#ifdef _DEBUG
void CEmailsDoc::AssertValid() const
{
    CDocument::AssertValid();
}

void CEmailsDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}
#endif // _DEBUG

////////////////////////////////////
// CEmailsDoc commands

void CEmailsDoc::DeleteContents()
{
    delete db;
    CDocument::DeleteContents();
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Usually, the final argument to **IMPLEMENT_SERIAL** specifies the version number of the class. Each time you make a change to the class, you should update the version (or schema) number. That way, if your program tries to read an old version of the object, MFC will throw an exception. When you add the **VERSIONABLE_SCHEMA** flag to the version number, you are telling MFC that you don't want it to throw that exception. That implies that you'll take care of reading older versions of the file.

The next version of the program makes allowances for those of us who have more than one email address. This means that **EMailDB** holds more data and gets a new version number (see Listing 2.5). Since the class uses the **VERSIONABLE_SCHEMA** flag, you can still read files from the first version.

Listing 2.5 Excerpt from version 2 of the email program.

```
IMPLEMENT_SERIAL(EMailDB,CObject,VERSIONABLE_SCHEMA|2)
```

```
void EMailDB::Serialize(CArchive& ar)
{
    int i;
    CObject::Serialize(ar);
    if (ar.IsLoading())
    {
        int version=ar.GetObjectSchema();
        if (version>2||version<1)
        {
            AfxMessageBox("Unknown file message");
            return;
        }
    }
}
```

```

    }
    ar>>m_count;
    for (i=0;i<=m_count;i++)
    {
        ar>>name[i];
        ar>>email[i];
        if (version>1) ar>>email1[i];
    }
}
else
{
    ar<<m_count;
    for (i=0;i<=m_count;i++)
    {
        ar<<name[i];
        ar<<email[i];
        ar<<email1[i];
    }
}
}

```

To make this work, you must call **GetObjectSchema** on the **CArchive** object when you detect that the archive is in the loading state. The documentation states that you must make this call first, but I haven't found that to be true. However, for some odd reason, the code that implements **GetObjectSchema** sets the schema to -1 after you read it one time. That means you can only read it once per object serialization. Subsequent calls will return -1 .

Once you have the version number, you can neglect to read data that didn't exist in earlier versions. In the example, only version 2 objects read the secondary email address. It would probably be a good idea to remember the version number. Then, when you save an old version object, you could prompt the user to decide whether to save it in the old format or if you want to convert it to the new format.

Don't forget: Your document object—contrary to appearance—doesn't really do object serialization. That means that even if you add an **IMPLEMENT_SERIAL** macro to your document object, it won't store a version number (or any other class information). If you call **GetObjectSchema** from inside your document's **Serialize** function, it will return -1 (unless you've taken steps to properly serialize the document using **>>** or **WriteObject**).

Custom Serialization

What happens if you want to serialize objects to something other than a file? In theory, you could derive a class from **CArchive**, but that is rarely practical. At a minimum, you'd need to replace (or modify) **Read**, **Write**, **Flush**, and **FillBuffer**. To do this, you'd need to understand how **CArchive** buffers data (and hope it doesn't change later). Of course, you could copy the existing code and modify.

Another approach is to make a custom **CFile**-derived class. This is the approach

MFC takes to make archives that work with network sockets (see Chapter 9). This is also a good bit of work. You need to supply **GetBufferPtr**, **Read**, **Write**, and **Seek** methods. The **GetBufferPtr** is only for file objects that have an internal buffer. If that doesn't apply to you, you can just return 0 for this function.

Once you have your custom **CFile** object, you'll override **CDocument::GetFile**. This virtual function is responsible for creating a **CFile**, calling **Open** on it, and returning a pointer to it.

There is a way to cheat to make things somewhat easier (albeit less efficient). You could use a **CMemFile** instead of a **CFile**. The process would be about the same as using a custom **CFile**-derived object (see the previous three paragraphs). If MFC is opening the file for writing, pass an empty **CMemFile** to the **CArchive** constructor. Then MFC will write all the information to a memory block controlled by the **CMemFile** object.

You'll also want to override **OnOpenDocument** and **OnSaveDocument** so you can insert some code after the **Serialize** call in these functions. When the **Serialize** function returns, you can use **CMemFile::Detach** to retrieve a pointer to the memory. Then you could do anything you wanted with that data. For example, you might put it in a database or write it inside another file that contains differently formatted data.

To read the data back in, just retrieve it to a memory block and then pass this memory block and its size to the **CMemFile** constructor (be sure to pass 0 for the **nGrowBytes** parameter). Return the **CMemFile** from **GetFile**. Now, **Serialize** will read the objects back as though it used an ordinary file. As usual, you can start with a copy of the existing **OnOpenDocument** and **OnSaveDocument** functions.

If you'd rather avoid overriding **OnOpenDocument** and **OnSaveDocument**, you could derive a new class from **CMemFile**. The **Open** call would take care of setting up the file and the **Close** call can take care of transferring the data to its eventual destination.

You can find an example of this technique in Listing 2.6. This is the same familiar email program, but this time, the program encrypts the phone book file to prevent snoopers from reading your address book.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH

ITKNOWLEDGE

Brief Full

- [Advanced](#)
- [Search](#)
- [Search Tips](#)

BROWSE

BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

The key to this process is the **CCryptFile** class (shown in Listing 2.6). It is a special kind of **CMemFile** that provides custom **Open** and **Close** functions. If the **Open** function detects that you are reading the file, it reads the entire physical file into memory, decodes it (using an XOR algorithm), and attaches the memory to the **CMemFile**.

Listing 2.6 The encrypted file class.

```
#include "stdafx.h"
#include "cryptfile.h"

BOOL CCryptFile::Open( LPCTSTR lpszFileName, UINT nOpenFlags,
                      CFileException* pError)
{
    CFile raw;

    // You'd like to test for modeRead, but since it is 0, that doesn't work
    if ((nOpenFlags & (CFile::modeReadWrite|CFile::modeWrite))==0)
    {
        if (!raw.Open(lpszFileName,nOpenFlags,pError))
        {
            return FALSE;
        }
        unsigned n;
        n=raw.GetLength();
        unsigned char *p=(unsigned char *)malloc(n);
        raw.Read(p,n); // read entire file
        raw.Close();
        // decrypt
        for (unsigned i=0;i<n;i++)
            p[i]^=0xFF;
        Attach(p,n);
    }
    else
        basefile=lpszFileName;
    return TRUE;
}
```

```

void CCryptFile::Close( )
{
    CFileException pError;
    CFile raw;
    // if saving, encrypt data and write to real file
    if (!basefile.IsEmpty())
    {
        unsigned n=GetLength();
        if (!raw.Open(basefile,
            CFile::modeWrite|CFile::modeCreate|CFile::shareExclusive,
            &pError))
            throw &pError;
        unsigned char *p=Detach();
        // encrypt
        for (unsigned i=0;i<n;i++)
            p[i]^=0xFF;
        raw.Write(p,n);
        raw.Close();
        free(p);
    }
}

```

Writing is simpler. The code saves the file name (in the **basefile** member) and allows **CMemFile** to create the memory buffer as needed. Then, the **Close** function encodes the block of memory and writes it to the actual file. Here's how you'd use the encrypted class:

```

CFile* CEmailsDoc::GetFile(LPCTSTR lpszFileName, UINT nOpenFlags,
    CFileException * pError)
{
    CCryptFile *pFile=new CCryptFile;
    if (!pFile->Open(lpszFileName, nOpenFlags, pError))
    {
        delete pFile;
        pFile = NULL;
    }
    return pFile;
}

```

Of course, encryption is just one thing you could do with this technique. You could read and write the data to a database, for example. Using **CMemFile** isn't very efficient because you have to buffer the entire file in memory. The alternative is to subclass **CFile** and implement **Read**, **Write**, and other methods to map to your alternative storage (for example, encrypting or accessing the database). You'd still use **GetFile** in the same way with your new **CFile**-derived class.

Simple Customizations

Sometimes you just want to make a simple change to the file format. For example, I've always thought it was clever how some files start with a message describing the type of file followed by a Ctrl-Z. That way, if someone types the file from the DOS prompt, it displays an informative message and stops before it dumps out the data in the file.

Of course, you could use the custom **CFile** technique described earlier to do this, but there is an easier way. Inside the **Serialize** function, you can call the archive's **GetFile** member to retrieve a **CFile** pointer. Using this pointer, you can make minor modifications to the file format.

If you haven't used the archive yet, you can go ahead and call **GetFile**. However, if the archive has buffered data, you need to flush the archive before calling **GetFile**. You can do this by calling the archive's **Flush** function.

Listing 2.7 shows yet another version of the email list program. This time, the **Serialize** function uses **GetFile** to add a header and a trailer to the file. When the program is reading the archive back it, it skips the header (and, of course, ignores the header).

Listing 2.7 Excerpt from another email program.

```
void CEmailsDoc::Serialize(CArchive& ar)
{
    CObject *ob;
    if (ar.IsStoring())
    {
        ar.GetFile()->Write("File by AWC\r\n\x1a",14);
        ar<<db;
        ar.Flush(); // must flush before using CFile
        ar.GetFile()->Write("AWC-EOF\r\n",9);
    }
    else
    {
        ar.GetFile()->Seek(14,CFile::begin); // skip bytes
        ar>>ob;
        db=(EMailDB *)ob;
        CEmailsView *v;
        POSITION pos;
        pos=GetFirstViewPosition();
        v=(CEmailsView *)GetNextView(pos);
        v->m_current=0;
        v->UpdateView();
    }
}
```

Portability Issues

The **CArchive** class makes a half-hearted attempt to make your file formats portable. Why half-hearted? Consider this fragment of code (where **ar** is an archive):

```
int x=10;
ar<<x;
```

MFC won't allow you to compile this code. The problem is that an **int** is 16-bits wide on some platforms and 32-bits wide on others. So **CArchive** doesn't have a function to read and write integers. In general, the only simple data types that you can use are the portable ones (see Table 2.5). Of course, you can cast to make your selection, if you like:

```
int x=10;
ar<<(DWORD)x;
```

Table 2.5 Simple serializable types.

Data type
BYTE
WORD
LONG
DWORD
float
double

So this is a good thing, right? Well, consider a **CRect** (the MFC class that contains a rectangle). You can serialize them just like most other objects. However, they aren't the same size on 16-bit and 32-bit platforms. The same goes for **CPoint** and **CSize**. Seems like if the archive is going to let you use any non-portable types, it might as well let you use all of them. Otherwise, it is just an inconvenience.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Summary

Archives are one of those things you don't need to worry about often. When you need to do something special with them, you find that you don't know much about them. Although archives are a good idea, if you have a custom file format, you might as well subvert serialization.

If you are flexible with your file format, making serializable objects is a good idea. You get a robust file format and versioning support. Of course, your file format will be inscrutable to non-MFC programs, but that's rarely a problem.

Practical Guide To Serialization

- Making A Class Serializable
- Customizing File Prompting
- Using Existing Or Custom File Code
- Creating Archives On Nonstandard Streams
- Reading Old File Versions

MFC relies on serialization to make objects persistent and to load and save documents. For simple programs, you can get away with minimal attention to serialization, but many practical programs require more. For example, many programs must conform to existing file formats. Others have to store data inside another file or some other storage medium. The material in this chapter will help you mold serialization to fit your needs.

Making A Class Serializable

To make a class serializable, you must:

1. Derive the class from **CObject** or a class derived from **CObject**.
2. Add the **DECLARE_SERIAL** macro to the header. If the class already has **DECLARE_DYNCREATE** or **DECLARE_DYNAMIC** defined, use **DECLARE_SERIAL** *in place of* the other macro.
3. Add the **IMPLEMENT_SERIAL** macro to the class definition. If the class already has **IMPLEMENT_DYNCREATE** or **IMPLEMENT_DYNAMIC** defined, use **IMPLEMENT_SERIAL** *in place of* the other macro.
4. Implement the **Serialize** function to read and write the constituent data and objects to an archive.

Once you've completed these steps, you can use the << and >> operators to read and write the object to an archive.

Customizing File Prompting

Many programmers like to provide a custom file dialog in their applications. Perhaps you need a file preview, or you want to customize the list of files, or even the starting directory. It would be a shame to have to rewrite the entire file open and save logic just to customize the file open dialog.

There are two simple ways to add a custom file prompt. One way is to override **DoPromptFileName** in the application object. A simple strategy is to copy the **DoPromptFileName** routine from the **DOCMGR.CPP** file in the MFC source code and change it to work with your application object and custom dialog. Because this function isn't virtual, you also need to replace the functions that call it (**OnFileOpen** in the application object and **DoSave** in the document). You'll also need to copy the static function **AppendFilterSuffix**. You can find a complete example of this in Listing 2.1.

An alternate method is to override **OnFileOpen** in the application object and the document's **OnFileSave** and **OnFileSaveAs** functions. You can then get the file name any way you like and pass it back into the normal MFC chain of events (see Tables 2.2 and 2.3). This method is well suited to cases where you don't want to select a file. For example, if your program might display a fixed list of 10 documents that the user can select, this might be a good idea. The first method is best when you want to use the file filters in the document templates to select files.

Using Existing Or Custom File Code

If you have existing code to read and write files, you may not be able to adapt easily to the MFC archive format—especially if you can't change the format of the file. That's not a problem. Although MFC likes to use archives and serialization, it doesn't require it.

The easiest way to use custom code to open and save files is to override **CDocument::OnSaveDocument** and **CDocument::OnOpenDocument**.

MFC calls these functions with a file name. You can do anything you see fit in these routines.

Of course, the default versions open the file, bind it to a **CArchive** object, and call **Serialize**. That means you don't want to call the base class if you are trying to prevent this behavior.

Creating Archives On Nonstandard Streams

Sometimes, you may want to create an archive that actually stores and loads data to a stream that is not a file. You might want an archive that works against a database, for example. Even if the stream is a file, you might want to encrypt it, or add things to the file to fit some particular format.

There are at least four ways you can alter the archive format:

- Get the **CFile** pointer from the archive using **GetFile** and use the file object to directly manipulate the underlying file. If the archive is in use, flush it before calling **GetFile**. This method is best if you are adding information to the beginning or the end of the file.
- Derive a new class from **CFile** and provide a suitable implementation to read and write bytes to the medium of your choice. The document object has an overrideable function named **GetFile** that you can use to return a pointer to your file object. This is a very general-purpose solution, but it is a bit of work.
- Derive a new class from **CMemFile**. When you open and close the memory based file, you can manipulate the memory buffer and then read or write the buffer to the medium you want to use. Again, you return a pointer to an instance of your new class by overriding the document's **GetFile** function. This is a simple technique, but not very efficient because it requires that you hold the entire file in memory at once.
- Create a new class based on **CArchive** and substitute it for the regular object. However, this is relatively difficult since **CArchive** has logic to save and load objects that you don't want to disturb.

You can find examples of retrieving the **CFile** object and supplying a **CMemFile**-derived object in Listing 2.6.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Reading Old File Versions

When you use **IMPLEMENT_SERIAL** in a class, the final argument to the macro is a version (or schema) number. MFC writes this schema number out with the class information for the object.

Ordinarily, if MFC reads an object from an archive and its schema number doesn't match the schema number you specified, MFC throws an exception and aborts the loading process.

However, you might want to actually adapt your serialization code based on the object version. In that case, use the logical OR operator (|) to add the constant **VERSIONABLE_SCHEMA** flag to your version number. Then, you can call **GetObjectSchema** during loading to find out the version for the object.

Your code can then modify the logic based on the version:

```
void CXXXClass::Serialize(CArchive& ar)
{
    int i;
    CObject::Serialize(ar);
    if (ar.IsLoading())
    {
        int version=ar.GetObjectSchema();
        if (version>2||version<1)
        {
            AfxMessageBox("Unknown file format");
            return;
        }
    }
}
```

```

    }
    ar>>m_count;
    for (i=0;i<=m_count;i++)
    {
        ar>>v1_data;;
        ar>>v1_moredata;;
        if (version>1) ar>>v2_data;
    }
}
.
.
.

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Chapter 3 Printing

App Wizard programs do printing automatically, right? Sure, any document/view program that can draw to the screen can draw to the printer, too. But often the results aren't what you want. Scaling differences, pagination, and print-time features (like headers) all complicate the problem. Print preview is another area where you often want customizations, but the framework hides the details. You'll see how to add editing during print preview in this chapter.

Programmer's Notes...

If you read the last chapter, it won't surprise you to know that I'm a big *Star Trek* fan. Judging by some of the people I see on TV, though, I'm probably not the biggest *Star Trek* fan. My car doesn't look like a tribble, and I don't own any Starfleet uniforms. On the other hand, I do have copies of the *Star Trek Technical Manuals* and blueprints signed by Leonard Nimoy. I have even had the privilege to talk semi-privately with Gene Roddenberry right after the first *Star Trek* movie. So, while I'm no zealot, I'm a pretty respectable fan.

I always try to get my kids interested in *Star Trek*. You can certainly have worse role models than Kirk and company. I suppose even the new guys have some merit. Of course, my kids are studiously uninterested in anything I like, so it is a rare occasion that I am not watching *Trek* alone.

For Columbus Day, we decided to take a short trip—sort of an end of the summer mini-vacation. After spending the day at Sea World, we decided, on the spur of the moment, to make the short drive to cross the Mexican border so that we could say we’d been to Mexico. I told my youngest son this was his chance to boldly go where no man had gone before (well, at least where he had not gone before). Of course, he rolled his eyes at the *Trek* reference.

Well, I found out just how unlike the bold Captain Kirk I really am. When we arrived at the border, we realized we had no identification for our 7-year-old son, Patrick. Think about it. Do you carry ID for your 7-year-old? If we had planned the trip, we’d have brought his birth certificate. Too bad we hadn’t thought about making the trip until we were there. We seriously debated on crossing the border—after all, Patrick might not get back in the country!

After talking with the U.S. Border Patrol (nice folks), we decided to be brave and chance it. I admit I felt an odd thrill in crossing the border knowing we were boldly taking a risk. Sure, we knew we could eventually prove Patrick’s citizenship, but we could be stuck in Mexico for days or even weeks if things didn’t go well.

In the end, it played out like a *Next Generation* episode where everything sort of resolves itself as the show runs out of time. Patrick bought a toy drum at the *mercado* (market). When we returned, the Border Patrol asked us if we were citizens. We said yes, and they let us back in the U.S. I didn’t even show my driver’s license. Quite anti-climatic, eh?

I think programming is a lot like this. It is natural to be apprehensive when venturing into new territory (a new operating system, language, or technique). Then again, it is kind of thrilling to figure out something new.

When I talk to MFC programmers, I find many of them are uneasy about printing. Sure, everyone can use the “free” printing. But how can you print correctly scaled, multi-page documents? How about headers, footers, or page numbers?

To begin with, I’ll show you a simple tick-tack-toe program (or naughts and crosses, if you prefer). By itself, the program is unremarkable (see Figure 3.1), but I’ll make it print appropriately. Later, I’ll show you how to take control of print preview and customize it. Before we play games, though, let’s talk about MFC printing support.



Figure 3.1 Example program.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

MFC Printing—The Big Lie?

MFC touts that you can draw and print with the same code. But is this really practical? Sometimes. More often, you have to write special code to handle printing. The good news is that the code you have to write is much simpler than the corresponding Windows printing code.

When the user selects the print command from the file menu, the active view receives an **OnPreparePrinting** call. This function has a single argument, a pointer to a **CPrintInfo** structure (see Table 3.1). You can set the members of this structure to describe the print job. If you know at this point the number of pages you will print, you can specify it at this time. When you pass the structure to **DoPreparePrinting**, it affects the values in the print dialog. The default code that App Wizard creates assumes you don't know anything about the printing job and simply passes the structure unchanged.

Table 3.1 CPrintInfo details.

Member	Type	Description
m_bDocObject	variable	Indicates whether the document being printed is a DocObject
m_dwFlags	variable	Specifies DocObject printing operations
m_nOffsetPage	variable	Specifies offset of a particular DocObject's first page in a combined DocObject print job

m_pPD	variable	Pointer to the corresponding CPrintDialog object
m_bDirect	variable	Indicates whether the document is being printed without displaying the Print dialog box
m_bPreview	variable	TRUE if print preview mode
m_bContinuePrinting	variable	Determines if current page should print (see text)
m_nCurPage	variable	Current page number
m_nNumPreviewPages	variable	Number of pages in preview mode (1 or 2)
m_lpUserData	variable	Pointer to a user-created structure
m_rectDraw	variable	Rectangle that defines the current usable page area (not valid until printing actually begins)
m_strPageDesc	variable	Contains a format string for page-number display
SetMinPage	function	Sets the number of the first page of the document
SetMaxPage	function	Sets the number of the last page of the document
GetMinPage	function	Returns the number of the first page of the document
GetMaxPage	function	Returns the number of the last page of the document
GetOffsetPage	function	Returns the number of the pages preceding the first page of a DocObject item being printed in a combined DocObject print job
GetFromPage	function	Returns the number of the first page being printed
GetToPage	function	Returns the number of the last page being printed

To start a printing job, the framework calls **OnBeginPrinting**. This function takes two arguments: a **CDC** pointer and the **CPrintInfo** pointer. You can use the **CDC** to determine the characteristics of the printer (just call **GetDeviceCaps**). This is your last meaningful chance to set the number of pages by calling the **CPrintInfo::SetMaxPage** function. If the number of pages you print depends on the printer characteristics, you should set the page count here. This allows the framework to call subsequent functions once for each page. If you still can't figure out how many pages you have, you'll have to manually set the **CPrintInfo** structure's **m_bContinuePrinting** member during the **OnPrepareDC** member (**OnPrepareDC** executes during printing, as you'll see shortly).

The **OnBeginPrinting** routine is a good place to create resources that you will

need for the entire print job. For example, it is not unusual to create printer fonts here.

As MFC prints a page, it makes the following calls:

1. OnPrepareDC
2. OnPrint
3. OnDraw

These three calls occur for each page. Notice that **OnPrepareDC** and **OnDraw** also occur for regular screen drawing. If you need to differentiate between the two cases, you can call the **CDC::IsPrinting**.

There are several reasons you might override **OnPrepareDC**. First, you can alter the device context. You might do this to change the viewport origin so that your drawing code will draw the correct page. You can also change the device context's mapping mode so that the printed output will scale differently from the screen drawing.

You can also decide if you want to continue printing or not. This is useful if you have no way to ascertain the number of pages to print beforehand. To do this, you need to make sure the **pInfo** argument is not **NULL**. Remember that **OnPrepareDC** occurs during drawing and printing. If **pInfo** is valid, you can determine if you want to print the page (examine the **m_nCurPage** member). If you want to print the page, set the **m_nContinuePrinting** to **TRUE** after calling the base class **OnPrepareDC**. The base class will set the flag to **FALSE**, so make sure to alter it after calling the base class. You should call the base class, by the way. Many special views use **OnPrepareDC** to do special work.

Tip: General Printing

Simple programs usually don't worry about all of these details. That's because **OnDraw**, the call that draws the view, automatically takes care of drawing to the printer. However, you need to understand these details if you want to change scaling, add headers and footers, or otherwise customize the printing process.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

If you want the code in **OnDraw** to handle screen drawing and printing, you don't need to override **OnPrint**. The default version calls **OnDraw** for you. However, if you want special printing code that doesn't really relate to screen drawing, you can do it in **OnPrint**. For example, suppose you have a program that displays a network diagram on the screen but prints a table of node connections when printing. You can draw the diagram in **OnDraw** and print the table in **OnPrint**. Another useful thing you can do in **OnPrint** is draw items that are only for the printed output. For example, it is common to draw headers and footers here. Then you have to change the viewport origin to prevent the drawing code from overwriting the header. You might also have to set the clipping region to protect your footer. When you are ready, you can call **OnDraw** to complete the page.

Of course, an alternate approach to headers and footers would be to put the code to draw them in **OnDraw**. You could only execute that code when **IsPrinting** returns **TRUE**. This is mainly a matter of personal preference.

The **OnDraw** function is what draws on the screen. If you properly manipulate the device context in earlier calls, you never need to know if the output is printing or going to the screen. However, you can use **IsPrinting** to differentiate the two cases if you find it necessary.

The framework calls the previous three calls repeatedly for each page. It knows how many pages should print based on the settings in the **CPrintInfo** structure (or based on your manipulation of the **m_bContinuePrinting** flag). At the end of the print job, the framework calls **OnEndPrinting**. This allows you to free printer resources that you allocated during **OnBeginPrinting**.

That's the printing architecture in a nutshell. You can find the entire process

outlined in Table 3.2. The best part of this is that the framework also handles print preview via the same mechanism. If your printing works, you should get print preview for free.

Table 3.2 The printing process.

MFC calls...	Why?	Override when...
CView::OnFilePrint	Menu selection	You want to handle everything
CView::OnPreparePrinting	To start the process	To insert information into the print dialog box
CView::DoPreparePrinting	To show the print dialog box	To show a custom print dialog
CView::OnBeginPrinting	To allocate resources	You want to allocate GDI resources once
CView::OnPrepareDC	To set up the DC	You want to allocate GDI resources or meddle with the device context
CView::OnPrint	To perform printing	You want to print a different view than you display or you want to print additional data (e.g., headers)
CView::OnDraw	To update the screen	You want to draw (nearly always)
CView::OnEndPrinting	To clean up	You need to release GDI resources

The Dilemma

Ideally, you could use exactly the same code for drawing as you use for printing. In practice, there are a few problems with doing this. The primary problem is scaling. The screen resolution and the printer resolution are usually quite different. A line that is 100 pixels long will be fairly long on the screen, but it will be quite short on a 300 or 600 dpi (dots per inch) printer. If you draw something of reasonable size on the screen, it will print out much smaller unless you do something special.

There are two ways around this problem. One way is to use a mapping mode. To do this, call **CDC::SetMapMode**, which allows you to draw in logical units. For example, if you select the **MM_LOENGLISH** mode, you can specify drawing dimensions in units of 1/10 inch (see Table 3.3). Of course, the screen size won't be exact because the driver doesn't exactly know the screen dpi. Also, most screen drivers purposely inflate dimensions so that small elements (10 point fonts, for example) will be easily visible. Printers, however, have a well-defined number of dots per inch and the dimensions will translate exactly.

Table 3.3 Mapping modes.

Mode	Description
MM_TEXT	1 logical unit == 1 pixel
MM_HIENGLISH	1 logical unit == .001 inch
MM_LOENGLISH	1 logical unit == .01 inch
MM_HIMETRIC	1 logical unit == .01 millimeters
MM_LOMETRIC	1 logical unit == .1 millimeters
MM_TWIPS	1 logical unit == 1/1440 inch (1/20 of a point; useful for typesetting)
MM_ISOTROPIC	Uses SetWindowExt and SetViewportExt to scale the x and y axis while preserving the aspect ratio; a circle in logical coordinates will form a circle on the screen
MM_ANISOTROPIC	Uses SetWindowExt and SetViewportExt to scale the x and y axis arbitrarily; a circle in logical coordinates may form an ellipse on the screen unless you calculate scaling factors that preserve the aspect ratio manually

Tip: Using Mapping Modes

Because they simplify printing, it is always a good idea to use a mapping mode other than **MM_TEXT**, if you can. Just remember that modes other than **MM_TEXT** have their y-axis flipped. In other words, use negative numbers for the y values (unless you moved the origin yourself). Of course, sometimes it isn't convenient to stop using pixels. The tick-tack-toe program is a good example. Because the board changes size depending on the window size, it isn't easy to think of the board in terms of inches or millimeters.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

The default mode is **MM_TEXT**, which maps 1 logical unit to 1 device pixel. In **MM_TEXT** mode, the horizontal direction is the X axis and the values increase from right to left. The vertical axis (Y) increases from top to bottom. Other modes (such as **MM_LOENGLISH**) invert the Y axis. That is, Y values start at the top of the device and increase as you go up. That means that all of the useful values are negative! The X axis remains the same. You can change this with **CDC::SetViewportOrg**, but this is the default behavior.

Although it's attractive to use the same dimensions for screen and printer, there are many cases in which this introduces as many problems as it solves. Often, you don't want to work in inches or centimeters when you are drawing things on the screen. Mouse coordinates are always in pixels, so that's not convenient either. Many times, you'll want a drawing to fill up the screen and the printer page, regardless of their sizes. In this case, the logical mapping modes aren't very useful.

When you want to stick with pixels, you'll probably want to scale the printer DC so that the printer and the screen units are similar. This is a perfect job for **OnPrepareDC** or **OnPrint**. You could also put the scaling code in **OnDraw** and use the **CDC::IsPrinting** call to skip the scaling if you are not printing. It just depends on how much you want your **OnDraw** to know about printing.

A Complete Printing Example

Consider the program in Listing 3.1. This is a tick-tack-toe program that allows you to print the playing board. All by itself, it is an unremarkable program. The only flashy features I used were the splash screen and the status bar clock components from Component Gallery (try them under the Insert Component menu item).

Listing 3.1 Printing view.

```
// mfctttView.cpp
```

```

//

#include "stdafx.h"
#include "mfcttt.h"

#include "mfctttDoc.h"
#include "mfctttView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CMfctttView

IMPLEMENT_DYNCREATE(CMfctttView, CView)

BEGIN_MESSAGE_MAP(CMfctttView, CView)
   //{{AFX_MSG_MAP(CMfctttView)
    ON_WM_LBUTTONDOWN()
    ON_COMMAND(ID_EDIT_UNDO, OnEditUndo)
    ON_UPDATE_COMMAND_UI(ID_EDIT_UNDO, OnUpdateEditUndo)
   //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()

////////////////////////////////////
// CMfctttView construction/destruction

CMfctttView::CMfctttView()
{
    // TODO: add construction code here
}
CMfctttView::~CMfctttView()
{
}

BOOL CMfctttView::PreCreateWindow(CREATESTRUCT& cs)
{
    return CView::PreCreateWindow(cs);
}

////////////////////////////////////
// CMfctttView drawing

// Offset from edge of tick-tack-toe cell
#define OFFSET 15

```



```

        pDC->SelectStockObject(HOLLOW_BRUSH);
        pDC->SelectObject(&op);
        pDC->Ellipse(pt.x+OFFSET,pt.y+OFFSET,
                    pt1.x-OFFSET,pt1.y-OFFSET);
        break;
    }
}
pDC->SelectObject(old);
}

////////////////////////////////////
// CMfctttView printing

BOOL CMfctttView::OnPreparePrinting(CPrintInfo* pInfo)
{
    pInfo->SetMaxPage(2);
    return DoPreparePrinting(pInfo);
}

// This OnPrint allows you to use MM_TEXT in your View
// and the print out scales appropriately
void CMfctttView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
    CDC *dc=GetDC();
    CString s;
    int x=dc->GetDeviceCaps(LOGPIXELSX);
    int y=dc->GetDeviceCaps(LOGPIXELSY);
    int x1=pDC->GetDeviceCaps(LOGPIXELSX);
    int y1=pDC->GetDeviceCaps(LOGPIXELSY);
    // print header when printing only
    s.Format("Tick-tack-toe game: %s",
        GetDocument()->GetTitle());
    pDC->TextOut(0,0,s);
    pDC->MoveTo(0,75);
    pDC->LineTo(pInfo->m_rectDraw.Width(),75);
    if (pInfo->m_nCurPage==2)
    {
        CMfctttDoc *doc=GetDocument();
        s.Format("Games I Won=%d, Games I Lost=%d"
            " Draw Games=%d",doc->wins,doc->loss,
            doc->draw);
        pDC->TextOut(0,100,s);
        return;
    }
    // Alter mapping mode so that pixels scale correctly
    pDC->SetMapMode(MM_ISOTROPIC);
    pDC->SetWindowExt(x,y);
    pDC->SetViewportExt(x1,y1);
    // top margin of 100 units
    pDC->SetViewportOrg(0,100);
    CView::OnPrint(pDC, pInfo);
}

```

```

void CMfctttView::OnBeginPrinting(CDC* /*pDC*/,
    CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CMfctttView::OnEndPrinting(CDC* /*pDC*/,
    CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}

////////////////////////////////////
// CMfctttView diagnostics

#ifdef _DEBUG
void CMfctttView::AssertValid() const
{
    CView::AssertValid();
}

void CMfctttView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}
CMfctttDoc* CMfctttView::GetDocument()
{
    ASSERT(m_pDocument->IsKindOf(
        RUNTIME_CLASS(CMfctttDoc)));
    return (CMfctttDoc*)m_pDocument;
}
#endif // _DEBUG

////////////////////////////////////
// CMfctttView message handlers

// Convert mouse coordinate to grid coordinate
void CMfctttView::MouseToGrid(CPoint &pt)
{
    CRect r;
    GetClientRect(&r);
    pt.x/=r.right/3;
    pt.y/=r.bottom/3;
}

// Convert grid coordinate to mouse coordinate
void CMfctttView::GridToMouse(CPoint &pt)
{
    CRect r;
    GetClientRect(&r);
    pt.x*=r.right/3;

```

```

        pt.y*=r.bottom/3;
    }

void CMfcttttView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CMfcttttDoc *doc=GetDocument();
    // Convert to grid position
    MouseToGrid(point);
    // if not empty, beep and forget it
    if (doc->GetBoardState(point.x,point.y)!=EMPTY)
    {
        MessageBeep(MB_ICONEXCLAMATION);
        return;
    }
    // Empty -- put an X or an O here depending on the turn #
    // Note: this was here for debugging,
    // now the computer always plays O, so the
    // move is always an X
    doc->SetBoardState(point.x,point.y,
        (doc->turn&1)?O:X);
    doc->turn++;
    doc->UpdateAllViews(NULL);
    doc->Play(); // Do our move
}

void CMfcttttView::OnEditUndo()
{
    CMfcttttDoc *doc=GetDocument();
    doc->undo(TRUE);
}

void CMfcttttView::OnUpdateEditUndo(CCmdUI* pCmdUI)
{
    CMfcttttDoc *doc=GetDocument();
    pCmdUI->Enable(doc->undo(FALSE));
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

The tricky part of the program is the printing. In Listing 3.1, you can omit **OnPrint** (or make it delegate directly to the base class). If you do that, you'll see that everything works fine, even the printing and print preview. However, the printed board will be very small compared to the image on screen.

To fix the problem, there are several options. First, you could draw using a mode other than **MM_TEXT**. However, that introduces another problem. The board on the screen should shrink and expand as the window size changes. This is easier to do when using **MM_TEXT** mode. A better answer is to scale the printer so that a line that goes all the way across the window also goes all the way across the printed page. Finally, you can arrange it so that a logical inch on the screen is equal to a physical inch on the printer.

Here is a simplified version of the code that accomplishes this in the **OnPrint** routine (see Listing 3.1 for the complete code):

```
void CMfctttView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
    CDC *dc=GetDC();
    int x=dc->GetDeviceCaps(LOGPIXELSX);
    int y=dc->GetDeviceCaps(LOGPIXELSY);
    int x1=pDC->GetDeviceCaps(LOGPIXELSX);
    int y1=pDC->GetDeviceCaps(LOGPIXELSY);
    // Alter mapping mode
    pDC->SetMapMode(MM_ISOTROPIC);
    pDC->SetWindowExt(x,y);
    pDC->SetViewportExt(x1,y1);
    CView::OnPrint(pDC, pInfo);
}
```

}

The first line obtains a DC that refers to the view (the window). The DC that the call receives (**pDC**) is for the printer. The subsequent lines determine the number of pixels per inch on each device in each direction. After that, the code sets the **MM_ISOTROPIC** mapping mode and sets the extents so that an inch on the screen is approximately equal to an inch on the printer. By the way, after you set the **MM_ISOTROPIC** mode, it is important that you set the window extent first and then the viewport extent. After the code above executes, drawing a line *x* units across on the printer DC will result in a line *x1* physical units long.

There is one quirk you need to know about when using this (or any similar scheme). You can't use the default font for a device context after you have changed the mapping mode in this way. The code that draws needs to create fonts after the mapping mode change. Otherwise, the results will not be pleasant.

The actual code in Listing 3.1 also prints a header (some text and a line; see Figure 3.2). The code calls **SetViewportOrg** so that the **OnDraw** routine will begin printing 100 units from the top to make room for the header.

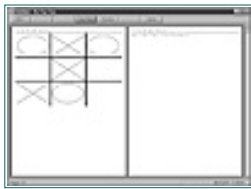


Figure 3.2 Print preview.

Because the MFC print architecture is symmetrical between printing a page and print preview, the same code takes care of both. This leaves only the pagination to the imagination. If you know the maximum number of pages while **OnPreparePrinting** is running, you can supply it there. Otherwise, you could set it during **OnBeginPrinting**. Finally, you could do runtime pagination by examining the page number in the **CPrintInfo** structure. If you want to print the page, set the **CPrintInfo**'s **m_bContinuePrinting** flag to **TRUE**. The program prints game statistics on page 2 using the first method. Notice that **OnPrint** entirely handles the second page printing.

Customizing Print Preview

When I was a kid, whittling fascinated me. I suspect that kids today don't whittle (not that they don't have the knives; just watch the evening news). Don't get me wrong—I was never any good at whittling, but I wanted to be.

No matter what I started out whittling, it generally turned out to be a toothpick in the end. I did make many fine toothpicks, however. I had the same problem with shop in high school. I'd have failed shop if not for the section on electricity. I did fairly well in plastics, too. I started making a cross, but I ended up turning in a number seven. Luckily, the instructor didn't ask us what we were making until we turned it in to him.

These experiences taught me two things:

- I should stay away from tools and knives

- It is often easier to reduce things than it is to add to them

This last axiom is certainly true of print preview. It isn't very hard to prevent MFC's print preview from doing things you don't want it to do. It is a bit more work to make print preview do new things. I'll show you how to do both.

To illustrate customized print preview, I wrote a very simple connect the dots program (see Figure 3.3). When you left-click the mouse, the program draws a line from the last point you picked to where the mouse is. When you right-click the mouse, the program doesn't draw a line, but it does change the current position to draw from. Not real rocket science, but it will serve as an example for printing.



Figure 3.3 Customizing the Preview tool bar.

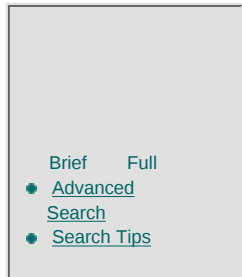
When you write a simple program like this one, the first thing you'll notice is that the print preview does too good a job for you. For example, MFC provides you with multiple pages and even a way to show two pages at a time. That's great, but this program only shows one page.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97



Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Stripping Down Print Preview

Stripping down print preview is relatively easy. All you really need to do is take away the tool bar buttons you don't want. Sure, the code is still there inside MFC, but so what? If the user can't run the code, it's as good as gone.

The trick is to take control of the **OnFilePrintPreview** handler. Normally, App Wizard adds a handler for this menu item in the view's message map. However, it places the message map entry outside of Class Wizard's comments and uses the base class' handler. Therefore, the code isn't in your source file and you can't see the entry with Class Wizard.

The first step, then, is to hoist the **ON_COMMAND** macro that contains the **OnFilePrintPreview** call into the ordinary, Class Wizard-managed area of the message map. Delete the scope override operator that names the base class (often **CView**) and provide a function named **OnFilePrintPreview** in your H and CPP files. When you are done, your message map should look like this:

```
BEGIN_MESSAGE_MAP(CConndotView, CView)
   //{{AFX_MSG_MAP(CConndotView)
    ON_WM_LBUTTONDOWN()
    ON_WM_RBUTTONDOWN()
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, OnFilePrintPreview)
   //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
END_MESSAGE_MAP()
```

If you look at MFC's original version of the function (in VIEWPREV.CPP), it looks like this:

```
void CView::OnFilePrintPreview()
{
    // In derived classes, implement special window handling here
    // Be sure to Unhook Frame Window close if hooked.

    // NOTE FROM AL: In the next comment, frame means stack frame,
    // not frame window!
    // Must not create this on the frame. Must outlive this function
    CPrintPreviewState* pState = new CPrintPreviewState;
```

```

        // DoPrintPreview's return value does not necessarily indicate that
        // print preview succeeded or failed, but rather what actions are
        // necessary at this point.
        // If DoPrintPreview returns TRUE, it means that OnEndPrintPreview
        // will be (or has already been) called and the pState structure
        // will be/has been deleted.
        // If DoPrintPreview returns FALSE, it means that OnEndPrintPreview
        // WILL NOT be called and that cleanup, including deleting pState,
        // must be done here.

        if (!DoPrintPreview(AFX_IDD_PREVIEW_TOOLBAR, this,

            RUNTIME_CLASS(CPreviewView), pState))
        {
            // In derived classes, reverse special window handling here
            for
            // Preview failure case.

            TRACE0("Error: DoPrintPreview failed.\n");
            AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
            // preview failed to initialize, delete State now.
            delete pState;
        }
    }
}

```

It should be obvious that the real work here is occurring in **DoPrintPreview**. If you knock out the comments and debug tracing, there are only four lines of real code. This call to **DoPrintPreview** is the key to customizing print preview.

Let's look at the four arguments to this call. The first one is a resource ID that contains the preview tool bar. Changing that will allow us to change the tool bar. It's that simple. You could draw your own tool bar, but it's easier to rip off the existing one (in AFXPRINT.RC, which, oddly enough, is in the MFC\INCLUDE directory). Then you can simply change the tool bar to suit your tastes. Don't forget to change the ID, and to use the ID in your new version of **OnFilePrintPreview**.

A Custom Print Preview Example

The code that does this is shown in Listing 3.2 (you can see the program's modified preview screen in Figure 3.3). The only tricky part about moving the **OnFilePrintPreview** code is that it requires a reference to **CPreviewView**. This is the class MFC uses to represent the preview window. However, the only header that defines it is AFXPRIV.H. You'll need to include it in your view to get the code to compile.

Tip: Using AFXPRIV.H

In general, the declarations in AFXPRIV.H are subject to change between versions of MFC. However, Microsoft has been good about not changing things that break too many people's code. In fact, some things that used to be in AFXPRIV.H (like conversion macros) are now officially documented and in another header file because so many people used them.

The bottom line? If you want professional-looking programs, there are times you are going to have to take some risks and use parts of MFC that aren't official. When the next version of MFC comes out, you may have to make changes, but that's the price of living on the edge.

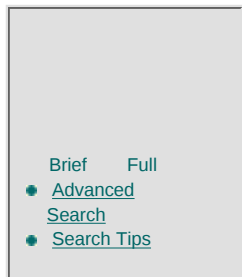
Here's the official Microsoft comment from the AFXPRIV.H file:

"This header file contains useful classes that are documented only in the MFC Technical Notes. These classes may change from version to version, so be prepared to change your code accordingly if you utilize this header. In the future, commonly used portions of this header may be moved and officially documented."

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97



Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

The **OnDraw** function in Listing 3.2 does something special for printing. When it determines you are printing (using the **IsPrinting** function of **CDC**), it changes the mapping mode so that the printed output will more closely match the screen. To do this, the code retrieves a device context for the actual window and sets an isotropic mapping between the screen pixels and the printer pixels.

Listing 3.2 A custom Print Preview tool bar.

```
// conndotView.cpp : implementation of the CConndotView class
//
```

```
#include "stdafx.h"
#include "conndot.h"
#include <afxpriv.h> // include preview window
```

```
#include "conndotDoc.h"
#include "conndotView.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
////////////////////////////////////
// CConndotView
```

```
IMPLEMENT_DYNCREATE(CConndotView, CView)
```

```
BEGIN_MESSAGE_MAP(CConndotView, CView)
   //{{AFX_MSG_MAP(CConndotView)
    ON_WM_LBUTTONDOWN()
    ON_WM_RBUTTONDOWN()
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, OnFilePrintPreview)
   //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
```

```

END_MESSAGE_MAP()

////////////////////////////////////
// CConndotView construction/destruction

CConndotView::CConndotView()
{
    // TODO: add construction code here
}

CConndotView::~CConndotView()
{
}

BOOL CConndotView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CView::PreCreateWindow(cs);
}

////////////////////////////////////
// CConndotView drawing

void CConndotView::OnDraw(CDC* pDC)
{
    CConndotDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    CPoint pt;
    if (pDC->IsPrinting())
    {
        CDC *vdc=GetDC();
        int n;
        n=vdc->GetDeviceCaps(LOGPIXELSX);
        pDC->SetMapMode(MM_ISOTROPIC);
        pDC->SetWindowExt(n,n);
        n=pDC->GetDeviceCaps(LOGPIXELSX);
        pDC->SetViewportExt(n,n);
    }
    pDC->MoveTo(0,0);
    for (int i=0;i<pDoc->points.GetSize();i++)
    {
        pt=pDoc->points[i];
        if (pt.x<0)
        {
            pt.x=-pt.x;
            pt.y=-pt.y;
            pDC->MoveTo(pt);
        }
        else
            pDC->LineTo(pt);
    }
}

////////////////////////////////////
// CConndotView printing

BOOL CConndotView::OnPreparePrinting(CPrintInfo* pInfo)

```

```

{
    // default preparation
    return DoPreparePrinting(pInfo);
}

void CConndotView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CConndotView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}

////////////////////////////////////
// CConndotView diagnostics

#ifdef _DEBUG
void CConndotView::AssertValid() const
{
    CView::AssertValid();
}

void CConndotView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CConndotDoc* CConndotView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CConndotDoc)));
    return (CConndotDoc*)m_pDocument;
}
#endif // _DEBUG

////////////////////////////////////
// CConndotView message handlers

void CConndotView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CConndotDoc *doc=GetDocument();
    // Draw
    doc->points.Add(point);
    doc->UpdateAllViews(NULL);
    CView::OnLButtonDown(nFlags, point);
}

void CConndotView::OnRButtonDown(UINT nFlags, CPoint point)
{
    // move current point only
    CPoint mover=point;
    mover.x=-mover.x;
    mover.y=-mover.y;
    GetDocument()->points.Add(mover);
    CView::OnRButtonDown(nFlags, point);
}

void CConndotView::OnFilePrintPreview()
{
    CPrintPreviewState* pState = new CPrintPreviewState;

```

```

        if (!DoPrintPreview(IDD_PREVIEW_TOOLBAR, this,
                            RUNTIME_CLASS(CPreviewView), pState))
        {
            // In derived classes, reverse special window handling here
            for
            // Preview failure case

            TRACE0("Error: DoPrintPreview failed.\n");
            AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
            delete pState;          // preview failed to initialize, delete
            State
        }
    }
}

```

Advanced Customizations

If you looked carefully at the other three arguments to **DoPrintPreview**, you probably figured out what they do. The second argument is the view to mimic (usually **this**). The next argument is the runtime class of the preview window. This is usually **CPreviewView**, but why not use something else? More specifically, why not derive a class from **CPreviewView** that does something interesting and use its name here instead?

What would you do in a custom preview class? Anything you want. You might draw something special on the page (like the word “preview” in big letters). The ultimate customization, however, would be to make the preview window active so the user can continue to make changes while previewing the print (this is sort of like the page layout view of many word processors).

As you customize your derived class, you may find you need some extra things. First, you can easily add commands to the tool bar that apply to your new view. Just change the template the same way the last example did. Also, you can derive a class from **CPrintPreviewState** and add any data you want to carry around in your derived class. Just be sure to use your new class in the **new** statement (inside **OnFilePrintPreview**).

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Deriving The Class

Unfortunately, Class Wizard won't allow you to select **CPreviewView** as a base class. **CPreviewView** itself derives from **CScrollView**, but because the class deals with all the **CScrollView** details, you are better off deriving a class from **CView**. Then change each occurrence of **CView** to **CPreviewView** in both the H and CPP files. Be sure to change it everywhere it occurs. You'll also need to include AFXPRIV.H to get the definition for **CPreviewView**.

The next step is to modify your existing view. The procedure is very similar to the steps you take when you want to customize the tool bar. The only difference is that you use your new class header instead of AFXPRIV.H and you substitute your class name for **CPreviewView** in the call to **DoPrintPreview**.

If you like, you can override **OnDraw** to draw on the page (or even in the area outside of the page). The problem is finding out where to draw. The same problem arises when you want to handle mouse clicks. You must convert the position of the mouse into coordinates that make sense to your program.

Preview Internals

The preview code resides in three MFC source files. The preview window itself is in VIEWPREV.CPP. A special device context for previewing is in DCPREV.CPP. Finally, definitions are in AFXPRIV.H. Studying these files can shed a lot of light on what's going on inside print preview.

The primary source of information for previewing is in the **m_pPageInfo** member. This is an array of **PAGE_INFO** structures (see Table 3.4). There is

an element in this array for each displayed page. If you are only showing one page, that means you'll always work with element 0. If you are showing two pages, you may have to use elements 0 and 1.

Table 3.4 The PAGE_INFO structure.

Member	Definition
rectScreen	Screen coordinates of this page
sizeUnscaled	Unscaled screen rectangle
sizeScaleRatio	Ratio (cx/cy) of printer units to preview screen units
sizeZoomOutRatio	Scale ratio used when zoomed out

You can find the screen coordinates of the page in question using the **rectScreen** member. The other three items in this structure are technically **CSize** objects. However, MFC doesn't use them as sizes. Instead, it uses them to store a fraction (cx/cy). You have to keep this in mind when you are reading the MFC source code.

By manipulating these various scaling factors, you can transform points on the printer to screen points and vice versa. You'll see the exact procedure in the example program. Unfortunately, I haven't found an easy way to do the transformation from points on the screen back to points in your program. The good news is that if you write the code to do it once, you never have to write it again. Of course, you can also just borrow the code from the example in this chapter.

CPreviewView has many other members, but most of them are not very interesting. You can find the current page by examining **m_nCurrentPage**. You can also find the preview DC in **m_pPreviewDC**.

Creating An Editable Print Preview

Figure 3.4 shows another version of the connect the dots program. This version provides its own **CPreviewView** object. The print preview screen looks almost the same as before (see Figure 3.4), but there is an extra button on the tool bar marked Edit. When you push this button, the cursor changes to an arrow, and you can draw directly on the print preview window. Pushing the button again (it now says Zoom) puts you back in the usual zoom cursor mode.

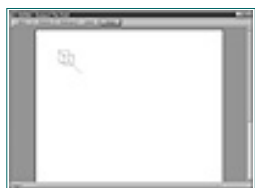


Figure 3.4 *Editing during preview.*

How does this work? The custom preview class maintains a Boolean variable that tracks if the view is in edit mode or zoom mode (**editmode** in Listing 3.3). It also provides message map handlers for both mouse buttons, **WM_SETCURSOR**, and the **ID_EDITMODE** button (the button that

switches between zoom and edit modes).

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

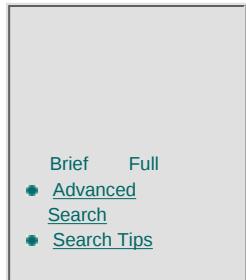
[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE



BROWSE BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

When the user pushes the edit mode button, the program simply reverses the state of the **editmode** flag. There is also an **UPDATE_COMMAND_UI** handler that places the correct caption in the button (see Chapter 1 for more about update handlers). When the program receives a **WM_SETCURSOR** message, it checks to see if **editmode** is **TRUE** and the cursor is over the window's client area. If both of these conditions are true, the program sets a standard arrow cursor. Otherwise, the code delegates to the base class so that everything works normally.

Listing 3.3 Editing during Print Preview.

```
// CustomPreview.cpp : implementation file
//

#include "stdafx.h"
#include "conndot.h"
#include "CustomPreview.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CCustomPreview

IMPLEMENT_DYNCREATE(CCustomPreview, CPreviewView)

CCustomPreview::CCustomPreview()
{
    editmode=FALSE;
}

CCustomPreview::~CCustomPreview()
{
}

BEGIN_MESSAGE_MAP(CCustomPreview, CPreviewView)
```

```

//{{AFX_MSG_MAP(CCustomPreview)
ON_WM_RBUTTONDOWN()
ON_WM_LBUTTONDOWN()
ON_COMMAND(ID_EDITMODE, OnEditMode)
ON_UPDATE_COMMAND_UI(ID_EDITMODE, OnUpdateEditMode)
ON_WM_SETCURSOR()
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CCustomPreview drawing

////////////////////////////////////
// CCustomPreview diagnostics

#ifdef _DEBUG
void CCustomPreview::AssertValid() const
{
    CView::AssertValid();
}

void CCustomPreview::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}
#endif //_DEBUG

////////////////////////////////////
// CCustomPreview message handlers

void CCustomPreview::OnRButtonDown(UINT nFlags, CPoint point)
{
    if (editmode)
    {
        ConvertPoint(point);
        // move current point only
        CPoint mover=point;
        mover.x=-mover.x;
        mover.y=-mover.y;
        GetDocument()->points.Add(mover);
    }
    else
        CPreviewView::OnRButtonDown(nFlags, point);
}

void CCustomPreview::OnLButtonDown(UINT nFlags, CPoint point)
{
    if (editmode)
    {
        CConndotDoc *doc=GetDocument();
        ConvertPoint(point);
        doc->points.Add(point);
        doc->UpdateAllViews(NULL);
    }
    else
        CPreviewView::OnLButtonDown(nFlags, point);
}

CConndotDoc *CCustomPreview::GetDocument()

```

```

{
    return (CConndotDoc *)CPreviewView::GetDocument();
}

void CCustomPreview::ConvertPoint(CPoint & point)
{
    // Assuming 1 page only visible.
    // If that isn't a good assumption, you'd need to test to see which page
    // you were on by testing the rectScreen member of each element in the

    // m_pPageInfo array.
    // Don't forget preview uses sizeScaleRatio as a fraction, not
    // a x,y scaling factor.
    CPoint ViewportOrg;
    if (m_nZoomState != ZOOM_OUT)
        ViewportOrg = -GetDeviceScrollPosition();
    else
        ViewportOrg=GetDC()->GetViewportOrg();
    m_pPreviewDC->SetScaleRatio(m_pPageInfo[m_nCurrentPage-1]
        .sizeScaleRatio.cx,
        m_pPageInfo[m_nCurrentPage-1].sizeScaleRatio.cy);

    // figure size of margin
    CSize PrintOffset;
    m_pPreviewDC->Escape(GETPRINTINGOFFSET, 0, NULL, (LPVOID)&PrintOffset);
    m_pPreviewDC->PrinterDPtoScreenDP((LPPOINT)&PrintOffset);
    PrintOffset += (CSize)m_pPageInfo[m_nCurrentPage-1].rectScreen
        .TopLeft();
    PrintOffset += CSize(1, 1);
    PrintOffset += (CSize)ViewportOrg; // For Scrolling
    point-=PrintOffset;

    // adjust point for page position
    point.x = MulDiv(point.x, m_pPageInfo[m_nCurrentPage-1]
        .sizeScaleRatio.cy,
        m_pPageInfo[m_nCurrentPage-1]
        .sizeScaleRatio.cx);
    point.y = MulDiv(point.y, m_pPageInfo[m_nCurrentPage-1]
        .sizeScaleRatio.cy,
        m_pPageInfo[m_nCurrentPage-1]
        .sizeScaleRatio.cx);
}

void CCustomPreview::OnEditMode()
{
    editmode=!editmode;
}

void CCustomPreview::OnUpdateEditMode(CCmdUI* pCmdUI)
{
    pCmdUI->SetText(editmode?"Zoom":"Edit");
    pCmdUI->Enable();
}

BOOL CCustomPreview::OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT message)
{
    if (editmode && nHitTest==HTCLIENT)
    {

```

```

        ::SetCursor(AfxGetApp()->LoadStandardCursor(IDC_ARROW));
        return TRUE;
    }
    else
        return CPreviewView::OnSetCursor(pWnd, nHitTest, message);
}

```

The real work occurs in the mouse button handlers. There, the code has to examine the **editmode** flag. If it is **FALSE**, the program simply hands the message to the base class. However, when **editmode** is **TRUE**, the program converts the mouse point into a normal point for the program and adds it to the document (just like the ordinary view).

To keep things simple, the mouse handlers call a function named **ConvertPoint** to do all the dirty work. If you look at the function, you'll see that it is dirty indeed. It isn't that hard to convert the point using **MulDiv** and the scaling ratios. The hard part is computing how big of a margin the printed page has and then converting the margin into a pixel measurement. You also have to take scrolling into account. Luckily, you can copy this function directly into your own custom preview classes.

The only other tricky part is getting a pointer to the document. In Chapter 1, you saw that App Wizard overrides your main view's **GetDocument** function so that it returns your main document type instead of the generic **CDocument**. However, when you create a new view class with Class Wizard, it has no idea what document that view corresponds to. Therefore, it doesn't provide a **GetDocument** override. When you call **GetDocument** in the new class, it returns a **CDocument** pointer. Trying to call your personal document functions (like **Add** in the example) isn't allowed without casting. Of course, there is nothing to prevent you from providing your own **GetDocument** override, which is exactly what the example code does.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Summary

If you're like most programmers, you put off printing until the last minutes of the development cycle. Luckily, MFC encourages you to do just that. Because drawing and printing are so closely related, you can frequently let MFC handle your printing if you just plan ahead.

However, if your printing needs are more demanding, you'll have to understand how printing and print preview work. Armed with the information in this chapter, you should be able to make printing work the way you want it to work.

Practical Guide To Printing

- Controlling The Print Dialog
- Scaling Printing
- Printing Something Different
- Printing Headers And Footers
- Customizing Print Preview's Tool Bar
- Customizing Print Preview

For many applications, using a mapping mode other than **MM_TEXT** is all you'll need to do to get printing to work. App Wizard sets up simple-minded printing support in your program and any of the normal mapping modes will automatically scale your printing.

Controlling The Print Dialog

By default, App Wizard is an **OnBeginPrinting** function in your view class. This function simply calls **DoPreparePrinting**. However, the **CPrintInfo** object that this function receives allows you to take control of the print dialog (and, of course, the actual printing process).

The **CPrintInfo** object has two main members of interest: **m_pPD**, which is a pointer to the **CPrintDialog** box MFC will display, and **SetMaxPage**, which allows you to specify the number of pages.

If you don't want to display a print dialog at all, set the **m_bDirect** flag to **TRUE** before calling **DoPreparePrinting**. This will cause the print job to print all pages to the default printer without user intervention.

Scaling Printing

If you are using **MM_TEXT** or one of the isotropic mapping modes, your printout will probably look different from your screen drawing. If you think about it, it makes sense. Your screen has about 72 dots per inch. So if you draw 75 dots, that's just over an inch (more or less—screen measurements aren't exact). On a 300 dpi printer, 75 dots are exactly 1/4 of an inch. At 600 dpi, you've got about 1/8 of an inch.

There are several ways to deal with this problem. The most straightforward method is to avoid using device dependent modes. Instead, stick to modes like **MM_LOENGLISH** or **MM_HIMETRIC** (see Table 3.3).

As this isn't always possible, you'll have to scale your printing to match your drawing when you use the more typical mapping modes. The idea is to set your device context to **MM_ISOTROPIC** and set the scaling such that when you draw 75 dots on the printer (for example), it turns out to be just over an inch.

Listing 3.1 shows an **OnPrint** function that automatically scales the printer to match the screen. Of course, you might want to do something slightly different. For example, suppose you want printouts to be twice as big as what you see on the screen. You could do that, too, by adjusting the scaling factors.

Printing Something Different

MFC generally assumes you want to print the same thing you draw on the screen. This is usually a good assumption, but it isn't always the case. Suppose you have a data entry form as part of a view. The form shows you a single record in a database. When you print, you don't want the one record that happens to be current. You want a listing of the entire database printed in tabular format.

This is easy to do. Just override **OnPrint**. The default **OnPrint** calls **OnDraw**, but you can write your own **OnPrint** to print anything you want. This is especially important for **CFormView**-based programs because **CFormView** controls are unable to print themselves anyway.

Printing Headers And Footers

Another reason to override **OnPrint** is to draw extra things when printing. A good example is when you want to draw headers or footers during printing but not for ordinary screen output.

You can find an example of this in the **OnPrint** handler in Listing 3.1. It is important to set the clipping region to protect footers and change the DC's origin to protect the header. If you think that is too much trouble, you could always use **IsPrinting** inside **OnDraw** and just make **OnDraw** smart enough to add headers and footers.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

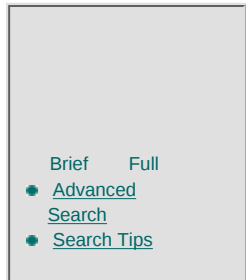
[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE



BROWSE BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Customizing Print Preview's Tool Bar

If you want to change the standard print preview tool bar, you'll need to add your own **OnFilePrintPreview** handler. Here are the steps you should take:

1. Hoist the **ON_COMMAND** macro that contains **OnFilePrintPreview** into the Class Wizard portion of the view's message map.
2. Remove the base class modifier from the existing call to **OnFilePrintPreview**.
3. Add your own **OnFilePrintPreview** member function (in your H and CPP files).
4. In your function, create a new **CPrintPreviewState** object on the heap (using **new**).
5. Call **DoPrintPreview**, passing the resource ID you want to use for your tool bar, the pointer, **RUNTIME_CLASS(CPreviewView)**, and the object you created in the previous step.
6. If **DoPrintPreview** returns **FALSE**, display an error message and delete the object you made in Step 4.
7. Be sure to include **AFXPRIV.H** in your view class' CPP file.
8. Create an appropriate tool bar resource with the same ID you used in Step 5 (you can start with the template from **AFXPRINT.RC** in the **MFC\INCLUDE** directory).

When you are done, your message map should look like this:

```
BEGIN_MESSAGE_MAP(CConndotView, CView)
   //{{AFX_MSG_MAP(CConndotView)
    ON_WM_LBUTTONDOWN()
    ON_WM_RBUTTONDOWN()
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, OnFilePrintPreview)
   //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
END_MESSAGE_MAP()
```

Your function will look something like this:

```
void CMyView::OnFilePrintPreview()
{
```

```

CPrintPreviewState* pState = new CPrintPreviewState;
if (!DoPrintPreview(MY_PREVIEW_TOOLBAR, this,
    RUNTIME_CLASS(CPreviewView), pState))
{
    AfxMessageBox(AFX_IDP_COMMAND_FAILURE); // default error message
    delete pState;
}
}

```

You can find a complete example of this technique in Listing 3.2.

Customizing Print Preview

You can follow the same steps you use to customize the tool bar (see the previous topic) if you want to completely customize how your print preview looks and behaves. However, instead of using **CPreviewView** as the third argument to **DoPrintPreview**, you'll use your own class. You can find an example of this technique in Listing 3.3.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Chapter 4 Windows, Views, And Controls

Controls are one of the best ways to re-use code under Windows. Using MFC, you can customize these controls to best suit your program. You'll see how to customize list views, use owner-draw controls, and more.

Programmer's Notes...

When people ask me questions, I'm always struck by how often it is that what they ask doesn't really relate to what they want to know. For example, several times I've had non-technical friends and neighbors ask me how they can get more memory on their PC. If I'm not careful, I'll launch into a discussion of 30 pin SIMMs versus 72 pin SIMMs, parity memory, and the like. Then, after 10 minutes of watching their faces fall, I'll discover they just wanted more disk space.

After many similar experiences, I've become wary of questions that are too specific. For example, someone will ask me, "How do I write a serial device driver?" I'll answer with another question: "What do you want to do?" When I find out that the person wants to work with the modem, I'll suggest a more sensible approach using TAPI or at least ordinary Win32 calls.

I recently went to Walt Disney World and realized that the people who built the park (the Imagineers, I'm told) had a great understanding of this approach. When you walk down Main Street in the Magic Kingdom, you can smell

cookies baking near the bakery. Your natural inclination is to assume that they are baking cookies, right? Wrong. It seems there are two problems with just venting cookie fumes into the street. The first is that the bakery doesn't bake cookies all the time, so it would be impossible to keep up the constant scent. Second, blowing air over hot cookies causes them to crack. So instead of giving up on cookie aroma, the Imagineers manufacture artificial cookie smell and blow that out into the street. Again, the effect is more important than the means by which it is achieved.

The same principle comes into play when you write programs. Sometimes getting what you want is more important than how you get it. Back when Windows 3.0 was entering the scene, a friend of mine called me one day. He told me that he had read about Windows, but he wasn't sure how to handle the **WM_PAINT** message. Instead of explaining exactly how to do it, I asked him what kind of program he was writing. He went on to describe a simple program that pulled up some data from a database and displayed it on the screen.

After hearing the problem, I told him to just use a dialog box with edit fields. Then you don't have to worry about **WM_PAINT** (and a host of other problems). Sure, the edit controls have to handle **WM_PAINT** messages somewhere, but who cares? It doesn't matter as long as the edit controls work properly.

Usually, you want someone else to worry about as much of the details as possible. But there's a danger here, too. What happens when the control doesn't do exactly what you want? Then you have to start from scratch, find another control, or bend the control to your will. For example, if my friend's database program needed to display text in different colors, that would have posed a problem. Because edit controls typically show text in one color, he might have had to use a more traditional programming approach with all the hassles of handling **WM_PAINT**.

Compounding this problem is the fact that MFC provides its own set of controls, windows, and views that help you write programs faster. Again, if they work for you, great! If they don't, you'll have to figure out a different strategy.

Generally, duplicating a control's functionality from scratch should be a last resort. Not only is this much more work, it also poses problems if the control's functions change over time. Then you'll have to work to stay in sync with the original control. The same goes for views and windows that may change between versions of Windows or MFC.

A better strategy is to alter the control. How? There are a variety of methods. With MFC, it is usually easiest to derive a new class based on the old control (or view, or window) and make the changes in the new class. Then the only obstacle is forcing the program to use your new class.

An Improved CListCtrl

To illustrate adding functionality to an existing control, consider the **CListCtrl**

class. MFC uses this class to encapsulate the new-style list control (part of the common controls package). This control offers several powerful features. Perhaps the most often used feature is the report style. This allows you to build columns of data in the list control (see Figure 4.1). You see this style of control in many places in the Windows shell. For example, the file search function on the Start menu button places its results in this type of control.

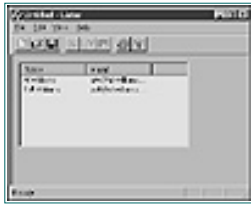


Figure 4.1 A list control.

There are two things I don't like about this control. First, to select an item, you must click on the first column of the item you want. Clicking on any other column has no effect. My second complaint is that when you select an item, only the first column changes appearance; the other columns remain the same.

It isn't very difficult to change the behavior of the list control. It's certainly easier than trying to duplicate the behavior of the list control. One approach would be to understand the internal workings of the list control and replace the entire selection logic in a derived class. Sure, you could do that, but there is an easier way.

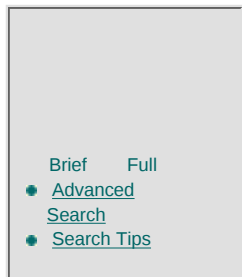
Instead of worrying about the selection yourself, just create a derived class that alters the user's mouse clicks so that they occur in a place the control is expecting. In other words, if the user clicks anywhere in an item, adjust the coordinates so that the click falls somewhere in the first column.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97



Search this book:



Altering The Control

To successfully convert the mouse clicks, you'll need to provide handlers for **WM_LBUTTONDOWN**, **WM_LBUTTONDOWN**, and **WM_LBUTTONDOWNBLCLK**. Your first attempt at the button down handler might look like this:

```
void CFullList::OnLButtonDown(UINT nFlags, CPoint point)
{
    point.x=0;
    CListCtrl::OnLButtonDown(nFlags, point);
}
```

This seems like a good idea. Set the x coordinate to zero and call the base class handler. However, there are two problems with this approach. First, setting **point.x** to zero doesn't work as you would expect. Why not? Because MFC doesn't really handle the **WM_LBUTTONDOWN** message in the **CListCtrl** class. The default for this message is to forward the message to the underlying Windows control.

When MFC receives a **WM_LBUTTONDOWN** message, it parses the **wParam** and **lParam** arguments (included with the message) and converts them to the **nFlags** and **point** arguments you see in the message handler function. Changing the **point** argument and calling the base class with the new value will only work if the base class does further processing within MFC. In this case, it simply forwards the message to the control.

To perform this message forwarding, MFC uses a function called **Default**. Here is what it looks like:

```
LRESULT CWnd::Default()
{
    // call DefWindowProc with the last message
    _AFX_THREAD_STATE* pThreadState = _afxThreadState.GetData();
    return DefWindowProc(pThreadState->m_lastSentMsg.message,
        pThreadState->m_lastSentMsg.wParam,
        pThreadState->m_lastSentMsg.lParam);
}
```

There are two important points here. First, **DefWindowProc** is not necessarily the standard Windows function by the same name. It is a **CWnd** member that calls the correct window procedure (which might be **::DefWindowProc**). In this case, it isn't calling **::DefWindowProc**; it's calling the default handler for the list control. Second, notice that MFC doesn't unparse the parameters that it previously parsed. Instead, it retrieves

the message's **wParam** and **lParam** values directly from a thread-specific data block.

This last tidbit is why changing the **point** parameter isn't sufficient to fool the control. The control only sees the original **wParam** and **lParam** arguments. In this case, the base class doesn't process arguments at all. However, to be safe, your best bet is to change the arguments in both places:

```
void CFullList::OnLButtonDown(UINT nFlags, CPoint point)
{
    int margin;
    point.x=0; // fool MFC
    // fool everyone else
    _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
    pThreadState->m_lastSentMsg.lParam=MAKELONG(point.x,
        HIWORD(pThreadState->m_lastSentMsg.lParam));
    CListCtrl::OnLButtonDown(nFlags, point);
}
```

This still won't work. The problem is that there is a small margin to the left of each item. Clicking on this margin has no effect. Therefore, setting the x coordinate to 0 ensures that each click is in the margin. You could probably guess the size of the margin, but I prefer to calculate it. The margin is the same for every item, so if you can learn it for one item, you know it for all the items. Here's the final version of the code:

```
void CFullList::OnLButtonDown(UINT nFlags, CPoint point)
{
    int margin;
    CRect r;
    GetItemRect(0,&r,LVIR_LABEL);
    margin=r.left;
    point.x=margin; // fool MFC
    // fool everyone else
    _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
    pThreadState->m_lastSentMsg.lParam=MAKELONG(point.x,
        HIWORD(pThreadState->m_lastSentMsg.lParam));
    CListCtrl::OnLButtonDown(nFlags, point);
}
```

Showing The Selection

Duplicating the above code for the other mouse handlers takes care of my first complaint. But what about the selection problem? Remember, my second complaint was that the selection only highlights the first column. Fixing this is a bit more tricky. The best way would be to draw the selection yourself. But that's too much work.

An easier answer is to allow the control to draw itself and then modify the drawing in some way. To keep things simple, I decided to draw a rectangle around the entire selection (see Figure 4.2). That leaves the first column highlighted and the other columns surrounded by a rectangle.

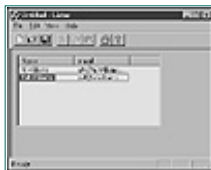


Figure 4.2 A list control with selection.

The first step is to override **OnPaint**. Class Wizard will tell you not to call the base class in this function. Ignore Class Wizard. You also won't use the suggested **CPaintDC** device context. That's because the base class has already used it. You'll need to get an ordinary DC for the window after the base class is finished drawing.

Here's the entire painting code:

```
void CFullList::OnPaint()
{
```

```

CRect r;
int n,n0,n1;
CListCtrl::OnPaint();
n0=GetTopIndex();
n1=GetCountPerPage();
n1+=n0;
CDC *dc=GetDC(); // don't use CPaintDC
for (n=n0;n!=n1;n++)
{
    if (GetItemState(n,LVIS_SELECTED)!=LVIS_SELECTED) continue;
    GetItemRect(n,&r,LVIR_BOUNDS);
    r.InflateRect(-2,0);
    dc->SelectStockObject(HOLLOW_BRUSH);
    dc->Rectangle(&r);
}
}

```

The first step is to call the base class so that the ordinary painting occurs. Next, the code learns the item number of the first visible item and the number of items that occur on the page (**n0** and **n1**). Then the program converts **n1** to contain the index of the last item that is visible.

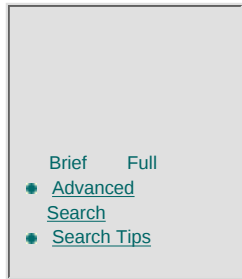
Armed with an ordinary **CDC**, the program enters a **for** loop that checks each visible item to see if it is selected. If it isn't, the **for** loop continues. If it is, the code learns the rectangle for the item, shrinks it horizontally so it fits better, and draws a rectangle around it. Overall, this is much simpler than taking responsibility for all the drawing.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97



Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Using The Modified List

Listing 4.1 show the complete code for the modified list control. The simple example program that uses the control just displays some fixed data.

Listing 4.1 An enhanced list control.

```
// FullList.cpp : implementation file
// A ListCtrl that allows you to select full lines

#include "stdafx.h"
#include "lister.h"
#include "FullList.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////
// CFullList

CFullList::CFullList()
{
}

CFullList::~CFullList()
{
}

BEGIN_MESSAGE_MAP(CFullList, CListCtrl)
   //{{AFX_MSG_MAP(CFullList)
    ON_WM_LBUTTONDOWN()
    ON_WM_LBUTTONUP()
    ON_WM_LBUTTONDBLCLK()
    ON_WM_PAINT()
    }
```

```

        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CFullList message handlers
// The idea is that when you click,
// we move the mouse coordinates to the far left.
// However, you can't move it to 0 since that
// isn't a legit value. Also, MFC doesn't really
// use the values you pass to the base class when
// it defers to the default window handler!
// It uses the last message values. So, if
// the base class uses MFC, we can just change the
// point argument. But if it goes directly to
// Windows (and it does), we have to modify the
// m_lastMsgSent in the thread state.
// This code does both just in case one day MS
// adds MFC code to the base class.
void CFullList::GoMargin(UINT nFlags, CPoint &point)
{
    int margin;
    CRect r;
    GetItemRect(0,&r,LVIR_LABEL);
    margin=r.left;
    point.x=margin; // fool MFC
    // fool everyone else
    _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
    pThreadState->m_lastSentMsg.lParam=
        MAKELONG(point.x,HIWORD(pThreadState->
            m_lastSentMsg.lParam));
}

void CFullList::OnLButtonDown(UINT nFlags, CPoint point)
{
    GoMargin(nFlags,point);
    CListCtrl::OnLButtonDown(nFlags, point);
}

// Same logic here
void CFullList::OnLButtonUp(UINT nFlags, CPoint point)
{
    GoMargin(nFlags,point);
    CListCtrl::OnLButtonUp(nFlags, point);
}

void CFullList::OnLButtonDblClk(UINT nFlags, CPoint point)
{
    GoMargin(nFlags,point);
    CListCtrl::OnLButtonDblClk(nFlags, point);
}

void CFullList::OnPaint()
{
    CRect r;
    int n,n0,n1;
    CListCtrl::OnPaint(); // Let control draw itself
    // find out first and last line
    n0=GetTopIndex();
    n1=GetCountPerPage();
    n1+=n0;

```

```

        CDC *dc=GetDC(); // don't use CPaintDC
// If we have a selected item, draw a rect around it
for (n=n0;n!=n1;n++)
{
    if (GetItemState(n,LVIS_SELECTED)!=LVIS_SELECTED) continue;
    GetItemRect(n,&r,LVIR_BOUNDS);
    r.InflateRect(-2,0); // shrink a bit
    dc->SelectStockObject(HOLLOW_BRUSH);
    dc->Rectangle(&r);
}
}

```

To make the form view use the new class, you first have to associate the list with your new class (**CFullList**) using Class Wizard. Here is the relevant code from the view's header file:

```

public:
    //{AFX_DATA(CListView)
    enum { IDD = IDD_LISTER_FORM };
    CFullList      m_list;
    //}AFX_DATA

```

Dialog Controls

There is more than one way to create a **CFullList** control, of course. If you just need the control, you could call **Create**. You can handle ordinary dialogs just like the example program handles a **CFormView** (that is, use DDX, a topic fully covered in Chapter 5).

There is another way you can convert an existing list control into a **CFullList**: subclassing. To subclass an existing window, construct a **CFullList** object and then call the **SubclassWindow** or **SubclassDlgItem** members. This assumes that the control isn't already attached to a **CWnd**-derived object. This might be the case if a third-party DLL created the control, or perhaps it is on a dialog box and you want to explicitly set the connection between your variable and the control.

Tip: Attaching Dialog Controls

Although you can use **SubclassDlgItem** to force a dialog control to attach to a **CWnd**-derived class, you'll usually use DDX instead. However, you might see **SubclassDlgItem** occasionally, so it is good to understand both methods.

General Window Operations

Everywhere you look in a program, there are windows. I guess that's why Microsoft called the operating system Windows. However, MFC often conceals these windows from the programmer. App Wizard creates frames and views without any intervention from you. This is great until you need to change something in a window.

MFC, as usual, gives you many opportunities to control the window creation process (see Table 4.1). By electing to handle portions of the process yourself, you can exert great control over the appearance and behavior of windows (including dialogs, frames, views, controls, and anything else that is really a window).

Table 4.1 The window creation process.

MFC calls...	Override when...
CWnd::Create or CWnd::CreateEx	You want to provide defaults or specialized creation processing
CWnd::PreCreateWindow	You want a chance to alter the standard Windows CREATESTRUCT (good place to register a class, for example)
CWnd::OnGetMinMaxInfo	You want to change the minimum, maximum, or tracking sizes of a window (called many times; not just during creation)
CWnd::OnNCCreate	You need to draw your own non-client area (rare)

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Setting Styles And Initial Conditions

Windows (the operating system) allows you to customize windows (on the screen) by setting style bits when you create them. For example, you can elect to have a nonresizable frame (**WS_THICKFRAME**) or create the window maximized (**WS_MAXIMIZE**). If you want a minimize button in the caption bar, you can set the **WS_MAXIMIZEBOX** style. You can pass these styles (and many more) as flags to **CWnd::Create** or **CWnd::CreateEx**.

However, what happens when you want to create a window that other programmers might use? It isn't very polite to tell them what flags they must pass to **Create**. Also, what about windows that some other part of MFC might create for you? You can't alter the styles it might pass. One way would be to override **Create** so you could merge the styles you wanted with the styles specified by the user.

A more common method in MFC is to override the **PreCreateWindow** member for the class. MFC calls this member just before it calls the actual Windows API **Create** function. It passes in a **CREATESTRUCT** (see Table 4.2). This structure contains all the information you need to create a window. If your overriding **PreCreateWindow** function changes this information, MFC will use the new values in place of the old ones. You can set the styles (or any other initial conditions) in this way.

Table 4.2 CREATESTRUCT.

Member	Type	Description
lpCreateParams	LPVOID	User-provided 32-bit parameter

hInstance	HINSTANCE	Handle to window creator
hMenu	HMENU	Handle for window's menu
HWND	hwndParent	Handle to parent window
cx	int	Window's width
cy	int	Window's height
x	int	Window's x position
y	int	Window's y position
style	LONG	Window styles (for example, WS_OVERLAPPED)
lpszName	LPCSTR	Window caption
lpszClass	LPCSTR	Class name (Windows class, not C++ class)
dwExStyle	DWORD	Extended style

Of course, you usually want to blend the styles you are interested in with the default styles using the bitwise-OR operator (`|`) or reset particular styles using a bitwise-AND (`&`). Your window needs many styles and you rarely want to take control of all of them; you only want to set or to clear specific flags.

One of the fields in the **CREATESTRUCT** structure is the class name. This is the Windows class name and has nothing to do with the MFC class. Windows provides many predefined window class names (BUTTON, EDIT, and so on). You can also create your own window classes. A window class defines the window's background color, the message handling function, the cursor for the window, and other items. If you don't provide one here (or earlier in the creation process), MFC will choose an appropriate one for you.

Custom Window Classes

Table 4.3 shows a **WNDCLASS** structure. This structure contains the information you can set in the window class. If you are satisfied with the defaults, you don't need to register a window class. Sure, the message handling function sounds important, but because MFC uses message maps, this isn't important for MFC programs.

Table 4.3 WNDCLASS structure.

Member	Type	Description
style	UINT	Window styles
lpfnWndProc	WNDPROC	Pointer to window's message-handling function
cbClsExtra	int	Number of extra (user-defined) bytes for class
cbWndExtra	int	Number of extra (user-defined) bytes for each window
hInstance	HINSTANCE	Handle of program creating window
hIcon	HICON	Window's icon

hCursor	HCURSOR	Window's default cursor
hbrBackground	HBRUSH	Window's default background color
lpzMenuName	LPCTSTR	Menu for window
lpzClassName	LPCTSTR	Name to assign this class

For most MFC programs, you'll use **AfxRegisterWndClass** to create custom classes. This call takes the class styles you want (if any), a handle to a cursor, a brush handle for the window background, and the window's icon handle. It returns the name of the class that will satisfy these requests. You can call this function as many times as you like with the same arguments and it will return the same name. What name? You don't care exactly what the name is—you just pass it to **Create** or **CreateEx** to make a window with the specified attributes.

If you need to control the name of the class, you'll have to call **AfxRegisterClass** and pass a filled-in **WNDCLASS** structure (see Table 4.3). For an EXE program, this is the same as calling **::RegisterClass** in the Windows API. However, for DLLs, **AfxRegisterClass** does some extra work, so you might as well develop the habit of using it.

Why do you care about the name of a class? Generally, you don't. However, if you want to use the class in a dialog, you'll need the name (see Chapter 5 for more about custom windows in dialogs).

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)BROWSE
BY TOPIC

Restricting Window Size

One of the most basic customizations you might like to make to a frame window is to restrict its size. Of course, if you only want it to be one size, you could set that size when you create the window and force it to have a nonresizeable border. However, this makes the window look odd. Also, it doesn't help in the case in which you want the user to resize it within a certain range.

You might first consider handling the **WM_SIZE** message to create this effect. However, this isn't a good idea. The **WM_SIZE** message occurs *after* the window changes size. If you don't like the size, you could resize the window again, but that is visually annoying. Besides, that will also generate another **WM_SIZE** message, giving you an excellent chance of coding a dynamic halt (or endless loop, if you prefer).

A better answer reveals itself if you study the steps that MFC and Windows go through when your program creates a window (see Table 4.1). Notice the call **OnGetMinMaxInfo** (which corresponds to the **WM_GETMINMAXINFO** message). Windows triggers this call every time it needs to know the maximum and minimum dimensions of the window. This occurs when you create the window, of course. It also occurs at other times, such as when the user resizes the window.

The function fills in an array with several items. You can specify the minimum permissible size, the maximum permissible size, the size the window should be when it is maximized, and the maximized top left corner position. Usually, you think of a maximized window as a window that consumes the entire screen, but that isn't always true. To see an example, maximize a DOS window. The window will become large enough to hold 25 lines of 80 characters. There is no point in the window growing any larger.

Once you know about **OnGetMinMaxInfo**, restricting window sizes is a snap. Just call the base class to pick up any defaults and then set the sizes you want to change. Because Windows will call this function again when it changes the window's size, you can change the dimensions later. For example, you might want your initial window to be no larger than 100×100 pixels. Then later, after you have some real data, you might want to relax the restriction to 100×600. You can easily do this by handling **OnGetMinMaxInfo**.

When you want a window of a certain size, remember that you specify window sizes based on the rectangle the entire window occupies. Only some of that rectangle is available for your client area. If you know how big your client area should be, you can use the Windows API call **::AdjustWindowRectEx** to convert it into the correct window size.

Setting the view size, by the way, requires a little legerdemain. The problem is that views are inside frames.

That means you have to make the frame window big enough to contain the entire view. The view has a border, so you really have to call **::AdjustWindowRectEx** twice: once to calculate the real size of the view and once to calculate the size the frame has to be to hold that view. For example, suppose you want a view that is 200×200 pixels. You need to call **::AdjustWindowRectEx** with the view's styles. How do you get the styles? Well, if you already have the view, you can call **GetStyle** and **GetExStyle**. But what happens if you don't already have a view to work with?

That's not as big a problem as it sounds. The styles shouldn't change (at least, not within one version of MFC), so you could just read them out once and hard code them. You could also examine the view with Spy++ (or a similar tool) and code the styles in that way. The best answer, however, is probably to use the constant **AFX_WS_DEFAULT_VIEW** (defined in AFXRES.H).

Suppose that the result from calling **::AdjustWindowRectEx** tells you that the view should be 202×202 pixels. Be careful when interpreting the result. The returned rectangle might end at point 201, 201. But you'll also find that the rectangle begins at -1, -1 for a total width and height of 202 pixels (of course, the exact numbers will vary depending on your system). That means that the outside part of the view must be 202×202 so that the inside can be 200×200.

Armed with this information, you can call **::AdjustWindowRectEx** with the frame's styles. Be sure to indicate if your frame has a menu, too. Of course, MDI child frames and views never have menus. An SDI frame, on the other hand, probably does have a menu. This call might adjust the 202×202 rectangle to 210×242. This is how big the frame must be so that the view's client area will be 200×200. You'll find a view that does just this in Listing 4.2.

Listing 4.2 A 200×200 view.

```
// sizeView.cpp : implementation of the CSizeView class
//

#include "stdafx.h"
#include "size.h"

#include "sizeDoc.h"
#include "sizeView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CSizeView

IMPLEMENT_DYNCREATE(CSizeView, CView)

BEGIN_MESSAGE_MAP(CSizeView, CView)
    //{AFX_MSG_MAP(CSizeView)
    // NOTE -- the ClassWizard will add and remove mapping macros here.
    // DO NOT EDIT what you see in these blocks of generated code!
    //}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////
// CSizeView construction/destruction

CSizeView::CSizeView()
{
    // TODO: add construction code here
```

```

}

CSizeView::~CSizeView()
{
}

BOOL CSizeView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CView::PreCreateWindow(cs);
}

////////////////////////////////////
// CSizeView drawing

void CSizeView::OnDraw(CDC* pDC)
{
    CSizeDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // just draw 2 200 unit lines with arrow heads
    pDC->MoveTo(5,194);
    pDC->LineTo(10,199);
    pDC->LineTo(10,0);
    pDC->LineTo(5,5);
    pDC->MoveTo(10,0);
    pDC->LineTo(15,5);
    pDC->MoveTo(10,199);
    pDC->LineTo(15,194);

    pDC->MoveTo(5,95);
    pDC->LineTo(0,100);
    pDC->LineTo(199,100);
    pDC->LineTo(194,95);
    pDC->MoveTo(199,100);
    pDC->LineTo(194,105);
    pDC->MoveTo(0,100);
    pDC->LineTo(5,105);
}

////////////////////////////////////
// CSizeView diagnostics

#ifdef _DEBUG
void CSizeView::AssertValid() const
{
    CView::AssertValid();
}

void CSizeView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CSizeDoc* CSizeView::GetDocument() // non-debug version is inline

```

```

{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CSizeDoc)));
    return (CSizeDoc*)m_pDocument;
}
#endif // _DEBUG

////////////////////
// CSizeView message handlers

void CSizeView::OnInitialUpdate()
{
    DWORD style,exstyle;
    CRect framerect, viewrect(0,0,200,200);
    CFrameWnd *frame=GetParentFrame();
    style=GetStyle();
    exstyle=GetExStyle(); // view styles!
    ::AdjustWindowRectEx(&viewrect,style,FALSE,exstyle);
    style=frame->GetStyle();
    exstyle=frame->GetExStyle(); // frame styles!
// NOTE: SDI frame has menu (TRUE), but an MDI child frame would not
    (FALSE)
    ::AdjustWindowRectEx(&viewrect,style,TRUE,exstyle);
    frame->GetWindowRect(&framerect); // don't lose top left corner
    framerect.right=framerect.left+viewrect.Width();
    framerect.bottom=framerect.top+viewrect.Height();
    frame->MoveWindow(&framerect);
    CView::OnInitialUpdate();
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Setting The Title

Setting the title of a frame window is easy, right? It seems to be. You simply call **SetWindowText** for the window in question. You can also set the title in a document string for the frame window (see Chapter 1 and Table 1.12). However, MFC often does something annoying. It adds the current document title to the frame window in most cases.

If your program really uses the document, this is actually a feature. But it is very annoying when your arcade game has the title “BitZapper - Untitled”. The answer is to clear the **FWS_ADDTOTITLE** bit in the frame window’s style bits. You can do that easily in the **PreCreateWindow** function:

```
BOOL CMyFrame::PreCreateWindow(CREATESTRUCT &cs)
{
    BOOL rv=CFrameWnd::PreCreateWindow(cs); // base class
    cs.style&=~FWS_ADDTOTITLE;
    return rv;
}
```

You can also set the **FWS_PREFIXTITLE** bit to force the document name to precede the title, if you prefer.

Tip: FWS_ADDTOTITLE Hints

I often find that I use **FWS_ADDTOTITLE** with form views. Most form views that I write don’t correspond to disk files, and I usually just want a fixed title for them. If you are still writing for Windows 3.1, you need to cast **FWS_ADDTOTITLE** to a **DWORD** before you invert it so that you get all 32 bits set properly.

Using UpdateCMDUI

Controlling the titles and attributes of control windows often requires using an **OnUpdateCmdUI** handler (see Chapter 1). MFC constantly adjusts menu items, tool bar buttons, and other similar windows based on this handler, so any change you try to make will not persist unless you make it inside the handler.

About CScrollView

CScrollView is one of the most powerful views that MFC provides for you to use. It is also the most frustrating to try to use in real life. Why? There are two major problems with **CScrollView**. First, there's no keyboard scrolling behavior built into the view. Second, any attempt to set the virtual size to more than 32,767 units causes odd behavior under Windows 95.

You may think, "I don't ever need to scroll 32,767 items." That may be true—however, think about text. Each line of text in a scrolling list takes a certain number of pixels (say, 16). That means that, using the **MM_TEXT** mapping mode, you can only scroll 2,048 lines of text if each line is 16 pixels high. As the font size goes up, the number of lines goes down.

Before you attack these problems, though, let's examine how **CScrollView** works. Simply put, **CScrollView** lies to your program so that you are not aware of the scrolling that occurs. You tell the view the logical size of your total document. That is, you tell it how big an area you would like to have (regardless of the actual window size). You also tell it (via **SetScrollSizes**) the mapping mode you'd like to use, the size of a line, and the page size.

The first thing the scroll view does is to determine if your logical document size is smaller than the physical window. If it is, the view doesn't show scroll bars. It is possible that your document is smaller than the window in one direction (say, up and down) but larger in the other direction. In that case, only one scroll bar will appear.

If there are no scroll bars or the scroll bars are all the way at position zero, nothing special happens. You paint all of the data you have in your **OnDraw** routine. What happens if you have more data than will fit on the screen? Nothing. Windows simply discards the extra output. If this sounds inefficient, it is. However, it isn't as bad as you might think because drawing things that Windows will discard (or clip, if you prefer) is much faster than real drawing. Later on, I'll show you how to improve efficiency. There is one special note about your **OnDraw** function: It should not set a mapping mode. **CScrollView**'s **OnPrepareDC** function already set it to what you asked for in the **SetScrollSizes** call.

Say the user now scrolls down 50 percent. Your **OnDraw** routine doesn't know the difference. However, **CScrollView** adjusts the view port and window origins so that when you draw the first 50 percent of your document, it is actually *above* the window. Again, Windows discards your drawing. Eventually, you will draw the area that appears in the window and the operating system will really perform the drawing. If your drawing exceeds the lower boundary, the system discards it, too.

If your program is simple enough, that's all there is to it. Set the scroll sizes and draw the entire document every time. You can call **SetScrollSizes** during **OnInitialUpdate** if the size of your document never changes, or you can dynamically resize the document during **OnUpdate**.

There are two cases in which you have to take the scroll view's position into account. The first is when you receive a mouse message. Mouse messages always report their coordinates in pixels (**MM_TEXT** mode). You need to adjust these pixels by the amount of scrolling. You can easily do this by deducting the value returned by **GetDeviceScrollPosition**. Then, convert pixels to logical units as usual (unless pixels are your logical units, in which case, you're done).

The second case in which things get tricky is when you get a **CDC** object outside of your **OnDraw** routine. **CScrollView** automatically manipulates the origin of the **CDC** that your **OnDraw** routine receives. Suppose you want to draw directly on the screen during a mouse message (this is common practice when drawing selection rectangles, for example). If you call **GetDC** and then don't call **OnPrepareDC**, things won't work right. That also means that you don't need to call **SetMapMode** on the device context, either.

If you forget to offset your mouse coordinates or call **OnPrepareDC**, the symptoms are very similar. Everything will work fine when the scroll bars are in the home position (that is, at zero). But when the scroll bars have non-zero positions, things will be off a bit. The more you scroll, the further off things will be.

It bears mentioning that you can call **SetScaleToFitSize** and force **CScrollView** to fit the entire logical document in the window, stretching or shrinking it as appropriate. Of course, that doesn't have much to do with scrolling.

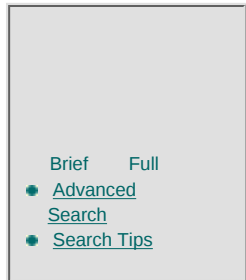
Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE



BROWSE BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Adding Keyboard Scrolling

The first problem with the scroll view is probably the easiest to fix: the lack of keyboard support. Listing 4.3 shows a simple scroll view that supplies an **OnKeyDown** handler. When the view detects a keystroke, it calls **KeyScroll**, a function you can easily cut and paste into your own view. This function examines the virtual key code and simulates a scroll event based on the key.

Listing 4.3 Keyboard scrolling.

```
// scrollerView.cpp : implementation of the CScrollerView class
//

#include "stdafx.h"
#include "scroller.h"

#include "scrollerDoc.h"
#include "scrollerView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CScrollerView

IMPLEMENT_DYNCREATE(CScrollerView, CScrollView)

BEGIN_MESSAGE_MAP(CScrollerView, CScrollView)
   //{{AFX_MSG_MAP(CScrollerView)
    ON_WM_SIZE()
    ON_WM_KEYDOWN()
   //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CScrollView::OnFilePrint)
```

```

        ON_COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
END_MESSAGE_MAP()

//////////
// CScrollerView construction/destruction

CScrollerView::CScrollerView()
{
    // TODO: add construction code here

}

CScrollerView::~CScrollerView()
{
}

BOOL CScrollerView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the window class or styles here by modifying
    // the CREATESTRUCT cs

    return CScrollView::PreCreateWindow(cs);
}

//////////
// CScrollerView drawing

void CScrollerView::OnDraw(CDC* pDC)
{
    CScrollerDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    int ct=sizeof(pDoc->list)/sizeof(pDoc->list[0]);
    pDC->SetBkMode(TRANSPARENT);
    for (int i=0;i<ct;i++)
    {
        CString s;
        s.Format("(%d)=%u",i,pDoc->list[i]);
        pDC->TextOut(0,i*lineht,s);
    }
}

void CScrollerView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    CScrollerDoc* doc = GetDocument();
    CSize sizeTotal,sizePg;
    CRect r;
    GetClientRect(&r);
    CDC *pDC=GetDC();
    TEXTMETRIC tm;
    pDC->GetTextMetrics(&tm);
    lineht=tm.tmHeight+tm.tmExternalLeading;
    sizeTotal.cx = 100;
    sizeTotal.cy = (sizeof(doc->list)/sizeof(doc->list[0]))*lineht;
    sizePg.cx=10;
    sizePg.cy=r.Height();
    // Set beginning size
    SetScrollSizes(MM_TEXT, sizeTotal,sizePg,CSize(10,lineht));
}

```

```

////////////////////////////////////
// CScrollerView printing

BOOL CScrollerView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}

void CScrollerView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CScrollerView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}

////////////////////////////////////
// CScrollerView diagnostics

#ifdef _DEBUG
void CScrollerView::AssertValid() const
{
    CScrollView::AssertValid();
}

void CScrollerView::Dump(CDumpContext& dc) const
{
    CScrollView::Dump(dc);
}

CScrollerDoc* CScrollerView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CScrollerDoc)));
    return (CScrollerDoc*)m_pDocument;
}
#endif //_DEBUG

////////////////////////////////////
// CScrollerView message handlers

void CScrollerView::OnSize(UINT nType, int cx, int cy)
{
    int dummy;
    CScrollView::OnSize(nType, cx, cy);
    CSize sizeTotal,pg,dummysize;
    CRect r;
    GetClientRect(&r);
    pg=CSize(10,r.Height());
    GetDeviceScrollSizes(dummy,sizeTotal,dummysize,dummysize);
    // Set size so page is always a screen full
    SetScrollSizes(MM_TEXT, sizeTotal,pg,CSize(10,lineht));
}

void CScrollerView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{

```

```

        BOOL processed;
        for (unsigned int i=0;i<nRepCnt&&processed;i++)
            processed=KeyScroll(nChar);
        if (!processed)
            CScrollView::OnKeyDown(nChar, nRepCnt, nFlags);
    }

    // Convert keystrokes to quasi-mouse messages
    BOOL CScrollerView::KeyScroll(UINT nChar)
    {
        switch (nChar)
        {
            case VK_UP:
                OnVScroll(SB_LINEUP,0,NULL);
                break;
            case VK_DOWN:
                OnVScroll(SB_LINEDOWN,0,NULL);
                break;
            case VK_LEFT:
                OnHScroll(SB_LINELEFT,0,NULL);
                break;
            case VK_RIGHT:
                OnHScroll(SB_LINERIGHT,0,NULL);
                break;
            case VK_HOME:
                OnHScroll(SB_LEFT,0,NULL);
                break;
            case VK_END:
                OnHScroll(SB_RIGHT,0,NULL);
                break;
            case VK_PRIOR:
                OnVScroll(SB_PAGEUP,0,NULL);
                break;
            case VK_NEXT:
                OnVScroll(SB_PAGEDOWN,0,NULL);
                break;
            default:
                return FALSE; // not for us
        }
        return TRUE;
    }
}

```

It's easy to modify the **KeyScroll** function to handle other keystrokes. You could also respect the current status of the shift and control keys. For example, you might make Ctrl-HOME do something different from just HOME. Another possibility is that you want scrolling to occur only if the SHIFT key is down. You can do all of that by using the **GetKeyState** call to determine if the particular modifier key is up or down. Don't forget that **GetKeyState** doesn't return **TRUE** or **FALSE**. It returns the topmost bit set if the key is down. A handy way to check for this condition is to test if the key state is less than zero. If it is, then the key is down.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Optimizing Scrolling

The program in Listing 4.3 only shows 100 even numbers, so optimization isn't very important. You can easily draw 100 lines of text without affecting performance. Even if you boost the number up to 1,000, you probably won't notice any problems (just change the size of the array in the document's header file). However, if you bring the number up to 10,000, you'll notice a big slowdown. You'll also find that the screen looks very odd if you are running Windows 95 (see Figure 4.3). You'll see what to do about the screen corruption in the next section.

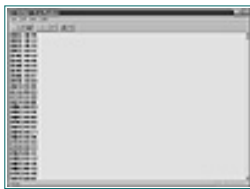


Figure 4.3 Windows 95 scrolling (or lack thereof).

Although MFC doesn't require you to draw only the visible portion of your display in a **CScrollView**, there is no reason why you can't. This is a good idea if you have a complex drawing or if you have lots of data.

The idea is simple: Learn the top coordinate of the window (use **GetDeviceScrollPosition**) and don't draw anything above that. Then, find out how big the window is (**GetClientRect**) and don't draw anything that would go below that. Here is an optimized **OnDraw** for the even number program:

```
void CScrollerView::OnDraw(CDC* pDC)
{
```

```

CScrollerDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
int ct;
pDC->SetBkMode(TRANSPARENT);
int start=GetDeviceScrollPosition().y/lineht-1;
if (start<0) start=0;
CRect client;
GetClientRect(&client);
// draw extra so scrolling works on incomplete lines
ct=start+client.Height()/lineht+3;
if (ct>sizeof(pDoc->list)/sizeof(pDoc->list[0]))
    ct=sizeof(pDoc->list)/sizeof(pDoc->list[0]);
for (int i=start;i<ct;i++)
{
    CString s;
    s.Format("(%d)=%u", i, pDoc->list[i]);
    pDC->TextOut(0,i*lineht,s);
}
}

```

In practice, you have to be careful to draw lines that are at least partially visible in the window. The reason for this is that the scroll view tries to optimize your drawing for you. Where possible, it scrolls the existing bits on the screen and asks you to draw only the area that changes. If you didn't draw partial lines that should have been visible, you'll have big gaps in your drawing when you scroll. To see this, remove the -1 from the calculation for the **start** variable and the +3 from the **ct** variable's calculation.

Tip: Practical Optimization

In most cases, you don't need to worry too much about optimizing your drawing for a scroll view. The exceptions to this rule are when you have a complex drawing (such as a fractal) data that is expensive to get (a real time network status map, for example), or very large amounts of data (see the next section). For simple drawings that aren't very large, it's usually not worthwhile to optimize the drawing.

Scrolling More Than 32K Units

If you're using Windows 95 and you set the number of even numbers to display to a large number (say 10,000), you will notice output like that seen in Figure 4.3. The reason for this is that the logical document's pixel size exceeds 32,767. Although Windows 95 allows you to specify 32-bit graphics coordinates, it actually discards the top 16 bits before processing them.

This poses a major problem for potential users of the **CScrollView** class. How can you use **CScrollView** to display more than 32,767 units? The answer is not simple. In fact, for many programmers, it may be easier to just manually handle scrolling in an ordinary view. However, you can coax the **CScrollView** to deal with about 30,000 items without too much trouble. How is this better? Say a line of text is 16 pixels high. 30,000 items is the same as 480,000 pixels. You can't do that with an ordinary **CScrollView** and Windows 95.

Here's the trick: Tell **CScrollView** how many logical items you have, not how many pixels. Pretend that you will use **MM_TEXT** mode. To make sure you can display all of

the items, you need to tell **CScrollView** that you have slightly more items than you really do. How many depends on the size of the window you are in.

The reason for this is that the view will only scroll until the bottom of the window holds the maximum logical pixel. We aren't working in pixels, so this is a problem. Suppose you have 10,000 lines of text, so you tell the view the logical size is 10,000 pixels. Then suppose the window is 500 pixels high. When the top line corresponds to pixel 9,500, the scroll view quits scrolling. The problem is that you still have 500×16 , or 8,000, pixels left to display. The easiest way to correct for this is to calculate how many pixels are in the window and subtract the height of one line. Then add that value to the total number of pseudo pixels the view will display.

This extra space is why you can't get the full 32,767 items. You need to leave room for the largest screen size you expect. I usually assume I can get away with $32,767 - 2,048$, or 30,719, items at the most. Any larger than that and you are probably going to have to write your own scrolling view (or require users to use Windows NT).

Of course, it isn't that simple. You now have to draw the correct data during **OnDraw**. This is almost the same as the optimization you saw in the last section. First, you read the starting line using **GetDeviceScrollPosition**. Next, you calculate the number of lines that can fit in the window.

So far, this is just like the optimizing draw discussed in the previous section. The trick is to then clobber the existing mapping mode in favor of the one you want to use. Now the origin of the screen is at point (0,0) and you can draw the correct number of lines from the correct starting position.

Is it that simple? Not quite. Because **CScrollView** tries to scroll bits on the screen, you'll find that your display is a mess if you just follow these instructions. The final step is to catch **OnVScroll** (and **OnHScroll**, if applicable), call the base class version of the function, and then invalidate the entire window to force a repaint. This isn't as efficient as an ordinary **CScrollView**, but an ordinary **CScrollView** can't handle 30,000 items either.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

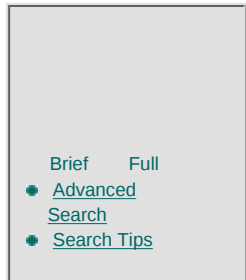
[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE



BROWSE BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Listing 4.4 shows the even number program that incorporates these ideas. Notice that because the logical document size changes depending on the size of the window, the program calls **SetScrollSizes** during the **OnSize** handler. You can compile this code with simple optimization (**METHOD=0**) or with the rescaling for Windows 95 (**METHOD=1**).

Tip: Horizontal Scrolling

The example program really only scrolls vertically. However, the same ideas are usable for horizontal, as well. In this example, an item is a line of text, but it might just as well be a degree of latitude and longitude on a map.

Listing 4.4 Sophisticated scrolling.

```
// scrollerView.cpp : implementation of the CScrollerView class
//

#include "stdafx.h"
#include "scroller.h"

#include "scrollerDoc.h"
#include "scrollerView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

#define METHOD 1 // 0 is optimized; 1 is per line method

////////////////////////
// CScrollerView

IMPLEMENT_DYNCREATE(CScrollerView, CScrollView)

BEGIN_MESSAGE_MAP(CScrollerView, CScrollView)
   //{{AFX_MSG_MAP(CScrollerView)
        ON_WM_SIZE()
    }}
```

```

        ON_WM_VSCROLL()
        //}}AFX_MSG_MAP
        // Standard printing commands
        ON_COMMAND(ID_FILE_PRINT, CScrollView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_DIRECT, CScrollView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
    END_MESSAGE_MAP()

    //////////////////////////////////////
    // CScrollerView construction/destruction

    CScrollerView::CScrollerView()
    {
        // TODO: add construction code here

    }

    CScrollerView::~CScrollerView()
    {
    }

    BOOL CScrollerView::PreCreateWindow(CREATESTRUCT& cs)
    {
        // TODO: Modify the Window class or styles here by modifying
        // the CREATESTRUCT cs

        return CScrollView::PreCreateWindow(cs);
    }

    //////////////////////////////////////
    // CScrollerView drawing
    void CScrollerView::OnDraw(CDC* pDC)
    {
        #if 0 // know nothing draw -- accounts for nothing
            CScrollerDoc* pDoc = GetDocument();
            ASSERT_VALID(pDoc);
            int ct=sizeof(pDoc->list)/sizeof(pDoc->list[0]);
            pDC->SetBkMode(TRANSPARENT);
            for (int i=0;i<ct;i++)
            {
                CString s;
                s.Format("(%d)=%u",i,pDoc->list[i]);
                pDC->TextOut(0,i*lineht,s);
            }
        #endif

        #if METHOD==0 // optimized draw
            CScrollerDoc* pDoc = GetDocument();
            ASSERT_VALID(pDoc);
            int ct;
            pDC->SetBkMode(TRANSPARENT);
            int start=GetDeviceScrollPosition().y/lineht-1;
            if (start<0) start=0;
            CRect client;
            GetClientRect(&client);
            // draw extra so scrolling works on incomplete lines
            ct=start+client.Height()/lineht+3;
            if (ct>sizeof(pDoc->list)/sizeof(pDoc->list[0]))
                ct=sizeof(pDoc->list)/sizeof(pDoc->list[0]);
            for (int i=start;i<ct;i++)

```

```

        {
            CString s;
            s.Format("(%d)=%u", i, pDoc->list[i]);
            pDC->TextOut(0, i*lineht, s);
        }
#endif
#if METHOD==1
    CScrollerDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    int ct;
    pDC->SetBkMode(TRANSPARENT);
    int start=GetDeviceScrollPosition().y;
    CRect client;
    GetClientRect(&client);
    // draw extra so scrolling works on incomplete lines
    ct=start+client.Height()/lineht+3;
    if (ct>sizeof(pDoc->list)/sizeof(pDoc->list[0]))
        ct=sizeof(pDoc->list)/sizeof(pDoc->list[0]);
    pDC->SetMapMode(MM_TEXT); // destroy fake mapping mode!
    pDC->SetViewportOrg(0,0);
    pDC->SetWindowOrg(0,0);
    for (int i=start; i<ct; i++)
    {
        CString s;
        s.Format("(%d)=%u", i, pDoc->list[i]);
        pDC->TextOut(0, (i-start)*lineht, s);
    }
#endif
}

void CScrollerView::OnInitialUpdate()
{
    #if METHOD==0
        CScrollView::OnInitialUpdate();
        CScrollerDoc* doc = GetDocument();
        CSize sizeTotal, sizePg;
        CRect r;
        GetClientRect(&r);
        CDC *pDC=GetDC();
        TEXTMETRIC tm;
        pDC->GetTextMetrics(&tm);
        lineht=tm.tmHeight+tm.tmExternalLeading;
        sizeTotal.cx = 100;
        sizeTotal.cy = (sizeof(doc->list)/sizeof(doc->list[0]))*lineht;
        sizePg.cx=10;
        sizePg.cy=r.Height();
        SetScrollSizes(MM_TEXT, sizeTotal, sizePg, CSize(10, lineht));
    #endif
    #if METHOD==1
        CScrollView::OnInitialUpdate();
        CScrollerDoc* doc = GetDocument();
        CSize sizeTotal, sizePg;
        CRect r;
        GetClientRect(&r);
        CDC *pDC=GetDC();
        TEXTMETRIC tm;
        pDC->GetTextMetrics(&tm);
        lineht=tm.tmHeight+tm.tmExternalLeading;
        sizeTotal.cx = 100;

```

```

        sizeTotal.cy = sizeof(doc->list)/
            sizeof(doc->list[0])+
            r.Height()-r.Height()/lineht;
        sizePg.cx=10;
        sizePg.cy=r.Height()/lineht;
        SetScrollSizes(MM_TEXT, sizeTotal, sizePg, CSize(10,1));
#endif
}

////////////////////
// CScrollerView printing

BOOL CScrollerView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}

void CScrollerView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CScrollerView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}

////////////////////
// CScrollerView diagnostics

#ifdef _DEBUG
void CScrollerView::AssertValid() const
{
    CScrollView::AssertValid();
}

void CScrollerView::Dump(CDumpContext& dc) const
{
    CScrollView::Dump(dc);
}

CScrollerDoc* CScrollerView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CScrollerDoc)));
    return (CScrollerDoc*)m_pDocument;
}
#endif // _DEBUG

////////////////////
// CScrollerView message handlers

void CScrollerView::OnSize(UINT nType, int cx, int cy)
{
    #if METHOD==0
        int dummy;
        CScrollView::OnSize(nType, cx, cy);
        CSize sizeTotal, pg, dummysize;
        CRect r;
    #endif
}

```

```

        GetClientRect(&r);
        pg=CSize(10,r.Height());
        GetDeviceScrollSizes(dummy,sizeTotal,dummysize,dummysize);
        SetScrollSizes(MM_TEXT, sizeTotal,pg,CSize(10,lineht));
    #endif
    #if METHOD==1
        CScrollView::OnSize(nType, cx, cy);
        CSize sizeTotal,pg;
        CRect r;
        CScrollerDoc *doc=GetDocument();
        GetClientRect(&r);
        pg=CSize(10,r.Height()/lineht);
        sizeTotal.cx=100;
        sizeTotal.cy=sizeof(doc->list)/sizeof(doc->list[0])+r.Height()
-r.Height()/
        lineht;
        SetScrollSizes(MM_TEXT, sizeTotal,pg,CSize(10,1));
    #endif
}

void CScrollerView::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar*
pScrollBar)
{

    CScrollView::OnVScroll(nSBCode, nPos, pScrollBar);
    #if METHOD==1 // Must undo scrolling behavior
        InvalidateRect(NULL);
    #endif
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US



SEARCH

ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)



BROWSE

BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

About CEditView

The **CEditView** class seems like it would be one of the most useful tools in a MFC programmer's arsenal. What could be better than a little text editor window that does all the work for you? It's too bad that several design flaws add up to make the control much less than perfect. First and foremost, the view stores the data internally (because it really just wraps an ordinary edit control). That means that the view doesn't work well with a document. Sure, you have to supply a document, but its only purpose is to pass serialization calls to the view.

The other problems that **CEditView** suffers from are the same ones that the internal Windows edit control has: one font display and severe memory limitations under Windows 95. These last two problems are not so easy to solve (unless you switch to the **CRichEditView** you'll see later in this chapter). However, you may be able to make the view and document work together better, depending on your application.

The basic idea is simple: Intercept all input to the view and send it to the document instead. Then the document distributes the input to all views attached to it. Of course, it isn't quite that simple. You also have to manage the selection (and caret) so that the characters show up in the same place in each view.

Fixing CEditView

Listings 4.5 and 4.6 show an edit view and document that implement this strategy. When the view receives a printable character (via **OnChar**), it passes it to the document and does not call the base class **OnChar**. Later, the document will pass the character to the view's **DoChar** routine. That function calls the base class to allow the view to respond to the keystroke.

The **OnChar** routine in the document simply enumerates the list of views, sets each view's selection to be the same as the current view's selection, and sends the character to **DoChar**. The only tricky part is that the document must set and reset the selection of all the subordinate views (that is, the views that don't have the focus). Of course, this assumes that the views are always synchronized.

The only chance the views have to desynchronize is when you create a new window. The new window will not have any of the contents of the previous window. To solve this, the document object notices when the view list changes. Each view has a custom **initialized** flag that is **FALSE** when the program creates the view. If the document finds that a view hasn't been initialized, it sets its text to the same as the first view in the list. You can find the code in the **OnChangedViewList** function in Listing 4.6.

Listing 4.5 A CEditView that cooperates with its document.

```

// multeditView.cpp : implementation of the CMulteditView class
//

#include "stdafx.h"
#include "multedit.h"

#include "multeditDoc.h"
#include "multeditView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CMulteditView

IMPLEMENT_DYNCREATE(CMulteditView, CEditView)

BEGIN_MESSAGE_MAP(CMulteditView, CEditView)
   //{{AFX_MSG_MAP(CMulteditView)
    ON_WM_CHAR()
   //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CEditView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CEditView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CEditView::OnFilePrintPreview)
END_MESSAGE_MAP()

////////////////////
// CMulteditView construction/destruction

CMulteditView::CMulteditView()
{
    // Document will set this TRUE after setting our
    // text buffer to match our sister edit controls
    initialized=FALSE;
}

CMulteditView::~CMulteditView()
{
}

BOOL CMulteditView::PreCreateWindow(CREATESTRUCT& cs)
{
    BOOL bPreCreated = CEditView::PreCreateWindow(cs);
    cs.style &= ~(ES_AUTOHSCROLL|WS_HSCROLL);    // Enable word-
wrapping

    return bPreCreated;
}

////////////////////
// CMulteditView drawing

void CMulteditView::OnDraw(CDC* pDC)
{

```

```

        CMulteditDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
    }

    //////////////////////////////////////
    // CMulteditView printing

    BOOL CMulteditView::OnPreparePrinting(CPrintInfo* pInfo)
    {
        // default CEditView preparation
        return CEditView::OnPreparePrinting(pInfo);
    }

    void CMulteditView::OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo)
    {
        // Default CEditView begin printing
        CEditView::OnBeginPrinting(pDC, pInfo);
    }

    void CMulteditView::OnEndPrinting(CDC* pDC, CPrintInfo* pInfo)
    {
        // Default CEditView end printing
        CEditView::OnEndPrinting(pDC, pInfo);
    }

    //////////////////////////////////////
    // CMulteditView diagnostics

    #ifdef _DEBUG
    void CMulteditView::AssertValid() const
    {
        CEditView::AssertValid();
    }

    void CMulteditView::Dump(CDumpContext& dc) const
    {
        CEditView::Dump(dc);
    }

    CMulteditDoc* CMulteditView::GetDocument() // non-debug version is inline
    {
        ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMulteditDoc)));
        return (CMulteditDoc*)m_pDocument;
    }
    #endif // _DEBUG

    //////////////////////////////////////
    // CMulteditView message handlers

    // Pass characters to document without processing
    void CMulteditView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
    {
        CMulteditDoc *doc=GetDocument();
        doc->OnChar(this,nChar,nRepCnt,nFlags);
    }

    // Document sends us characters here, so process them
    void CMulteditView::DoChar(UINT nChar, UINT nRepCnt, UINT nFlags)
    {

```

```

        CEditView::OnChar(nChar, nRepCnt, nFlags);
    }

```

Listing 4.6 A document for the new CEditView.

```

// multeditDoc.cpp : implementation of the CMulteditDoc class
//

#include "stdafx.h"
#include "multedit.h"

#include "multeditDoc.h"
#include "multeditView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CMulteditDoc

IMPLEMENT_DYNCREATE(CMulteditDoc, CDocument)

BEGIN_MESSAGE_MAP(CMulteditDoc, CDocument)
    //{AFX_MSG_MAP(CMulteditDoc)
        // NOTE - the Class Wizard will add and remove mapping
        macros here.
        // DO NOT EDIT what you see in these blocks of generated
        code!
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////
// CMulteditDoc construction/destruction

CMulteditDoc::CMulteditDoc()
{
    // TODO: add one-time construction code here
}

CMulteditDoc::~CMulteditDoc()
{
}

BOOL CMulteditDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())

        return FALSE;
    return TRUE;
}

////////////////////
// CMulteditDoc serialization

void CMulteditDoc::Serialize(CArchive& ar)
{

```

```

        // CEditView contains an edit control which handles all
        serialization
    //      ((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);
        CString s="Unimplemented!";
        WriteString(s);
    }

    //////////////////////////////////////
    // CMulteditDoc diagnostics

#ifdef _DEBUG
void CMulteditDoc::AssertValid() const
{
    CDocument::AssertValid();
}

void CMulteditDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}
#endif // _DEBUG

    //////////////////////////////////////
    // CMulteditDoc commands

    // Process a key
void CMulteditDoc::OnChar(CMulteditView *view0,UINT nChar,
    UINT nRepCnt, UINT nFlags)
{
    int ct;
    CMulteditView *view;
    POSITION pos;
    int st,en,st0,en0;
    view0->GetEditCtrl().GetSel(st,en);
    pos=GetFirstViewPosition();
    // Send to all views
    while (view=(CMulteditView *)GetNextView(pos))
    {
        ct=nRepCnt;
        while (ct-)
        {
            int offset=1;
            if (nChar==8) offset=-1; // backspace
            view->GetEditCtrl().GetSel(st0,en0);
            if ((st==st0&&st!=en0)|| (st>st0 && st<=en0)) en0+=offset;
            if (st<st0) st0+=offset,en0+=offset;
            view->GetEditCtrl().SetSel(st,en); // set position
            view->DoChar(nChar,1,nFlags); // do it
            if (view!=view0) view->GetEditCtrl().SetSel(st0,en0); // set
            position back
        }
    }
}

    // Write a string to all views (very similar to keystroke code)
void CMulteditDoc::WriteString(char const * s)
{
    CMulteditView *view;
    POSITION pos;

```

```

        pos=GetFirstViewPosition();
        while (view=(CMulteditView *)GetNextView(pos))
        {
            int n=view->GetEditCtrl().
                GetWindowTextLength();
            view->GetEditCtrl().SetSel(n,n);
            view->GetEditCtrl().ReplaceSel(s);
        }
    }

    // If a view is added, we have to update its buffer!
    void CMulteditDoc::OnChangedViewList()
    {
        POSITION pos=GetFirstViewPosition();
        CMulteditView *view;
        view=(CMulteditView *)GetNextView(pos);
        if (view)
        {
            CString txt;
            view->GetWindowText(txt); // assume 1st view is OK
            while (view=(CMulteditView *)
                GetNextView(pos))
                if (!view->initialized) // if new view
                {
                    view->SetWindowText(txt); // set it
                    view->initialized=TRUE; // forget it
                }
        }
        CDocument::OnChangedViewList();
    }
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

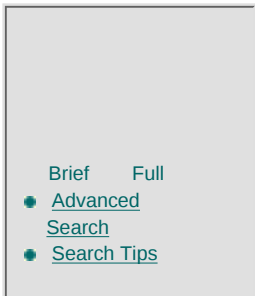
[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE



BROWSE BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

CEditView And Splitters

It would be nice to be able to use **CEditView** and splitter windows together. However, this isn't as easy as it sounds because both the edit view and the splitter window want to supply scroll bars. If you turn off the splitter window scroll bars, you lose the default user interface for splitting the window. However, if you supply an alternate method to split the window, you can use splitters by simply removing the scroll styles from the splitter window's **Create** call.

Listing 4.7 shows a child frame that includes a splitter with no scroll bars; it works with the document and view from the previous section. Figure 4.4 shows the entire program in action. Notice that the scroll bars belong to the edit control. Therefore, you have to split the window by sending commands from the program. That's the purpose of the Window|Split menu option. It tracks the current split state and calls **SplitRow** or **DeleteRow** to control the splitter window. You might consider calling **DoKeyboardSplit** instead if you want to allow a mouse interface.

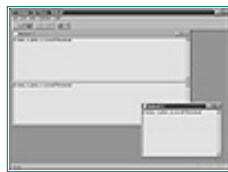


Figure 4.4 Multiple edit controls.

Listing 4.7 Splitting the edit view.

```
// ChildFrm.cpp : implementation of the CChildFrame class
//

#include "stdafx.h"
#include "multedit.h"

#include "ChildFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
```

```

#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////
// CChildFrame

IMPLEMENT_DYNCREATE(CChildFrame, CMDIChildWnd)

BEGIN_MESSAGE_MAP(CChildFrame, CMDIChildWnd)
   //{{AFX_MSG_MAP(CChildFrame)
    ON_COMMAND(IDM_SPLIT, OnSplit)
    ON_UPDATE_COMMAND_UI(IDM_SPLIT, OnUpdateSplit)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

//////////
// CChildFrame construction/destruction

CChildFrame::CChildFrame()
{
    split=FALSE;
}

CChildFrame::~CChildFrame()
{
}

BOOL CChildFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CMDIChildWnd::PreCreateWindow(cs);
}

//////////
// CChildFrame diagnostics

#ifdef _DEBUG
void CChildFrame::AssertValid() const
{
    CMDIChildWnd::AssertValid();
}

void CChildFrame::Dump(CDumpContext& dc) const
{
    CMDIChildWnd::Dump(dc);
}

#endif // _DEBUG

//////////
// CChildFrame message handlers

```

```

BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext*
pContext)
{
    return splitter.Create(this,2,1, CSize(1,1),pContext,WS_CHILD | WS
_VISIBLE |
    SPLS_DYNAMIC_SPLIT);
}

void CChildFrame::OnSplit()
{
    CRect r;
    GetClientRect(r);
    if (split)
        splitter.DeleteRow(1);
    else
        splitter.SplitRow(r.Height()/2);
    split=!split;
}

void CChildFrame::OnUpdateSplit(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck(split);
    pCmdUI->Enable();
}

```

About CRichEditView

Microsoft's answer to the problems with **CEditView** is to use **CRichEditView** instead. This text editor control uses the common rich edit control, so it doesn't have severe memory limitations, it supports OLE, and it can support formatted text. It also behaves better (but not perfectly) as a view.

Figure 4.5 shows a very simple word processor implemented with **CRichEditView**. Even though App Wizard wrote most of it, it has full support for OLE, multiple fonts, and paragraph formatting. In addition, it handles ASCII or RTF files.



Figure 4.5 A simple word processor.

The easiest way to write a program like EZWP is to run App Wizard. Select full OLE support (if you want it) and set the other options according to your desires. At the last screen (Figure 4.6), select your view class and change the base class from **CView** to **CRichEditView**. App Wizard will also automatically change your document class.



Figure 4.6 Selecting **CRichEditView**.

The program that results will look much like EZWP, but it won't have any provisions for changing character or paragraph formatting. Adding these features isn't very hard. A quick look at Listing 4.8 shows that most of the features just require hooking up built-in handlers. For example, to set characters to bold, you just call **OnCharEffect** with a few simple arguments. There is also an

OnUpdateCharEvent function provided to set the status of menu items and tool bar buttons. Similar functions handle the paragraph formatting. These functions are part of **CRichEditView**. You only need to hook them up to a menu item, a key command, or a tool bar button.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Listing 4.8 Using CRichEditView.

```
// ezwpView.cpp : implementation of the CEzwpView class
//

#include "stdafx.h"
#include "ezwp.h"

#include "ezwpDoc.h"
#include "CntrItem.h"
#include "ezwpView.h"

#include "CharPage.h"
#include "LinePage.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CEzwpView

IMPLEMENT_DYNCREATE(CEzwpView, CRichEditView)

BEGIN_MESSAGE_MAP(CEzwpView, CRichEditView)
//{{AFX_MSG_MAP(CEzwpView)
ON_COMMAND(ID_CANCEL_EDIT_SRVR, OnCancelEditSrvr)
ON_COMMAND(ID_CHARACTER_BOLD, OnCharacterBold)
ON_UPDATE_COMMAND_UI(ID_CHARACTER_BOLD, OnUpdateCharacterBold)
ON_COMMAND(ID_PARAGRAPH_CENTER, OnParagraphCenter)
ON_UPDATE_COMMAND_UI(ID_PARAGRAPH_CENTER, OnUpdateParagraphCenter)
ON_COMMAND(ID_CHARACTER_ITALIC, OnCharacterItalic)
ON_UPDATE_COMMAND_UI(ID_CHARACTER_ITALIC, OnUpdateCharacterItalic)
```

Brief Full

- Advanced
- Search
- Search Tips

BROWSE
BY TOPIC

```

        ON_COMMAND(ID_PARAGRAPH_LEFT, OnParagraphLeft)
        ON_UPDATE_COMMAND_UI(ID_PARAGRAPH_LEFT, OnUpdateParagraphLeft)
        ON_COMMAND(ID_PARAGRAPH_RIGHT, OnParagraphRight)
        ON_UPDATE_COMMAND_UI(ID_PARAGRAPH_RIGHT, OnUpdateParagraphRight)
        ON_COMMAND(ID_CHARACTER_FONT, OnCharacterFont)
        ON_COMMAND(IDM_Statistics, OnStatistics)
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CRichEditView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CRichEditView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CRichEditView::OnFilePrintPreview)
END_MESSAGE_MAP()

////////////////////
// CEzwpView construction/destruction

CEzwpView::CEzwpView()
{
    // TODO: add construction code here
}

CEzwpView::~~CEzwpView()
{
}

BOOL CEzwpView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CRichEditView::PreCreateWindow(cs);
}

void CEzwpView::OnInitialUpdate()
{
    CRichEditView::OnInitialUpdate();
    CHARFORMAT cf;
    memset(&cf, 0, sizeof(cf));
    cf.cbSize=sizeof(cf);
    cf.dwMask=CFM_FACE|CFM_SIZE|CFM_BOLD|CFM_ITALIC;
    cf.yHeight=400; // 20 points
    cf.bCharSet=DEFAULT_CHARSET;
    cf.bPitchAndFamily=DEFAULT_PITCH|FF_DONTCARE;
    strcpy(cf.szFaceName, "Times New Roman");
    SetCharFormat(cf);
}

////////////////////
// CEzwpView printing

BOOL CEzwpView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}

```

```

////////////////////
// OLE Server support

// The following command handler provides the standard keyboard
// user interface to cancel an in-place editing session. Here,
// the server (not the container) causes the deactivation.
void CEzwpView::OnCancelEditSrvr()
{
    GetDocument()->OnDeactivateUI(FALSE);
}

////////////////////
// CEzwpView diagnostics

#ifdef _DEBUG
void CEzwpView::AssertValid() const
{
    CRichEditView::AssertValid();
}

void CEzwpView::Dump(CDumpContext& dc) const
{
    CRichEditView::Dump(dc);
}

CEzwpDoc* CEzwpView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CEzwpDoc)));
    return (CEzwpDoc*)m_pDocument;
}
#endif // _DEBUG

////////////////////
// CEzwpView message handlers

void CEzwpView::OnCharacterBold()
{
    OnCharEffect(CFM_BOLD,CFE_BOLD);
}

void CEzwpView::OnUpdateCharacterBold(CCmdUI* pCmdUI)
{
    OnUpdateCharEffect(pCmdUI,CFM_BOLD,CFE_BOLD);
}

void CEzwpView::OnParagraphCenter()
{
    OnParaAlign(PFA_CENTER);
}

void CEzwpView::OnUpdateParagraphCenter(CCmdUI* pCmdUI)
{
    OnUpdateParaAlign(pCmdUI,PFA_CENTER);
}

```

```

}

void CEzwpView::OnCharacterItalic()
{
    OnCharEffect(CFM_ITALIC, CFE_ITALIC);
}

void CEzwpView::OnUpdateCharacterItalic(CCmdUI* pCmdUI)
{
    OnUpdateCharEffect(pCmdUI, CFM_ITALIC, CFE_ITALIC);
}

void CEzwpView::OnParagraphLeft()
{
    OnParaAlign(PFA_LEFT);
}

void CEzwpView::OnUpdateParagraphLeft(CCmdUI* pCmdUI)
{
    OnUpdateParaAlign(pCmdUI, PFA_LEFT);
}

void CEzwpView::OnParagraphRight()
{
    OnParaAlign(PFA_RIGHT);
}

void CEzwpView::OnUpdateParagraphRight(CCmdUI* pCmdUI)
{
    OnUpdateParaAlign(pCmdUI, PFA_RIGHT);
}

void CEzwpView::OnCharacterFont()
{
    CHARFORMAT cf;
    cf=GetCharFormatSelection();
    CFontDialog dlg(cf, CF_FORCEFONTEXIST|CF_INITTTOLOGFONTSTRUCT|CF_
SCREENFONTS);
    if (dlg.DoModal()==IDOK)
    {
        dlg.GetCharFormat(cf);
        SetCharFormat(cf);
    }
}

void CEzwpView::OnStatistics()
{
    CCharPage cp;
    CLinePage lp;
    CString tmp;

```

```

CPropertySheet sheet("Statistics");
sheet.AddPage(&cp);
sheet.AddPage(&lp);
tmp.Format("%d", GetTextLength());
cp.m_count=tmp;
tmp.Format("%d", GetRichEditCtrl().GetLineCount());
lp.m_count=tmp;
sheet.DoModal();
}

```

At first glance, it might appear difficult to provide for font selection. **CFontDialog** (the standard font dialog) returns font information in a **LOGFONT** structure and via member function calls. However, the **CRichEditView** expects the same information in a **CHARFORMAT** structure. You could read the current formatting, convert it to a **LOGFONT**, bring up the font dialog, and then translate the **LOGFONT** back into a **CHARFORMAT**, but that's a lot of trouble.

Luckily, Microsoft built support for **CRichEditView** into the **CFontDialog** class. They just forgot to document it. **CFontDialog** has two constructors: the documented one and another that is exactly the same except that the first argument is a **CHARFORMAT** structure pointer. The other piece to the puzzle is the undocumented **GetCharFormat** call. This returns the font information in the format that **CRichEditView** can understand. Here's all the code you need:

```

void CEzwpView::OnCharacterFont()
{
    CHARFORMAT cf;
    cf=GetCharFormatSelection();
    CFontDialog dlg(cf, CF_FORCEFONTEXIST|CF_INITTOLOGFONTSTRUCT|CF_
SCREENFONTS);
    if (dlg.DoModal()==IDOK)
    {
        dlg.GetCharFormat(cf);
        SetCharFormat(cf);
    }
}

```

I don't understand why this isn't documented. It makes using a font dialog in conjunction with a **CRichEditView** trivial.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Working With Owner-Draw Controls

In traditional Windows programming, owner-draw controls are inconvenient at best. What's an owner-draw control? It's a button, menu, static, list box, combo box, or similar item that has no default appearance. When Windows wants to display the item (or otherwise manipulate it), it sends a message to the control's parent window. The parent window then draws the object in the correct state.

Traditionally, programs used owner-draw buttons to provide buttons that contained bitmaps. However, modern versions of Windows support pictures in buttons with no special effort. On the other hand, suppose you want a button that shows a graphic that changes. For example, it might contain a gas gauge with a needle somewhere between empty and full. Then, you will probably turn to owner-draw controls.

Owner-draw controls have several problems. First, you are responsible for drawing every aspect of the control. For a button, that means drawing the button as it appears normally, when it has the focus, when the user presses it, and so forth. Of course, this is also a feature because you can exercise as much control as you like.

The most significant problem with owner-draw controls is that they are not modular. Imagine that you have a dialog box that contains three owner-draw buttons and an owner-draw list box. All four controls will ask the dialog box to draw them using the same message. The dialog box code has to examine data that Windows includes with the message to determine what to draw.

Suppose that one of the three buttons has a pulsating DNA helix in it and someone wants to reuse that button in another program. You'll have to pull out all the code relating to the button from the dialog. That includes any code that sets up a timer and handles **WM_TIMER** messages to create the pulsating effect. Then, the new program will somehow have to incorporate that code. Because all the controls use the same message, this can get very ugly.

The MFC Solution: Self-Draw

MFC solves this problem by using a variation on owner-draw controls called *self-draw* controls. The idea is simple: Have parent windows reflect owner-draw messages to the window that generates them. In the example I used above, suppose Windows wants to draw the DNA helix button. It sends a message to the parent window (the dialog, derived from **CDialog**). The default handlers reflect the message back to the helix button.

Presumably, the helix button is a special subclass of **CButton** that has the handlers required to do the drawing. You still have to handle the variations in appearance, but you have a modular component that you can easily reuse. To move the helix button to another project, you need only move the **CHelixButton** class (or whatever you called it).

Other Solutions

Although modern buttons can accept bitmaps or icons, they look exactly like ordinary buttons that may not be what you want (see Figure 4.7). MFC can help with its built-in class **CBitmapButton**. **CBitmapButton** is little more than a built-in self-draw button that MFC implements. To use this button, you must supply one or more bitmaps in your resource script. The bitmaps must not use numeric IDs. Each bitmap specifies the appearance of the button in some state. You must provide the bitmap for the button's normal appearance. However, the other bitmaps are optional. If you supply them, fine. If you don't, MFC will show something (although you may not like it since it won't appear any different than before).



Figure 4.7 Various controls.

Suppose that you have a button bearing the caption "OK". You must provide bitmap OKU (the normal image). You may also provide OKD (down), OKF (focused), and OKX (disabled). Why provide them when MFC doesn't require it? You might like a completely different image for some states. For example, what if you wanted a thumbs-up for OKD, but a big X for OKX. You'd need to provide your own OKX bitmap for that.

Tip: Resource IDs

Windows stores resources using an ID. The ID is usually numeric. When you name a resource (like a bitmap, an icon, or a menu), Visual C++

automatically relates the name to a number and generates a **#define** statement in your RESOURCE.H file. This allows you to use the name (which is case-sensitive) in place of the automatically generated number.

However, resource IDs can also be case-insensitive strings. This isn't as efficient as numbers, but it is possible. However, because Visual C++ automatically converts your strings into numbers, you might wonder how to get a string ID. The answer is to place the string in double quotes. This prevents Visual C++ from converting the string.

CBitmapButton requires string IDs, as it determines what bitmap to use by loading the button's caption, appending a letter, and using that string to look up the ID. If Visual C++ converted that ID to a number, it would not be found.

If you don't want to use this scheme to relate your images with the button, you can call **SetBitmap** to set each bitmap individually. Then you can use string IDs or the more common numeric IDs.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

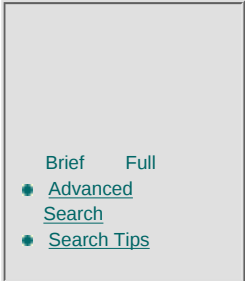
[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE



BROWSE BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97



Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Listing 4.9 contains a simple program that implements a standard bitmap button and a **CBitmapButton** (and some other self-draw and related controls) in a **CFormView**. The standard button uses the Bitmap style. You may notice that although you can set the Bitmap style at design time, you can't set the actual bitmap. To do that, you call **SetBitmap** (see the **OnInitialUpdate** member in Listing 4.9). The example code uses a DDX variable to obtain a **CButton** variable that corresponds to the button.

Listing 4.9 Customized controls.

```
// buttonsView.cpp : implementation of the CButtonsView class
//
```

```
#include "stdafx.h"
#include "buttons.h"
#include "odcombo.h"
#include "selfmenu.h"
#include "buttonsDoc.h"
#include "buttonsView.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
////////////////////////////////////
// CButtonsView
```

```
IMPLEMENT_DYNCREATE(CButtonsView, CFormView)
```

```
BEGIN_MESSAGE_MAP(CButtonsView, CFormView)
   //{{AFX_MSG_MAP(CButtonsView)
        ON_BN_CLICKED(IDC_STDBTN, OnStdbtn)
        ON_COMMAND(ID_SELFDRAW, OnSelfdraw)
        ON_COMMAND(ID_BITMAP, OnBitmap)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```

//////////
// CButtonsView construction/destruction

CButtonsView::CButtonsView()
    : CFormView(CButtonsView::IDD)
{
    //{AFX_DATA_INIT(CButtonsView)
    // NOTE: the Class Wizard will add member initialization
here
    //}}AFX_DATA_INIT
    // TODO: add construction code here
}

CButtonsView::~CButtonsView()
{
}

void CButtonsView::DoDataExchange(CDataExchange* pDX)
{
    CFormView::DoDataExchange(pDX);
    //{AFX_DATA_MAP(CButtonsView)
    DDX_Control(pDX, IDC_ODCOMBO, m_odcombo);
    DDX_Control(pDX, IDC_STDBTN, m_stdbtn);
    //}}AFX_DATA_MAP
}

BOOL CButtonsView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the window class or styles here by modifying
    // the CREATESTRUCT cs

    return CFormView::PreCreateWindow(cs);
}

//////////
// CButtonsView diagnostics

#ifdef _DEBUG
void CButtonsView::AssertValid() const
{
    CFormView::AssertValid();
}

void CButtonsView::Dump(CDumpContext& dc) const
{
    CFormView::Dump(dc);
}

CButtonsDoc* CButtonsView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CButtonsDoc)));
    return (CButtonsDoc*)m_pDocument;
}
#endif // _DEBUG

//////////
// CButtonsView message handlers

```

```

// Here's where all the work takes place
void CButtonsView::OnInitialUpdate()
{
    CFormView::OnInitialUpdate();
    // load bitmap for button (if not already done)
    if (bm.m_hObject==NULL)
        bm.LoadBitmap(IDB_BOMB);
    // Set standard bitmap button
    m_stdbtn.SetBitmap(bm);
    // Auto load MFC bitmap button
    if (m_MFCBtn.m_hWnd==NULL)
        m_MFCBtn.AutoLoad(IDC_MFCBTN,this);
    // Real owner/self draw button attach (could have used DDX)
    if (m_odbtn.m_hWnd==NULL)
        m_odbtn.SubclassDlgItem(IDC_ODBTN,this);
    // Owner/self draw static attach (could have used DDX)
    if (m_odstatic.m_hWnd==NULL)
        m_odstatic.SubclassDlgItem(IDC_ODSTATIC,this);
    // Set style to OWNERDRAW
    SetWindowLong(m_odstatic.m_hWnd,GWL_STYLE,
        m_odstatic.GetStyle()|SS_OWNERDRAW);
    // Used DDX to attach combo box; could have used SubclassDlgItem instead.
    // Since the box already exists, we have to compute the item height
    // and set it now, so we will...
    MEASUREITEMSTRUCT mis;
    m_odcombo.MeasureItem(&mis);
    m_odcombo.SetItemHeight(0,mis.itemHeight);
    // Add some items, the strings don't really matter as we use the
    // item data in this case.
    m_odcombo.ResetContent();
    int n=m_odcombo.AddString("0");
    m_odcombo.SetItemData(n,0);
    n=m_odcombo.AddString("1");
    m_odcombo.SetItemData(n,1);
    // Now for a menu
    CMenu *mainmenu=AfxGetApp()->m_pMainWnd->GetMenu(); // Get main
menu
    CMenu *tempmenu=mainmenu->GetSubMenu(1); // find our sub menu
    m_selfMenu.Attach(tempmenu->m_hMenu); // attach it to special
class
    // set owner draw flag
    m_selfMenu.ModifyMenu(ID_SELFDRAW,MF_BYCOMMAND|MF_OWNERDRAW,
        ID_SELFDRAW,(char *)1);
    // Standard bitmap Menu
    tempmenu=mainmenu->GetSubMenu(2);
    tempmenu->ModifyMenu(ID_BITMAP,MF_BYCOMMAND|MFT_BITMAP,
        ID_BITMAP,(char *)bm.m_hObject);
}

void CButtonsView::OnStdbtn()
{
    m_MFCBtn.EnableWindow(!m_MFCBtn.IsWindowEnabled());
}

void CButtonsView::OnSelfdraw()
{
    MessageBox("Unimplemented");
}

```

```
void CButtonsView::OnBitmap()  
{  
    MessageBox("Unimplemented");  
}
```

If you prefer to use an icon, you can set the icon style and use **SetIcon** instead. This button has a nice 3D appearance with little effort. One fine point: If you create the bitmap on the stack, the button will appear to be blank. That's because the bitmap goes out of scope, quickly destroying the button's image. The example program uses a member variable to make sure the bitmap doesn't go out of scope. It might be a good idea to destroy the bitmap when you are done, or to cache it in a static variable that all instances of the buttons could use.

The **CBitmapButton** looks like an owner-draw control. You need only to set the Owner Draw style and the caption. The button provides three bitmaps: the normal bitmap, one for the down state, and one for the disabled state. The **AutoLoad** member function takes care of associating the bitmaps with the button. When you click the standard button, it disables this button so you can see the disabled state.

You'll notice that the MFC button doesn't look as nice as the standard button. On the other hand, the MFC button is probably more flexible than the standard button. If you wanted to draw all the 3D effects, you could make the MFC button look better, but it would be more work. On the other hand, you can create unusual effects with the MFC button.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

 **SEARCH**
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97



Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Using Self-Draw Controls

Self-draw controls under MFC are fairly simple. The trick is overriding the correct functions in your derived class (see Table 4.4). Buttons (see Listing 4.10) and static controls (Listing 4.11) are simplest with only one override (**DrawItem**). This call receives a **DRAWITEMSTRUCT** (see Table 4.5). This structure tells you how you should draw the item (and which item you should draw—remember, in plain old Windows, that isn't obvious).

Table 4.4 Implementing self-draw controls.

	DrawItem	MeasureItem	CompareItem	DeleteItem
Button	ë			
Menu	ë	ë		
List Box	ë	ë	ë	ë
Combo Box	ë	ë	ë	ë
Static	ë			

Table 4.5 DRAWITEMSTRUCT.

Member	Description	Type
CtlType	Type of control (ODT_BUTTON, ODT_MENU, etc.)	In
CtlID	ID of button, combo box, list box, or static	In
itemAction	Command (ODA_DRAWENTIRE, ODA_DRAWFOCUS, or ODA_SELECT)	In
itemState	State (ODS_CHECKED, ODS_GRAYED, and so on)	In
hwndItem	HWND of control or HMENU of menu	In
hDC	Device context	In
rcItem	Bounding rectangle (Warning: menus don't clip to the rectangle)	In
itemData	User-defined data for menu, combo box, or list box (Note: not string)	In

Of course, you aren't obligated to draw anything. Self-draw buttons with no drawing are handy for making areas of a window clickable without having to write hit-testing code. While this is an unorthodox use of a self-draw button, it illustrates that you can draw as much or as little as you like.

Notice that the device context, window handle, and other quantities are not MFC objects, but just regular handles. You'll have to convert these before using them (for example, you might use **CDC::FromHandle** for the device context).

Tip: DrawItem Vs. OnDrawItem

Don't get confused between calls like **DrawItem** and **OnDrawItem** (or the other similar pairs of calls). **OnDrawItem** is the actual message handling function that receives **WM_DRAWITEM** messages. This function belongs in the parent window for the owner-draw control. The default code in **CWnd** calls the correct control's **DrawItem** function. So these calls do the same thing in different places: **OnDrawItem** is in the parent window and **DrawItem** is in the child. The same goes for **MeasureItem** and **OnMeasureItem**, as well as the other pairs of calls.

Listing 4.10 A self-draw button.

```
// ODButton.cpp : implementation file
//

#include "stdafx.h"
#include "buttons.h"
#include "ODButton.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// COButton

COButton::COButton()
{
}

COButton::~COButton()
{
}

BEGIN_MESSAGE_MAP(COButton, CButton)
    //{AFX_MSG_MAP(COButton)
    // NOTE - the Class Wizard will add and remove mapping macros here
    //}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////
// COButton message handlers

void COButton::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
    CDC *dc=CDC::FromHandle(lpDrawItemStruct->hDC);
    CBrush *btnface;
    CString caption;
    GetWindowText(caption);
    btnface=CBrush::FromHandle(GetSysColorBrush(COLOR_BTNFACE));
```

```

        if (lpDrawItemStruct->itemState&ODS_SELECTED==ODS_SELECTED)
            btnface=(CBrush *)dc->SelectStockObject(WHITE_BRUSH);
        else
            btnface=dc->SelectObject(btnface);
        dc->Ellipse(&lpDrawItemStruct->rcItem);
        dc->SetBkMode(TRANSPARENT);
        dc->DrawText(caption, -1, &lpDrawItemStruct->rcItem,
            DT_SINGLELINE|DT_CENTER|DT_VCENTER);
        dc->SelectObject(btnface);
    }

```

Listing 4.11 A self-draw static.

```

// ODStatic.cpp : implementation file
//

#include "stdafx.h"
#include "buttons.h"
#include "ODStatic.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CODStatic

CODStatic::CODStatic()
{
}

CODStatic::~CODStatic()
{
}

BEGIN_MESSAGE_MAP(CODStatic, CStatic)
    //{AFX_MSG_MAP(CODStatic)
// ON_WM_DRAWITEM()
    ON_WM_DRAWITEM_REFLECT()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////
// CODStatic message handlers

void CODStatic::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
    CDC *dc=CDC::FromHandle(lpDrawItemStruct->hDC);
    dc->Ellipse(&lpDrawItemStruct->rcItem);
}

```

Static controls pose a special problem because the current version of MFC doesn't understand that statics can be self-drawn (an owner-draw static in older versions of Windows did not exist). The resource editor doesn't even show owner-draw as an option for static controls. However, you can set the style at runtime (as Listing 4.9 does). Although MFC reflects the owner-draw message to the static, the default implementation does not handle it. You'll have to manually add an **ON_WM_DRAWITEM_REFLECT**

message map macro to your subclass of **CStatic**. You can see the details in Listing 4.11.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Self-Draw List And Combo Boxes

List boxes and combo boxes are a bit more complicated. They come in two flavors: fixed-height and variable-height. In a fixed-height box, each item is the same size. Variable-height boxes can handle items that are different sizes.

The **MeasureItem** call uses the **MEASUREITEMSTRUCT** (see Table 4.6) to find the measurements from your program. Notice that for a fixed-height box, Windows only makes this call once right after it creates the box. Variable-height boxes make the call each time they need to know the height of an item.

Table 4.6 MEASUREITEMSTRUCT.

Member	Description	Type
CtlType	Type of control	In
CtlID	ID of control (not used for menus)	In
itemID	Menu ID or combo/list box item ID (if box is variable-height)	In
itemWidth	Width of item	Out
itemHeight	Height of item	Out
itemData	User-defined data for combo or list box	In

This poses a problem if you are using the box in a dialog. The dialog manager creates the box and Windows measures the items right away. However, you haven't had the chance to attach your MFC class to the box yet. Therefore, you never have a chance to set the height of the item.

There are two solutions to this problem. First, you use a variable-height box and always set the same height for each item. A more efficient solution is to call the box's **SetItemHeight** function soon after creating it. The program in Listing 4.9 uses this approach. Just for completeness, it calls **MeasureItem** directly (although in this case, **MeasureItem** really isn't called from anywhere else).

If you don't allow sorting, you don't need to handle **CompareItem** (and the corresponding **COMPAREITEMSTRUCT** in Table 4.7). If you want to sort your boxes, however, you do need to provide a way to compare items in the same way that **strcmp** compares strings. That is, you return zero if the items are equal, -1 if the first item is less than the second item, and 1 if the second item is smaller.

Table 4.7 COMPAREITEMSTRUCT.

Member	Description	Type
CtlType	Type of control	In
CtlID	ID of control	In
hwndItem	Item's window handle	In
itemID1	First item's index	In
itemData1	First item's user data	In
itemID2	Second item's index	In
itemData2	Second item's user data	In

Sometimes during the course of handling a self-draw box, you might allocate memory or other resources for a particular item. If the program removes that item from the box, you can release any resources you are using. That is the purpose of **DeleteItem** and **DeleteItemStruct** (see Table 4.8). This call is for your information only. If you don't need to know when items are no longer in use, you don't need to supply this function.

Table 4.8 DELETEITEMSTRUCT.

Member	Description	Type
CtlType	Type of control	In
CtlID	ID of control	In
hwndItem	Item's window handle	In
itemID	Item's index	In
itemData	Item's user data	In

Just like buttons, the box's **DrawItem** call is responsible for the entire drawing process. The code in Listing 4.12 is very simplistic, so the combo box doesn't have selected appearances and other niceties. However, it does use two different fonts to display the selections.

Listing 4.12 A self-draw combo box.

```
// ODCCombo.cpp : implementation file
//

#include "stdafx.h"
#include "buttons.h"
#include "ODCCombo.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////

// CODCombo

CODCombo::CODCombo()
{
}

CODCombo::~CODCombo()
{
}
```

```

BEGIN_MESSAGE_MAP(CODCombo, CComboBox)
//{{AFX_MSG_MAP(CODCombo)
// NOTE - the Class Wizard will add and remove mapping macros here.
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

//////////////////////////
// CODCombo message handlers

void CODCombo::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
    CDC *dc=CDC::FromHandle(lpDrawItemStruct->hDC);
    CString txt;
    if (lpDrawItemStruct->itemData==0)
    {
        txt="Fixed";
        dc->SelectStockObject(ANSI_FIXED_FONT);
    }
    else
    {
        txt="Variable";
        dc->SelectStockObject(ANSI_VAR_FONT);
    }
    dc->DrawText(txt, -1,
        &lpDrawItemStruct->rcItem, DT_SINGLELINE|DT_LEFT|DT_VCENTER);
}

void CODCombo::MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct)
{
    CDC *dc=GetDC();
    TEXTMETRIC tm1, tm2;
    dc->SelectStockObject(ANSI_FIXED_FONT);
    dc->GetTextMetrics(&tm1);
    dc->SelectStockObject(ANSI_VAR_FONT);
    dc->GetTextMetrics(&tm2);
    lpMeasureItemStruct->itemWidth=max(tm1.tmAveCharWidth*25,
        tm2.tmAveCharWidth*25);
    lpMeasureItemStruct->itemHeight=max(tm1.tmHeight, tm2.tmHeight);
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

Self-Draw Menus

Unlike most of the other self-draw controls, you can't specify owner-draw menus at design time. Instead, you must set them up at runtime (see Listing 4.13). You only need the **DrawItem** and **MeasureItem** functions, but Class Wizard won't help you because **CMenu** isn't one of Class Wizard's base classes. You'll have to derive the class by hand.

Listing 4.13 A self-draw menu.

```
#include "stdafx.h"
#include "selfmenu.h"

void CSelfMenu::DrawItem(DRAWITEMSTRUCT *dis)
{
    CDC *dc=CDC::FromHandle(dis->hDC);
    CFont *oldfont;
    oldfont=(CFont *)dc->SelectStockObject(ANSI_FIXED_FONT);
    dc->DrawText("Fixed Font",-1,&(dis->rcItem),DT_SINGLELINE|DT_LEFT|DT_VCENTER);
    dc->SelectObject(oldfont);
}

void CSelfMenu::MeasureItem(MEASUREITEMSTRUCT *mis)
{
    // Get some window
    CWnd *w=AfxGetApp()->m_pMainWnd;
    CDC *dc;
    CRect r(0,0,0,0);
    dc=w->GetDC();
    CFont *oldfont;
    oldfont=(CFont *)dc->SelectStockObject(ANSI_FIXED_FONT);
    dc->DrawText("Fixed Font",-1,&r,DT_SINGLELINE|DT_LEFT|DT_VCENTER|BDT_CALCRECT);
    dc->SelectObject(oldfont);
}
```

```

r.InflateRect(5,5); // leave a little room
    mis->itemWidth=r.Width();
    mis->itemHeight=r.Height();
w->ReleaseDC(dc);
}

```

Another problem you'll run into is attaching your menu object to the actual menu to which MFC will reflect owner-draw messages. Your first task is to find the correct submenu that contains the owner-draw menu item (or items). Then, you call **Attach** for that submenu. This implies that one MFC object handles an entire submenu, even if there is more than one self-draw menu item in that submenu.

You can find the code to draw the menu item in Listing 4.13. As usual, it is simplistic and doesn't handle things like disabled menu items, checked menu items, or even selected menu items. Windows does absolutely no drawing for these menu items. You'll have to do all that yourself.

If you just want a bitmap picture in the menu, you can use the **MFT_BITMAP** flag with **ModifyMenu** (see Listing 4.13). MFC no longer documents this, although it is still in the SDK documentation (under **SetMenuItemInfo**, which MFC doesn't directly expose). However, it works fine. You merely specify **MFT_BITMAP** in the call to **ModifyMenu** and pass a bitmap handle as the string data for the menu item. You can also use **SetMenuItemBitmaps** if you want to modify the bitmaps Windows uses for checked and unchecked menu items.

Editing Tree Or List View Items In Dialogs

Tree and list views were one of the most welcome new controls Microsoft introduced along with Windows 95. These super list boxes allow you to use small icons, support hierarchical data, and generally work better than old-fashioned list boxes. MFC supports these with the standard **CListCtrl** and **CTreeCtrl** classes, so there shouldn't be much to say about them here, right?

That would be true if the controls worked well, but there is one surprisingly common case where these controls fail: inside dialog boxes. In all fairness, this isn't an MFC problem—the problem manifests itself with ordinary Windows programming, too. Also, the controls work in dialog boxes as long as you don't try to use the label-editing feature. You've seen label editing before. It is the feature where you can click on an item and transform it into an edit control. Then you can type a new value for the item and press Enter.

Pressing Enter turns out to be the problem. When you press Enter, the dialog box decides you are clicking on the OK button and dismisses itself. Not exactly what you usually have in mind when you are editing a piece of data in a tree view.

The answer is fairly simple, but it isn't easy to generalize. The program in Listing 4.14 and Figure 4.8 (which doesn't really do anything useful) solves the problem by intercepting the dialog's **OnOK** call. Before relinquishing control to the base class, the code finds the control that has the focus. Then it finds the parent window of that control. For a tree (or list) view's edit control, the parent will be the view control itself (not the dialog, as will usually be the case of other controls).

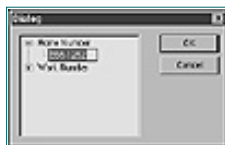


Figure 4.8 A tree control in a dialog.

Listing 4.14 A corrected tree control dialog.

```

// BadDlg.cpp : implementation file
//

#include "stdafx.h"
#include "treebug.h"

```

```

#include "BadDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CBadDlg dialog

CBadDlg::CBadDlg(CWnd* pParent /*=NULL*/)
: CDialog(CBadDlg::IDD, pParent)
{
   //{{AFX_DATA_INIT(CBadDlg)
    // NOTE: the Class Wizard will add member initialization here
   //}}AFX_DATA_INIT
}

void CBadDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(CBadDlg)
    DDX_Control(pDX, IDC_TREE1, m_tree);
   //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CBadDlg, CDialog)
   //{{AFX_MSG_MAP(CBadDlg)
    ON_NOTIFY(TVN_ENDLABELEDIT, IDC_TREE1, OnEndlabeleditTree1)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CBadDlg message handlers

BOOL CBadDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    HTREEITEM item;
    item=m_tree.InsertItem("Home Number");
    m_tree.InsertItem("555-1252",item);
    item=m_tree.InsertItem("Work Number");
    m_tree.InsertItem("555-1253",item);
    return TRUE;
}

void CBadDlg::OnOK()
{
    if (Fix) // only do this if we turned off bug mode
    {
        if (CWnd::GetFocus()->GetParent()==&m_tree) // bogus OK from edit
            control
            {
                CWnd *ectl=CWnd::GetFocus();
                ectl->SendMessage(WM_KEYDOWN,VK_RETURN);
                // As odd as it seems, don't send a KEYUP
            }
    }
}

```

```

        // because by the time it gets there, the edit control
        // will be gone!
        // ect1->SendMessage(WM_KEYUP, VK_RETURN);
        return;
    }
}
CDialog::OnOK();

}

void CBadDlg::OnEndlabeleditTree1(NMHDR* pNMHDR, LRESULT* pResult)
{
    TV_DISPINFO* info = (TV_DISPINFO*)pNMHDR;
    if (info->item.pszText)
    {
        info->item.mask=TVIF_TEXT;
        m_tree.SetItem(&info->item);
    }
    *pResult = 0;
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

When the program detects this case, it simulates an Enter keystroke to the edit control and quietly sends the program about its business. In other cases, it passes control to the base class. The base class code then performs the usual data transfer and dismisses the dialog.

Splitter Windows

Splitter windows have the distinction of being a unique feature of MFC. Usually, if you see a splitter window in a program, you can bet it was written with MFC (although Delphi 3 now supports a form of splitter windows). Even within MFC, splitter windows don't behave like other windows that you use.

Potentially, splitter windows have the capability of making your programs stand out in the crowd. Along with the document/view architecture, splitters allow you to do some interesting things. However, if you try to go beyond the basics, you'll find that splitters are poorly documented and don't work right in many common cases.

What The User Sees

Before you tackle the problem with splitters, consider how they work when they operate correctly. Splitters come in two flavors: dynamic and static. A static splitter takes the form of a small control right next to a scroll bar. When you click and drag the control, it forms a split that you can move with the mouse. On each side of the split is a copy of the current view. After you split the window, you'll usually scroll one side of the split to view some different data (see Figure 4.9). Double-clicking the splitter bar makes the splitter go away (until you split the window again). You can split some windows more

than once, and in either direction. Some programs may limit you to splitting only once, or only in one direction.



Figure 4.9 Nested splitters.

Static splitters are similar, but they are always present. Unlike a dynamic splitter, the views on either side of a static split don't have to be of the same type. As an example, on one side of the split you might have a spreadsheet. On the other side, you might have a pie chart representing the data in the spreadsheet. You can resize static splitters, but you can't get rid of them.

Programming Splitters

If you want to automatically add dynamic splitters to your program, it's easy. The Advanced button in App Wizard has a check box that allows you to create a dynamic splitter for your program. Also, Class Wizard allows you to create frame windows that contain a splitter.

That's what's odd about splitters: They are part of your frame window. Usually, when an MFC program wants to add some function to a window, it derives the window from a different class. For example, if you want your view to scroll, you derive from **CScrollView** instead of **CView**.

Splitters are different. You don't derive any of your classes from a splitter window (**CSplitterWnd**). Instead, you embed an instance of the class in a frame window. The frame window owns the splitter window. Then in the frame's **OnCreateClient** call, you can initialize the splitter to be dynamic or static. By default, the Wizards put prototype code in **OnCreateClient** to create a simple dynamic splitter. When you create a splitter window in a frame, don't call the base class **OnCreateClient**.

If you look up **Create**, **CreateStatic**, and **CreateView** in the MFC help menu, you'll see that the calls are fairly straightforward. The only unusual thing is that some of the calls take a **CCreateContext** structure. If you haven't heard of **CCreateContext**, don't worry. Just look at the arguments for **OnCreateClient**. The final argument is a **CCreateContext** object. For simple splitters, you can pass the parameter on to the splitter functions and you're in business.

Nesting Splitters

The real fun comes when you want to nest one or more splitter windows inside a static splitter. (Think about it: You can't nest splitters inside of a dynamic splitter.) This is important when you want to divide the screen into nonsymmetrical areas, or if you want a dynamic splitter inside of a static splitter.

In theory, this shouldn't be hard to do. Suppose you want to use a static splitter

that breaks the window horizontally. Then imagine you want the top part of the split to have a dynamic splitter. Here are the steps:

1. Create the static splitter as usual.
2. Call **CreateView** for the lower portion of the split.
3. Change the usual **CCreateContext** object so that its **m_pNewViewClass** member is the runtime class of the view class you want the dynamic splitter to contain (this step only applies if you are creating a nested dynamic splitter).
4. Use the splitter's **IdFromRowCol** function to determine what ID a window at row 1, column 0 should have (this is the child ID for the nested splitter).
5. Create the nested splitter using the first splitter as the parent window and the child ID determined in the last step (this requires you to supply arguments you normally don't supply).

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US



SEARCH

ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)



BROWSE

BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

You can find this code in Listing 4.15. The only difference you'll notice is that the nested splitter uses class **CNestSplit** instead of **CSplitterWnd**. **CNestSplit** is a non-standard class from the listing CD.

Listing 4.15 Nesting splitters.

```
// ChildFrm.cpp : implementation of the CChildFrame class
//

#include "stdafx.h"
#include "split.h"

#include "nestsplt.h" // nested splitter bug fix
#include "ChildFrm.h"
#include "statuspane.h"
#include "splitdoc.h"
#include "splitview.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CChildFrame

IMPLEMENT_DYNCREATE(CChildFrame, CMDIChildWnd)

BEGIN_MESSAGE_MAP(CChildFrame, CMDIChildWnd)
//{{AFX_MSG_MAP(CChildFrame)
// NOTE - the Class Wizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

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

```

// CChildFrame construction/destruction

CChildFrame::CChildFrame()
{
    // TODO: add member initialization code here
}

CChildFrame::~CChildFrame()
{
}

BOOL CChildFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
    CCreateContext* pContext)
{
    CRect r;
    BOOL rv;
    GetClientRect(&r);
    rv=m_StatusSplit.CreateStatic(this,2,1); // Create #1 splitter
    if (rv) rv=m_StatusSplit.CreateView(1,0,RUNTIME_CLASS(CStatusPane),
    CSize(640,r.Height()/3),pContext); // second view
    m_StatusSplit.SetRowInfo(0,2*r.Height()/3,10);
    pContext->m_pNewViewClass=RUNTIME_CLASS(CSplitView);
    if (rv) rv=m_wndSplitter.Create( &m_StatusSplit, // Create dynamic
                                     // part (first view)
                                     2, 1,
                                     CSize( 10, 10 ),
                                     pContext,
                                     WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL | SPLS_DYNAMIC_SPLIT,
                                     m_StatusSplit.IdFromRowCol(0,0) );
    return rv;
}

BOOL CChildFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    return CMDIChildWnd::PreCreateWindow(cs);
}

//////////
// CChildFrame diagnostics

#ifdef _DEBUG
void CChildFrame::AssertValid() const
{
    CMDIChildWnd::AssertValid();
}

void CChildFrame::Dump(CDumpContext& dc) const
{
    CMDIChildWnd::Dump(dc);
}

#endif // _DEBUG

//////////
// CChildFrame message handlers

```

Why Not Use CSplitterWnd?

If you'd like to see what's wrong with **CSplitterWnd**, just modify the source code so that **m_wndSplitter** is of type **CSplitterWnd** and run the program. Click the Reset button so that the lower pane has the focus. Then try to split the top window. You'll get an assertion. Why? Because the dynamic splitter assumes it is the only splitter window, and it also assumes that whenever it wants to create a new view, one of its views must be active.

When you nest splitters, this isn't a good assumption. Luckily, this is easy to fix. **CNestSplit** (see Listing 4.16) intercepts the calls that create splits and forces the first view to be active before passing control to the base class. This may cause odd focus semantics in some rare cases, but it sure beats causing an assertion or a protection fault.

Listing 4.16 Fixing the splitter bug.

```
#include "stdafx.h"
#include "nestsplt.h"

void CNestSplit::SetMeActive(void)
{
    CFrameWnd *frame=GetParentFrame();
    if (GetDlgItem(IdFromRowCol(0, 0)) == NULL) return; // 1st time?
    CView *view=(CView *)GetPane(0,0);
    if (view->IsKindOf(RUNTIME_CLASS(CView)))
        frame->SetActiveView(view, TRUE);
}

BOOL CNestSplit::SplitRow(int cybefore)
{
    SetMeActive();
    return CSplitterWnd::SplitRow(cybefore);
}

BOOL CNestSplit::SplitColumn(int cxbefore)
{
    SetMeActive();
    return CSplitterWnd::SplitColumn(cxbefore);
}
```

Summary

Controls and other reusable components are what make Windows a powerful operating system. However, as this chapter points out, many of the controls and MFC classes have odd quirks. Luckily, MFC's architecture allows you to change things you don't like by just deriving a new class.

The MFC source code and a good understanding of Windows helps when you want to fix things that are not right. Also, you should resist the urge to place too much functionality in these repair components. Why? Microsoft might fix the same bug someday, and then you'll want to remove your altered components in favor of the shipping versions.

Practical Guide To Windows, Views And Controls

- Setting Window Styles
- Removing The Document Title
- Setting A Custom Icon, Cursor, Or Background
- Setting A View To A Specific Size
- Making List Controls Select In All Columns
- Scroll Using The Keyboard
- Scrolling Many Items In Windows 95
- Using Multiple **CEditViews** With The Same Document

- Setting Formatting For **CRichEditView**
- Using Owner-Draw (Or Self-Draw) Controls
- Effectively Using Label Editing For List And Tree Controls In Dialog Boxes
- Nesting Splitter Windows

Using window classes and predefined controls are crucial to developing Windows applications efficiently. The problem is that these controls constrain the way you work. This is especially bothersome when the controls work in some way that doesn't fit your application's look and feel. With a little ingenuity, you can make controls work the way you want them to work.

Setting Window Styles

When you want to set window styles, the logical way is to supply a style argument to the **Create** call. However, MFC creates many of the most interesting windows internally, so you can't do that.

A better, but less intuitive, approach is to set the styles in the window class' **PreCreateWindow** override. This technique makes the window more of a self-contained object. The window will always have the correct styles set (or cleared, since you can also clear bits during this call).

PreCreateWindow allows you to modify a **CREATESTRUCT** (see Table 4.2). This structure allows you to modify many important attributes of a window.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced](#)
- [Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Removing The Document Title

Sometimes it is bothersome to have the document title appear in the frame of your application. You can use the previous tip to banish the title. All you need to do is clear the **FWS_ADDTOTITLE** bit in the styles. For example:

```
BOOL CMyFrame::PreCreateWindow(CREATESTRUCT &cs)
{
    BOOL rv=CFrameWnd::PreCreateWindow(cs); // base class
    cs.style&=~FWS_ADDTOTITLE;
    return rv;
}
```

Setting A Custom Icon, Cursor, Or Background

Using **PreCreateWindow** allows you to alter many things, including the style, size, and position of the window. One thing that isn't obvious, however, is how to set the icon, cursor, or default background color. That's because those things are in the Windows class structure.

Ordinarily, you don't supply a class name for a window, and MFC assigns an appropriate one for you. However, you can supply a class name when calling **Create** or during **PreCreateWindow**.

How do you get a class name? The easiest way is to use **AfxRegisterWndClass**. This call allows you to specify the icon, cursor, and background brush you want to use. It returns a class name. What's the name? Who cares? It is a name that specifies the things you want to use.

Sometimes (when working with dialogs, for example) you need to assign the class name yourself. In those cases, use **AfxRegisterClass** instead of **AfxRegisterWndClass**. This function takes a **WNDCLASS** structure as an argument (see Table 4.3). Don't worry about the message handling function. MFC will replace it with code to process your message map. You can specify **DefWinProc** (the standard Windows default) and MFC will take care of the rest.

Of course, you can also change icons, cursors, and the background by handling Windows messages.

For example, you can specify a **NULL** icon and then draw your own icon during **OnPaint** (when **IsIconic** is **TRUE**). You can select your own cursor during **WM_SETCURSOR**. The **WM_ERASEBKGD** message allows you to draw whatever you like for the window background.

Setting A View To A Specific Size

Although overriding **PreCreateWindow** ostensibly allows you to set the size of the window, you won't find that very useful in many cases—at least, not directly. The problem is that MFC's architecture gets in the way of directly setting the size of windows.

Usually, when you are interested in setting the size of a window, you really want to set the size of the view. However, views are usually tied to the size of a frame. Also, the view has a small border that you have to account for.

To calculate the correct frame size for a given view, you have to call **::AdjustWindowRectEx** twice. You'll call it once to calculate the size of the view and again to calculate the size of the frame. You can find the code in Listing 4.3. This call isn't hard to figure out. It merely takes a rectangle and the styles of the window in question. Then it inflates the rectangle based on those styles. Notice that no window is required.

A handy place to calculate the sizes is from within the view's **OnInitialUpdate**. At that point, you can learn the styles of both windows by calling **GetStyle** and **GetExStyle**. An alternate approach would be to size the frame when it is created (for example, during **PreCreateWindow**). However, the view doesn't exist at this time; you would need to hard code the styles for the view.

If you want to enforce the size of a view, you'll need to use the same procedure while processing the **WM_GETMINMAXINFO** message. This message allows you to set the minimum and maximum size of the frame. Again, if you can set the size of the frame, the view will take care of itself.

Making List Controls Select In All Columns

One annoying feature of list controls is that you have to click in the first column of data to select a row. Also, only that column will have a highlight. You can use the code in Listing 4.1 (see Figure 4.1) as a direct replacement for the list control.

This special control forces mouse input to appear to be at the left-hand side of the control. It also paints a selection rectangle around any currently selected items.

Scroll Using The Keyboard

One feature sorely lacking in **CScrollView** is the ability to scroll with the keyboard. Fortunately, this is easy to fix. Instead of trying to duplicate scrolling semantics, it's easier to just fake scroll bar events in response to keyboard messages. This forces the scroll view to do all the hard work. Here's some sample code:

```
void CScrollerView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    BOOL processed;
    for (unsigned int i=0;i<nRepCnt&&processed;i++)
        processed=KeyScroll(nChar);
    if (!processed)
        CScrollView::OnKeyDown(nChar, nRepCnt, nFlags);
}
BOOL CScrollerView::KeyScroll(UINT nChar)
{
    switch (nChar)
    {
```

```

        case VK_UP:
            OnVScroll(SB_LINEUP, 0, NULL);
            break;
        case VK_DOWN:
            OnVScroll(SB_LINEDOWN, 0, NULL);
            break;
        case VK_LEFT:
            OnHScroll(SB_LINELEFT, 0, NULL);
            break;
        case VK_RIGHT:
            OnHScroll(SB_LINERIGHT, 0, NULL);
            break;
        case VK_HOME:
            OnHScroll(SB_LEFT, 0, NULL);
            break;
        case VK_END:
            OnHScroll(SB_RIGHT, 0, NULL);
            break;
        case VK_PRIOR:
            OnVScroll(SB_PAGEUP, 0, NULL);
            break;
        case VK_NEXT:
            OnVScroll(SB_PAGEDOWN, 0, NULL);
            break;
        default:
            return FALSE; // not for us
    }
    return TRUE;
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Scrolling Many Items In Windows 95

Scroll views don't handle more than 32,767 items under Windows 95. Why? Because Windows 95 is no more than a thin 32-bit wrapper over (mostly) 16-bit code. That means that scroll bar ranges must be less than 32K and that GDI coordinates have to fall below that magic number, as well. Worse still, you won't see any error messages or warnings. Things start working funny (refer back to Figure 4.3).

Of course, you could simply write your own scrolling behavior into **CView**, but that's a lot of work. A better way is to trick the scroll view into thinking each pixel of scrolling is really some larger unit (for example, a line of text).

There are a few things to remember when using this technique. First, the scroll view will only show enough pixels so that your selected number is at the bottom of the window. Because you are going to expand the pixels into lines (or whatever), you have to specify more than you actually need. How much more will depend on the size of the window.

The second thing you need to do when faking scrolling is to reset the mapping mode in **OnDraw** so that the scrolling information programmed by the default implementation of **OnPrepareDC** doesn't bother your drawing. Then it's up to you to adjust your drawing based on the scroll amounts.

You can find a complete example of this technique in Listing 4.4. Another solution to this problem is to run your program under Windows NT (where this isn't a problem).

Using Multiple CEditViews With The Same Document

CEditView isn't well matched with the document/view architecture. The MFC designers opted for the simplicity of a system edit control (I would have, too). However, because the edit control contains the data (and the document doesn't), you generally can't use edit views with advanced features like splitter windows, multiple views, and so on without special techniques.

One approach is to intercept all keystrokes that can change the edit control. Then, send them to the document for processing. The document can then distribute the correct data to all related views.

There are some complexities involved in this approach. For example, pasted data from the clipboard should go to all views, but a copy operation only affects the current view. You'll have to code around these kinds of problems as your situation warrants. The basic code appears in Listings 4.5 and 4.6.

Setting Formatting For CRichEditView

There is a little-known connection between MFC's **CFontDialog** and **CRichEditView**. The font dialog has a constructor that accepts a **CHARFORMAT** structure. You can then retrieve a **CHARFORMAT** structure using the call **GetCharFormat**.

Because the **CHARFORMAT** structure is what the **CRichEditView** expects, this makes it trivial to add sophisticated character formatting to a **CRichEditView** (refer back to Figure 4.5 and Listing 4.8). You can get the current formatting of a selection using **GetCharFormatSelection** and set the format using **SetCharFormat**.

Using Owner-Draw (Or Self-Draw) Controls

Traditional owner-draw controls send messages to their parent window when they require drawing. MFC reflects these messages back to the control so it can draw itself (hence, the name self-draw).

The trick is to derive a class for each control. In this derived class, you override **DrawItem**, **MeasureItem**, **CompareItem**, and **DeleteItem** to handle the various cases. Exactly what you need to override depends on what kind of control you are creating (see Table 4.4).

You can create buttons (**CButton**), statics (**CStatic**), menus (**CMenu**), list boxes (**CListBox**), and some kinds of combo boxes (**CComboBox**). Each type has its own peculiarities that you can read about in the main text.

Effectively Using Label Editing For List And Tree Controls In Dialog Boxes

List and tree controls support label editing. This allows the user to click on an item and type over it. You see this in the Windows 95-style shell and the registry editor, among other places.

If you try to use this feature in a dialog box, you'll be disappointed. When the user presses Enter to terminate the edit, it also terminates the dialog box. This is sure to surprise the user.

Luckily, the answer is relatively simple. In the dialog box's **OnOK** handler, you need to find out what window has the focus. Then find the parent window. If the parent window is the list (or tree) control, you can send it a fake Enter key and prevent the base class code from executing.

Here's a snippet of code to illustrate:

```
void CBadDlg::OnOK ()
{
    if (CWnd::GetFocus()->GetParent()==&m_tree) // bogus OK from edit
        control
        {
            CWnd *ectl=CWnd::GetFocus();
            ectl->SendMessage(WM_KEYDOWN,VK_RETURN);
            // As odd as it seems, don't send a KEYUP
            // because by the time it gets there, the edit control
            // will be gone!
            // ectl->SendMessage(WM_KEYUP,VK_RETURN);
            return;
        }
    CDialog::OnOK();
}
```

Nesting Splitter Windows

Splitter windows are one of MFC's more unique features. However, splitters don't work properly when they are nested inside of one another. In theory, nesting a splitter window isn't difficult. You simply make one splitter the parent of another. You also have to manipulate the child ID of the inner splitter so that it appears in the right place (see Listing 4.15).

The problem is that dynamic splitters assume they are always the only splitter window. Therefore, one of the views that it owns must be active. This isn't always the case, and it causes an assertion or exception under certain circumstances.

To correct this problem, you can force an arbitrary view to have the focus when the splitter expects it. That is the purpose of the **CNestSplit** class in Listing 4.16. You can use it as a direct replacement for the normal **CSplitterWnd** class and it corrects all the focus problems I've found so far.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Chapter 5 Dialogs

Dialogs are the easiest way to interact with the user. MFC's DDX/DDV facility makes using dialogs simpler than ever before. This chapter shows you how to customize DDX and DDV for your own data types. You can also customize the common dialogs to better fit your program's needs.

Programmer's Notes...

I'm one of those annoying people who overanalyze TV shows and movies. It's probably one of my worst habits (and a constant source of aggravation to my poor wife, Pat). Consider *M*A*S*H*, for example. A great show, of course. But did you ever wonder how all those people stayed assigned to the same posts for so long? The regulars were stuck in Korea at the same billet for years. Of course, to produce a show where the audience gets to know and like the characters, some suspension of disbelief is necessary. I know it has to be that way, but it still strikes me as funny.

As you already know, I love *Star Trek*, so I probably pick on it more than any other show. The original TV series was made in a different time, so you have to overlook the big bulky lamps and switches. You even have to overlook the chugging computer noises (gee, they did seem to have 3.5 inch disks, though).

The movies, on the other hand, don't have this excuse. Here's another case of where practically the entire crew stays together for years and years. With each

movie, they totally redesign the ship, and often the uniforms, too. In some of the movies, Sulu uses what looks like a transmission shifter from an old Mustang to accelerate to warp speed.

Worse still is some of the computer technology—especially in the area of user interface design. I remember at least one movie where Spock is sitting in front of a panel with at least 200 neon bulbs all blinking at random. The bulbs are in a tight matrix with no labels or other distinguishing characteristics. Offhand, this strikes me as a bad user interface. To think that you could reasonably assimilate hundreds of unmarked light bulbs is silly. Even if Spock could do it (hey, he can do most anything), who among the human crew could?

Good user interface design is why dialog boxes exist. Dialogs give you a standard way to put controls (and labels for them) up for the user. The user knows how to navigate between the controls because dialog boxes (usually) work the same no matter what program you are using.

Not that dialog boxes guarantee good communications. I love the dialog boxes I sometimes get from Microsoft's Internet Explorer. Every once in a while, I get something along the lines of "Could not open <http://www.al-williams.com/awc>. The operation completed successfully." This proves the old adage "Garbage in, garbage out." Dialog boxes aren't a cure-all, but they do offer a great way to present and collect data.

MFC And Dialogs

MFC has a strange relationship with dialogs. It offers a special class for them (**CDialog**), but that class primarily handles modal dialogs. If you want modeless dialogs, you'll have to do a bit of extra work.

Modal Vs. Modeless Dialogs

Most users think of dialog boxes as modal. When you bring them up, they lock up your program until you dismiss them (typically with an OK or Cancel button). However, you can also use modeless dialogs that allow you to switch to other windows in the application without dismissing the dialog. The most common example of a modeless dialog is the find and replace dialog you see in many word processor and text editor programs.

What's the difference between a modeless dialog and a regular window? Not much. A dialog, of course, can load from a resource. Also, dialogs have the smarts to handle things like the Tab key. Of course, you could do all of these things with a regular window, too—it would just be more work.

It is interesting to note that MFC dialogs are not real dialogs in the classical sense. Instead, MFC emulates traditional dialog boxes. It does this so it can add special features (like ActiveX containment).

Technically, MFC uses **CDialog** for both modal and modeless dialogs. However, in reality, the class expects to work with a modal dialog. Here's why:

- When **CDialog** detects the OK or Cancel buttons, it calls **::EndDialog**. The **::EndDialog** call is only for modal dialogs. Use **DestroyWindow** for modeless dialogs.
- The dialog box does not automatically free its memory. This isn't a

problem for modal dialogs because you will usually create them on the stack. However, modeless dialogs have a longer life. You'll often create them on the heap and want them to delete themselves before they disappear.

What happens if you accidentally allow the framework to call **EndDialog**? Although the documentation isn't very specific, it is documented that the function doesn't destroy the dialog. Instead, it hides the dialog and sets a flag instructing the modal event loop to destroy the dialog. Because a modeless dialog doesn't use the modal event loop, the dialog still exists after a call to **EndDialog** and you'll have problems trying to bring it up again.

How does MFC know if you are using a modal or modeless dialog? Modal dialogs begin when you call **DoModal**. You use **Create** to start a modeless dialog. Other than that, there is little difference.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Implementing Modeless Dialogs

Of course, it is easy to derive a new class (say, **CModelessDialog**) to handle dialog problems. You can't make Class Wizard base a new class from your class, but you can always use **CDialog** and then manually touch up the file.

You'll find an example **CModelessDialog** in Listing 5.1. This simple class changes the dialog to work properly as a modeless dialog created with **new** (that is to say, on the heap).

You might think that if you simply omit buttons that have IDs of **IDOK** and **IDCANCEL** that you can side-step calling the **OnOK** and **OnCancel** routines. This isn't sufficient, however. In some cases, Windows generates events in response to the keyboard that simulates these buttons. That means that even if your dialog box doesn't have OK and Cancel buttons, it can still get events that those buttons would generate.

The only other task the new dialog class handles is the deletion of the dialog from memory when the program destroys it. When Windows destroys a dialog (or any window, for that matter), the last message it sends is **WM_NCDESTROY**. This message tells the window to delete its non-client area (the menu bar, the caption, borders, and so on). MFC knows that this is the last message the window receives, so after it handles the message, it calls **PostNcDestroy**. This is your opportunity to destroy the dialog instance.

Listing 5.1 CModelessDlg.

```
// modeless.cpp : implementation file
//
#include "stdafx.h"
#include "modeless.h"
#ifdef _DEBUG
```

```

#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif
//////////
// CModelessDlg dialog

CModelessDlg::CModelessDlg()
{
    //{{AFX_DATA_INIT(CModelessDlg)
        // NOTE: the Class Wizard will add member
        // initialization here
    //}}AFX_DATA_INIT
}

void CModelessDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CModelessDlg)
        // NOTE: the Class Wizard will add DDX and DDV
        // calls here
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CModelessDlg, CDialog)
    //{{AFX_MSG_MAP(CModelessDlg)
        // NOTE: the Class Wizard will add message map
        // macros here
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

//////////
// CModelessDlg message handlers

void CModelessDlg::PostNcDestroy()
{
    delete this;
}

void CModelessDlg::OnCancel()
{
    DestroyWindow();
}

void CModelessDlg::OnOK()
{

```

```
UpdateData(TRUE);  
DestroyWindow();  
}
```

```
void CModelessDlg::OnClose()  
{  
    DestroyWindow();  
}
```

Unfortunately, you have to write an ugly line of code to delete the object. The line is:

```
delete this;
```

While this might look odd, it is perfectly legal. Just don't try to do anything after writing this line of code because the object's instance variables will all be invalid.

Using DDX/DDV

Dynamic data exchange (DDX) and dynamic data validation (DDV) are the features that make MFC dialogs seem almost like magic. In theory, DDX makes a connection between a control on your dialog and a variable in your **CDialog**-derived class. That's how it appears to work (at least, most of the time).

The reality of DDX is quite different. The connection between the variable and the control is really just an illusion. When you ask Class Wizard to connect a variable to a control (via the Member Variables tab), it makes an entry in the data map. All this really means is that it adds a function call inside the dialog's **DoDataExchange** function (a function that Class Wizard maintains).

When you call **UpdateData(FALSE)**, MFC calls **DoDataExchange**. The functions that Class Wizard puts in **DoDataExchange** copy the data from your variables to the corresponding controls. If you call **UpdateData(TRUE)**, MFC reverses its steps and the **DoDataExchange** functions copy data (and possibly validate it) back to the variables.

The reason this appears automatic is that **CDialog** always calls **UpdateData(FALSE)** in the **OnInitDialog** function. That means that as long as you call the base class **OnInitDialog** (or fail to override it), the member variables mysteriously appear on your dialog when it starts. The default **OnOK** function also calls **UpdateData**, but with a **TRUE** argument. So a modal dialog seems to take care of itself. You can write code like this:

```
CNamedDlg dlg;  
dlg.m_name="New User";  
if (dlg.DoModal()==IDOK) MessageBox(dlg.m_name, "Greetings");
```

Consider what happens when you use a modeless dialog. The dialog still receives an **OnInitDialog** message, so the initial transfer works. However, modeless dialogs usually don't wait for an OK button to process their data. That means you'll need to call **UpdateData** when you want data to transfer.

UpdateData: True Or False?

Here's a good way to remember what argument you should use when calling **UpdateData**. Just remember that **TRUE** is like the OK button: It transfers data from the controls to the variables. That, of course, implies that **FALSE** does the opposite.

Sometimes modal dialogs need a little help, too. For example, suppose you are writing an email program. The user enters a name and an email address into a dialog box. You also have a button next to the address field labeled Lookup. If the user presses that button, your program will get the name and automatically fill in the address field.

When you write the code for the Lookup button, you have two choices: Either read the name directly from the edit control (using **GetDlgItemText**, for example) or call **UpdateData(TRUE)** to transfer data to your variables (like **m_name**). The same thing happens when you want to fill in the address. You'll either set it directly or fill in the variable and call **UpdateData(FALSE)**.

With modal dialogs, it is often useful to have the variables appear to track each user input. The brute force method for doing this is to look for every control's signal that its state has changed and call **UpdateData**.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

Conventional controls always signal this with a **WM_COMMAND** message. That means a better way to handle this is to install a **WM_COMMAND** handler using Class Wizard. This places an **ON_WM_COMMAND** macro in your message map. This is different from how you usually handle command messages because this routine will see all command messages, not just ones for a specific control. Call the base class, then call **UpdateData(TRUE)**, and you're in business. If you use controls that send **WM_NOTIFY** messages, you can do the same trick with **OnNotify**.

You can find a simple example of this technique in Listing 5.2. Run the program and select the Go|Go menu item. Move the dialog box out of the way so you can see the main window and the dialog at the same time. As you enter data into the dialog, the main view will reflect the changes.

There are several details worth mentioning about this program. First, there is a chance that the **WM_COMMAND** message will destroy the dialog and make it invalid. That's why it's important not to call **UpdateData** unless **::IsWindow** returns **TRUE** (see the **OnCommand** function in Listing 5.2). Also, if some part of the code calls **UpdateData(FALSE)**, the controls may fire command messages at that time. This will cause an assertion as you try to recursively call **UpdateData**. That's why the **OnInitDialog** function sets the **in_init** flag. The **OnCommand** function will not call **UpdateData** when **in_init** is **TRUE**. You should set this flag before calling **UpdateData** if you call it from elsewhere in your code.

In this particular case, the main view needs to know when something changes. The code calls **UpdateAllViews** for the controlling document when anything changes. Although your first temptation might be to simply find the dialog's parent window, this won't work. The parent window will never be the view (even if you explicitly set it that way). The dialog wants a non-child window as its parent. A simple workaround is to set a pointer to the view, which is exactly what the code in Listing 5.2 does.

Listing 5.2 Instant DDX.

```
// LiveDialog.cpp : implementation file
//
```

```
#include "stdafx.h"
#include "livedlg.h"
#include "LiveDialog.h"
```

```

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CLiveDialog dialog

CLiveDialog::CLiveDialog(CWnd* pParent /*=NULL*/)
: CDialog (CLiveDialog::IDD, pParent)
{
   //{{AFX_DATA_INIT(CLiveDialog)

    m_email = _T("");
    m_name = _T("");
    //}}AFX_DATA_INIT
    m_View=NULL;
}

void CLiveDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CLiveDialog)
    DDX_Text(pDX, IDC_EMAIL, m_email);
    DDX_Text(pDX, IDC_NAME, m_name);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CLiveDialog, CDialog)
    //{{AFX_MSG_MAP(CLiveDialog)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CLiveDialog message handlers

BOOL CLiveDialog::OnCommand(WPARAM wParam, LPARAM lParam)
{
    BOOL rv=CDialog::OnCommand(wParam, lParam);
    // Don't do it if this command destroyed us or we are initializing
    if (::IsWindow(m_hWnd)&&!in_init)
    {
        UpdateData(TRUE); // Update on any change
        // Because this particular program wants updates to show up in the
        view
        ASSERT(m_View!=NULL);
        m_View->GetDocument()->UpdateAllViews(NULL);
    }

    return rv;
}

```

```
void CLiveData::OnOK()
{
    DestroyWindow();
}

void CLiveData::OnCancel()
{
    DestroyWindow();
}

BOOL CLiveData::OnInitDialog()
{
    in_init=TRUE;
    CDialog::OnInitDialog();
    in_init=FALSE;
    return TRUE;
}
```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

About Data Validation

In addition to data exchange, the data map can also validate data. Typically, validation means ensuring a string is less than a certain number of characters, or that a number is within a particular range. You set this up with Class Wizard, and it appears to be more or less automatic.

However, data validation doesn't live up to most user's expectations. Why? Because validation only occurs along with control-to-variable data transfers. That usually means that the user enters all the data, clicks on OK, and then gets an error message.

Live Data Validation

Is there a way to do "live" data validation? That is, validation that occurs right after the user enters data? That depends. Validating data on a keystroke-by-keystroke basis is a better job for a customized edit control (or even an ActiveX control). However, it is possible to validate fields immediately after the user enters them. It just takes a little extra work.

In principle, the idea isn't much different from exchanging data every time something changes (as in Listing 5.2). You intercept all command messages, but this time, you look for **EN_KILLFOCUS** in the high word of **wParam**. This indicates that an edit control is losing focus. This is a good time to validate the value of the field.

The problem is that you don't want to validate everything. You only want MFC to examine the field that is losing the focus (this field's ID is in the low word of **wParam**). There are several ways you might accomplish this. My solution is to manually modify the data map.

The first step is to get the data map working the way you like (except, of course, that it still uses regular delayed validation). Next, you move all the code in **DoDataExchange** (the data map) outside of Class Wizard's special comments. Add a member variable to the dialog that contains the ID you want to validate and set it to zero in the class constructor. Then change each validation and exchange line to only execute if the validation ID is zero, or the ID that the line applies to (see Listing 5.3). Finally, add a variable to let your code know that you are in the process of validating data. Set the flag to **FALSE** in the constructor and **TRUE** at the beginning of **DoDataExchange**. You can reset it at the end of the **DoDataExchange** function.

When you want to validate a particular field (for example, when you detect an **EN_KILLFOCUS**), you

set the validation variable to the control ID and call **UpdateData(TRUE)**. Be sure you aren't already in the middle of validation (that's why you have a flag set at the start of **DoDataExchange**).

The only catch to this is that if the data validation fails, MFC throws an exception to abort **DoDataExchange**. That means that if you have a number out of range, for example, your validation flag doesn't reset. The simplest solution is to reset it after calling **UpdateData**. Another solution is to catch the exception, clear the flag, and rethrow the exception.

Listing 5.3 is an example of a **CFormView** that validates on the fly. The same technique works fine for dialogs, too. All the real action is in the **OnCommand** and **DoDataExchange** functions. The modified **DoDataExchange** function catches exceptions so that the **validate** flag is kept correct even if validation fails.

Listing 5.3 Instant DDV.

```
// validView.cpp : implementation of the CValidView class
//

#include "stdafx.h"
#include "valid.h"

#include "validDoc.h"
#include "validView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CValidView

IMPLEMENT_DYNCREATE(CValidView, CFormView)

// Class Wizard won't put this here because it thinks
// dialog boxes handle OnOK. They do, but this is a
// form view, not a dialog box.
BEGIN_MESSAGE_MAP(CValidView, CFormView)
    //{AFX_MSG_MAP(CValidView)
    ON_COMMAND(IDOK, OnOK)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////
// CValidView construction/destruction

CValidView::CValidView()
    : CFormView(CValidView::IDD)
{
    validating=FALSE;
    vid=0;
    //{AFX_DATA_INIT(CValidView)
    m_age = 18;
    m_name = _T("");
    m_wager = 1.0f;
    //}}AFX_DATA_INIT
    // TODO: add construction code here
```

```

}

CValidView::~CValidView()
{
}

void CValidView::DoDataExchange(CDataExchange* pDX)
{
    CFormView::DoDataExchange(pDX);
    validating=TRUE; // prevent recursion
    // moved these out of class wizard comments and changed them
    try
    {
        if (!vid||vid==IDC_AGE) DDX_Text(pDX, IDC_AGE, m_age);
        if (!vid||vid==IDC_AGE) DDV_MinMaxInt(pDX, m_age, 18, 150);
        if (!vid||vid==IDC_NAME) DDX_Text(pDX, IDC_NAME, m_name);
        if (!vid||vid==IDC_NAME) DDV_MaxChars(pDX, m_name, 64);
        if (!vid||vid==IDC_WAGER) DDX_Text(pDX, IDC_WAGER, m_wager);
        if (!vid||vid==IDC_WAGER) DDV_MinMaxFloat(pDX, m_wager, 1.f,
            100.f);
        //{AFX_DATA_MAP(CValidView)
        //}AFX_DATA_MAP
        validating=FALSE;
    }
    catch (...)

    {
        validating=FALSE; // make sure this is clear
        throw; // go ahead
    }
}

BOOL CValidView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CFormView::PreCreateWindow(cs);
}

////////////////////////
// CValidView diagnostics

#ifdef _DEBUG
void CValidView::AssertValid() const
{
    CFormView::AssertValid();
}

void CValidView::Dump(CDumpContext& dc) const
{
    CFormView::Dump(dc);
}

CValidDoc* CValidView::GetDocument() // non-debug version is inline
{

```

```

        ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CValidDoc)));
        return (CValidDoc*)m_pDocument;
    }
#endif // _DEBUG

////////////////////////////////////
// CValidView message handlers

void CValidView::OnOK()
{
    if (UpdateData(TRUE))
        MessageBox("Wager placed");
}

BOOL CValidView::OnCommand(WPARAM wParam, LPARAM lParam)
{
    if (HIWORD(wParam)==EN_KILLFOCUS&&!validating)
    {
        vid=LOWORD(wParam);
        UpdateData(TRUE);
        // reset conditions
// You'd need to do the line below if you don't catch
// exceptions in DoDataExchange.
//         validating=FALSE;
        vid=0;
    }
    return CFormView::OnCommand(wParam, lParam);
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Other Data Map Tricks

Once you realize that the so-called data map is just a function, it opens up many possibilities. For example, the wager validation line might look like this:

```
DDV_MinMaxFloat(pDX, m_wager, 1.f, credit_limit);
```

Of course, Class Wizard doesn't know about this, so you'd need to move outside the Wizard comments.

Another possibility is selective validation. For example, suppose you had a custom zip code validator (you'll see how to write custom validation routines in the next section). You might write:

```
if (country==USA) DDV_ZipCode(pDX,m_zip);
```

Just be sure that the validation code appears immediately after the corresponding exchange (DDX) call. Otherwise, your program may not correctly identify which field is in error when validation fails.

Adding Custom DDX/DDV

Once you understand data maps, you have to start thinking about writing your own exchange and validation routines. There's no reason you can't do it; the exchange and validation functions are just global functions that know how to deal with a **CDataExchange** object. There is nothing particularly unusual about them.

Sometimes it is handy to do your validation at the same time you do the exchange. This is particularly true if validation doesn't require any arguments. For example, if you wrote an exchange routine to convert a host name to an IP address, you could easily validate it at the same time you transfer the data.

Other times, you'll want to write a custom validator that can take arguments. In either case, you can integrate your new routines with Class Wizard so that they appear along with the standard DDX/DDV routines.

The first step is to get your exchange and validation routines working. For exchange functions, write a global function that takes a **CDataExchange** pointer, a control ID, and a reference to a variable. Although it's tempting to not name the function with the **DDX_** prefix, resist the urge (you'll see why shortly).

In the exchange function, you can examine the **CDataExchange** pointer to learn the details you need (see Table 5.1). Usually, you want to examine the **m_bSaveAndValidate** member. This variable corresponds to the argument you supply to **UpdateData** (**TRUE** means transfer from the control to the variable; **FALSE** implies the other direction).

Table 5.1 The CDataExchange class.

Member	Description
m_bSaveAndValidate	TRUE to transfer from controls to variables
m_pDlgWnd	Window handle for controlling window or dialog
PrepareControl	Call this function to mark the current control (unless it is an edit control)
PrepareEditCtrl	Call this function to mark the current control if it is an edit control
Fail	Causes a validation failure on the last prepared control (you may call this in either a DDX or DDV routine)

If there is any chance that the transfer might fail, you need to call **PrepareEditCtrl** (for edit controls) or **PrepareCtrl** (for all other controls). After you make this call, any call to **Fail** will put the focus back on that control. That's true even if another routine (like the companion validation routine) signals the failure.

If you decide that the data is not valid, you should display a message and call **Fail**. This throws an exception that disrupts the **DoDataExchange** function and places the input focus back on the last prepared control. Of course, you should only validate when **m_bSaveAndValidate** is **TRUE**. Transfers from your program to the controls usually assume that the data is valid.

Writing a validation routine is almost the same as writing an exchange routine except that the arguments are different. Use the **DDV_** prefix in the function name. For arguments, the function accepts a **CDataExchange** pointer, a value of the appropriate type, and one or two validation parameters (for example, the lower and upper limits for the value).

Here, your job is simple. If **m_bSaveAndValidate** is **TRUE**, do whatever logic you want to do to make sure the value is legitimate. If the data is good, return from the function. If it isn't, call **Fail**. The previous exchange function has already marked which control you are working with. That's why you always want DDV functions in your data map to immediately follow their corresponding DDX function.

Listing 5.4 shows a version of the OTB program (from Listing 5.2) that uses a custom validator (Listing 5.5). The validation code makes sure that you don't enter more than two digits after the decimal point. It also passes control to the standard **DDV_MinMaxFloat** function to make sure that the number is a legitimate floating-point value. After all, why reinvent the wheel?

Notice that the validation routine doesn't directly get the ID of the control like the exchange function does. That's because the validation routine usually is only worried about the control's value. However, you can get the control's window handle by examining the **CDataExchange** object's **m_hWndLastControl** member.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US



SEARCH

ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)



BROWSE

BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Listing 5.4 Using custom DDX and DDV.

```
// validView.cpp : implementation of the CValidView class
//
```

```
#include "stdafx.h"
```

```
#include "valid.h"
```

```
typedef float Currency; // used for DDV
```

```
#include "validDoc.h"
```

```
#include "validView.h"
```

```
#include "customdd.h"
```

```
#ifdef _DEBUG
```

```
#define new DEBUG_NEW
```

```
#undef THIS_FILE
```

```
static char THIS_FILE[] = __FILE__;
```

```
#endif
```

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

```
// CValidView
```

```
IMPLEMENT_DYNCREATE(CValidView, CFormView)
```

```
// Class Wizard won't put this here because it thinks
```

```
// dialog boxes handle OnOK. They do, but this is a
```

```
// form view, not a dialog box.
```

```
BEGIN_MESSAGE_MAP(CValidView, CFormView)
```

```
   //{{AFX_MSG_MAP(CValidView)
```

```
    ON_COMMAND(IDOK, OnOK)
```

```
   //}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()
```

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

```

// CValidView construction/destruction

CValidView::CValidView()
    : CFormView(CValidView::IDD)
{
    validating=FALSE;
    vid=0;
    //{AFX_DATA_INIT(CValidView)
    m_age = 18;
    m_name = _T("");
    m_wager = 1.0;
    m_btnenable = TRUE;
    //}}AFX_DATA_INIT
    // TODO: add construction code here

}

CValidView::~CValidView()
{
}

void CValidView::DoDataExchange(CDataExchange* pDX)
{
    CFormView::DoDataExchange(pDX);
    //{AFX_DATA_MAP(CValidView)
    DDX_Text(pDX, IDC_AGE, m_age);
    DDV_MinMaxInt(pDX, m_age, 18, 150);
    DDX_Text(pDX, IDC_NAME, m_name);
    DDV_MaxChars(pDX, m_name, 64);
    DDX_Text(pDX, IDC_WAGER, m_wager);
    DDV_MinMaxCurrency(pDX, m_wager, 1.f, 100.f);
    DDX_EnableWindow(pDX, IDOK, m_btnenable);
    //}}AFX_DATA_MAP
}

BOOL CValidView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CFormView::PreCreateWindow(cs);
}

//////////
// CValidView diagnostics

#ifdef _DEBUG
void CValidView::AssertValid() const
{
    CFormView::AssertValid();
}

void CValidView::Dump(CDumpContext& dc) const
{
    CFormView::Dump(dc);
}

CValidDoc* CValidView::GetDocument() // non-debug version is inline
{

```

```

        ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CValidDoc)));
        return (CValidDoc*)m_pDocument;
    }
#endif // _DEBUG

//////////
// CValidView message handlers

void CValidView::OnOK()
{
    if (UpdateData(TRUE))
    {
        MessageBox("Wager placed");
        m_btnenable=FALSE;
        UpdateData(FALSE);
    }
}

```

Listing 5.5 The custom DDX and DDV routines.

```

#include <stdafx.h>
#include "customdd.h"

// Custom Exchange
void DDX_EnableWindow(CDataExchange *pDX, int id, BOOL &flag)
{
    CWnd *ctl=pDX->m_pDlgWnd->GetDlgItem(id);
    if (pDX->m_bSaveAndValidate)
        flag=ctl->IsWindowEnabled();

    else
        ctl->EnableWindow(flag);
}

// Custom validator

void DDV_MinMaxCurrency(CDataExchange *pDX, float val, float min,
float max)
{
    CWnd *editctl=CWnd::FromHandle(pDX->m_hWndLastControl);
    CString s;
    int n;
    if (pDX->m_bSaveAndValidate)
    {
        // Using math to decide if anything is left over is bad because of
        // rounding
        // errors, so use a string method instead.
        editctl->GetWindowText(s);
        n=s.Find('.');
        if (n!=-1 && n+3<s.GetLength())
        {
            AfxMessageBox("Please enter the data to the nearest penny!");
            pDX->Fail();
        }
        DDV_MinMaxFloat(pDX, val, min, max); // let the existing one do the job
    }
}

```

Listing 5.5 also contains an exchange function that maps a **BOOL** variable to a control's enabled status.

Traditionally, you would use Class Wizard to map an entire control variable (like a **CButton**, for example) just to enable and disable the control. With this exchange function, you can map it to a simple **BOOL** variable and treat it like any other exchanged variable. Just remember that changes are not valid until a call to **UpdateData** occurs.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

Brief Full

Advanced

Search

Search Tips

To access the contents, click the chapter and section titles.

MFC Black Book
(Publisher: The Coriolis Group)
Author(s): Al Williams
ISBN: 1576101851
Publication Date: 12/01/97

Bookmark It

Search this book:

Previous Table of Contents Next

Integrating With Class Wizard

Once you have your custom DDX and DDV routines, you can integrate them directly with Class Wizard (see Figure 5.1). This is especially helpful if you are managing a large project with many programmers.

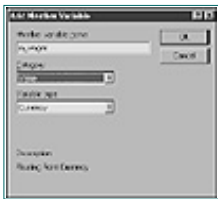


Figure 5.1 Custom DDX and Class Wizard.

You can add your custom routines to your project’s CLW file if you only want it to apply to one project. Of course, if your CLW file ever goes bad (and they do sometimes), you’ll have to reenter the custom DDX data. You can also create a DDX.CLW file in the BIN directory that contains MFCCLWZ.DLL (usually \Program Files\DevStudio\SharedIDE\Bin). Then your DDX routines will apply to all projects.

Here’s how it works: If you create a DDX.CLW file, you need to add a section to it named **[ExtraDDX]** (see Listing 5.6). This looks like an INI file section, but the name is case-sensitive. If you are working with your project’s CLW file, you can simply work in the existing [General Info] section. Then add a line that says:

ExtraDDXCount=x

Substitute the number of DDX items you wish to add for x. Of course, this number is cumulative. If you add more items later, you’ll increase this count. For the first item, you write a line that begins with **ExtraDDX1=**. Subsequent items will be **ExtraDDX2**, **ExtraDDX3**, and so on.

What goes in those lines? Either 7, 10, or 12 fields, depending on what you want to do. Each field ends with a semicolon. You can find the meaning of these fields in Table 5.2 and see an example of the lines in Listing 5.6. (Please note that due to page width constraints, the code line beginning **ExtraDDX1=E...** is shown here broken in two lines, but it’s actually one.)

Table 5.2 Registering custom DDX/DDV.

Field	Description
1	Type of controls this DDX applies to (E=edit control, for example)

2	Not used
3	Property type (usually Value; this corresponds to the first combo box in Class Wizard)
4	Data type for variable
5	Initial value
6	Name of DDX routine without DDX_ prefix
7	Comment
8	Name of DDV routine without DDV_ prefix (optional)
9	Name of first DDV argument (optional)
10	Type of first DDV argument (for example, f=float; optional)
11	Name of second DDV argument (optional)
12	Type of second DDV argument (optional)

Listing 5.6 The DDX.CLW

```
[ExtraDDX]
ExtraDDXCount=2
ExtraDDX1=E;;Value;Currency;0.0;Text;Floating Point Currency;
MinMaxCurrency;
    Mi&nimum;f;Ma&ximum;f
ExtraDDX2=bBECcRLlMnN;;Enable State;BOOL;TRUE;EnableWindow;Window Enabled
Status
```

Notice that the third field is usually “Value” if you want your DDX to show up with all the usual data types. That also means that you should define your own data type instead of reusing one of the standard types. That’s why the currency validator uses the **Currency** type (a **typedef** for **float**) instead of using **float** directly. You can specify unique names, if you wish. The enable exchange definition, for example, uses the string “Enable State”. Class Wizard then shows that choice along with the usual “Control” and “Value” categories.

You don’t have to specify any DDV routines. You can also mix a new validation routine with a standard exchange function (that’s exactly what **ExtraDDX1** does for the currency type). Notice that the names of the DDX and DDV functions don’t begin with **DDX_** and **DDV_**, but Class Wizard does add these prefixes to the code it generates. That’s why it’s important to name your functions with these prefixes.

Using Dialog Bars

A close relative to dialog boxes are dialog bars, which you don’t see very often. A dialog bar is like a tool bar, but it uses a dialog template instead of an array of bitmap buttons. You can see a dialog bar in Figure 5.2. Not very flashy, but you could easily make it flashier with bitmapped buttons and the like if you wanted to do so.

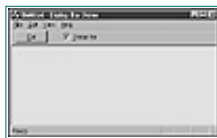


Figure 5.2 A dialog bar.

Adding a dialog bar is easy. It’s just like a tool bar except that you use the **CDialogBar** class. The Component Gallery will add one to your project automatically, if you like. Just go to the Project|Add and then to Project|Components and Controls menu item. Select Developer Studio Components on the subsequent dialog box and then pick Dialog Bar.

As you can see from Figure 5.2, you can add more than just buttons to a dialog bar. Why do people still use regular tool bars, then? There are several reasons. First, App Wizard always generates a tool bar, so that’s what most people stick with. Second, under Windows 3.1 (and to a lesser extent, Windows 95), having dialog bars with lots of controls would be a significant drain on system resources. A standard tool bar requires far fewer resources. Lastly, dialog bars can’t dock on any edge of the window because MFC doesn’t know how to rotate them the way it rotates a standard tool bar.

To understand the problems inherent in rotating dialog bars, grab a standard tool bar (like one of those in the Developer's Studio) and drag it from the top edge of the window to the right edge (or vice versa). The tool bar rotates to accommodate its new orientation. Now look at Figure 5.3. This is a screen shot of what the dialog bar from Figure 5.2 looks like on the right-hand side of its window. It's not a pretty sight.



Figure 5.3 Rotating a dialog bar (or not).

To prevent this from happening, the Component Gallery asks you which side of the window is the default for the dialog bar. Then it enforces that the bar only docks at that edge or the parallel edge of the window. I had to modify the code in **CMainFrame** to get the picture in Figure 5.3 because the default code won't allow the dialog to dock on the wrong edge.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

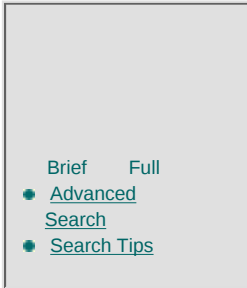
[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE



BROWSE BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Customizing Tool Bars

Another advantage that tool bars have over dialog bars is that they have support for user customization. Well, sort of. Here's the story. Older versions of MFC provided all of the code required for creating tool bars. Newer versions simply wrap the new tool bar common control. The common control has complete support for customization. However, because MFC strives to make **CToolBar** look the same as it always has, it is very difficult to make a standard tool bar customizable. Difficult, but not impossible.

What kind of customization is possible? There is support for dragging buttons around on tool bars (use the shift key along with the mouse to drag the buttons). If you drag a button off of the tool bar, it vanishes. The common control also defines a dialog box (see Figure 5.4) that allows users to add, delete, and rearrange buttons. Finally, the control has methods to save and restore the customized state to the registry.



Figure 5.4 Customizing tool bars.

The problem is that the **CToolBar** class is essentially a one-way wrapper. It uses a **CToolBarCtrl** (the common control that implements toolbars), but any changes that you make to the underlying control don't reflect in the **CToolBar** wrapper. That makes the methods to save and restore the tool bar useless because when the tool bar initializes, it won't properly reflect the changes. If you want to save and restore the tool bar, you'll need custom functions to do it.

The other features aren't too difficult to arrange for, however. To make the tool bar customizable, you just need to set the **CCS_ADJUSTABLE** style in the common control window. You can't set it when you create the normal tool bar because Microsoft uses the same flag for a different purpose. However, you can set it right afterwards, like this:

```
::SetWindowLong(m_wndToolBar.GetToolBarCtrl().m_hwnd, GWL_STYLE,
    m_wndToolBar.GetToolBarCtrl().GetStyle() | CCS_ADJUSTABLE);
```

You'll also need to handle some **WM_NOTIFY** messages sent to the main frame (or whichever window owns the tool bar). You can easily handle these using **ON_NOTIFY** message map macros. Of course, Class Wizard doesn't understand what you are doing, so you'll have to put them in the map manually.

You can find a complete list of notifications that the tool bar sends in the online documentation that comes with Visual C++. Table 5.3 shows the ones that are important for the kind of customization attempted in this chapter (see Listing 5.7).

Table 5.3 Tool bar notifications.

Notification	Description
TBN_QUERYINSERT	Can user insert this button?
TBN_QUERYDELETE	Can user delete this button?
TBN_GETBUTTONINFO	Get information about button in a TBNOTIFY structure
TBN_RESET	Reset to original tool bar
TBN_TOOLBARCHANGE	Tool bar has changed

Listing 5.7 A frame with a customizable tool bar.

```
// MainFrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
#include "tools.h"

#include "MainFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
   //{{AFX_MSG_MAP(CMainFrame)
    ON_WM_CREATE()
    ON_COMMAND(ID_CUSTOMIZE, OnCustomize)
    ON_NOTIFY(TBN_QUERYINSERT, AFX_IDW_TOOLBAR, NotifyQI)
    ON_NOTIFY(TBN_QUERYDELETE, AFX_IDW_TOOLBAR, NotifyQI)
    ON_NOTIFY(TBN_GETBUTTONINFO, AFX_IDW_TOOLBAR, NotifyInfo)
    ON_NOTIFY(TBN_RESET, AFX_IDW_TOOLBAR, NotifyReset)
    ON_NOTIFY(TBN_TOOLBARCHANGE, AFX_IDW_TOOLBAR, NotifyChange)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

static UINT indicators[] =
{
    ID_SEPARATOR,           // status line indicator
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};

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

```

// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
    // TODO: add member initialization code here
}

CMainFrame::~CMainFrame()
{
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;
    // Setting CCS_ADJUSTABLE here doesn't seem to work
    if (!m_wndToolBar.Create(this, WS_CHILD | WS_VISIBLE |
        CBRS_TOP | CCS_ADJUSTABLE | CCS_ADJUSTABLE) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;        // fail to create
    }

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
            sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar\n");
        return -1;        // fail to create
    }

    // TODO: Remove this if you don't want tool tips or a resizable
    tool bar
    m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
        CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC );

    // TODO: Delete these three lines if you don't want the tool bar to
    // be dockable
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndToolBar);
    // Make adjustable
        ::SetWindowLong(m_wndToolBar.GetToolBarCtrl().m_hWnd, GWL_STYLE,
            m_wndToolBar.GetToolBarCtrl().GetStyle() | CCS_ADJUSTABLE);
    // Restore saved state of toolbar
    RestoreTBState();
    return 0;
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CFrameWnd::PreCreateWindow(cs);
}

```

```

////////////////////
// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
    CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
    CFrameWnd::Dump(dc);
}

#endif // _DEBUG

////////////////////
// CMainFrame message handlers

// Handle QueryInsert and Delete as always true
void CMainFrame::NotifyQI(NMHDR *hdr, LRESULT *res)
{
    *res=(LRESULT)TRUE;
}

void CMainFrame::NotifyChange(NMHDR *hdr, LRESULT *res)
{
    SaveTBState();    // save all changes
}

void CMainFrame::NotifyReset(NMHDR *hdr, LRESULT *res)
{
    m_wndToolBar.LoadToolBar(IDR_MAINFRAME); // put old tool bar up
    RecalcLayout();
}

void CMainFrame::NotifyInfo(NMHDR *hdr, LRESULT *res)
{
    // The intent here is for you to supply info on all buttons,
    // even if they
    // don't appear in the bar right now.
    // That's great, but hard to do for MFC because the buttons are
    // locked up
    // in a resource.
    // My answer: Just load whatever buttons are there; if you want
    // more, reset
    // the bar and start over.
    TBNOTIFY *nfy=(TBNOTIFY *)hdr;
    int n=nfy->iItem;
    *res=(n<=m_wndToolBar.GetCount());

    if (*res) m_wndToolBar.GetToolBarCtrl().GetButton(n,&nfy->tbButton);
    //MFC Tool bars don't usually have text, so no need to put it in
}

```

```

// Restore TB from registry
void CMainFrame::RestoreTBState()
{
    CWinApp *app=AfxGetApp();
    int n=app->GetProfileInt("Toolbar","Count",0);
    if (n==0) return;
    m_wndToolBar.SetButtons(NULL,n);
    int i;
    for (i=0;i<n;i++)
    {
        int cmd,style,image;
        CString tag,root="TBTN%s%d";
        tag.Format(root,"CMD",i);
        cmd=app->GetProfileInt("Toolbar",tag,0);
        tag.Format(root,"STY",i);
        style=app->GetProfileInt("Toolbar",tag,0);
        tag.Format(root,"IMG",i);
        image=app->GetProfileInt("Toolbar",tag,0);
        m_wndToolBar.SetButtonInfo(i,cmd,style,image);
    }
    RecalcLayout();
}

// Save TB to registry
void CMainFrame::SaveTBState()
{
    CWinApp *app=AfxGetApp();
    int n=m_wndToolBar.GetToolBarCtrl().GetButtonCount();
    app->WriteProfileInt("Toolbar","Count",n);
    int i;
    for (i=0;i<n;i++)
    {
        unsigned int cmd,style;
        int image;
        CString tag,root="TBTN%s%d";
        m_wndToolBar.GetButtonInfo(i,cmd,style,image);
        tag.Format(root,"CMD",i);
        app->WriteProfileInt("Toolbar",tag,cmd);
        tag.Format(root,"STY",i);
        app->WriteProfileInt("Toolbar",tag,style);
        tag.Format(root,"IMG",i);
        app->WriteProfileInt("Toolbar",tag,image);
    }
}

// Bring up customize dialog
void CMainFrame::OnCustomize()
{
    m_wndToolBar.GetToolBarCtrl().Customize();
}

```

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

The **TBN_QUERYINSERT** and **TBN_QUERYDELETE** notifications allow you to make some buttons non-customizable. The example program simply returns **TRUE** for both of these notifications so that the entire bar is customizable.

When the user selects the Edit|Customize Toolbar menu item, the code calls **Customize**. This is the built-in function that displays the customization dialog. The control then sends **TBN_GETBUTTONINFO** notifications. This is where you tell the dialog about all possible buttons the user could place in the tool bar. This is difficult with MFC tool bars because the buttons are locked up in resources. I elected to take a different approach. In response to **TBN_GETBUTTONINFO**, I list all the buttons that are on the tool bar now. The user can then rearrange or delete buttons, but he or she can't really add any buttons that were not on the tool bar when the dialog opened.

So what happens when you remove a button and later change your mind? Simply press the Reset button. This fires off a **TBN_RESET** notification. The code reloads the original tool bar from the resources. Then you can customize it further, if you like.

The only remaining piece to the puzzle is to save the customizations. The frame window in Listing 5.7 also has two custom functions named **SaveTBState** and **RestoreTBState**. These functions use the button information for each button in the **CToolBar** (not the **CToolBarCtrl**). Simple calls to the application object take care of the registry.

While it isn't perfect, this simple collection of functions can add real pizzazz to your tool bars. The alternative is to give up on MFC tool bars and code your own around the common controls. While this is possible, it isn't an enviable

task, and it is much easier to handle it the way the code in this section does.

Customizing Common Dialogs

Common dialogs are a great idea. They allow you to easily incorporate dialogs you often need in your programs. Users like them because they make common functions work the same between different programs. MFC makes it even easier to use most of the dialogs (see Table 5.4).

Table 5.4 Common dialogs.

Dialog	MFC Class	Windows Structure	Template Location*
Color dialog	CColorDialog	CHOOSECOLOR	COLOR.DLG/COLORDLG.H
File dialog	CFileDialog	OPENFILENAME	FILEOPEN.DLG/DLGS.H
Font dialog	CFontDialog	CHOOSEFONT	FONT.DLG/DLGS.H
Print dialog	CPrintDialog	PRINTDLG	PRNSETUP.DLG/DLGS.H
Find/replace dialog [†]	CFindReplace Dialog	FINDREPLACE	FINDTEXT.DLG/DLGS.H

[†]This dialog is modeless and generally unpleasant to use from Windows or MFC

*May change depending on VC++ version

However, like most things, you often want something slightly different from the standard. The designers of the common dialogs knew this and went to great lengths to allow you to customize the dialogs. If you've ever tried to do this under the Windows API, you probably found it to be a great deal of work. MFC simplifies it to a manageable task.

All the common dialogs follow the same regimen for customization except for the explorer-style file dialog found in Windows 95 and NT 4.0. You'll see how those work in the next section. For now, any mention of common dialogs means common dialogs, except for the explorer-style file dialog.

Customizing Step By Step

Sometimes, you'll want to make simple changes to a common dialog. For example, you might want to change the label of a button. For these customizations, you can use the existing, predefined template. You'll just modify what's already there. For more complicated changes, you might want to supply your own template. Usually, you'll start with the existing templates and work from there. Here are the basic steps you'll need to customize a common dialog in either case:

1. Import the appropriate resource template; modify it to suit your needs.
2. Create a derived class with Class Wizard.

3. In the constructor, customize the underlying Windows structure. If you are replacing the resource template, set the **lpTemplateName** member to the new resource ID and add **xx_ENABLETEMPLATE** (where **xx** is the correct prefix for the dialog type) to the flags. Also, set the **hInstance** member using **AfxGetInstanceHandle**.
4. Use the standard message map mechanism to handle messages.

A Color Dialog Bug!

For some unknown reason, the **CHOOSECOLOR** structure defines its **hInstance** field as an **HWND** instead of as an **HINSTANCE**. In old-fashioned C compilers, this didn't make any difference. With C++, however, it causes errors. The only answer is to cast your instance handle to an **HWND**, even though that's clearly wrong.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

An Example Color Dialog

That sounds simple enough, right? It usually is simple, but you may have to experiment a bit to get the results you want. For example, consider the custom color dialog in Figure 5.5 (and Listing 5.8). This seems like an easy customization. It changes the label of two buttons and adds a new third button. If all you wanted to do was change the labels, you could do it all without a template. Of course, it would be possible to add the third button at runtime, too, but it is somewhat more difficult.



Figure 5.5 A custom color dialog.

Listing 5.8 A custom color dialog.

```
// customco.cpp : implementation file
//

#include "stdafx.h"
#include "color.h"
#include "customco.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

////////////////////
// CCustomColor dialog

CCustomColor::CCustomColor(COLORREF color, DWORD flag, CWnd* pParent)
```

```

        : CColorDialog (color,flag,pParent)
    {
        //{{AFX_DATA_INIT(CCustomColor)
        // NOTE: the Class Wizard will add member initialization here

        //}}AFX_DATA_INIT
        m_reset=FALSE;
    /* m_cc is a CHOOSECOLOR */
        m_cc.lpszTemplateName="CHOOSECOLOR";
        m_cc.Flags|=CC_ENABLETEMPLATE;    // don't disturb existing flags!
    // funny that the hInstance variable is declared as an HWND?
        m_cc.hInstance=(HWND)AfxGetInstanceHandle();
    }

    void CCustomColor::DoDataExchange(CDataExchange* pDX)
    {
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(CCustomColor)
        // NOTE: the Class Wizard will add DDX and DDV calls here
        //}}AFX_DATA_MAP
    }

    BEGIN_MESSAGE_MAP(CCustomColor, CColorDialog)
        //{{AFX_MSG_MAP(CCustomColor)
        ON_BN_CLICKED(IDC_RESET, OnReset)
        //}}AFX_MSG_MAP
    END_MESSAGE_MAP()

    ///////////////////////////////////
    // CCustomColor message handlers

    void CCustomColor::OnReset()
    {
        m_reset=TRUE;
        EndDialog(IDOK);
    }

    BOOL CCustomColor::OnInitDialog()
    {
        BOOL result=CColorDialog::OnInitDialog();

        CWnd *newbutton=GetDlgItem(IDC_RESET);
        newbutton->ShowWindow(SW_SHOW);
        newbutton->EnableWindow();
        return result;
    }

```

The program in Listing 5.8 uses a custom template to do all the work. However, without some special work, the new button doesn't show up. Why? The color dialog is a bit odd. It starts off as a small box, with a button at the bottom that expands the dialog to its full size. Apparently, when the dialog starts, it hides all controls that belong to it. Then it decides what state it is in and shows the appropriate controls. However, your new control isn't on the list that it uses to initialize things. So if you want the button visible, you'll have to set it to visible yourself (for example, in the **OnInitDialog** call).

The reset button sets a flag (**m_reset**) and dismisses the dialog. The calling program has to examine the flag to determine if the user clicked on the reset button. This provides the most flexibility because any program that uses the color dialog can decide what reset means.

Customizing any of the other dialogs is almost the same as working with the color dialog. In fact, the others are generally easier because they don't play games with their controls and don't have bugs in their data structures. However, don't forget that the explorer-style file open dialog doesn't follow the same form as the other dialogs.

Customizing File Open

The new style file open dialog is a bit more trouble to customize if you need a template because this dialog doesn't allow you to replace the existing template. Instead, you give it a template, which makes that template a child dialog of the existing dialog. You can control the placement of the existing controls, and you can add your own. There are also special messages you can send to the dialog in order to control its appearance.

Your template should have the child style, the 3D look, and no border. It should also have the visible, control, and clip siblings styles set. If you want to control where the ordinary part of the dialog will appear, use an invisible control (a static control will do) with the special ID **stc32**. Wherever this control appears is where Windows will place the standard dialog controls. However, because Visual C++ doesn't understand this identifier (found in DLGS.H), it adds a new identifier with the same name. You'll have to open up RESOURCE.H and delete the **stc32** identifier. Unfortunately, this often confuses Class Wizard.

Once you have all of this done, you only need to derive a new class from **CFileDialog** (Class Wizard will do this for you). You'll find that because your controls aren't really part of the file dialog, you won't see them in Class Wizard. You'll have to manually handle your message map entries. To associate your template with the dialog, place the following line in the constructor:

```
SetTemplate(0, IDD_CUSTOMTEMPLATE);
```

Substitute the name of your template for the second argument. The first argument is the template to use for a non-explorer dialog, so you can write code that has a custom dialog on any platform, if you wish.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

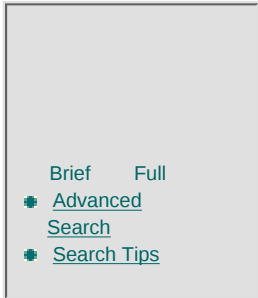
[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE



BROWSE BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

The MFC wrapper doesn't expose everything you might want as a member function. However, you can always send the dialog box messages (see Table 5.5). Just remember that your dialog class is the child dialog, so these messages must go to the parent window (the common dialog).

Table 5.5 File dialog messages.

Message	Meaning
CDM_GETFILEPATH	Get full file name
CDM_GETSPEC	Get file name (without path information)
CDM_GETFOLDERPATH	Get path only
CDM_HIDECONTROL	Hide a control (see DLGS.H and WINUSER.H for IDs)
CDM_SETCONTROLTEXT	Set text in a control (see DLGS.H and WINUSER.H for IDs)
CDM_SETDEFEXT	Set default extension

You'll find an example custom file dialog in Figure 5.6 and Listing 5.9. The dialog box has some extra text, including a box to show the current directory. The accompanying program doesn't use the dialog for its normal File menu operations (although it could, using the techniques in Chapter 2).



Figure 5.6 A custom file dialog box.

Listing 5.9 A custom file dialog.

```
// CustomFile.cpp : implementation file
//

#include "stdafx.h"
```

```

#include "custfile.h"
#include "CustomFile.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////
// CCustomFile

IMPLEMENT_DYNAMIC(CCustomFile, CFileDialog)

CCustomFile::CCustomFile(BOOL bOpenFileDialog, LPCTSTR lpszDefExt,
                        LPCTSTR lpszFileName,

                        DWORD dwFlags, LPCTSTR lpszFilter, CWnd* pParentWnd) :
                        CFileDialog(bOpenFileDialog, lpszDefExt, lpszFileName,
                        dwFlags,
                        lpszFilter, pParentWnd)
{
    SetTemplate(NULL, "FileTemplate"); // set resource template
}

BEGIN_MESSAGE_MAP(CCustomFile, CFileDialog)
   //{{AFX_MSG_MAP(CCustomFile)
    ON_COMMAND(IDC_TRYBTN, OnTry)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

void CCustomFile::OnTry() // from the ministry of silly messages
{
    MessageBox("You can try, but it doesn't do any good.");
}

// Set directory static text
void CCustomFile::OnFolderChange()
{
    {
        char dirbuf[256];
        if (GetParent()->SendMessage(CDM_GETFOLDERPATH, sizeof(dirbuf),
        (LPARAM)dirbuf))
            SetDlgItemText(IDC_DIR, dirbuf);
    }
}

```

Summary

Dialogs are a powerful part of Windows programming. Without them, even the simplest interfaces would require custom programming. MFC provides classes that let you work with dialogs as views, tool bars, or even ordinary dialogs. Even the common dialogs have their own classes that you can customize.

Thanks to DDX and DDV, you can often write simple modal dialogs that require practically no coding on your part. By adding your own DDX and DDV routines, you can increase your options for getting by with less work.

Although dialog bars provide an alternative to tool bars, tool bars are easier on your resources, and they can swivel around automatically. The customization of tool bars is another feature that would be difficult to implement using dialog bars.

Practical Guide To Dialogs

- Creating Modeless Dialogs
- Updating DDX Variables On Changes
- Live Data Validation
- Writing Custom DDX/DDV Routines
- Integrating Custom DDX/DDV With Class Wizard
- Dialog Bars Vs. Tool Bars
- Customizing Common Dialogs

Dialogs are the workhorses of Windows programs. MFC recognizes that and gives you a great deal of support for them in both the framework and the tools.

Creating Modeless Dialogs

Technically, creating a modeless dialog is exactly the same as creating a modal dialog. The only difference is that you call **Create** (and pass the resource ID as an argument) instead of calling **DoModal**. In practice, there are several problems you'll encounter. First, the default handlers for **OnOK** and **OnCancel** call **EndDialog**. This call only works for modal dialogs. Even if you don't have OK and Cancel buttons, you can still get these calls under certain circumstances. For example, pressing the Escape key can call **OnCancel** even if there is no Cancel button. The safe thing to do is override **OnOK** and **OnCancel**; do not call **EndDialog** (instead call **DestroyWindow**).

Usually, you create modal dialogs on the stack. Why not? Think about it. When does the dialog variable go out of scope? When the function that creates the dialog exits. However, the function can't exit until the user dismisses the dialog box. Therefore, by the time you destroy the dialog object, the user is done with it. You can't count on that with modeless dialogs because they have a life that normally far exceeds the scope of a stack variable. If you create modeless dialogs as part of another class (like a frame class, for example), you won't have any problems. However, it's not uncommon to create modeless dialogs on the heap using **new**. This isn't a problem, except that you need to be sure to eventually call **delete** in order to free the memory the dialog box uses.

You could also code your program to keep track of the dialog box and call **delete** at the correct time. This isn't a very elegant solution. You'd rather make the dialog box delete itself when the window terminates. There is a simple way to do this: Override **PostNCDestroy** and call **delete this** from inside that routine.

You can incorporate both of these solutions in a class that you can use especially for modeless dialogs (see **CModelessDialog** in Listing 5.1). The code in Listing 5.1 calls **delete** and **DestroyWindow** at the proper times. It also automatically initiates a DDX transfer in response to the OK button.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Updating DDX Variables On Changes

Although DDX makes it appear that you have variables attached to dialog box controls, the reality of the situation is a bit different. MFC copies the contents of your variables to the controls during dialog initialization, and then it reverses the copy when the user clicks on OK.

Sometimes you want transfers to occur at other times, so you need to call **UpdateData** manually. If you wanted any change on the dialog to reflect immediately when the user changes a field, you could manually wire each field's change event to a routine that calls **UpdateData**.

Although that will work, it isn't a very tidy solution because it requires a separate message map entry for each field. Also, when you add fields, you have to remember to add message map entries, too. A better approach is to override **OnCommand** (and possibly **OnNotify** if you are using controls that send **WM_NOTIFY** messages). Then, when any changes occur, your routine will execute so you can call **UpdateData**. This doesn't interfere with the normal command message routine. You can find an example of this technique in Listing 5.1.

Be careful if you are validating data with DDV when using this technique. Every change will trigger a transfer, even if the data isn't valid yet. This can cause spurious DDV errors. If you need to use DDV, you might consider turning DDV off using a flag that you set in your **OnCommand** handler. Then you reset the flag so that DDV functions for ordinary transfers. (See the next section for examples of modifying the DDV data map.)

Live Data Validation

DDV validates data based on criteria that you specify from Class Wizard. However, the validation step only occurs when the data transfers. Most users would prefer to know as soon as possible that the data is not good.

If you want to validate data on a character-by-character basis, you'll need to subclass the control (see Chapter 4 for more about subclassing). DDV, however, can be coerced into checking each field as the user completes the data entry in a field and moves to another one. The trick is to recognize that what MFC calls a data map is just a simple function called **DoDataExchange**. You can modify this function in many ways. For example, you can set a flag that instructs the function to validate only one field. Then, when a field loses the focus, you can set that flag and call **UpdateData**. The modified **DoDataExchange** function will then only validate the single field. You can find a complete example of this live validation technique in Listing 5.3.

Writing Custom DDX/DDV Routines

Armed with the realization that **DoDataExchange** is just a function, you can easily create your own custom DDX and DDV routines. The key is the **CDataExchange** structure (see Table 5.1). This structure contains information about the transfer in progress. The best way to get started is to look at some of the predefined functions in the MFC source or the examples in this chapter (see Listing 5.5).

You can also do interesting things by modifying **DoDataExchange** (outside of the Class Wizard comments, of course). For example, you might use a variable as one of the validation parameters. Or, you can control transfers or validation based on conditions you set.

Integrating Custom DDX/DDV With Class Wizard

Once you have custom DDX and DDV routines, you can add them to Class Wizard. The process requires that you place some lines in your .CLW file (if you want the changes for one project) or the DDX.CLW file (if you want custom routines for all projects).

The syntax is a bit strange, but it's not too difficult. Just make sure your routines begin with the prefix **DDX_** and **DDV_**. You can find details about the syntax for the .CLW file in Table 5.2.

Dialog Bars Vs. Tool Bars

By default, App Wizard places a tool bar at the top of your main frame. This tool bar acts as a graphical menu and contains buttons. Or does it? The buttons in a tool bar are actually pieces of a single bitmap (sometimes disguised as a tool bar resource). If you want to place buttons (and other controls) in a bar, you'd need to use the **CDialogBar** class instead of the **CToolBar** class.

Each class has its advantages and disadvantages. **CToolBar** doesn't consume many resources. It also can pivot automatically from a horizontal to a vertical

orientation. On the other hand, it is difficult to place non-button controls in a tool bar. Even if you want different size buttons, you'll find it difficult to manage.

Dialog bars are easy to create (even if you want non-button controls). However, they can't pivot around automatically, and they usually consume more resources than a similar tool bar.

One other advantage to a tool bar is that it is (theoretically) possible to write code to customize one using code built-in to Windows. You can drag buttons around, delete buttons, and use a system-defined dialog to organize buttons. However, with MFC, this turns out to be fairly difficult (although you'll find an actual example in Listing 5.7).

If you want a dialog bar instead of a tool bar, you can create a program without a tool bar using App Wizard and then use Component Gallery to add a dialog bar to the program.

Customizing Common Dialogs

Common dialogs are easy to customize using MFC. If you don't need to supply a custom dialog template, you can easily derive a new class from the corresponding MFC class (see Table 5.4). Then use the standard message map mechanism to handle messages (including things like **WM_INITDIALOG**) and do your work in the message handlers.

If you need a dialog template and you are not working with the explorer-style file dialog, you'll need to take the following steps:

1. Import the appropriate resource template and modify it to suit your needs.
2. Create a derived class with Class Wizard.
3. In the constructor, customize the underlying Windows structure. If you are replacing the resource template, set the **lpTemplateName** member to the new resource ID and add **xx_ENABLETEMPLATE** (where **xx** is the correct prefix for the dialog type) to the flags. Also, set the **hInstance** member using **AfxGetInstanceHandle**.
4. Use the standard message map mechanism to handle messages.

If you are using the explorer-style file dialog, you'll need to supply a template that has only your custom controls and (optionally) a hidden control with the special identifier **stc32**. This hidden control tells Windows where to place the default controls. Your template needs to have the **WS_CHILD**, **WS_CLIPSIBLINGS**, **WS_VISIBLE**, **DS_3DLOOK**, and **DS_CONTROL** styles. Inside the class constructor, call **SetTemplate** to inform the class what template you want to use.

Once you open the dialog, you can control and query the dialog by sending the messages in Table 5.5 to the parent window of your custom dialog. There is an example of this in Listing 5.9.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Chapter 6

Property Sheets And Wizards

Property sheets are like dialog boxes with an attitude. Although many of the techniques you'll use with property sheets are similar to dialog box manipulation, property sheets have some unique aspects. You can construct modeless property sheets and wizards, and you can even create custom App Wizards to generate new programs to your own specifications.

Programmer's Notes...

I don't have many hobbies. I don't seem to have time to do very much, I guess. I like to fish, and I coach Little League on occasion. Other than that, all of my hobbies fall into the geek category. Obviously, I enjoy computers. I've also been an amateur (or ham) radio operator for more than 20 years (my call sign, if you care, is WD5GNR).

A lot has changed in ham radio over the past 20 years. Most of the changes can be traced back to computers. For example, most ham radios now contain microprocessors. When I celebrated my twentieth anniversary as a ham, I decided to treat myself to a brand new radio with all the modern gadgets, a Kenwood TS570D. This isn't the top of the line, but it isn't the bottom of the barrel, either. In fact, if you watch *Jurassic Park: The Lost World*, you'll see one in the souped-up RV they drive around.

All of the new radios on the market have fewer switches and knobs than older

models. Of course, they do more than their older counterparts. They just do it with a better user interface. An old radio would have dozens of knobs and switches. However, you didn't use more than a few of them at a time. You also didn't use but a handful of controls on a regular basis. Still, you needed each knob and switch sometime, so the radio had to include them.

Today, a microprocessor controls all the functions. Therefore, the switches and knobs mostly just send signals to the processor. The processor actually does the work. That means that a single knob can perform different functions.

Some very small radios intended for use in a car, boat, or plane hardly have any knobs anymore. The whole radio runs from a menu and only needs a handful of buttons. Hand-held radios are another good example. In times past, a hand-held radio never had many features. Where would you put all the knobs? Now, using a menu system, the radio can have practically unlimited features.

If you think about it, radio designers use microprocessors to organize functions, showing only the ones you need to work with at a particular time. You can see the same evolution in computer user interfaces. With visual tools, it is easy to get carried away and create huge dialog boxes full of every control you might want. However, this quickly intimidates and confuses users. Even developers and power users don't want to see 100 different options in a dialog box.

At first, programmers tried to hide parts of the dialog box. If you wanted more control (or information), you pressed a button titled Advanced, or Details. Then the dialog would grow in size and more controls would appear. That was a step in the right direction, but it only hid the complex dialog until you needed it.

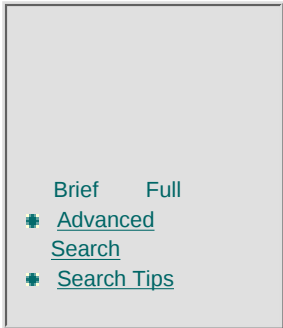
A better idea is to use a dialog box that contains tabs. Each tab corresponded to a related page of controls. Radio operators only need a few buttons at a time; users usually only need a few controls at a time.

Some intrepid programmers started creating tabbed dialogs by hand (no mean feat), and the idea became very popular. Microsoft added support for them in MFC; it also added some support for tabs in general in the base Windows API (actually, in the common controls). Microsoft calls tabbed dialogs *property sheets*.

Some prime examples of property sheets appear in Visual C++ itself. Pull down the Tools|Options or Project|Settings menu items. How would you like all that information to appear in one dialog? Not that having it on 15 or more dialogs would be any better. The property sheets make a daunting number of options manageable.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97



Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Property Sheet Overview

If you haven't worked with property sheets before, you won't find them much different from ordinary dialogs. You simply create a dialog template for each tab. The title of the dialog becomes the text that appears in the corresponding tab. You can use DDX, DDV, and Class Wizard. The dialog template needs to have certain styles turned on (and off), but you don't have to worry about that. Just right-click in the resource view and select Insert from the menu (don't pick Insert Dialog). This shows you a dialog like the one in Figure 6.1. Click on the plus sign next to Dialog to expand it and you'll see that there are three sizes of predefined property sheets there. Select one and you are ready to go.

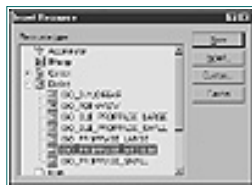


Figure 6.1 Inserting a property sheet.

When you create a dialog template and invoke Class Wizard, the Wizard notices that you have a new dialog and offers to create a new class for you. It suggests using **CDialog** as the base class. When you create a property sheet dialog, the same thing happens. However, you'll want to choose **CPropertyPage** as the base class instead of **CDialog**.

When you are ready to create an instance of the property sheet, you'll need an instance of **CPropertySheet** (or a class derived from **CPropertySheet**). You also make an instance of each **CPropertyPage**-derived class you are using.

These are the classes Class Wizard creates for you after you finish your dialog template.

You can use ordinary DDX calls on the **CPropertyPage** objects. You attach the templates to the property sheet using **CPropertySheet::AddPage**. Finally, you call **CPropertySheet::DoModal** to show the property sheet.

Here is a typical example:

```
CPropertySheet sheet("Example Property Sheet");
CPropPg1 prop1; // derived from CPropertyPage
CPropPg2 prop2; // derived from CPropertyPage
sheet.AddPage(&prop1);
sheet.AddPage(&prop2);
prop1.m_value1=100; // DDX
prop1.m_value2 = "Test"; // DDX
. . .
prop2.m_position=25; // more DDX
if (sheet.DoModal()==IDOK)
{
    // reverse DDX
    MessageBox(prop1.m_value2,"Returned data");
    . . .
}
```

This code is similar to what you'd expect for an ordinary dialog. The only difference is that you can have two or more “dialogs” (actually **CPropertyPage** objects).

There are several members you might want to override (see Table 6.1). With these functions, you can learn when the user moves to another tab or clicks on OK, Apply, or Cancel. You can also call members to alter some of these buttons. The **CancelToClose** function, for example, changes the caption of the Cancel button to read Close. This is usually because you've made some change that can't be canceled and you want the user to know that pressing the Cancel button will only dismiss the property sheet. Another similar call is **SetModified**. This call enables the Apply button.

Using A Single Template

Sometimes you'd like to have a property sheet that shows the same dialog template on more than one tab. For example, look at Figure 6.2. This shows a property sheet you might see in a checker game. It shows the number of red and black pieces. Each page is the same, but one shows red pieces and the other displays black pieces.



Figure 6.2 A checker board property sheet with one template.

Table 6.1 CPropertyPage overrideables.

Function	Description
OnCancel	Called by the framework when the Cancel button is clicked.
OnKillActive	Called by the framework when the current page is no longer the active page. Perform data validation here.
OnOK	Called by the framework when the OK, Apply Now, or Close button is clicked.
OnSetActive	Called by the framework when the page is made the active page.
OnApply	Called by the framework when the Apply Now button is clicked.
OnReset	Called by the framework when the Cancel button is clicked.
OnQueryCancel	Called by the framework when the Cancel button is clicked, and before the cancel has taken place.

The obvious approach would be to design two identical templates and name them “Red” and “Black.” That would work, but it is an unappealing solution. Every time you change one template, you have to remember to change the other one. Plus it isn’t a very efficient answer because it requires two templates where common sense tells you that you should only need one.

You can use a single template for multiple pages in your dialog. The trick is that each one needs to set a different title. The way to do this is to pass the correct title as the second argument to the **CPropertyPage** constructor. There are a couple of catches, however. First, the string must be a numeric ID in the string table. You can’t just pass a string.

The other problem is that the normal Class Wizard-generated class has only a default constructor. This constructor calls the base class constructor with the correct first argument (the dialog template ID). When you try to change the constructor to take an argument for the tab caption, your program will no longer compile. Why? Because the property page uses the **DECLARE_DYNCREATE** macro. This macro expects a default constructor. It is very unlikely that you really need this macro, but Class Wizard puts it in anyway.

You have three choices on how to resolve this problem. First, you could remove the **DECLARE_DYNCREATE** macro (and the associated **IMPLEMENT_DYNCREATE** macro). Second, you could leave the default constructor alone and add a second constructor that takes an argument. The third alternative is to provide a default argument to the constructor so that it takes 0 or 1 arguments. This satisfies the dynamic creation macro and allows you to set the ID when you want.

You can find an example property sheet page in Listing 6.1. The view that uses it appears in Listing 6.2. Notice that unlike the previous example, each property page requires a constructor argument (and the corresponding string

table entry).

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

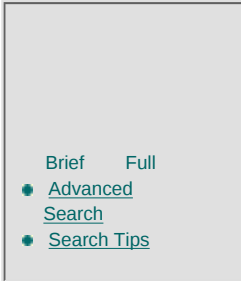
[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE



BROWSE BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Would it work to make two separate templates? Of course. But what a drag to have two identical templates. Although using a single template takes more work, it is well worth the effort when you (or someone) must maintain your program later.

Listing 6.1 A property sheet.

```
// statuspg.cpp : implementation file
//

#include "stdafx.h"
#include "chkprop.h"
#include "statuspg.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

////////////////////////
// StatusPage property page

IMPLEMENT_DYNCREATE(StatusPage, CPropertyPage)

StatusPage::StatusPage(UINT title):CPropertyPage(StatusPage::IDD,title)
{
    //{AFX_DATA_INIT(StatusPage)
    m_kings = 0;
    m_pieces = 0;
    //}}AFX_DATA_INIT
}

StatusPage::~StatusPage()
{
}

void StatusPage::DoDataExchange(CDataExchange* pDX)
```

```

{
    CPropertyPage::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(StatusPage)
    DDX_Text(pDX, IDC_KINGS, m_kings);
    DDX_Text(pDX, IDC_PIECES, m_pieces);
   //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(StatusPage, CPropertyPage)
   //{{AFX_MSG_MAP(StatusPage)
    // NOTE: the ClassWizard will add message map macros here
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// StatusPage message handlers

```

Listing 6.2 Using the property sheet.

```

// chkpropView.cpp : implementation of the CChkpropView class
//

#include "stdafx.h"
#include "chkprop.h"

#include "chkpropDoc.h"
#include "chkpropView.h"
#include "statuspg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CChkpropView

IMPLEMENT_DYNCREATE(CChkpropView, CView)

BEGIN_MESSAGE_MAP(CChkpropView, CView)
   //{{AFX_MSG_MAP(CChkpropView)
    ON_WM_LBUTTONDOWN()
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CChkpropView construction/destruction

CChkpropView::CChkpropView()
{
    // TODO: add construction code here
}

CChkpropView::~CChkpropView()
{
}

BOOL CChkpropView::PreCreateWindow(CREATESTRUCT& cs)

```

```

{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CView::PreCreateWindow(cs);
}

////////////////////////////////////
// CChkpropView drawing

void CChkpropView::OnDraw(CDC* pDC)
{
    CChkpropDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: add draw code for native data here
}

////////////////////////////////////
// CChkpropView diagnostics

#ifdef _DEBUG
void CChkpropView::AssertValid() const
{
    CView::AssertValid();
}

void CChkpropView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CChkpropDoc* CChkpropView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CChkpropDoc)));
    return (CChkpropDoc*)m_pDocument;
}
#endif // _DEBUG

////////////////////////////////////
// CChkpropView message handlers

void CChkpropView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CPropertySheet sheet("Checker Counts");
    StatusPage redpage(IDS_REDSTRING), blackpage(IDS_BLKSTRING);
    redpage.m_kings=0;
    redpage.m_pieces=9;
    blackpage.m_kings=2;
    blackpage.m_pieces=10;
    sheet.AddPage(&redpage);
    sheet.AddPage(&blackpage);
    if (sheet.DoModal()==IDOK)
    {
        // prop sheet was for information only, so we don't really care
    }
}

```

Wizard Mode

Of course, another time you use property sheets is when you write wizards. You see wizards all the time in various Microsoft products, but you don't seem to see them that often anywhere else. There's no reason for that because MFC makes wizards quite simple.

Creating a wizard is just like creating a property sheet with the exception of two minor differences. First, before calling **DoModal**, you have to call **CPropertySheet::SetWizardMode**. Second, in each property page object, override **OnSetActive**. Within that function, get the parent window (the actual property sheet) and call **CPropertySheet::SetWizardButtons** to set the buttons that will appear (see Table 6.2). You'll notice that the Next and Finish buttons are actually the same button—you can't turn them both on at once.

You can find an example wizard dialog page and the program that invokes it in Listings 6.3 and 6.4. If you don't override **OnSetActive**, the buttons will read Back and Next. At first, you might think that only the last page needs to change the Next button to Finish. That isn't the case, however. Suppose that the user goes to the last page and that page changes the Next button so that it reads Finish. Then the user presses Back. The Next button will still read Finish unless the second page explicitly changes it. That's why all of the pages need an **OnSetActive** handler. Besides, the first page doesn't need a Back button, either.

Table 6.2 Wizard buttons.

Button	Constant
Back	PSWIZB_BACK
Next	PSWIZB_NEXT
Finish	PSWIZB_FINISH
Disabled Finish	PSWIZB_DISABLEDFINISH

Listing 6.3 One page of a wizard-style property sheet.

```
// Page1.cpp : implementation file
//

#include "stdafx.h"
#include "wiz.h"
#include "Page1.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// Page1 property page

IMPLEMENT_DYNCREATE(Page1, CPropertyPage)

Page1::Page1() : CPropertyPage(Page1::IDD)
{
    //{AFX_DATA_INIT(Page1)
    m_name = _T("");
    //}}AFX_DATA_INIT
}

Page1::~Page1()
{
}
```

```

void Page1::DoDataExchange(CDataExchange* pDX)
{
    CPropertyPage::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(Page1)
    DDX_Text(pDX, IDC_NAME, m_name);
   //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(Page1, CPropertyPage)
   //{{AFX_MSG_MAP(Page1)
    // NOTE: the Class Wizard will add message map macros here
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

//////////////////////////
// Page1 message handlers

BOOL Page1::OnSetActive()
{
    CPropertySheet *sheet=(CPropertySheet *)GetParent();
    // each page sets appropriate buttons
    sheet->SetWizardButtons(PSWIZB_NEXT);
    return CPropertyPage::OnSetActive();
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

• [Advanced](#)[Search](#)• [Search Tips](#)BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Listing 6.4 Using the wizard.

```
// wizView.cpp : implementation of the CWizView class
//

#include "stdafx.h"
#include "wiz.h"

#include "wizDoc.h"
#include "wizView.h"
#include "page1.h"
#include "page2.h"
#include "page3.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CWizView

IMPLEMENT_DYNCREATE(CWizView, CView)

BEGIN_MESSAGE_MAP(CWizView, CView)
    //{AFX_MSG_MAP(CWizView)
    ON_WM_LBUTTONDOWN()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

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

```

// CWizView construction/destruction

CWizView::CWizView()
{
    // TODO: add construction code here
}
CWizView::~~CWizView()
{
}

BOOL CWizView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CView::PreCreateWindow(cs);
}

////////////////////////////////////
// CWizView drawing

void CWizView::OnDraw(CDC* pDC)
{
    CWizDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: add draw code for native data here
}

////////////////////////////////////
// CWizView diagnostics

#ifdef _DEBUG
void CWizView::AssertValid() const
{
    CView::AssertValid();
}

void CWizView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CWizDoc* CWizView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CWizDoc)));
    return (CWizDoc*)m_pDocument;
}
#endif // _DEBUG

////////////////////////////////////
// CWizView message handlers

void CWizView::OnLButtonDown(UINT nFlags, CPoint point)

```

```

{
    CPropertySheet sheet;
    Page1 pg1;
    Page2 pg2;
    Page3 pg3;
    sheet.AddPage(&pg1);
    sheet.AddPage(&pg2);
    sheet.AddPage(&pg3);
    sheet.SetWizardMode();
    if (sheet.DoModal()==ID_WIZFINISH)
    {
        CString greet="Thank You ";
        greet+=pg1.m_name;
        MessageBox("You may pass",greet);
    }
    else
    {
        MessageBox("You are flung into the abyss");
    }
}

```

Modeless Property Sheets

Creating a modeless property sheet isn't too much different from creating a modeless dialog. You simply call **Create** instead of **DoModal**. Of course, just like a dialog, a modeless property sheet doesn't explicitly transfer data, so you'll have to manually transfer using **UpdateData** (see Chapter 5).

One problem with modeless property sheets is scope. Modal dialogs and property sheets typically use stack variables because their lifetime will never extend past the scope of these variables. Modeless dialogs and property sheets, on the other hand, usually outlive the functions that create them. Therefore, you have to create them (and everything related to them) on the heap or as part of an object that will live throughout the property page's lifetime.

One handy technique (used in Listings 6.5 through 6.8) is to derive a class from **CPropertySheet**. This new class can contain member variables for each page in the property sheet. In the derived class constructor, you call **AddPage** for each page. You then create an instance of the property sheet class as you would any other property sheet object.

Listing 6.5 The modeless property sheet.

```

// MouseSheet.cpp : implementation file
//

#include "stdafx.h"
#include "mdlsprop.h"
#include "MouseSheet.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CMouseSheet

```

```

IMPLEMENT_DYNAMIC(CMouseSheet, CPropertySheet)

CMouseSheet::CMouseSheet(UINT nIDCaption, CWnd* pParentWnd, UINT
iSelectPage)
    :CPropertySheet(nIDCaption, pParentWnd, iSelectPage)
{
}

CMouseSheet::CMouseSheet(LPCTSTR pszCaption, CWnd* pParentWnd, UINT
iSelectPage)
    :CPropertySheet(pszCaption, pParentWnd, iSelectPage)
{
    leftct=rightct=midct=0;
    AddPage(&m_clickpage);
    AddPage(&m_pospage);
}

CMouseSheet::~CMouseSheet()
{
}

BEGIN_MESSAGE_MAP(CMouseSheet, CPropertySheet)
    //{AFX_MSG_MAP(CMouseSheet)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

//////////////////////////
// CMouseSheet message handlers

BOOL CMouseSheet::OnInitDialog()
{
    return CPropertySheet::OnInitDialog();
}

void CMouseSheet::SetXY(int x, int y)
{
    m_pospage.m_x=x;
    m_pospage.m_y=y;
    if (::IsWindow(m_pospage.m_hWnd)) m_pospage.UpdateData(FALSE);
}

void CMouseSheet::MouseDown(int cmd)
{
    switch (cmd)
    {
        case WM_LBUTTONDOWN:
            leftct++;
            break;
        case WM_RBUTTONDOWN:
            rightct++;
            break;
        case WM_MBUTTONDOWN:
            midct++;
            break;
    }
}

```

```

    }
    m_clickpage.m_left=leftct;
    m_clickpage.m_right=rightct;
    m_clickpage.m_middle=midct;
    if (::IsWindow(m_clickpage.m_hWnd)) m_clickpage.UpdateData(FALSE);
}

```

Listing 6.6 The position page.

```

// MouseSheet.cpp : implementation file
//

#include "stdafx.h"
#include "mdlsprop.h"
#include "MouseSheet.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////
// CMouseSheet

IMPLEMENT_DYNAMIC(CMouseSheet, CPropertySheet)

CMouseSheet::CMouseSheet(UINT nIDCaption, CWnd* pParentWnd, UINT
iSelectPage)
    :CPropertySheet(nIDCaption, pParentWnd, iSelectPage)
{
}

CMouseSheet::CMouseSheet(LPCTSTR pszCaption, CWnd* pParentWnd, UINT
iSelectPage)
    :CPropertySheet(pszCaption, pParentWnd, iSelectPage)
{
    leftct=rightct=midct=0;
    AddPage(&m_clickpage);
    AddPage(&m_pospage);
}

CMouseSheet::~CMouseSheet()
{
}

BEGIN_MESSAGE_MAP(CMouseSheet, CPropertySheet)
    //{AFX_MSG_MAP(CMouseSheet)
    //{AFX_MSG_MAP
END_MESSAGE_MAP()

//////////////////////////
// CMouseSheet message handlers

BOOL CMouseSheet::OnInitDialog()

```

```

{
    return CPropertySheet::OnInitDialog();
}

void CMouseSheet::SetXY(int x, int y)
{
    m_pospage.m_x=x;
    m_pospage.m_y=y;
    if (::IsWindow(m_pospage.m_hWnd)) m_pospage.UpdateData(FALSE);
}

void CMouseSheet::MouseDown(int cmd)
{
    switch (cmd)
    {
        case WM_LBUTTONDOWN:
            leftct++;
            break;
        case WM_RBUTTONDOWN:
            rightct++;
            break;
        case WM_MBUTTONDOWN:
            midct++;
            break;
    }
    m_clickpage.m_left=leftct;
    m_clickpage.m_right=rightct;
    m_clickpage.m_middle=midct;
    if (::IsWindow(m_clickpage.m_hWnd)) m_clickpage.UpdateData(FALSE);
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US



SEARCH

ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)



BROWSE

BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Listing 6.7 The Click page.

```
// ClickPage.cpp : implementation file
//

#include "stdafx.h"
#include "mdlsprop.h"
#include "ClickPage.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////
// CClickPage property page

IMPLEMENT_DYNCREATE(CClickPage, CPropertyPage)

CClickPage::CClickPage() : CPropertyPage(CClickPage::IDD)
{
   //{{AFX_DATA_INIT(CClickPage)
    m_left = 0;
    m_middle = 0;
    m_right = 0;
    //}}AFX_DATA_INIT
}

CClickPage::~CClickPage()
{
}

void CClickPage::DoDataExchange(CDataExchange* pDX)
{
    CPropertyPage::DoDataExchange(pDX);
```

```

   //{{AFX_DATA_MAP(CClickPage)
    DDX_Text(pDX, IDC_LEFT, m_left);
    DDX_Text(pDX, IDC_MIDDLE, m_middle);
    DDX_Text(pDX, IDC_RIGHT, m_right);
   //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CClickPage, CPropertyPage)
   //{{AFX_MSG_MAP(CClickPage)
    // NOTE: the Class Wizard will add message map macros here
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CClickPage message handlers

```

Listing 6.8 Using the modeless property sheet.

```

// mdlpropView.cpp : implementation of the CmdlpropView class
//

#include "stdafx.h"
#include "mdlprop.h"

#include "mdlpropDoc.h"
#include "mdlpropView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CmdlpropView

IMPLEMENT_DYNCREATE(CmdlpropView, CView)

BEGIN_MESSAGE_MAP(CmdlpropView, CView)
   //{{AFX_MSG_MAP(CmdlpropView)
    ON_WM_LBUTTONDOWN()
    ON_WM_MOUSEMOVE()
    ON_WM_RBUTTONDOWN()
    ON_WM_MBUTTONDOWN()
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CmdlpropView construction/destruction

CmdlpropView::CmdlpropView() : psheet("Mouse Properties")
{
    // TODO: add construction code here
}

CmdlpropView::~CmdlpropView()
{
}

```

```

BOOL CmdlspropView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CView::PreCreateWindow(cs);
}

////////////////////////////////////
// CmdlspropView drawing

void CmdlspropView::OnDraw(CDC* pDC)
{
    CmdlspropDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: add draw code for native data here
}

////////////////////////////////////
// CmdlspropView diagnostics

#ifdef _DEBUG
void CmdlspropView::AssertValid() const
{
    CView::AssertValid();
}

void CmdlspropView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CmdlspropDoc* CmdlspropView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CmdlspropDoc)));
    return (CmdlspropDoc*)m_pDocument;
}
#endif // _DEBUG

////////////////////////////////////
// CmdlspropView message handlers

void CmdlspropView::OnLButtonDown(UINT nFlags, CPoint point)
{
    if (::IsWindow(psheet.m_hWnd)) psheet.MouseDown(WM_LBUTTONDOWN);
    CView::OnLButtonDown(nFlags, point);
}

void CmdlspropView::OnMouseMove(UINT nFlags, CPoint point)
{
    if (::IsWindow(psheet.m_hWnd)) psheet.SetXY(point.x, point.y);
    CView::OnMouseMove(nFlags, point);
}

void CmdlspropView::OnRButtonDown(UINT nFlags, CPoint point)
{
    if (::IsWindow(psheet.m_hWnd)) psheet.MouseDown(WM_RBUTTONDOWN);
}

```

```

    CView::OnRButtonDown(nFlags, point);
}

void CmdlspropView::OnMButtonDown(UINT nFlags, CPoint point)
{
    if (::IsWindow(psheet.m_hWnd)) psheet.MouseDown(WM_MBUTTONDOWN);
    CView::OnMButtonDown(nFlags, point);
}

void CmdlspropView::OnInitialUpdate()
{
    CView::OnInitialUpdate();
    psheet.Create(this, DS_MODALFRAME | DS_3DLOOK | DS_CONTEXTHELP |
        DS_SETFONT | WS_CHILD | WS_VISIBLE | WS_CAPTION);
}

```

For example, the mouse property sheet in Figure 6.3 belongs to the containing view. The view contains a variable of type **CMouseSheet**. In turn, this class contains the two property pages (**CClickPage** and **CPosPage**). **CMouseSheet** also contains functions that handle the data transfer between the property pages and the outside world. Notice that these functions have to be careful not to call **UpdateData** before the page actually exists. That's why there are **::IsWindow** statements protecting each call to **UpdateData**. As you can see in the figure, modeless property sheets don't contain any default buttons. Anything you want you'll have to add yourself.

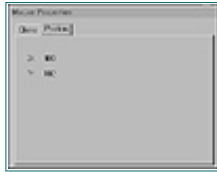


Figure 6.3 A modeless property sheet.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Custom App Wizards

I'm not much of a salesman. That's a bit surprising too, because my Dad, Cecil, was a great salesman. When I was a kid, Dad sold everything at one time or another: shoes, business printing, family Bibles—everything but brushes, vacuum cleaners, and encyclopedias, I believe. He loved to sell so much, he did it in addition to holding the same regular job until he retired.

When Mom and Dad started selling wedding invitations (a business my Mom still operates today), they started small—very small. They would read the wedding announcements in the paper, call the soon-to-be bride, and arrange to visit her home with their catalogs. When they first started doing this, I was very young and most of what grownups did was unfathomable, so it didn't bother me. As I got older, I sometimes felt embarrassed about my parents going to peoples' homes with their wares. Of course, every kid goes through that stage where they think their parents are weird (my kids certainly have). Still, to a young preteen, calling strangers and then dropping in on them with all your merchandise seemed particularly crass.

Later, they quit going out to make sales and relied on people coming to them instead. By that time, they had quite a reputation. Hardly anyone in town had a wedding without "Miz Bea" (my Mom) having a hand in it. Even after Dad retired from his regular job, he and my Mom still ran the little shop.

As I reflect on all this, I realize I was foolish to find my Dad's in-home selling a source of embarrassment. He was doing what any successful salesman does: making his product easy to buy.

You must be wondering what this has to do with programming. Consider this: Unless you only write programs for yourself, you have to sell. Maybe not in the traditional sense, but you are selling—yourself, your programs, or your ideas.

Do you want your client to use your user interface? Will other programmers adopt your library of code, or your coding style, or even your techniques? Do you want your boss to think of you for a particularly interesting project? All of that—in a way—comes down to salesmanship. One obstacle to “selling” other programmers your particular development solution is that often it isn’t easy to “buy.” As a programmer, if I have to decipher your code, pull it out of context, and modify the data structures, I’m going to consider writing it myself, instead.

Visual C++ offers several ways you can make your programming ideas easier to sell. If you’ve ever tried to write an icon handler shell extension for the Windows 95 shell, you probably found that it is no easy task. But what if you had an App Wizard that automatically generates an icon extension with very little effort or knowledge required (like the one in Figure 6.4)?

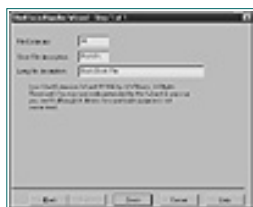


Figure 6.4 The custom App Wizard.

That’s more like it! Even if you don’t understand exactly how it works, you can still use the wizard to create icon extensions. Once you put the hard work into making a skeletal project, you can easily make an App Wizard from it. You can use it yourself later, or let others use it—even if they don’t understand much about your program. I won’t explain the code this wizard generates here. Instead, you’ll see how the wizard works. If you want to know more about icon handlers, look in Appendix A.

Creating A Wizard

Creating the wizard was simple. I started with a working icon handler project. Then I created a new project workspace. For the project type, I selected Custom AppWizard. Then, I had to tell Visual C++ that I wanted to base my new App Wizard on an existing project and specify which project to use (see Figures 6.5 and 6.6).



Figure 6.5 Creating a custom App Wizard (Step 1).



Figure 6.6 Creating a custom App Wizard (Step 2).

This results in a series of source files and resources. If you build them, they will create an AWX file (automatically installed in the \Program Files\DevStudio\SharedIDE\Template directory). Now when you create a new workspace, you'll see the custom App Wizard right along with the other ones (see Figure 6.7).

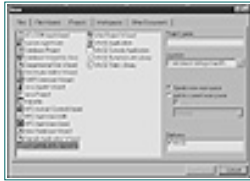


Figure 6.7 The custom App Wizard installed.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

BROWSE
BY TOPIC

Customizing The Customizer

This works surprisingly well. The custom App Wizard is smart enough to change your identifiers to match the project's name. However, it doesn't always make every change you might like. It also doesn't always figure out the right way to create the new project file.

For example, when I first created the icon handler wizard, it created the DEF file, but it didn't add it to the project. Also, it didn't change the GUID for the project. Obviously, you'll need to customize the generated code to take care of these problems.

If you look at the custom App Wizard code, it isn't hard to decipher it. First, there is a **CCustomAppWiz**-derived object that represents your custom App Wizard. This is only interesting if you want to customize particular steps or do a custom initialization (like generate a GUID, for example).

The template files make up the bulk of the project. These are skeletal files that control the generation of new files. If you open up the template files, they look similar to your original source code except that Visual C++ replaces parts of your code with macros. For example, everywhere your project name appeared now reads **\$\$ROOT\$\$**. This macro expands to the new project name. You can find a list of common macros in Table 6.3. There's no rocket science here—the wizard just copies these files into the new project after it expands any macros. The **\$\$IF** macro acts like a preprocessor **#if**. You can use it to select or discard portions of the template file. If you want to change these files, open them from the Dependencies portion of the file view—you can open them as resources, but then you'll have to edit them with a binary editor.

Table 6.3 Common App Wizard macros.

Macro	Meaning
\$\$IF	Implement If/Else logic
\$\$ELSE	Implement If/Else logic

\$\$ENDIF	Implement If/Else logic
\$\$ELIF	Implement If/Else logic
\$\$INCLUDE	Include another file
\$\$BEGINLOOP	Implements looping
\$\$ENDLOOP	Implements looping
\$\$SET_DEFAULT_LANG	Set default language
\$\$//	Comment
\$\$\$\$	Emit the string “\$\$”
\$\$ROOT	Project name with no extension (all uppercase)
\$\$Root	Like \$\$ROOT, but with case preserved
\$\$FULL_DIR_PATH	Path of directory for project

Creating The Project

The two files you won't recognize are CONFIRM.INF and NEWPROJ.INF. The CONFIRM.INF file generates the final text box you see just before App Wizard creates code. NEWPROJ.INF is a bit trickier. It controls how the wizard generates files and constructs the actual project. Unlike other files, App Wizard doesn't create the make file from a template. Instead, it creates the make file from the NEWPROJ.INF file. Here's a look at a NEWPROJ.INF file:

```

$$// newproj.inf = template for list of template files
$$//  format is 'sourceResName' \t 'destFileName'
$$//      The source res name may be preceded by any combination of
$$//          '=', '+', and/or '*'
$$//          '=' => the resource is binary
$$//          '+' => the file should be added to the project
$$//          '*' => bypass the custom AppWizard's resources when
$$//      loading
$$//  if name starts with / => create new subdir

```

/res

```

ROOT.CLW  $$root$.clw
+ROOT.CPP  $$root$.cpp
ROOT.H     $$root$.h
+ROOT.DEF  $$root$.def
STDAFX.H   StdAfx.h
+STDAFX.CPP StdAfx.cpp
RESOURCE.H  resource.h
+ROOT.RC   $$root$.rc
ROOT.REG   $$root$.reg
ICONHANDLER.H IconHandler.h
+ICONHANDLER.CPP IconHandler.cpp
=ICON1.ICO  icon1.ico
ROOT.RC2   res\$$root$.rc2

```

The first column indicates the template name and the second column indicates the name to assign to that file (using macro expansion). When I first generated this file, the DEF file was not in the project (this prevents the DLL from working properly). The solution: Add the + sign ahead of the

ROOT.DEF template name.

The GUID problem was nearly as simple to fix. The App Wizard object maintains a dictionary of macros in **m_Dictionary**. To create your own macros, just add them to the dictionary:

```
// In App Wizard object constructor  
m_Dictionary[_T("Favorite_Magazine")] = _T("Visual Developer");
```

Now **\$\$Favorite_Magazine\$\$** will give the results you'd expect. The **_T** macro type casts the string to a **LPTSTR** required by this and many OLE calls.

Armed with this information, it is simple to create a macro for the GUID. You have to call **CoCreateGuid** to generate the number. Then, you'll want to convert that to the string representation using **StringFromCLSID** (a **CLSID**, or Class ID, is a form of GUID). The string is in UNICODE format, so if you are working with ordinary strings, you should convert them to ANSI. The code requires the GUID as a long string and as a series of hexadecimal numbers separated by commas. The example code provides macros for both.

Touching up the templates is simple until you try to fix the REG file. The REG file provides the registry entries required for the shell to recognize the handler. Ideally, you'd like to be able to specify the file extension you will use as part of the wizard's steps. However, that would take a bit of code, and because we haven't written any real wizard code yet, it seems a shame to start now. Also, the registry needs the full path to the DLL. This shouldn't be a problem, except that the registry requires doubled backslashes to separate the path. Of course, you could read the **\$\$PATH\$\$** variable, double the backslashes in code, and provide a new macro for that value. Then you would need to decide how to handle the different locations for debug builds and release builds.

I decided it wasn't worth the effort. Instead, I had the REG file insert XXX into the suspicious places and I marked that section with a TODO comment. I also placed the DLL name with no directory information in the path. You have to put the DLL somewhere where Windows will search for it or you'll have to manually touch up the REG file.

Once you have the icon handler wizard, you can turn these shell extensions out over and over again—even if you don't completely understand how they work. That means you can write a wizard like this one and turn it over to other programmers who might not be able to write the code on their own.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Other Options

Basing your wizard on an existing project is one way to create a new wizard. You can also start with the usual custom steps (and presumably modify them) or you can opt to create all new steps. You can also provide customized help for your wizard.

There are three new MFC classes of interest to the wizard's author: **CCustomAppWiz**, **CAppWizStepDlg**, and **OutputStream**. Your code provides a class derived from **CCustomAppWiz** (this is similar to a regular program's application object). This object manages, for example, the dictionary I mentioned earlier. You can override member functions to customize the wizard's actions.

You usually won't be concerned with the **OutputStream** class (amazingly, this MFC class doesn't begin with the letter C). It contains two simple functions: **WriteLine** and **WriteBlock**. **WriteLine** moves lines of text from a template to an output stream, whereas **WriteBlock** moves binary resources (like bitmaps) to an output stream.

If you want to create your own custom steps, you'll need to create a dialog template for each step and bind it to a class derived from **CAppWizStepDlg** (which itself derives from **CDialog**). Each step's derived class overrides **CAppWizStepDlg::OnDismiss**. This function receives control when the user presses the Next, Back, or Finish buttons. This is your opportunity to update the dictionary, for example.

The custom App Wizard DLL also exports some functions you can use. **GetDialog** retrieves one of the standard steps that you can reuse. Your main DLL function calls **SetCustomAppWizClass** (App Wizard sets this up automatically for you). You can also set the number of steps (**SetNumberOfSteps**) or work with multiple languages (**ScanForAvailableLanguages/SetSupportedLanguages**).

Pressing On

I couldn't leave well enough alone, so I decided to add a step dialog to the wizard (see Figure 6.4). The changes to the code were minimal. Of course, the call to **SetNumberOfSteps** needed an argument of 1 instead of 0. I also had to draw the dialog box. You should make the box have the child style and the control style. If you don't include the control style, you won't be able to tab between your controls. Don't include a title bar or any of the standard buttons (Finish, Next, Back, and so on).

Class Wizard doesn't know about **CAppWizStepDlg**, so I had to use **CDialog** as a base class instead. Then I manually changed all the places **CDialog** occurs in the header and in the CPP files. I also had to change the constructor that Class Wizard creates. It passes the **CDialog** constructor two arguments: the resource ID and a parent window. **CAppWizStepDlg** only requires the first argument.

I used Class Wizard to set up DDX connections between the three edit fields and variables in the **CStep1Dlg** object (the class I created with Class Wizard). Then, I overrode **OnDismiss**. That routine calls **UpdateData(TRUE)** to transfer the data to the variables and then updates the dictionary. I also modified the REG and CONFIRM.INF files to reflect the new dictionary entries. This is the version of code you'll find in Listings 6.9 and 6.10.

Listing 6.9 ShIconWzAw.CPP.

```
// shiconwzaw.cpp : implementation file
//

#include "stdafx.h"
#include "shiconwz.h"
#include "shiconwzaw.h"
#include <objbase.h>
#include <afxpriv.h>

#ifdef _PSEUDO_DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

// This is called immediately after the custom App Wizard is
// loaded. Initialize the state of the custom App Wizard here.
void CShiconwzAppWiz::InitCustomAppWiz()
{
    // There are no steps in this custom App Wizard.
    SetNumberOfSteps(1);

    // Inform App Wizard that we're making a DLL.
    m_Dictionary[_T("PROJTYPE_DLL")] = _T("1");

    // TODO: Add any other custom App Wizard-wide initialization here.
    char clsid[66], part[66];
    GUID g;
    LPOLESTR str;
    CoCreateGuid(&g);
    StringFromCLSID(g, &str);
    USES_CONVERSION;
    strcpy(clsid, W2A(str));
    // Set up CLSID in various ways in the dictionary.
    m_Dictionary[_T("EXT_CLSID")] = _T(clsid);
    strcpy(part, "0x");
    clsid[9] = '\\0';
    strcpy(part+2, clsid+1);
    m_Dictionary[_T("CLSID_P1")] = _T(part);
    clsid[14] = '\\0';
    strcpy(part+2, clsid+10);
    m_Dictionary[_T("CLSID_P2")] = _T(part);
    clsid[19] = '\\0';
    strcpy(part+2, clsid+15);
```

```

        m_Dictionary[_T("CLSID_P3")] = _T(part);
        part[2] = clsid[20];
        part[3] = clsid[21];
        strcpy(part+4, ",0x");
        part[7] = clsid[22];
        part[8] = clsid[23];
        for (int i=0; i<6; i++)
        {
            strcpy(part+9+5*i, ",0x");
            part[9+5*i+3] = clsid[25+i*2];
            part[9+5*i+4] = clsid[26+i*2];
        }
        part[39] = '\\0';
        m_Dictionary[_T("CLSID_P4")] = _T(part);
    }

    // This is called just before the custom App Wizard is unloaded.
    void CShiconwzAppWiz::ExitCustomAppWiz()
    {
        // TODO: Add code here to deallocate resources
        // used by the custom App Wizard.
    }

    // This is called when the user clicks "Create..." on the
    // New Project dialog or next.
    CAppWizStepDlg* CShiconwzAppWiz::Next(CAppWizStepDlg* pDlg)
    {
        // Set template macros based on the project name entered by the user.
        // Get value of $$root$$ (already set by App Wizard).
        if (pDlg==NULL) // first time
        {
            CString strRoot;
            m_Dictionary.Lookup(_T("root"), strRoot);

            // Set value of $$Doc$$, $$DOC$$
            CString strDoc = strRoot.Left(6);
            m_Dictionary[_T("Doc")] = strDoc;
            strDoc.MakeUpper();
            m_Dictionary[_T("DOC")] = strDoc;

            // Set value of $$MAC_TYPE$$
            strRoot = strRoot.Left(4);
            int nLen = strRoot.GetLength();
            if (strRoot.GetLength() < 4)
            {
                CString strPad(_T(' '), 4 - nLen);
                strRoot += strPad;
            }
            strRoot.MakeUpper();
            m_Dictionary[_T("MAC_TYPE")] = strRoot;
            return &step_1; // bring up step 1
        }
    }

    // Only 1 step so we are done if we get here (and we
    // should never get here). The step_1 dialog updates

```

```
// the dictionary.  
return NULL;  
}
```

```
// Here we define one instance of the CShiconwzAppWiz class.  
// You can access m_Dictionary and any other public members  
// of this class through the global Shiconwzaw.  
CShiconwzAppWiz Shiconwzaw;
```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

Listing 6.10 Step1Dlg.CPP.

```
// Step1Dlg.cpp : implementation file
//

#include "stdafx.h"
#include "shiconwz.h"
#include "shiconwzaw.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////
// CStep1Dlg dialog

CStep1Dlg::CStep1Dlg(CWnd* pParent /*=NULL*/)
    : CAppWizStepDlg(CStep1Dlg::IDD)
{
    //{{AFX_DATA_INIT(CStep1Dlg)
    m_extension = _T("");
    m_desc = _T("");
    m_filetype = _T("");
    //}}AFX_DATA_INIT
}

void CStep1Dlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
```

```

        //{AFX_DATA_MAP(CStep1Dlg)
        DDX_Text(pDX, IDC_EXTENSION, m_extension);
        DDX_Text(pDX, IDC_DESC, m_desc);
        DDX_Text(pDX, IDC_FILETYPE, m_filetype);
        DDV_MaxChars(pDX, m_filetype, 8);
        //}}AFX_DATA_MAP
    }

BEGIN_MESSAGE_MAP(CStep1Dlg, CAppWizStepDlg)
    //{AFX_MSG_MAP(CStep1Dlg)
    // NOTE: the Class Wizard will add message map macros here
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

//////////////////////////
// CStep1Dlg message handlers

/* Do DDX and update dictionary before page is dismissed */
BOOL CStep1Dlg::OnDismiss()
{
    if (!UpdateData(TRUE)) return FALSE; // invalid data
    // Do custom validation
    // No empty fields
    if (m_extension.IsEmpty()||m_filetype.IsEmpty()||m_desc.IsEmpty())
    {
        MessageBox("Please complete all entries");
        return FALSE;
    }
    // If extension starts with '.' chop it off
    // A . anywhere else is an error
    if (m_extension.Find('.')!=-1)
    {
        if (m_extension.Find('.')==0)
            m_extension=((LPCSTR)m_extension)+1;
        else
        {
            CWnd *ext=GetDlgItem(IDC_EXTENSION);
            MessageBox("Do not include a period in the extension");
            ext->SetFocus();
            return FALSE;
        }
    }

    // No space or tab in file type
    if (m_filetype.Find(' ')!=-1||m_filetype.Find('\t')!=-1)
    {
        CWnd *ft=GetDlgItem(IDC_FILETYPE);
        MessageBox("File type can't contain blanks");
        ft->SetFocus();
        return FALSE;
    }
    // Update
    Shiconwzaw.m_Dictionary[_T("EXT")]=m_extension;

```

```
Shiconwzaw.m_Dictionary[_T("FILETYPE")] = m_filetype;  
Shiconwzaw.m_Dictionary[_T("DESC")] = m_desc;  
return TRUE; // allow wizard to proceed  
}
```

In the first version of the wizard, the code used the default **Next** routine. Now, **Next** requires modifications to bring up the custom step dialog. The main class (**CShiconwzAppWiz**) now has an extra member variable (**step_1**) that is a **CStep1Dlg** object. If the **Next** routine receives a **NULL** pointer, that indicates that you should return a pointer to the first step. The code returns the address of the **step_1** variable. If you wanted multiple pages, you'd need to examine the pointer to decide which dialog object to return. If you want one of the standard steps, you can do that by calling **GetDialog** to learn which pointer to use.

If I had specified that I was going to use steps in the original version, App Wizard would have provided a **CDialogChooser** class to manage the multiple dialogs. For one step, that's overkill; I decided to handle it myself. Also, when you base your wizard on an existing project, App Wizard doesn't add any help file support. That's easy to add, however. Just make your help file name the same as your wizard's name. When the user clicks help on one of your step dialogs, Visual C++ will start WinHelp with your help file name and a context ID equal to 131,072 plus the ID of your dialog box. As long as you have a topic in the help file with that context ID, everything will work fine.

Debugging Wizards

If you try to do anything fancy with a wizard, you'll need to debug it. Visual C++ arranges your project so that there are the customary release and debug builds, but that isn't the whole picture. You'll notice that the IDE labels the non-release build *pseudo-debug*. Visual C++ uses the release library in either case. However, in pseudo-debug mode, there is minimal debugging support. If you run the wizard, a new copy of Visual C++ executes, and you must use the New Project Workspace command to exercise your code.

More Ideas For Wizards

Try creating your own custom wizards from any kind of project. They are a great way to promote consistent style or leverage an experienced programmer on your staff. Just write a minimalist program, pepper it with TODO comments, and make it a wizard. If you put too much specific code in your wizard, you'll prevent others from using it, so try to generalize.

If you have an entrepreneurial bent, you should be able to sell your wizards. A full set of shell extension wizards would probably sell (any offers?). How about a wizard to write an adventure game, a database program, a web server, or a Winsock Internet server? There's money to be made here. Too bad I'm not a good salesman.

Summary

MFC makes it simple to add property sheets and wizards. The only hard part is making up excuses not to add them. Modal property sheets are easy. Modeless sheets are only a bit more difficult.

MFC practices what it preaches (at least, in this case). It is straightforward to create new wizards based on existing projects. This is one of the most exciting, and perhaps underutilized, features of Visual C++.

Practical Guide To Property Sheets And Wizards

- [Creating A Property Sheet](#)
- [Creating A Wizard](#)
- [Using A Single Template](#)
- [Modeless Property Sheets](#)
- [Making Custom App Wizards](#)

Working with property sheets isn't much different from handling ordinary dialogs. You simply have to get used to thinking of a block of dialogs all together because each tab (or page) in the property sheet corresponds to a separate dialog-like class.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Creating A Property Sheet

Creating a property sheet is just a matter of following a few simple steps:

1. In the resource view, right click and select Insert from the pop-up menu.
2. On the dialog that appears, click on the plus sign to expand the Dialog choice.
3. Select one of the property pages (small, medium, or large).
4. Populate the property page with controls just like an ordinary dialog box.
5. Bring up Class Wizard and use it to create a new class (derived from **CPropertyPage**) that corresponds to the new page.
6. Use Class Wizard's Member Variables tab to create variables for the controls you want to manipulate.
7. Repeat the above six steps until you have all the pages you need.
8. To display your sheet, create an instance of each page class you created.
9. Create a **CPropertySheet** object.
10. Alter any member variables in the pages that you want to initialize.
11. Call **CPropertySheet::AddPage** for each of the pages.
12. Call **CPropertySheet::DoModal** and handle like an ordinary dialog box.

Here is a typical example:

```

CPropertySheet sheet("Example Property Sheet");
CPropPg1 prop1; // derived from CPropertyPage
CPropPg2 prop2; // derived from CPropertyPage
sheet.AddPage(&prop1);
sheet.AddPage(&prop2);
prop1.m_value1=100; // DDX
prop1.m_value2 = "Test"; // DDX
. . .
prop2.m_position=25; // more DDX
if (sheet.DoModal()==IDOK)
{
    // reverse DDX
    MessageBox(prop1.m_value2,"Returned data");
    . . .
}

```

Creating A Wizard

To create a wizard, follow the same steps that you would for a property sheet. However, before you call **DoModal**, you must call **CPropertySheet::SetWizardMode**.

Also, if you want to manage the buttons correctly, you'll need to override the **OnSetActive** member of each property page object. Within the override, get the parent window (the actual property sheet) and call **CPropertySheet::SetWizardButtons** to set the buttons that should appear (see Table 6.2).

Using A Single Template

Sometimes you'd like to use the same property pages for multiple pages. It is permissible to use two instances of one page object in a property sheet. However, because the dialog's title will name the tab, this would result in two tabs with the same name.

The answer is simple. You must supply a resource string ID to the **CPropertyPage** constructor. This ID points to a string that contains the name. If you decide to place the string ID in your derived class' constructor, you'll find that MFC will complain that it requires a default constructor. Either overload the constructor or give the ID argument a default value so that it can serve as a default constructor. You can find details in Listings 6.1 and 6.2.

Modeless Property Sheets

It is possible to make a property sheet modeless by calling **Create** instead of **DoModal**. Of course, you'll have to handle **UpdateData** and manage the scope of the variables (just like a modeless dialog).

One possible way to handle the scope problem is to derive a new class from **CPropertySheet**. Your new class will contain member variables of each page

class you create. Then you can create your new sheet class on the heap (using **new**) and it will take care of the rest (see Listings 6.5 through 6.8).

Making Custom App Wizards

You can add your own custom App Wizards to Visual C++. This allows you to easily recreate projects or allow other programmers to create starter applications to your specifications.

The easiest way to do this is to start with an existing application that you want to duplicate. Then start a new project and select Custom AppWizard as the project type. Then you can tell the wizard where your existing project is and it will create a custom wizard for you. It is really that simple.

You can also start with the standard App Wizard steps, or even create a Wizard completely from scratch. However, this is a bit more work (you can find the details in this chapter and Listings 6.9 and 6.10).

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Chapter 7

DLLs And MFC

DLLs are a fundamental component of the Windows operating system. You can use code from DLLs and create DLLs using MFC. You can also extend MFC with custom DLLs and provide individualized programming support.

Programmer's Notes...

I grew up in Bay St. Louis, Mississippi, a very small town at that time. Now they have several major casinos there, but when I lived there, it was excruciatingly tiny. I remember what a big deal it was when they built a Burger King on the highway. Before that, it was about 30 miles to the nearest fast food joint. Bay St. Louis was (and is) certainly charming with its beaches and historic districts. But it wasn't a metropolis.

A town like Bay St. Louis is a great place to have a Norman Rockwell childhood. Of course, I never wanted a Norman Rockwell childhood—I wanted a Will Robinson childhood. For a kid who was interested in radios and mainframe computers (the only kind that there were in those days), Bay St. Louis seemed like a prison.

Not that there wasn't any technology in the area. NASA tests their rocket engines near the Bay (at what is now the Stennis Space Center). So there were a few people around who had science and engineering backgrounds. However, there just wasn't any support for folks like us there. There were no bookstores,

much less technical bookstores. We didn't even have a Radio Shack for most of the time I was there. If you wanted anything, you had to make a 30-mile drive to Biloxi (home to Keesler Air Force Base) or a 60-mile trek to New Orleans (where you could get anything—and I do mean anything—you wanted).

One thing Bay St. Louis did have was a nice library. They always had a new building and plenty of books. Just not my kind of books. Of course, all the electronics books were 20 years old or more. They had virtually no books on computers. So I learned from the books they had, but it gave me an older perspective on technology. I grew up reading about vacuum tubes in a world that was starting to embrace transistors. I spoke of condensers and rheostats while the rest of the world used capacitors and potentiometers. I still occasionally say “cycles per second” instead of hertz.

So, did the library do me more harm than good? I don't know. I felt like Buck Rogers (or Rip Van Winkle, if you prefer) when I moved to Starkville to attend Mississippi State University. They had a world-class library and more electronics books than I had ever seen in my life. I spent ages in that library soaking up everything I could find. I learned a lot at MSU, but most of it I learned at the library.

Libraries of a different kind are important in programming, too. Take away the C library and you'll have to do a lot of work. Do you really want to interface directly with the file system? Do you want to format output by hand? I don't, so I use the standard C library. The Windows API is really just a library, too.

It wasn't long ago that there was really only one kind of library programmers used: a static library. Of course, no one called it that because it was the only kind you knew about. The idea is that you took object code from the library and copied it into your program (using a linker). Then the library became a permanent part of your program.

There are several disadvantages to this approach. First, every program has its own copy of the library. That can waste space. Second, what happens when the library changes? You have to relink your code to take advantage of it. Imagine if you had to relink your programs for each new version of Windows!

Most operating systems (including Windows) still support static libraries. However, most systems also support some form of dynamic linking. This allows you to find libraries of code at runtime instead of link time. Each time your program runs, it locates the library on disk and loads it at that time. This means that the program doesn't have to contain a copy of the library. It also means that the program gets the latest version of the library every time.

Windows supports dynamic linking via dynamic link libraries (or DLLs). With MFC, you can create and use several types of DLLs. There are several ways you can incorporate DLLs into your programs, and there are a few special techniques you have to use to ensure compatibility.

The Link Process

When Visual C++ links your program, it combines the program with LIB files

that (ordinarily) contain static libraries. However, you can also specify import libraries (or IMPLIBs) that don't really contain library code. Instead of regular code, an import library contains "stubs" that force your program to load and execute code from a DLL.

What this means is that if you link with a library, you really don't need to know if that library incorporates code into your program or if it just instructs your program to load the library at runtime. That's very clever. In fact, you can set MFC to load its standard routines from a static library (which generates large, self-contained executables) or a DLL version of the library (which creates smaller executables that require a DLL to execute).

There is another way that you can load a library at runtime. You can use the **LoadLibrary** call to actually read a DLL and manually incorporate it at runtime. This requires more work, but it's useful in special cases. For example, suppose you develop a program that can support multiple languages. You might use a different DLL for each language and select which one you use at runtime. Each DLL would contain the same functions, but they would behave correctly for the selected language. This would be difficult to do with an IMPLIB, but it is relatively straightforward using **LoadLibrary**.

Language Considerations

Because DLLs form the backbone of Windows, it stands to reason that programs written in any language can call DLLs. Thus, DLL designers typically take special care to ensure that most programs can call their DLLs. Of course, you are not obligated to take these steps, but it is customary in most cases.

What this means is that most DLLs you encounter will have simple functions (not class members) that use the Pascal calling convention. There are exceptions, but this is the general rule.

Does this mean that you can't place classes or functions with variable argument lists in a DLL? Not at all. It just means that if a DLL provides these things, it becomes less general-purpose.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Using An Ordinary DLL

When you want to call a DLL, you'll need to define the functions it declares. Often, the DLL will have a header file associated with it that you can use. However, if the DLL doesn't use C++ functions (and most don't), you have to be careful that the header defines the functions as C functions instead of C++ functions. Otherwise, C++ will alter the names of the functions at link time, which will cause your calls to fail.

The header should look something like this:

```
extern "C" {  
    // function declarations  
}
```

However, if the file doesn't look like this, you can easily remedy the situation. Simply enclose your **#include** statement with the **extern** statement:

```
extern "C" {  
    #include "dllhfile.h"  
}
```

Once you have your declarations in place, you need only add the appropriate LIB file into your project. You can add a LIB file to your project just as you would any other file. If you prefer to hide your DLL's library, you can also go to the Project|Settings dialog box and add the LIB file to the list of libraries the project uses. You'll find the list under the Link tab.

The compiler will automatically generate the correct code with nothing more than the function names. However, if you mark the functions with **__declspec(dllimport)**, your code will be a bit more efficient. This is also the way you import variables. For example, you might write:

```
__declspec(dllimport) int dll_flag;
```

This will import a variable (named **dll_flag**) from a DLL. Note that you don't want to declare things with this notation when you are writing your DLL—you want to do this when you are actually importing from the DLL.

That's all there is to using an ordinary DLL, as long as you don't mind linking to the IMPLIB files. This is usually what you want, although sometimes you might want to put off deciding what library you load until runtime. You can do that with **LoadLibrary**, but it is a bit more work.

The problem is that loading the library doesn't give you a reference to the function the DLL contains. To do that, you'll need a function pointer for each function you call. **GetProcAddress** will give you that pointer, and then you have to call the function via that pointer.

One way you can do this is to initialize all the pointers you need before you call them. For example:

```
void (*dllfunc1)(int); // global
BOOL (*dllfunc2)(void); // another global
HANDLE theDLL;

...
theDLL = ::LoadLibrary("MyDll.dll");
dllfunc1 = ::GetProcAddress(theDLL, "dllfunc1");
dllfunc2 = ::GetProcAddress(theDLL, "dllfunc2");

...
dllfunc1(100);
```

Notice that in this case, the call to **GetProcAddress** requires the name of the function. Many DLLs export their functions by name. However, for maximum efficiency, some DLLs export their functions by ordinal number. This is just a fancy way of saying that each function has a unique integer number instead of a name. When you use an IMPLIB, this is invisible to you. But if you call **GetProcAddress**, you have to know the number (if any). Usually, the DLL's documentation will supply this information. If not, you may be able to run DUMPBIN on the DLL to discover its exported functions.

Using DUMPBIN

DUMPBIN is a program Microsoft supplies with Visual C++. Other vendors often supply similar programs. This is a command line program in the compiler's BIN directory. The purpose of DUMPBIN is to display an EXE or DLL file in a meaningful way. If you want to see what functions a DLL exports, use the /EXPORTS command line switch along with the file name of interest. If you want to discover what DLLs a program needs, use the /IMPORTS switch. There are many other switches DUMPBIN accepts (run

DUMPBIN with no arguments to see a list). The /ALL option causes the program to decode everything it can. Old assembly language hackers (like me) will enjoy the /DISASM option. Try it!

Creating An Ordinary DLL

The real challenge is creating a DLL of your own. With MFC's App Wizard, it's as simple as asking the wizard to do it for you. However, there are a few things you should know.

You'll notice in Figure 7.1 that the wizard only has one screen. For right now, the only thing you should worry about is a regular DLL (you'll see MFC DLLs later in the chapter). The two options that relate to regular DLLs allow you to use a static copy of MFC or link the DLL version of the library. Which should you choose? That depends. Making a static copy of the library allows your DLL to stand alone with no extra files (at least, not to support MFC). You'll never have to worry about what version of MFC is on the machine or if it exists at all because your DLL is self-contained. However, there is a price for this convenience. Your DLL will be quite large because it contains your code plus everything MFC uses.

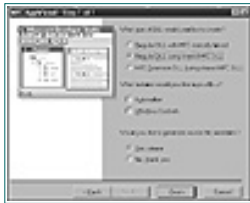


Figure 7.1 Creating a DLL.

The answer, of course, is to allow the DLL to link to MFC in another DLL. That means, however, that you can't expect your DLL to work without the MFC libraries. There is also the potential for having version problems with the MFC DLLs. On the other hand, your DLL will be small and efficient.

Another reason to use the DLL library strategy is to create many programs and DLLs that use MFC. Because all of your programs and libraries can share the same MFC DLL, you get smaller files, requiring fewer install disks (or less download time).

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

BROWSE
BY TOPIC

The Main File

You can find an example DLL in Listing 7.1. Notice that even though this is a DLL, it has a **CWinApp**-derived class. In this case, the class doesn't represent a running application. Instead, it stands in for the DLL's connection with MFC.

Listing 7.1 A sample DLL.

```
// dumbdll.cpp : Defines the initialization routines for the DLL.
//
#include "stdafx.h"
#include "dumbdllmfc.h" // changed from dumbdll.h

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//
// Note!
//
// If this DLL is dynamically linked against the MFC
// DLLs, any functions exported from this DLL which
// call into MFC must have the AFX_MANAGE_STATE macro
// added at the very beginning of the function.
//
// For example:
//
// extern "C" BOOL PASCAL EXPORT ExportedFunction()
// {
//     AFX_MANAGE_STATE(AfxGetStaticModuleState());
//     // normal function body here
// }
//
```

```

//      It is very important that this macro appear in each
//      function, prior to any calls into MFC.  This means that
//      it must appear as the first statement within the
//      function, even before any object variable declarations,
//      as their constructors may generate calls into the MFC
//      DLL.
//
//      Please see MFC Technical Notes 33 and 58 for additional
//      details.
//

////////////////////////////////////
// CDumbdllApp

BEGIN_MESSAGE_MAP(CDumbdllApp, CWinApp)
    //{AFX_MSG_MAP(CDumbdllApp)
        // NOTE -- the Class Wizard will add and remove mapping macros here.
        // DO NOT EDIT what you see in these blocks of generated code!
    //}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CDumbdllApp construction

CDumbdllApp::CDumbdllApp()
{
    // TODO: add construction code here
    // Place all significant initialization in InitInstance
}

////////////////////////////////////
// The one and only CDumbdllApp object

CDumbdllApp theApp;

// A Dumb exported function

__declspec(dllexport) void Alert(char *s)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    MessageBeep(-1);
    AfxMessageBox(s);
}

```

Because the DLL has an application object, it can do things you expect applications to do, such as load resources, learn its instance handle, and the like. It is important to realize, however, that the DLL becomes part of another application and executes in the host application's virtual address space. If that application is an MFC program, it will have its own application object.

I did make one change to the App Wizard-generated project. I want programs that will use this DLL to include DUMBDLL.H. However, as generated, that file is full of MFC idioms that aren't important (or even intelligible) to all programs. So I renamed the generated DUMBDLL.H to DUMBDLLMFC.H. I then made a simple DUMBDLL.H file for inclusion by other programs (see Listing 7.2). You'll also find a simple console application that uses the DLL in Listing 7.3.

Testing With Console Applications

You'll notice that the code in Listing 7.3 that tests the DLL isn't an MFC program. You may often want to rig up a little program just to test some part of your DLL. Why write a full-blown Windows application for that? Sure, sometimes you have no choice. However, if your DLL does calculations or has an independent user interface (as this one does), you can get away with a lot less work. Just start a new Win32 Console Application

project. At first, the project won't contain any source files. Use the New command to create a new CPP file, and you're ready to go. If you haven't made console applications before, don't worry—it's easy. Console applications are just like old DOS or Unix command-line programs. Use **printf** or **cout** and all the other things you used to use when you first learned C or C++. The difference? Console programs have full access to the Windows API. A console application can allocate megabytes of memory, bring up a message box, or even bring up a dialog. You'll find lots of other uses for console applications, too. Give it a try!

Listing 7.2 The DLL's header.

```
// Header file for DUMBDLL.DLL
#ifdef __cplusplus
extern "C" {
#endif

    __declspec(dllimport) void Alert(char *s);

#ifdef __cplusplus
}
#endif
```

Listing 7.3 Using the DLL (a console application).

```
#include <iostream.h>
#include "dumbdll.h" // not MFC version

void main()
{
    char c;
    cout<<"Get ready for the alert!\n";
    cout.flush();
    Alert("Alert! Alert!");
    cout<<"I told you it was a dumb dll\n";
    cout<<"Press Enter to continue";
    cin.get(c);
}
```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Exporting Functions

In consideration of other programmers (who might not use Visual C++), it's a good idea to declare your externally-accessible functions in such a way that everyone can call them. This means using global functions (not member functions) that have the C declaration syntax. That is:

```
extern "C" {
    __declspec(dllexport) void f(int a);
    __declspec(dllexport) int f1(void);
};
```

The **__declspec(dllexport)** statement marks the function as one that you wish the outside world to see. You can do the same thing by adding a DEF file to your project and naming the function in the **EXPORTS** section (see Listing 7.4).

Listing 7.4 A sample DEF file.

```
LIBRARY          DLLMania

EXPORTS
    ManiaBegin
    ManiaEnd
    ProduceMania
```

Without the **extern "C"** statement, the compiler would transform my innocent **Alert** function into `?Alert@YAXPAD@Z` (or something like that). Because

that would be very confusing, you'll want to make your functions use the C type, which prevents the name mangling (or decorating, if you prefer). You'll notice that this file avoids any MFC-related macros or other dependencies.

When you try to name the functions in a DEF file or in an external program, you'll find that the compiler can slightly change the name of your functions. In the DEF file, you can rename a function the compiler calls `_f` to a function named `f` using the syntax:

```
EXPORTS
f=_f
```

If you want to export functions by ordinal number (which is slightly more efficient than using names), you'll have to use a DEF file. To make the previous export ordinal number 5, you'd write:

```
EXPORTS
f=_f @5
```

Add the **NONAME** directive if you don't want the name even mentioned in the DLL.

There is a third method you can use to export a function. Simply mention the function in an `/EXPORT` option on the linker's command line (you can set the options in the Projects|Settings menu). Just follow the `/EXPORT` switch with a colon and the same line you would have put in the DEF file. It is perfectly permissible, by the way, to use all three methods (**__declspec**, a DEF file, and `/EXPORT`) together in the same program.

The DUMBDLLMFC.H file includes DUMBDLL.H so that it always has the most up-to-date information. However, DUMBDLL.H specifies that the function is **__declspec(dllimport)**. You don't want to do that in this case (here, you want to use **__declspec(dllexport)**). The code uses a macro to arbitrarily force all **__declspec** statements in DUMBDLL into export statements. However, you could do something more clever if you needed other **__declspec** statements. For example, you might write:

```
#ifndef DLLFUNC
#define DLLFUNC __declspec(dllexport)
#endif
```

Then you could define **DLLFUNC** as **__declspec(dllexport)** in DUMBDLLMFC.H before including DUMBDLL.H.

There is one other important thing to consider if you use MFC in a DLL from a DLL. More than one program might be using the same MFC DLL at the same time. Suppose your DLL is using the MFC library and a program using your DLL is also using the MFC DLL. It is important that MFC not get confused in this situation.

The trick is that if you are using MFC in a DLL from a DLL, you should start each exported function with the following statement:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
```

If you forget to do this, you'll generate an exception when you try to do things from within the function. If you use the static MFC library, you don't need this line of code.

You can export global variables from a DLL. Just use the same techniques you use for functions. However, variables can have some peculiar characteristics in a DLL that you should be aware of (see the next section).

Private And Shared Variables

Who owns a global variable in a DLL? Careful now. In Win16, the answer is everybody. That's no surprise because everyone really uses the same memory anyway. For Win32, the answer isn't so clear cut. By default, if you use Borland's 32-bit compiler, the same thing happens. If three programs are using your DLL, they all see the same global variables. With Microsoft Visual C++, it is quite different. By default, a Microsoft DLL always gives each program a private copy of global variables. Consider this partial code from a DLL:

```
int threshold;
void set_thresh(int n) // export this function
{
    threshold=n;
}

int get_thresh(void) // export this function
{
    return threshold;
}
```

If three Win16 programs loaded this DLL and one of them set the threshold value, the other two would see the same value. Note that if the DLL unloads (no one is using it), the value isn't preserved. If you compile the DLL with Borland's compiler, you'll get the same behavior (unless you specify MULTIPLE in the DEF file). With Microsoft, you'll have to do a bit of work to get the same effect.

Here's the plan:

- Mark the variable with a special pragma. This will place the variable in a data section.
- Make certain that the variable has an initial value.
- Use the linker options (or the DEF file) to set the attributes for the data section that holds the variable.

Here's the same variable declaration with the sharing pragma:

```
#pragma data_seg(".ASHARE")
int threshold=0;
// more shared data could go here
```

`#pragma data_seg()`

If you don't initialize the variable, it won't work. This is important to remember. Just defining it is not enough, you must provide an initial value.

The section name is special. It must be eight characters or less and start with a period. Otherwise, it can be pretty much any legal identifier. Of course, you can't use ones the compiler already uses for something else (that is, **.CODE**, **.DATA**, **.BSS**, and so forth).

Next, you need a special option passed to the linker. You can set this in the project settings (exactly where depends on what version of VC++ you are using). The setting dialog doesn't have a special place for the option you need; you'll have to modify the command line (at the bottom of the dialog). If you can't change it, make sure you have either Debug or Release build selected, but not both. If both are selected, VC++ won't allow you to edit the command line. Here's what you need:

`/SECTION: .ASHARE, RWS`

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Of course, if you called your section something else, use that instead of **ASHARE**. The **RWS** means Read, Write, and Shared. The shared attribute is what forces the compiler to share the segment (data area) between multiple programs.

If you prefer, you can create a DEF file and put a SECTIONS line in it:

LIBRARY DLL

SECTIONS

.ASHARE READ, WRITE, SHARED

If you use this method, be sure to add the DEF file to your project. MSVC doesn't use a DEF file by default. If you want to share everything, just set the **.DATA** and **.BSS** sections as shared using one of the above techniques (the command line option or a DEF file).

When you are using shared memory, you should be aware of how Windows NT loads DLLs. Each DLL has a preferred load address. When Windows NT loads a DLL for a process, it places the DLL, if possible, at that virtual address. If it isn't possible, NT has to reload the DLL (that is, move it and fix up its addresses). That means that while a page of variables may be shared, *their addresses may be different in different processes*. The bottom line: Don't store addresses to shared memory in shared memory.

You can use the REBASE utility to change the preferred load address. You can also use the linker's /BASE option. Adding /FIXED will prevent the system from relocating the DLL (which means if the DLL can't load at that address, you'll get an error). Another way to set the base address is using the **BASE** option in a DEF file's **LIBRARY** command. Technically, executable files have a preferred load

address, too. However, you usually don't care because it is the first thing loaded into the process space. Therefore, it will always load where it wants.

This is a non-issue for Windows 95, by the way. In Windows 95, all DLLs load in a single reserved block of memory. That means that once a DLL is loaded (and that probably won't be at the preferred address), it stays at that location for all processes until it unloads.

MFC DLLs

It is easy to create DLLs that extend the MFC library itself. The only limitation is that only programs that use the DLL version of MFC can use MFC extension DLLs. To create one of these extension DLLs, you can run App Wizard for DLLs and pick the third choice (MFC Extension DLL) in the dialog (see Figure 7.1). This results in an empty project.

To flesh out the extension DLL, you simply add classes and resources to suit your needs. To make the classes visible to other programs, you define them in your H file using the **AFX_EXT_CLASS** macro:

```
class AFX_EXT_CLASS CCustomClass : Public CWindow . . .
```

This causes the compiler to export the entire class. Although you can individually export functions (using **__declspec(dllexport)**), you'll find it's difficult to do so because many of the MFC macros generate functions you'll also need to export. Of course, as this exports everything using the long, decorated C++ names, this isn't a very efficient method. The alternative, however, is very tedious. You'd need to generate a listing file, decipher all of the long names, and manually build a DEF file so you could export by ordinal number. If you decided to do this, at least wait until you are about to ship the DLL, and then build the DEF file so you won't have to manually maintain it.

Extension DLLs don't have a **CWinApp** object—that's the purview of the main application. Instead, MFC DLLs have a **CDynLinkLibrary** object. If you need to keep separate variables for each MFC instance using your DLL, you can derive a new class from **CDynLinkLibrary** and use it instead in the **DLLMain** function. Just put whatever data you need to manipulate in the new class.

One other thing that extension DLLs share with their main program is resources. This is good if you want to share resources between your DLL and the program, but it does present a problem. Resource IDs are global, so if you want to supply a bitmap, for example, it had better have an ID that the main program isn't using. Most programmers reserve blocks of resource IDs for their DLLs, but this gets complicated if you are using multiple DLLs from several sources. MFC searches all the DLLs (including the system DLLs) and the main program for resources.

If you're certain that you want to load a resource from a specific module, you can arrange for MFC to only look in one place. First, call **AfxGetResourceHandle** to obtain the current resource handle and cache it away. Then you can call **AfxSetResourceHandle** to set the handle to the instance handle of the module you want to search. Once you have the resource handle you want, be sure to call **AfxSetResourceHandle** again to restore the previous setting.

To use an MFC extension DLL, you simply include the header and LIB files in your ordinary project. Why is this different from an ordinary DLL? You can pass MFC objects back and forth, export entire classes, and share resources, all things that would be difficult to do with an ordinary DLL. Of course, it goes without saying that MFC extension DLLs only work with MFC. You can't easily use them with any other programming environment or language.

You can find a simple example of an MFC extension in Listings 7.5 and 7.6. This extension is nothing more than the modeless dialog class from Chapter 5—there aren't any changes relevant to the DLL. The difference? A programmer who wants to use the class now only needs the DLL, the header file, and the LIB file.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

• [Advanced](#)[Search](#)• [Search Tips](#)BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Listing 7.5 shows the main part of the DLL. It creates the **CDynLinkLibrary** object. Of course, you could change it to use a derived class if you wanted to do so. You can also add code here to recognize when the system loads the DLL, although with MFC, that isn't usually very interesting.

Listing 7.5 An MFC extension main file.

```
// mlessdlg.cpp : Defines the initialization routines for the DLL.
//
```

```
#include "stdafx.h"
#include <afxdllx.h>
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
static AFX_EXTENSION_MODULE MlessdlgDLL = { NULL, NULL };
```

```
extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    // Remove this if you use lpReserved
    UNREFERENCED_PARAMETER(lpReserved);
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        TRACE0("MLESSDLG.DLL Initializing!\n");

        // Extension DLL one-time initialization
        if (!AfxInitExtensionModule(MlessdlgDLL, hInstance))
            return 0;
    }
}
```

```

        // Insert this DLL into the resource chain
        // NOTE: If this Extension DLL is being implicitly linked to by
        // an MFC Regular DLL (such as an ActiveX Control)
        // instead of an MFC application, then you will want to
        // remove this line from DllMain and put it in a separate
        // function exported from this Extension DLL. The Regular DLL
        // that uses this Extension DLL should then explicitly call that
        // function to initialize this Extension DLL. Otherwise,
        // the CDynLinkLibrary object will not be attached to the
        // Regular DLL's resource chain, and serious problems will
        // result.

        new CDynLinkLibrary(MlessdlgDLL);
    }
    else if (dwReason == DLL_PROCESS_DETACH)
    {
        TRACE0("MLESSDLG.DLL Terminating!\n");
        // Terminate the library before destructors are called
        AfxTermExtensionModule(MlessdlgDLL);
    }
    return 1;    // ok
}

```

Listing 7.6 The extension DLLs class.

```

// ModelessDlg.cpp : implementation file
//

#include "stdafx.h"
#include "stdafx.h"
#include "ModelessDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////
// CModelessDlg dialog
CModelessDlg::CModelessDlg(CWnd* pParent /*=NULL*/)
{
    //{{AFX_DATA_INIT(CModelessDlg)
    // NOTE: the Class Wizard will add member initialization here
    //}}AFX_DATA_INIT
}

void CModelessDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CModelessDlg)
    // NOTE: the Class Wizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
}

```

```

BEGIN_MESSAGE_MAP(CModelessDlg, CDialog)
//{{AFX_MSG_MAP(CModelessDlg)
// NOTE: the Class Wizard will add message map macros here
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CModelessDlg message handlers

void CModelessDlg::OnOK()
{
    UpdateData(TRUE);
    DestroyWindow();
}

void CModelessDlg::OnCancel()
{
    DestroyWindow();
}

void CModelessDlg::PostNcDestroy()
{
    delete this;
}

```

What About OLE (Or ActiveX) DLLs?

Visual Basic programmers often speak of OLE (or ActiveX) DLLs. C++ programmers usually refer to these as inproc ActiveX servers. Although these are physically DLLs, they are DLLs that the ActiveX system software uses to create objects. Your involvement with them will be purely as ActiveX objects. That's another chapter (Chapter 8, to be exact). For now, just realize that the material in this chapter only vaguely applies to ActiveX DLLs because these DLLs only slightly resemble ordinary ones.

Summary

DLLs are the cornerstone of Windows programming. Although code sharing is done more often with ActiveX these days, DLLs are still the first (and sometimes best) way to share one piece of code between multiple programs. DLLs are efficient and universally understood (if you construct them correctly).

One of the most interesting uses of DLLs is to allow field customization of your software. MFC exemplifies this by allowing you to add your DLLs to the MFC system DLLs, which provides custom programming objects very easily.

Practical Guide To DLLs And MFC

- Determining What DLLs A Program Uses Or Functions A DLL Exports
- Linking At Build Time
- Linking At Runtime
- Creating A DLL
- Exporting Functions And Data
- Creating An MFC Extension DLL
- Optimizing DLL Load Addresses

Determining What DLLs A Program Uses Or Functions A DLL Exports

You can examine programs and DLLs using Microsoft's DUMPBIN utility. This command line program takes any of several options and a file name. If you use the /EXPORTS option, the program displays the symbols the file exports. Using /IMPORTS shows the DLLs and the functions that the program requires to run. You can run DUMPBIN with no arguments to see a list of all the possible options.

Linking At Build Time

The easiest way to include a DLL into your program is to include its header file and link the associated LIB file with your project. If you are using a DLL that is meant for use with a C program, you should surround the **#include** statement with a special form of the **extern** statement:

```
extern "C" {  
    #include "dllheader.h"  
}
```

There is a slight performance benefit to placing the special keyword **__declspec(dllimport)** in front of functions you plan to get from a DLL. You must use the **__declspec** statement in conjunction with data you wish to import.

Linking At Runtime

You can also include a DLL at runtime. The trick is to call **LoadLibrary** with the DLL's file name. This returns a handle that you can pass to **GetProcAddress**. You also pass **GetProcAddress** the name or ordinal number of a function. You get back a pointer to the function. Be sure to call **FreeLibrary** when you no longer need the DLL.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Creating A DLL

There are three kinds of DLLs you can create using MFC. For all of them, you should use the MFC App Wizard for DLLs. The first kind of DLL is one that uses MFC and includes a complete copy of MFC. This makes for a large, but self-contained, DLL. You can also choose to create a DLL that uses MFC DLLs. This makes your DLL smaller, but it also makes it dependent on having the correct version of the MFC DLLs present. The third kind of DLL you can create is an MFC extension DLL.

Both types of DLLs have their own application object derived from **CWinApp**. This object (which isn't really an application) is how the DLL can load its resources, learn its instance handle, and so on.

Exporting Functions And Data

When you want to export global functions and data, you can do so in one of several ways. First, you can mark them with the **__declspec(dllexport)** modifier. Second, you can add a DEF file to your project and name the symbols in the **EXPORTS** section. Third, you can export symbols using the **/EXPORT** switch to the linker.

If you are using the MFC library in a DLL (as opposed to your own private copy), you need to inform MFC each time you enter a DLL function. You can accomplish this by adding this line to the beginning of each exported function:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState());
```

Creating An MFC Extension DLL

If you want to export entire classes to another MFC program, you'll need to create an MFC extension DLL. You still use the same wizard. However, the target program (that is, the program that will use the DLL) must use MFC DLLs in order to use your program.

To export a class, you can mark it with **AFX_EXT_CLASS**. You can also export individual items using any of the usual methods (see "Exporting Functions and Data," in this practical section). However, exporting only a few functions from a class can cause problems because MFC often inserts members you may need to export. It is usually best to export the entire class unless you have good cause to do otherwise.

Your DLL will contain a **CDynamicLinkLibrary** object that MFC uses to manage it. You can derive a new class and use it instead if you'd like to have private variables for each instance of your DLL.

Optimizing DLL Load Addresses

When Windows NT loads a DLL on behalf of a process, it tries to map the DLL into the process' virtual memory space at a preferred load address. The DLL is already set to load at that address. If that works, subsequent processes that load the DLL can use essentially the same copy of the DLL. However, if one of the processes doesn't have that area of memory available (in other words, something else is consuming that space), NT has to reload the DLL, which can be time-consuming.

The answer is to select different preferred addresses for each DLL that you plan to use. The system DLLs, for example, have their own addresses to prevent conflict with each other. You can set the base address with the /BASE option to the linker, or by using the REBASE command line utility. Simply use the -r option and specify a new load address.

Windows 95 loads DLLs into a memory area shared by all programs. Therefore, this isn't an issue under Windows 95.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Chapter 8 ActiveX

ActiveX brings the advantages of object-oriented programming to the binary level. With ActiveX-specific tools already built in, MFC makes using ActiveX easy, as long as you know how to use them.

Programmer's Notes...

How many times have you heard the old expression "The whole is greater than the sum of its parts?" Real-life examples of this axiom abound. Look at children's toys. Tinkertoys, Legos, and Lincoln Logs are all simple pieces of wood or plastic. Individually, they are completely unremarkable. But mix them together with some nine-year-old imagination and you really have something.

NASA recently toured an exhibit of spacecraft built from Legos. I've heard there are several Tinkertoy computers that play games, and my son Patrick and I built a very nice Morse code key from Legos. We couldn't have built a complex structure without the use of simple components.

The same holds true of computer hardware and software. How many clone makers could build PCs without Pentium chips? On the other hand, what good is a Pentium PC without memory and other circuitry? Years ago, I wrote software from the ground up using assembly language. Now I depend on libraries to read files, print, and do a lot of other simple, mundane tasks.

Software practice evolves over time to help build more reusable parts. Early

programs used subroutines as a way to reuse code. Later, other programming systems introduced object-code libraries and other ways to package reusable code.

Today, object-oriented programming is the preferred way to provide reusable parts. There are many ways to write object-oriented (OO) programs. It is easiest if you have a language that supports OO like SmallTalk. However, you can write OO programs in any language, if you have the discipline. C++ is odd because it allows you to use OO constructs when you want and to ignore them when you prefer.

But even within C++, how reusable is your code? Sure, you can create source code objects, but can you move those objects to other compilers? Usually not at the object code level, and often even the source code will require some work. All bets are off if you plan on reusing code with another language like Visual Basic or Delphi. Unless your code is very simple, you'll probably not have much luck going to other platforms like Unix, either.

To have reusable software, you'd like it to be easy to share code between various languages and platforms. This is especially important because the Internet has brought unprecedented connections between dissimilar machines.

Several solutions to this problem are now available, including ActiveX. ActiveX is a technology used to build binary-compatible objects. That might not be what you've heard. Everyone knows that ActiveX is about embedding documents in containers and putting controls on the Internet, right? Not exactly. Sure, you can do all of those things with ActiveX, but that's not what ActiveX is—any more than a bunch of two-by-fours are a house. You can create controls to use on the Internet by building ActiveX objects. These objects are binary-compatible with other ActiveX objects used by the Web browser. You can build ActiveX objects that allow you to place your document inside another document (or host other documents). These documents are ActiveX objects. You can use two-by-four lumber to build a house—you could also use it to build sawhorses. Similarly, ActiveX objects can construct big systems, or small (and efficient) objects.

How Is ActiveX Different From OLE, OCX, And COM?

When Microsoft first released technology that allowed you to put (for example) a spreadsheet into a word processing document, they christened it OLE, meaning object linking and embedding. However, they also used the same term to describe the system used to create binary-compatible objects. Because of the complexity involved in constructing the objects required to insert documents into one another, many programmers felt OLE was difficult, even though the base technology is simple.

Microsoft wanted more people to embrace OLE, so they decided to rename the base technology to the Component Object Model (COM). For some reason, this seemed to further terrify most programmers (perhaps they thought the Model in COM referred to a design methodology). After this, there were many attempts to rename the technology. Controls for VB and similar programs became known as OLE Control Extensions (or OCXs).

Finally, to clear the slate, Microsoft declared the next major release of the OLE specification to be ActiveX. You still hear the other terms occasionally. Personally, I like to call traditional document-inside-document programs OLE. Visual controls are still OCXs to me. COM is simple objects that are not designed to work inside containers like Visual Basic, and ActiveX is the technology that makes them all work. However, this is just my preference.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

What Is An ActiveX Object?

This may be the simplest statement in this book: An ActiveX object is any code that has one or more tables of function pointers. As long as the provider of the object (the server) and the user of the object (the client) agree on what those functions do, nothing else is important. The functions don't need to be in the same language, or even on the same machine.

So, you ask, where is the object-oriented part? Well, ActiveX is just a little more complicated than I've let on (but only a little). The first three functions in each table of function pointers always do the same things. ActiveX function pointer tables are known as *interfaces*, and the three functions that are always present are known as the **IUnknown** interface.

The three **IUnknown** functions aren't very complicated. Two of them (**AddRef** and **Release**) track how many copies of the interface are in use. Suppose you create an object that has two interfaces (it is perfectly normal for objects to have multiple interfaces). The object's reference count is one (because you created it). Now suppose you are going to pass the object to another thread in your program. Before you send the object, you need to call **AddRef** through one of the interfaces. This brings the count to two. Usually, it doesn't matter which interface you use because nearly all objects keep a single count for the object regardless of the number of interfaces it supports. Later, when you are done with the object, you call **Release**. This deducts one from the count. Presumably, the thread will do the same. When the object notices the count has dropped to zero, it's free to destroy itself.

The other function in **IUnknown** is **QueryInterface**. Keep in mind that,

because these are just function pointers, the name is only a convention. The purpose of **QueryInterface** is to ask one interface if it knows about another. Suppose an object has two interfaces, **IA** and **IB**. If you have a pointer to the **IA** interface, you could query for **IB** using **QueryInterface** (or vice versa). If you asked for a different interface, you'd get a NULL pointer. Each successful call to **QueryInterface** increments the reference count by calling **AddRef**. Because every interface starts with **IUnknown**, you can treat any interface pointer as though it were an **IUnknown** and query it until you get something that you understand. It isn't unusual to ask an interface if it knows about itself (at which point **QueryInterface** returns a pointer to the same interface you started with).

As you can see, these three functions aren't very complicated. Yet, like Tinkertoys, they can build a powerful object-oriented system. Before you decide if **IUnknown** really makes OO programming possible, consider what the goals of an object-oriented programming system are.

ActiveX And OOP

What are the fundamental goals of any object-oriented programming system? Be careful. I'm not talking about meta-goals such as better maintainability or specific language features in C++ or other languages. I mean concrete programming goals.

Most people will agree that the following items are three main things an OO system needs:

- Encapsulation—An object's implementation details should be private.
- Reuse—It should be easy for one object to reuse code from another object. It is even better if you can slightly modify the code if it isn't exactly what you want. You shouldn't have to change code that is exactly what you want.
- Polymorphism—This is a fancy way of saying that you want to treat objects of several classes as though they were members of a different, related class. The example I like to use is an auto dealership that needs a database. To start with, you design the Vehicle object. Suppose you also have types of Vehicle named Car and Truck. In the main database, you don't really want to have a list of Cars and another list of Trucks. Instead, polymorphism allows you to treat Cars and Trucks as if they were both Vehicles. Later, if you need to, you can decide the specific type of each object.

Notice that derivation is not on the list. That's because derivation isn't a goal. It is simply the way C++ (and many other languages) achieves code reuse and polymorphism.

ActiveX Encapsulation

How can ActiveX address each of these goals? Encapsulation is easy. Because the only part of your object the outside world sees is its interface tables, objects are tightly encapsulated. If you have code that isn't exposed in an interface, that function is private. You may notice that there is no such thing as

an ActiveX data member. If you have any variables the outside world will read or set, you must provide access functions for them. So it is easy to see that ActiveX objects are tightly encapsulated.

When you define an interface, you create a contract between the object's server and the users of the object. That contract should not change. As long as the interface does the same thing, you are free to change your implementation details (even the implementation language) at will.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

ActiveX Reuse

Reuse is trickier. There are two ways ActiveX objects can reuse other ActiveX objects: containment and aggregation. Containment is straightforward. Suppose you have a Vehicle object and a Car object. There's no reason the Car object can't create a Vehicle object to do some of its work. Imagine that these objects have an **IVehicle** interface. The Car object's interface could be nothing more than functions that wrap the same functions in the contained Vehicle object. Remember, you have tight encapsulation, so no outside entity will ever know this is happening.

C++ uses derivation to achieve reuse. When you derive from a base class, you can override functions. In the override, you can call into the base class as part of your new processing. In ActiveX, using a function that does nothing but call a contained object is similar to deriving from the contained object and not providing a function override. If your new function elects to do some work before or after calling the contained class, that's similar to calling a C++ base class from within an override. Finally, your object could do something unique for a particular function and elect to not call the corresponding function in the contained class. This is similar to overriding a class and providing a new override function that doesn't call into the base class.

The other way ActiveX objects can reuse another object is through aggregation. Aggregation is most useful when an object wants to expose an entire interface from another object, even if it doesn't know much about that object. For example, Visual Basic uses aggregation to add its own custom interfaces to third-party objects.

Both your object and the reused object must specifically support aggregation. That means that you might want to aggregate an object and find that it doesn't have the necessary support. Also, aggregation doesn't give you any opportunity to alter the interface you expose. To support aggregation, your **IUnknown** interfaces have to cooperate. Imagine that you have a Car object that aggregates a Vehicle object. It wouldn't do if you were to query the **IVehicle** interface and it didn't know about the **ICar** interface in the same object.

Why doesn't ActiveX use derivation? Certainly, derivation does offer a handy way to reuse code. However, languages such as C++ have great difficulty reusing code unless there is source code present. Even for a DLL, you need a header file. ActiveX's overriding concern is that it be language-independent. With containment and aggregation, any object can reuse any other object regardless of either object's choice of language. You don't need the source code at all. You might write a C++ object that contains a Visual Basic object. If the Visual Basic programmer has a change of heart and starts writing in Delphi, so what? As long as the interfaces remain the same, no one really cares what languages are in use.

ActiveX Polymorphism

You may have already guessed how ActiveX achieves polymorphism. Suppose you have a Car object that has an **ICar** interface and an **IVehicle** interface. Then imagine you have a Truck object that has an **ITruck** interface but also has an **IVehicle** interface. These objects are both Vehicles. If you have a pointer to an **IVehicle** interface, you don't care if it is a Car or a Truck. Later, if you do need to know, you can query the **IVehicle** interface for the interface of interest. If you query for **ICar** and **QueryInterface** returns NULL, then you must have a Truck (or some other object that is a kind of Vehicle).

Again, this is a very powerful idea. You can write objects that are polymorphic with other objects even if you don't have their source code. Also, objects can be polymorphic with any number of other objects. For example, the Car and Truck objects might have **IPersistFile** interfaces. This is a standard interface for objects that save their state in a file. By adding this interface, the objects become polymorphic with all other objects that provide **IPersistFile**.

Of course, you don't get the reuse that C++ derivation provides. Supporting an **IPersistFile** interface means writing all of the code yourself (unless you contain or aggregate an object that does what you want).

Fun With Interfaces

Each interface has a unique 128-bit number that identifies it (an IID). Also, each object has a 128-bit number (a CLSID) that identifies it. These 128-bit numbers are known as GUIDs (Globally Unique Identifiers) or UUIDs (Universally Unique Identifiers). The numbers appear in the system registry to uniquely identify the object and its interfaces. There are also ways to enter type library information into the registry. A type library describes an object and its interfaces. This is necessary because a client needs to know what functions are available in a server's objects.

Elementary ActiveX objects use early binding to find methods. That is, you have to know ahead of time that you want to call a particular slot in the interface table. However, in some cases, it is advantageous to use late binding (finding functions at runtime). However, that's no problem because you can easily define an interface to allow late binding. In fact, Microsoft does define such an interface and it is **IDispatch**.

By using **IDispatch**, objects can expose properties, methods, and events to the outside world. This is the base technology for ActiveX controls (OCXs), ActiveX scripting (automation), and a host of other technologies.

Properties

Properties appear as ordinary variables to the client program. Some properties will be read-only (or even write-only). Others allow full access. When a client reads or writes a property, ActiveX calls subroutines in the server. These subroutines may just access a real variable. However, it might also validate or convert the data. For example, suppose you write a Thermostat object. There is a property named **SetPoint** that controls the temperature at which the Thermostat will turn on. Originally, the object uses Fahrenheit temperature. Later, you decide to start using Centigrade temperatures internally. However, the public interface must not change, so it has to continue to use Fahrenheit. No problem. Your property subroutines can easily convert the value to Centigrade on the way in and reverse the conversion in the other direction. You could also add a new property that accepted Centigrade for new programs to use.

You can't use any data type you want for a property. You can only use certain types (sometimes known as OLE Automation types). However, most useful types are there, including strings, objects, currency, dates, and so forth.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Methods

Methods are nothing more than ordinary functions that the server exposes to the outside world. Of course, the arguments and return values must be of the approved types (just like properties). When the client calls a method, your function executes. It isn't unusual for the internal function to have the same name as the method, but it's not a requirement.

Events

Events are little more than functions that your object calls that reside in the client. You use events to notify the client when something happens. For example, you might have an event that signals the client when the user clicks the mouse over an object. Events are reverse method calls, so they are subject to the same limitations as methods.

Names Vs. Numbers

Each property, method, and event (collectively, the members) in a dispatch interface has a DISPID (or dispatch ID) that identifies it. Clients can use names or numbers to access the members.

Certain DISPIDs are reserved for common members. For example, the foreground color of a control has a reserved DISPID. This allows programs to manipulate these members without having to know their names (which might be in any language, for example). These members are known as stock members (for example, foreground color is a stock property).

There is another type of property available to ActiveX controls: ambient properties. These are properties that the client (or container, if you prefer) exposes to the control. For example, suppose you write an ActiveX control that displays a spinning logo on a Web page. You might want this control to use the same background color that other items on the page use. The Web browser's ambient background property will let you do this. Simply read the ambient property and use it when drawing your background.

ActiveX And MFC

If the preceding discussion seems general, that's because it is. MFC hides much of the details of ActiveX from you, so you won't often have to know all the gory internals (which is a good thing).

MFC has good support for creating objects that support **IDispatch**. If you want to create such an object, run App Wizard, and make sure to select the Automation check box (Automation is a synonym for ActiveX scripting, which means the object supports **IDispatch**).

If you are creating an EXE program, you can add **IDispatch** to your document object. Usually, you'll be creating a DLL. In that case, you need to create an object derived from **CCmdTarget**. This object will serve as the **IDispatch**-aware object. Select Add Class from the Class Wizard dialog to add a new class. In the dialog that appears, derive a new class from **CCmdTarget**. Be sure to check the Automation radio button (see Figure 8.1). Usually, you'll select the last radio button and provide a short name for the object. You can often use this short name instead of the full CLSID (although only the CLSID is sure to be unique).

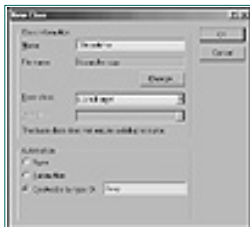


Figure 8.1 Creating a new ActiveX object.

Once you have an object, select Class Wizard's Automation tab (see Figure 8.2). This tab will allow you to add methods or properties. MFC supports the idea of common properties (like background color) for ActiveX controls. These are stock properties that have predefined names and semantics. However, in a general-purpose automation object like this one, there are no stock properties. You'll define any custom methods and properties that you require.

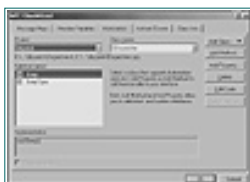


Figure 8.2 Defining properties and methods.

To define a property, fill in its name and type (see Figure 8.3). If you want to

map this property to a variable in your code, make sure the Member Variable radio button is on and enter the name of the variable in the Variable Name field. Class Wizard will automatically add this variable to your class. If you like, you can add a function in the Notification Function field that MFC will call any time another program changes your property.

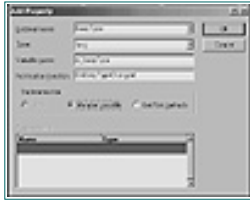


Figure 8.3 Defining a property.

Some properties don't map directly to variables. For example, suppose you want a property in centimeters, but you represent it internally in pixels. Then you might elect to select the Get/Set Methods radio button instead of the Member Variable button. This allows you to specify functions that MFC will call when a program attempts to read or write the property. You can take whatever action you require in these functions. If you need a variable, in this case, you'll have to define it yourself. If you want a read-only property, don't supply a write function. If you want a write-only property (as odd as that sounds), don't supply a read function.

MFC's implementation of **GetIDsOfNames** (the function that converts names to DISPIDs) uses the name of the property. If you use MFC, you don't need to care about the DISPIDs. You supply the names and MFC will automatically assign and convert appropriate DISPIDs.

Adding methods is very similar. The Class Wizard dialog (see Figure 8.4) allows you to select an external name (used in **GetIDsOfNames**) and an internal name (the name of your function). These names may be the same, if you like. You also select a return type and any arguments you want the method to accept (up to 16 arguments are possible). Again, Class Wizard knows about stock methods, but these are only useful for ActiveX controls.

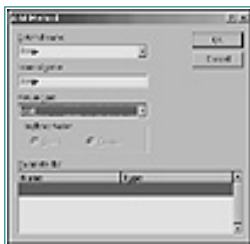


Figure 8.4 Adding a method.

The other related tab in Class Wizard is the ActiveX Events tab. This is only useful for ActiveX controls. You can define events here in a way similar to how you define methods (but only if you are creating an ActiveX control as opposed to an ordinary ActiveX dispatch object).

As you can see, building an **IDispatch** object with MFC is almost trivial. You use Class Wizard to do all the work. MFC provides a class factory (what ActiveX uses to create the object) and an implementation of **IDispatch** based on your input to Class Wizard. Better still, the MFC code is quite efficient.

You can read more about MFC's **IDispatch** implementation in the MFC technotes (specifically, look at notes 38 and 39).

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

To illustrate how simple this is with MFC, look at Listings 8.1 and 8.2. These show a simple **IDispatch** object that has a property and a method. The property selects a type of beep (the default is -1) and the method beeps using the selected type (by calling **MessageBeep** from the standard API).

It's amazing how little of this code did not come from MFC. I added a single line to the constructor to set the property variable to -1. I also wrote the **Beep** function (although Class Wizard already had a stub in place for me). I also had to add the **DECLARE_OLECREATE** line in the header and the **IMPLEMENT_OLECREATE** line in the CPP file. That's it. The rest of the code is from App Wizard or Class Wizard.

If you register the class appropriately (see Listing 8.3), you can use this object from Visual Basic or any other language that can act as an automation controller or ActiveX script language. You can also use Visual C++ and MFC to write controllers. Simply ask Class Wizard to add a class from a type library and select the automation object's DLL, EXE, or TLB (type library) file. This will cause Class Wizard to create a class that stands in for the automation object. It writes simple functions that call **Invoke** with the correct arguments for each case.

Listing 8.1 An IDispatch example.

```
// Dispatcher.cpp : implementation file
//

#include "stdafx.h"
#include "dispatch.h"
#include "Dispatcher.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CDispatcher
```

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC



```

IMPLEMENT_DYNCREATE(CDispatcher, CCmdTarget)
// Add to Class Factory
IMPLEMENT_OLECREATE(CDispatcher, "Beeper",
    0x8755aa4, 0xd365, 0x11cf, 0xa7, 0xb2, 0x44,
    0x45, 0x53, 0x54, 0x0, 0x0)
CDispatcher::CDispatcher()
{
    EnableAutomation();
    m_beepType=-1; // default
}
CDispatcher::~CDispatcher()
{
}

void CDispatcher::OnFinalRelease()
{
    CCmdTarget::OnFinalRelease();
}

BEGIN_MESSAGE_MAP(CDispatcher, CCmdTarget)
    //{AFX_MSG_MAP(CDispatcher)
    //}AFX_MSG_MAP
END_MESSAGE_MAP()

BEGIN_DISPATCH_MAP(CDispatcher, CCmdTarget)
    //{AFX_DISPATCH_MAP(CDispatcher)
    DISP_PROPERTY(CDispatcher, "BeepType", m_beepType, VT_I4)
    DISP_FUNCTION(CDispatcher, "Beep", Beep, VT_EMPTY, VTS_NONE)
    //}AFX_DISPATCH_MAP
END_DISPATCH_MAP()

// {08755AA4-D365-11CF-A7B2-444553540000}
static const IID IID_IDispatcher =
    { 0x8755aa4, 0xd365, 0x11cf, { 0xa7,
        0xb2, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0 } };

BEGIN_INTERFACE_MAP(CDispatcher, CCmdTarget)
INTERFACE_PART(CDispatcher, IID_IDispatcher, Dispatch)
END_INTERFACE_MAP()

////////////////////
// CDispatcher message handlers

void CDispatcher::Beep()
{
    MessageBeep(m_beepType);
}

```

Listing 8.2 The dispatch object header.

```

// dispatch.h : main header file for the DISPATCH DLL
//
#ifdef __AFXWIN_H__
#error include 'stdafx.h' before including this file
#endif

#include "resource.h" // main symbols

```



```

long IDispatcher::GetBeepType()
{
    long result;
    GetProperty(0x1, VT_I4, (void*)&result);
    return result;
}

void IDispatcher::SetBeepType(long propVal)
{
    SetProperty(0x1, VT_I4, propVal);
}

////////////////////////////////////////
// IDispatcher operations

void IDispatcher::Beep()
{
    InvokeHelper(0x2, DISPATCH_METHOD, VT_EMPTY,
        NULL, NULL);
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Armed with this wrapper class, it's merely a matter of creating and using this automation class. Listing 8.5 shows a simple program that displays a dialog. When you click on OK, it creates an automation object and asks it to beep. Notice that MFC's App Wizard wrote most of the code. The only original lines are the include lines for the dispatch class and the code that handles the OK case in **InitInstance**.

Listing 8.5 An automation controller.

```
// User.cpp : Defines the class behaviors
// for the application.
//
#include "stdafx.h"
#include "User.h"
#include "UserDlg.h"
#include "dispatch.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CUserApp

BEGIN_MESSAGE_MAP(CUserApp, CWinApp)
   //{{AFX_MSG_MAP(CUserApp)
```

```

        //}}AFX_MSG
        ON_COMMAND(ID_HELP, CWinApp::OnHelp)
    END_MESSAGE_MAP()

    //////////////////////////////////////
    // CUserApp construction

    CUserApp::CUserApp()
    {
        // TODO: add construction code here
        // Place all significant initialization in InitInstance
    }

    //////////////////////////////////////
    // The one and only CUserApp object

    CUserApp theApp;

    //////////////////////////////////////
    // CUserApp initialization

    BOOL CUserApp::InitInstance()
    {
        // Initialize OLE libraries
        if (!AfxOleInit())
        {
            AfxMessageBox(IDP_OLE_INIT_FAILED);
            return FALSE;
        }

#ifdef _AFXDLL
        Enable3dControls();
#else
        Enable3dControlsStatic();
#endif

        // Parse the command line
        if (RunEmbedded() || RunAutomated())
        {
            COleTemplateServer::RegisterAll();
            return TRUE;
        }
        COleObjectFactory::UpdateRegistryAll();

        CUserDlg dlg;
        m_pMainWnd = &dlg;
        int nResponse = dlg.DoModal();
        if (nResponse == IDOK)
        {
            IDispatcher disp;

```

```

        disp.CreateDispatch("Beeper"); // ProgID
        disp.Beep();
    }
    else if (nResponse == IDCANCEL)
    {
        // No action
    }
    return FALSE;
}

```

MFC And ActiveX Controls

The basic steps in using MFC to create an ActiveX control are quite simple. Remember, a control is a type of object that implements the interfaces required to exist within a Web browser, Visual Basic, or in an MFC window. First, you run a special wizard to create a basic control. The wizard automatically generates a UUID for your control and sets up all the necessary IDL and source files. If you build the project, you'll have a generic, useless (but functioning) control.

You'll want to add code to paint your control in a meaningful way. You'll also use a special tool, Class Wizard, to add properties, methods, and events. In some cases, Class Wizard will write all the code for you. In other cases, you'll have to write code to produce the desired effect. In either case, Class Wizard will at least start you in the right place and keep your IDL file and registry entries in line.

Of course, you'll also use the normal MFC mechanisms to handle messages, draw, and perform other mundane tasks. This is a plus, unless you don't know MFC. Don't worry, though. The amount of MFC you need to know to write most controls is minimal compared to the amount of MFC you need for a regular application. If you've struggled with document/view architecture and other MFC oddities, you'll be relieved to know that you don't need all of that for a control.

Using Control Wizard

To start your control, select New from the File menu. In the resulting dialog box, select the Workspace tab. This creates a new project with its own makefile and sample source files. Once you select Project Workspace, you'll see a dialog box that contains a list of project types (see Figure 8.5). You want to select the one labeled Control Wizard. You can select a project name and directory using the right side of the dialog. Press the Create button to proceed.

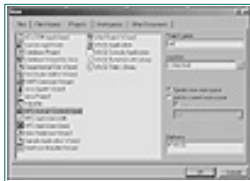


Figure 8.5 Starting a new control project.

You'll find that the Control Wizard produces two dialogs to collect information (see Figures 8.6 and 8.7). The first dialog allows you to specify several things:

- How many controls you want in your project
- If you want a runtime license (that is, if you want to support a mechanism that

requires developers to have a key to design with your control)

- If you want source file comments
- If you want skeletal help files



Figure 8.6 The Control Wizard's first screen.

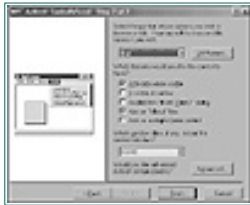


Figure 8.7 The Control Wizard's second screen.

Once you press the Next button, the final dialog box will appear. On this screen, you can edit the names of each class the wizard will create and change the file name it will use for each class (you'll usually accept the defaults). You can also select among several options:

- The control may always activate when visible
- The control may be invisible at runtime
- You may elect to have the control appear in an ActiveX program's insert list (like any other OLE-style document)
- The wizard will create an About box, if you request it
- The control can act as a simple frame container that can contain other controls

You can also elect to subclass a standard Windows control (such as a button). This is very useful if you want, for example, a list box that happens to expose functions through properties, methods, and events. You don't have to draw specific code to implement the drawing of such a class.

Code You Add

Often, the only code that you'll add directly to the files that the Control Wizard generates is the **OnDraw** function. This function paints the control on a device context. Instead of a normal Windows DC, it receives an MFC CDC.

On occasion, you might want to add some custom code to the control's constructor or destructor. However, when it comes to adding properties, methods, and events, you'll use Class Wizard to either write the code or to place stub functions in the code that you will finish writing.

The other code that you might add to a control is code to handle ordinary window events. For example, a control that acts like a button might accept mouse events. You use Class Wizard to handle these events in exactly the same way that you use Class Wizard in the usual MFC program.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Adding Properties

Properties are the heart and soul of ActiveX controls. Adding them with Class Wizard couldn't be easier. Start Class Wizard and select the Automation tab, then click on Add Property (see Figure 8.3). From here, you can select a stock property or name a new property. If you select a stock property, Class Wizard automatically arranges to store it and provides a notification function you can override to detect when the property changes. For example, when the stock **Text** property changes, the framework will call **OnTextChanged**. You can retrieve the text as a **BSTR** by using the **GetText** function, or as a **CString** by calling **InternalGetText**. You need only inform Class Wizard that you want the stock property.

When you add a custom property, you can select between two types. The member variable type allows you to define a member variable that corresponds to the property. If you like, you can also define a function that the framework will call when the property changes. Class Wizard automatically provides the function's skeleton.

Your other choice for properties is to use **get** and **set** member functions. In this case, you specify a member function that the framework calls when a container sets the property (the **set** function) and another function that supplies the value of the property (the **get** member function). You don't need to supply both functions if you want a read-only or write-only property.

When you create a member function property, Class Wizard automatically creates skeleton functions for you. The **get** function returns a value that has the same type as the property. The **set** function takes a value as an argument. What

you do with these functions is up to you. If you need a variable to store the property, you'll have to define it manually.

By default, stock properties are persistent. That is, the container saves them when it saves the control, and MFC arranges to load them back in when the container loads the control. If you want any custom properties to be persistent, you need to add a special **PX_** function to the **DoPropExchange** function. These functions (there is one for each common type) take the name of the property, the corresponding variable in your code, and a default value as arguments. Of course, some runtime properties may not require persistence. In that case, you don't need to take any special action in the **DoPropExchange** function.

Using Ambient Properties

Although you use them, you don't define ambient properties in your code. For common ambient properties, MFC provides simple functions to retrieve them (see Table 8.1). You can also get any arbitrary ambient property by calling **GetAmbientProperty**. Of course, then you need to know the property's DISPID.

Table 8.1 Standard ambient property functions.

Function
AmbientAppearance
AmbientBackColor
AmbientDisplayName
AmbientFont
AmbientForeColor
AmbientLocaleID
AmbientScaleUnits
AmbientTextAlign
AmbientUserMode
AmbientUIDead
AmbientShowGrabHandles
AmbientShowHatching

Adding Methods

When you want to add a method, select the Add Method button from Class Wizard's Automation tab. This brings up a dialog that you can use to create methods (see Figure 8.4). Each method can have up to 16 parameters.

You won't often need methods if you use properties correctly. For example, suppose you want to create a method called **Open** that opens a file. Why not provide a **FileName** property instead? Then, when the property changes, you can perform the open function.

The Web browser doesn't interact with control methods directly. However, scripting languages can call control methods. Web pages and Web servers can utilize scripting languages to orchestrate complex HTML content using ActiveX controls.

Adding Events

Class Wizard makes adding events trivial. First, select the OLE Events tabs. Then, just name the event (or select a stock event name). Events can have up to 15 arguments that you specify near the bottom of the dialog (see Figure 8.8).

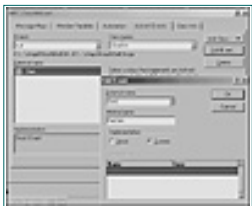


Figure 8.8 Defining an event.

Class Wizard automatically writes a complete function that will trigger the event (the function will start with **Fire**, as in **FireClick**). You don't need to take any further action. When you want to fire the event, just call this firing function. Of course, you have to pass any arguments that you specified when you created the event.

The default MFC code will fire some stock events by default. For example, when the control detects a character (a **WM_CHAR** message), it calls **FireKeyPress** unless you override the **OnChar** message and don't call the base class. You'll find a list of window messages that MFC converts to events in Table 8.2.

Table 8.2 Automatic events.

Message	Event
WM_KEYUP	FireKeyUp
WM_KEYDOWN	FireKeyDown
WM_CHAR	FireKeyPress
WM_?BUTTONDOWN	FireMouseDown
WM_?BUTTONUP	FireMouseUp, FireClick
WM_MOUSEMOVE	FireMouseMove
Note: The Symbol "?" denotes any of a family of messages.	

The Web browser doesn't really care about control events. However, scripting languages (like VBScript or JavaScript) can handle events that your controls generate.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Adding Property Sheets

Although it isn't mandatory, many ActiveX controls support property pages. This allows the user to display a property page to set all properties. Containers implement this command by requesting the **OLEIVERB_PROPERTIES** action using **IOleObject::DoVerb**. As usual, MFC makes it quite simple to implement this feature.

The Control Wizard adds a blank property sheet in the dialog section of the resources. You can add controls (such as edit controls) to the box in the usual way. To associate them with their corresponding ActiveX property, use Class Wizard's Member Variables tab. You can associate a control with a variable and property—Class Wizard takes care of the rest. You can even restrict entries by filling in the appropriate fields in the Class Wizard dialog. For example, you can limit integer properties to a certain range, or you can restrict a string property to a certain number of characters.

In addition to your normal property sheet, you can add additional sheets by adding them to the **BEGIN_PROPPAGEIDS** section of the source file. There are several standard property pages for things like fonts and colors that you can add with no effort. Use **CLSID_CColorPropPage** for colors, **CLSID_CPicturePropPage** for pictures, and **CLSID_CFontPropPage** for fonts. You'll see an example of adding a color property page later in this chapter.

Examining The Generated Files

When you use the Control Wizard to start a project, it generates three

significant CPP files. The first has the same name as your project. This file contains an object derived from **COleControlModule**. This is the base class that represents the ActiveX control. If you need any customized code to execute when the control loads or unloads, you can add it here.

The same file that contains the **COleControlModule**-derived object also contains functions to support self-registration. You usually won't need to modify these functions because they already do the right thing.

The most important file that MFC generates has the same name that your project has, with the letters CTL added to the end (for example, PROJCTL.CPP). This contains an object derived from **COleControl**. You'll make most of your changes, both automatic and manual, in this file. This is where the **OnDraw** function resides, along with all of the events, properties, and methods in your control.

The final source file that MFC creates has the same name as your project has, with the letters PPG appended (for example, PROJPPG.CPP). In this file, you'll find an object that represents your main property page. This object derives from the base class **COlePropertyPage**. You won't often need to change this file, except via Class Wizard.

Testing And Using The Control

You can embed an MFC control into any appropriate container. However, you may find it especially useful to use the test container that comes with the Visual C++ tools. This container (see Figure 8.9) allows you to embed any control into it. Then you can examine its properties, monitor its events, and set conditions, such as ambient properties.



Figure 8.9 The test container.

To insert your control into the test container, you'll need to run TSTCON32.EXE (usually available on the Tools menu of the Visual C++ menu). After you select the Edit menu's Insert Control item, you can use items on the Edit, View, and Options menus to exercise and monitor the control. In particular, you can use the event log window (found on the View menu) to monitor events that the control fires.

If you want to test your control with other containers (such as the Web browser), you'll need to insert the control using the method appropriate for that container. For the Web browser, you'll need to examine your source code to find the control's CLSID and write an HTML script that uses it. Alternately, you can use one of the HTML script generators to automatically insert your control from a list (much like the test container).

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

A Simple Control

Figure 8.9 shows a very simple control, called the Bull control. This simplistic control looks somewhat like an archery target. You can control the colors and the size of the concentric circles via properties. You can also detect (by monitoring events) when the user clicks on the control.

As controls go, this is very simplistic. The control defines three properties and one event. It also uses the ambient background color to fill in the target's background. With MFC, this control is very easy to construct. The only part that requires any significant code is the **OnDraw** function (see Listing 8.6).

Listing 8.6 The Bull control.

```
// BullCtl.cpp : Implementation of the CBullCtrl OLE control class.
```

```
#include "stdafx.h"
#include "bull.h"
#include "BullCtl.h"
#include "BullPpg.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
IMPLEMENT_DYNCREATE(CBullCtrl, COleControl)
```

```
////////////////////////////////////
// Message map
```

```
BEGIN_MESSAGE_MAP(CBullCtrl, COleControl)
```

```
//{{AFX_MSG_MAP(CBullCtrl)
ON_WM_LBUTTONDOWN()
//}}AFX_MSG_MAP
ON_OLEVERB(AFX_IDS_VERB_PROPERTIES, OnProperties)
END_MESSAGE_MAP()

////////////////////////////////////
// Dispatch map

BEGIN_DISPATCH_MAP(CBullCtrl, COleControl)
//{{AFX_DISPATCH_MAP(CBullCtrl)
DISP_PROPERTY_NOTIFY(CBullCtrl, "Step",
    m_step, OnStepChanged, VT_I2)
DISP_STOCKPROP_FORECOLOR()
DISP_STOCKPROP_BACKCOLOR()
//}}AFX_DISPATCH_MAP
DISP_FUNCTION_ID(CBullCtrl, "AboutBox",
    DISPID_ABOUTBOX, AboutBox, VT_EMPTY, VTS_NONE)
END_DISPATCH_MAP()

////////////////////////////////////
// Event map

BEGIN_EVENT_MAP(CBullCtrl, COleControl)
//{{AFX_EVENT_MAP(CBullCtrl)
EVENT_STOCK_CLICK()
//}}AFX_EVENT_MAP
END_EVENT_MAP()

////////////////////////////////////
// Property pages

// TODO: Add more property pages as needed.
// Remember to increase the count!
BEGIN_PROPPAGEIDS(CBullCtrl, 2)
    PROPPAGEID(CBullPropPage::guid)
    PROPPAGEID(CLSID_CColorPropPage)
END_PROPPAGEIDS(CBullCtrl)

////////////////////////////////////
// Initialize class factory and guid

IMPLEMENT_OLECREATE_EX(CBullCtrl, "BULL.BullCtrl.1",
    0x59e8a903, 0xe0c6, 0x11cf, 0xa7, 0xb2, 0x44,
    0x45, 0x53, 0x54, 0, 0)

////////////////////////////////////
// Type library ID and version

IMPLEMENT_OLETYPELIB(CBullCtrl, _tlid, _
    wVerMajor, _wVerMinor)

////////////////////////////////////
// Interface IDs
```

```

const IID BASED_CODE IID_DBull =
    { 0x59e8a901, 0xe0c6, 0x11cf, { 0xa7, 0xb2,
    0x44, 0x45, 0x53, 0x54, 0, 0 } };
const IID BASED_CODE IID_DBullEvents =
    { 0x59e8a902, 0xe0c6, 0x11cf, { 0xa7, 0xb2,
    0x44, 0x45, 0x53, 0x54, 0, 0 } };

////////////////////////////////////
// Control type information

static const DWORD BASED_CODE _dwBullOleMisc =
    OLEMISC_ACTIVATEWHENVISIBLE |
    OLEMISC_SETCLIENTSITEFIRST |
    OLEMISC_INSIDEOUT |
    OLEMISC_CANTLINKINSIDE |
    OLEMISC_RECOMPOSEONRESIZE;

IMPLEMENT_OLECTLTYPE(CBullCtrl, IDS_BULL, _dwBullOleMisc)

////////////////////////////////////
// CBullCtrl::CBullCtrlFactory::UpdateRegistry -
// Adds or removes system registry entries for CBullCtrl

BOOL CBullCtrl::CBullCtrlFactory::UpdateRegistry(
    BOOL bRegister)
{
    if (bRegister)
        return AfxOleRegisterControlClass(
            AfxGetInstanceHandle(),
            m_clsid,
            m_lpszProgID,
            IDS_BULL,
            IDB_BULL,
            afxRegApartmentThreading,
            _dwBullOleMisc,
            _tlid,
            _wVerMajor,
            _wVerMinor);
    else
        return
            AfxOleUnregisterClass(m_clsid, m_lpszProgID);
}

////////////////////////////////////
// CBullCtrl::CBullCtrl - Constructor

CBullCtrl::CBullCtrl()
{
    InitializeIIDs(&IID_DBull, &IID_DBullEvents);
}

////////////////////////////////////
// CBullCtrl::~CBullCtrl - Destructor

CBullCtrl::~CBullCtrl()

```

```

{
    // TODO: Cleanup your control's instance data here.
}

/////////////////////////////////////////////////////////////////
// CBullCtrl::OnDraw - Drawing function

void CBullCtrl::OnDraw(
    CDC* pdc, const CRect& rcBounds,
    const CRect& rcInvalid)
{
    COLORREF fore=TranslateColor(GetForeColor());
    COLORREF back=TranslateColor(GetBackColor());
    COLORREF backgrnd=TranslateColor(AmbientBackColor());
    CBrush br1(fore);
    CBrush br2(back);
    CBrush *old;
    CRect r=rcBounds;
    int step=min(r.Width(),r.Height())/m_step;
    pdc->SetBkColor(backgrnd);
    pdc->ExtTextOut(0,0,ETO_OPAQUE,&r,"",NULL);
    old=pdc->SelectObject(&br1);
    pdc->Ellipse(&r);
    pdc->SelectObject(&br2);
    r.InflateRect(-step,-step);
    pdc->Ellipse(&r);
    pdc->SelectObject(&br1);
    r.InflateRect(-step,-step);
    pdc->Ellipse(&r);
    pdc->SelectObject(old);
}

/////////////////////////////////////////////////////////////////
// CBullCtrl::DoPropExchange - Persistence support

void CBullCtrl::DoPropExchange(CPropExchange* pPX)
{
    ExchangeVersion(pPX,
        MAKELONG(_wVerMinor, _wVerMajor));
    COleControl::DoPropExchange(pPX);
    PX_Short(pPX,"Step",m_step,8);
}

/////////////////////////////////////////////////////////////////
// CBullCtrl::OnResetState - Reset control's state

void CBullCtrl::OnResetState()
{
    // Resets defaults found in DoPropExchange
    COleControl::OnResetState();
    // TODO: Reset any other control state here.
}

/////////////////////////////////////////////////////////////////
// CBullCtrl::AboutBox - Display an "About" box

```

```

void CBullCtrl::AboutBox()
{
    CDialog dlgAbout(IDD_ABOUTBOX_BULL);
    dlgAbout.DoModal();
}

////////////////////////////////////
// CBullCtrl message handlers

void CBullCtrl::OnStepChanged()
{
    InvalidateRect(NULL);
    SetModifiedFlag();
}

void CBullCtrl::OnLButtonDown(UINT nFlags, CPoint point)
{
    FireClick();
    // actually, base class fires this event for you...
    // COleControl::OnLButtonDown(nFlags, point);
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

- HOME
- ACCOUNT INFO
- SUBSCRIBE
- LOGIN
- SEARCH
- MY ITKNOWLEDGE
- FAQ
- SITEMAP
- CONTACT US

SEARCH

ITKNOWLEDGE

Brief

Full

Advanced Search

Search Tips

BROWSE

BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book
(Publisher: The Coriolis Group)
Author(s): Al Williams
ISBN: 1576101851
Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

The **OnDraw** function must respect the color properties that the control supports. Because the foreground and background colors are stock properties, the code simply calls **GetForeColor** and **GetBackColor** to learn what values to use. These functions return an **OLE_COLOR**. This is similar to, but not exactly the same as, a regular **COLORREF**. Although a **COLORREF** is 32 bits wide, it only uses the lower 24 bits (8 bits each for red, green, and blue). An **OLE_COLOR** uses the top 8 bits to specify the type of color it contains. If the top byte is zero, the bottom 24 bits are a **COLORREF**. That means you can pass an RGB value as an **OLE_COLOR** with no conversion. However, if the top 8 bits are, for instance, 0x80, then the bottom byte is a system color index (the same values you pass to **GetSysColor**). There are more rules, but you don't need to know them. You can just call **TranslateColor** to get the correct RGB value.

The control draws the area outside the target using the ambient background color. The **OnDraw** function retrieves this with the **AmbientBackColor** function. Again, this function returns an **OLE_COLOR** and requires **TranslateColor** to convert it into an RGB value.

The remaining drawing code is just ordinary MFC programming. The **step** variable corresponds to the custom property that defines the size of the concentric circles. Although stock properties take care of themselves, you do have to add a line of code to make a custom property like **Step** work in many cases. If you don't want to save the property in the control's persistent state, you don't have to do anything outside of Class Wizard to define a property. However, if you do want the property to be persistent, you'll need to add a **PX_** function to the **DoPropExchange** function (see Listing 8.6).

DoPropExchange maps properties to persistent storage. There is a **PX_** function for most common types (see Table 8.3). You provide the context (an argument passed into **DoPropExchange**), the external property name, the internal property name, and, optionally, a default value for the property.

Table 8.3 PX_ exchange functions.

Functions
PX_Blob
PX_Bool
PX_Color
PX_Currency
PX_Double
PX_Float
PX_Font

PX_IUnknown
PX_Long
PX_Picture
PX_Short
PX_String
PX_ULONG
PX_UShort
PX_VBFontConvert

Why does **Step** need to be persistent? A programming tool might want to save the control and reload it later. It should look the same when it reloads. Also, if you plan on using the Bull control with the Web browser, it uses persistence to load the values specified in the HTML. For example, consider the following HTML code:

```
<OBJECT CLASSID=clsid:59e8a903-e0c6-11CF-A7B2-  
444553540000 WIDTH=100  
HEIGHT=100>  
Error! Object doesn't exist!  
<PARAM NAME="Step" VALUE=10>  
</OBJECT>  
end
```

Try taking out the **PX_** function and running this code. You'll see that the control will ignore the **Step** value unless you include the **PX_** function.

Another piece of code associated with the **Step** property is the change notification. If the container changes the step size, the control should redraw itself. This is the purpose of the **OnStepChanged** function. This simple function has two lines: **InvalidateRect** causes the control to redraw, and **SetModifiedFlag** marks the control as "dirty"—that is, the control needs to be saved.

One other refinement that is a good idea to implement is property pages. MFC makes these easy to implement, too. Near the top of Listing 8.7 is a **BEGIN_PROPPAGEIDS** macro. The last argument to this macro is the number of property pages defined for the control. This is initially set to one, and the MFC tools create a skeleton dialog box for that property page. You can customize the page (using the resource editor) to handle the **Step** property by inserting an edit control (see Figure 8.10). Then, use Class Wizard to connect the edit control to the property (by using the Member Variables tab). Be sure to include the ActiveX name of the property when asked for it.

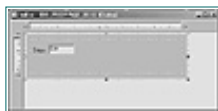


Figure 8.10 Creating the property page.

The color properties are more problematic. How can you easily supply a nice interface for changing colors? Well, you don't have to, because MFC provides one for you. Simply change the number of property pages in the **BEGIN_PROPPAGEIDS** line to two and add a **PROPPAGEID(CLSID_CColorPropPage)** command to the list of property pages. This produces the color dialog you'll see in Figure 8.11.

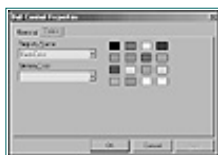


Figure 8.11 The default color page.

MFC also produces a file to represent the default property page (BULLPPG.CPP; see Listing 8.7). This is a relatively straightforward MFC property page with some ActiveX enhancements. Usually, you won't need to make any manual changes to this file. Class Wizard's Member Variables tab does all the work.

Listing 8.7 Bulls property page class.

```

// BullPpg.cpp : Implementation of the CBullPropPage property page class.

#include "stdafx.h"
#include "bull.h"
#include "BullPpg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

IMPLEMENT_DYNCREATE(CBullPropPage, COlePropertyPage)

////////////////////////////////////
// Message map

BEGIN_MESSAGE_MAP(CBullPropPage, COlePropertyPage)
   //{{AFX_MSG_MAP(CBullPropPage)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// Initialize class factory and guid

IMPLEMENT_OLECREATE_EX(CBullPropPage, "BULL.BullPropPage.1",
    0x59e8a904, 0xe0c6, 0x11cf, 0xa7,
    0xb2, 0x44, 0x45, 0x53, 0x54, 0, 0)

////////////////////////////////////
// CBullPropPage::CBullPropPageFactory::UpdateRegistry -
// Adds or removes system registry entries

BOOL
CBullPropPage::CBullPropPageFactory::
    UpdateRegistry(BOOL bRegister)
{
    if (bRegister)
        return AfxOleRegisterPropertyPageClass(
            AfxGetInstanceHandle(),
            m_clsid, IDS_BULL_PPG);
    else
        return AfxOleUnregisterClass(m_clsid, NULL);
}

////////////////////////////////////
// CBullPropPage::CBullPropPage - Constructor

CBullPropPage::CBullPropPage() :
    COlePropertyPage(IDD, IDS_BULL_PPG_CAPTION)
{
   //{{AFX_DATA_INIT(CBullPropPage)
    m_step = 0;
   //}}AFX_DATA_INIT
}

////////////////////////////////////
// CBullPropPage::DoDataExchange - Moves data between page and properties

```

```
void CBullPropPage::DoDataExchange(CDataExchange* pDX)
{
   //{{AFX_DATA_MAP(CBullPropPage)
    DDP_Text(pDX, IDC_EDIT1, m_step, _T("Step") );
    DDX_Text(pDX, IDC_EDIT1, m_step);
    DDV_MinMaxInt(pDX, m_step, 1, 10);
   //}}AFX_DATA_MAP
    DDP_PostProcessing(pDX);
}

//////////////////////////////////////
// CBullPropPage message handlers
```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

The other source file that MFC automatically creates represents the entire control (BULL.CPP; see Listing 8.8). Again, this file is usually exactly what you need, requiring no changes. You can spot the DLL registration calls, and other items that you have to put in by hand, if you are not using MFC.

Listing 8.8 BULL.CPP.

```
// bull.cpp : Implementation of CBullApp

#include "stdafx.h"
#include "bull.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

CBullApp NEAR theApp;

const GUID CDECL BASED_CODE _tlid =
    { 0x59e8a900, 0xe0c6, 0x11cf,
      { 0xa7, 0xb2, 0x44, 0x45, 0x53,
        0x54, 0, 0 } };
const WORD _wVerMajor = 1;
const WORD _wVerMinor = 0;

////////////////////////////////////
// CBullApp::InitInstance - DLL initialization

BOOL CBullApp::InitInstance()
{
    BOOL bInit = COleControlModule::InitInstance();
    if (bInit)
    {
        // TODO: Add your own module initialization code
    }
}
```

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC



```

    return bInit;
}

////////////////////////////////////
// CBullApp::ExitInstance - DLL termination

int CBullApp::ExitInstance()
{
    // TODO: Add your own module termination code here.

    return COleControlModule::ExitInstance();
}

////////////////////////////////////
// DllRegisterServer - Adds entries to the registry

STDAPI DllRegisterServer(void)
{
    AFX_MANAGE_STATE(_afxModuleAddrThis);

    if (!AfxOleRegisterTypeLib(AfxGetInstanceHandle(), _tlid))
        return ResultFromCode(SELFREG_E_TYPELIB);

    if (!COleObjectFactoryEx::UpdateRegistryAll(TRUE))
        return ResultFromCode(SELFREG_E_CLASS);
    return NOERROR;
}

////////////////////////////////////
// DllUnregisterServer - Removes entries from registry

STDAPI DllUnregisterServer(void)
{
    AFX_MANAGE_STATE(_afxModuleAddrThis);

    if (!AfxOleUnregisterTypeLib(_tlid))
        return ResultFromCode(SELFREG_E_TYPELIB);

    if (!COleObjectFactoryEx::UpdateRegistryAll(FALSE))
        return ResultFromCode(SELFREG_E_CLASS);

    return NOERROR;
}

```

Using ActiveX Controls

Incorporating an ActiveX control into a dialog box or **CFormView** is simple. Of course, you need to build ActiveX support into your project. (If you forgot to do that, you can add a call to **AfxEnableControlContainer** in your application's **InitInstance** method and include AFXDISP.H in your STDAFX.H file.)

Assuming your system has the ActiveX control you want to use installed, you select Project|Add to Project|Components and Controls and select the ActiveX control you want to use. The previous version of MFC, by the way, called this the Component Gallery. This tool will insert one or more C++ classes that represent the control (and any ancillary objects it might use). In the simplest case, you really won't need to use these classes directly.

If you are using a dialog (or a **CFormView**), you can select the control from the control palette as you would any standard component (like a button, for example). Once the control is on the dialog, you can use Class Wizard to trap its events or hook up variables to properties. You can also attach a member variable to the entire control so that you can access the properties and methods of the control directly.

You'll find a form view that uses an ActiveX control in Listing 8.9 and Figure 8.12. The ActiveX control in question is a calendar control from Microsoft. The wrapper class that Visual C++ automatically generates is based on **CWnd**. The header file contains inline versions of **Create** that call **CreateControl** instead of the usual **CWnd** code. In addition, the CPP file has simple functions for each property and method. Each method has a corresponding C++ function. The wrapper class also has functions to read and set each property. Pay special attention to the **OnClickCalendar1** function. Not only is it an ActiveX event handler, but it also accesses a property (**ShowDays**).

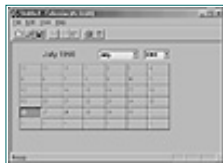


Figure 8.12 The ActiveX calendar.

Using the control with a regular view is only slightly more difficult. Just use **Create** to construct the control in your code as you would any other window type. Of course, Class Wizard isn't very helpful in this case.

Be aware that although calling **Create** on an ActiveX control seems like calling the same call on an ordinary **CWnd**, the underlying code is different. In particular, an ActiveX control only supports the **WS_VISIBLE**, **WS_DISABLED**, **WS_BORDER**, **WS_GROUP**, and **WS_TABSTOP** styles.

Listing 8.9 Using an ActiveX control.

```
// axcalView.cpp : implementation of the CAxcalView class//
#include "stdafx.h"
#include "axcal.h"

#include "axcalDoc.h"
#include "axcalView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CAxcalView

IMPLEMENT_DYNCREATE(CAxcalView, CFormView)

BEGIN_MESSAGE_MAP(CAxcalView, CFormView)
   //{{AFX_MSG_MAP(CAxcalView)
        // NOTE - the Class Wizard will add and remove mapping macros here.
        // DO NOT EDIT what you see in these blocks of generated code!
   //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CFormView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CFormView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CFormView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

```

////////////////////
// CAxcalView construction/destruction

CAxcalView::CAxcalView()
: CFormView(CAxcalView::IDD)
{
    //{AFX_DATA_INIT(CAxcalView)
    // NOTE: the Class Wizard will add member initialization here
    //}}AFX_DATA_INIT
    // TODO: add construction code here

}

CAxcalView::~CAxcalView()
{
}

void CAxcalView::DoDataExchange(CDataExchange* pDX)
{
    CFormView::DoDataExchange(pDX);
    //{AFX_DATA_MAP(CAxcalView)
    DDX_Control(pDX, IDC_CALENDAR1, m_calendar);
    //}}AFX_DATA_MAP
}

BOOL CAxcalView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CFormView::PreCreateWindow(cs);
}

////////////////////
// CAxcalView printing

BOOL CAxcalView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}

void CAxcalView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CAxcalView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}

void CAxcalView::OnPrint(CDC* pDC, CPrintInfo*)
{
    // TODO: add code to print the controls
}

```

```

////////////////////
// CAxcalView diagnostics

#ifdef _DEBUG
void CAxcalView::AssertValid() const
{
    CFormView::AssertValid();
}

void CAxcalView::Dump(CDumpContext& dc) const
{
    CFormView::Dump(dc);
}

CAxcalDoc* CAxcalView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CAxcalDoc)));
    return (CAxcalDoc*)m_pDocument;
}
#endif //_DEBUG

////////////////////
// CAxcalView message handlers
BEGIN_EVENTSINK_MAP(CAxcalView, CFormView)
    //{AFX_EVENTSINK_MAP(CAxcalView)
    ON_EVENT(CAxcalView, IDC_CALENDAR1, -600 /* Click */,
    OnClickCalendar1, VTS_NONE)
    //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()

void CAxcalView::OnClickCalendar1()
{
    m_calendar.SetShowDays(!m_calendar.GetShowDays());
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc.
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Summary

If Microsoft has its way (and it usually does), everything you do will one day use ActiveX. All of the new APIs (like MAPI, or the new shell) use ActiveX interfaces. Don't think of this as a bad thing. ActiveX brings the advantages of object-oriented programming to the binary level—if you have the tools and the skills to use it.

Luckily, MFC makes it fairly easy to use ActiveX. Class Wizard hides most of the details from you one way or another—at a price, of course. ActiveX controls that use MFC require the MFC runtime DLLs (or a static copy of the library). This can add up quickly if you are trying to load controls over the Internet. Of course, if the user already has the DLLs, then you don't have a problem. Other languages (VB, for example) suffer from this same condition.

You can use the ATL (Advanced Template Library) to generate lean, efficient ActiveX controls using Visual C++, but that is very difficult to do. You also forego the MFC class library to get this efficiency. Sure, you can use MFC with ATL, but what's the point? Then you get an ActiveX control that is hard to create and uses the MFC DLLs.

Still, if you can assume that the user has the MFC DLLs, or that you can deliver them quickly, MFC is a very simple way to make and use ActiveX controls. Is it as simple as VB? No, not really. But on the other hand, MFC is much more capable than VB in general.

Although this chapter will help you get started with ActiveX, you should know that ActiveX is a field of study all its own. If you want to go further, try a book

especially about ActiveX (I suggest *Developing ActiveX Web Controls*, also published by The Coriolis Group).

Practical Guide To ActiveX

- Making An MFC Object With An **IDispatch** Interface
- Interpreting CLSIDs, PROGIDs, And The Registry
- Creating ActiveX Controls
- Debugging ActiveX Controls
- Allowing VB Or Web Developers To Initialize Your ActiveX Control
- What Is ATL?
- Adding Property Sheets
- Using ActiveX Controls

ActiveX objects are the closest thing to component software available today. Using ActiveX, you can create reusable components for a variety of languages, including Visual C++, Visual Basic, Delphi, PowerBuilder, and others. You can also use these functional components in your own programs.

Making An MFC Object With An IDispatch Interface

All MFC automation objects (that is, objects with an **IDispatch** interface) derive from **CComdTarget** (or a class that derives from **CComdTarget**). You need only to enable automation when creating the class with Class Wizard.

You can use the ActiveX-specific tabs in Class Wizard to create properties and methods for your object. Remember, automation objects don't support events (ActiveX controls support events).

Interpreting CLSIDs, PROGIDs, And The Registry

ActiveX uses a CLSID (a 128-bit number) to uniquely identify various entities, including objects. These are the very long numbers you see in the system registry that look like this:

```
{00000010-0000-0010-8000-00AA006D2EA4}
```

These numbers are automatically generated by a number of tools (including App Wizard and Class Wizard). You can also generate them using UUIDGEN, a command line tool included with Visual C++. These tools use an algorithm that makes it very unlikely that any two numbers will ever be the same (remember, there are 2^{128} numbers to choose from). If you've ever run across UUIDs (part of the Open System's Foundation Distributed Computing Environment—OSF/DCE), then you have already seen CLSIDs, because UUIDs and CLSIDs are the same thing.

Although MFC usually sets up the registry for you, it is often useful to know a little bit about what it is doing. All the entries for ActiveX are in the HKEY_CLASSES_ROOT registry tree. You'll find a subkey of

HKEY_CLASSES_ROOT with the short name of the ActiveX class (the PROGID). This is a handy, but not necessarily unique, name for the object. Beneath this key, you'll find a subkey named CLSID. This key contains the actual class ID that corresponds to the PROGID.

You can then look up the CLSID under HKEY_CLASSES_ROOT\CLASSES. The entry that corresponds to the CLSID contains a wealth of information about the ActiveX object, including the location of the DLL or EXE that serves the object, the PROGID, the version number, and a great deal of other data.

Creating ActiveX Controls

Creating ActiveX controls with MFC is easy if you use the special wizard provided. You simply answer a few easy questions and the wizard generates the files you need. In particular, it generates an object derived from **COleControl**. Whatever you draw in this object's **OnDraw** member becomes the appearance of your ActiveX control.

Of course, ActiveX controls have properties, methods, and events. You can add all of these by using Class Wizard. Class Wizard allows you to treat ordinary functions in your code as methods that external programs can call. When external programs read or set properties, you can map the action to a variable in your program or into a pair of functions. The advantage to using functions is that you can convert or validate data as programs read and write it. You can also calculate answers on the fly. Finally, adding an event named **Phasers** (for example) causes the wizard to write a function by the name of **FirePhasers**. You call this function whenever you want to raise the event for the external program.

Debugging ActiveX Controls

A handy way to debug ordinary ActiveX controls is to use the supplied test container (TSTCON32). This container allows you to insert controls, set properties, call methods, and log events. You can also manipulate ambient properties and load and save properties to a file.

The test container is very helpful when you want to debug a control but you don't want to write a special container for it. Of course, you'll also want to use ordinary Visual C++ debugging to set breakpoints, examine variables, and so on.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Allowing VB Or Web Developers To Initialize Your ActiveX Control

It isn't uncommon for an ActiveX control to work properly in the test container as you manually set properties but not work with the Web browser, Visual Basic, or another general-purpose container. The symptom is that the properties set at design-time (or in the HTML file) are not set when the program runs. Invariably, this is a result of not making properties persistent. To do this, you need to fill in the control's property map with **PX_** functions (see Table 8.3). The property map is in a function named **DoPropExchange**.

Why is this a problem? Programs like Visual Basic allow the programmer to set properties for an object. However, at runtime, the environment stores all of the design-time properties into a stream and loads them into the object. That means that if your code doesn't exchange persistent properties, it can't initialize to the correct values. Simply add the **PX_** functions and your control should work fine.

What Is ATL?

ATL, or the Advanced Template Library, is a set of C++ templates that ships with Visual C++. ATL gives you tools that help you build ActiveX interfaces from scratch. If you build the right interfaces, you can duplicate the behavior of any ActiveX object, including a full-blown control. However, ATL adds very little overhead for its own code. The size of the control depends mainly on the size of your code.

Contrast this to MFC, where any control you write requires the entire MFC library. Of course, if you use the DLL library and the user's machine already has the DLLs, that isn't a big problem. But suppose an Internet user has to download your control and the giant MFC DLLs. Then your control is very large. ATL controls don't require any libraries (unless, of course, you use them in your code).

That means that you can't use MFC in an ATL control unless you want your control to require the MFC libraries. That is possible, of course, but it doesn't make much sense. If you are going to use MFC, go ahead and use the simplified ActiveX model that MFC provides. If you need a lean control, you'll have to rough it with ATL alone.

Adding Property Sheets

If you generate an ActiveX control with the wizard, it produces a default property page for you. You can find the template in your resources, customize it, and use Class Wizard to map member variables as you would for any dialog.

You can also add standard color and font property boxes by adding one to the number of property pages in the **BEGIN_PROPPAGEIDS** macro and adding **PROPPAGEID(CLSID_CColorPropPage)** to the list of property pages for color or **PROPPAGEID(CLSID_CFontPropPage)** for a font property page.

Using ActiveX Controls

MFC allows you to import an ActiveX control by inserting it into your project (use the Project|Add to Project|Components and Controls menu). Once you do this, you'll have C++ objects in your program that represent the control and any objects it requires. These C++ objects stand in directly for the control, and you can insert them into dialogs and **CFormView** templates.

The wrapper object that MFC creates will have functions corresponding to each method. It will also have a pair of functions to read and write each property. You can handle events from the control by mapping them with Class Wizard.

If you want to create a control without a dialog template, just call the wrapper class **Create** function. This is exactly what you would call to create an ordinary **CWnd** object. However, this **Create** function actually calls **CreateControl** for you and instantiates the control.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Chapter 9

MFC And The Internet

Microsoft has entered the Internet race with vigor. Virtually every product that Microsoft offers has been designed for use on the Internet, and MFC is no exception. Although the MFC socket and ISAPI extensions have their uses, you may prefer to use ActiveX for simple data transfers.

Programmer's Notes...

In the movies, it's always easy to tell the good guys from the bad guys. Unfortunately, it isn't the same in real life. Motives aren't always clear with real people in real situations. Even inanimate objects aren't always simple to figure out.

Alfred Nobel felt so guilty after he invented dynamite that he started the Nobel peace prize. He was afraid that he had assured the destruction of the world. In retrospect, dynamite is probably more useful than it is dangerous. Like most tools, dynamite can be used for good things or for evil purposes.

There are many examples of tools that can be used for good and bad purposes. A knife has many good uses, but when misused, it can be harmful or even fatal. Television, although not a traditional tool, has a similar duality. TV can be a great tool when used to educate and inform, but it may also be used to propagate misinformation. When TV was in its infancy, people disagreed over its relative merits; some thought that TV would change the world for the

better, and others believed that it would destroy our society. A few people figured that TV was just a fad. TV did change the way in which we live, but was it for better or worse? It's a difficult question, isn't it? Besides, early TV only barely resembles TV today.

In just a few short years, the Internet has transformed the computer world. Sure, the Net has been around for a long time, but its explosion in popularity has been recent. And I do mean explosion. Even my mother surfs the Web! When you watch television commercials, it seems that every ad has a URL attached to it. You have to wonder who rushes to their computer to check out the Web site for razor blades or antacids.

Some people think that the Internet is evil, full of stalkers, child molesters, and killers. Others think that it will create a new utopia. I think the truth is somewhere in between. Just like most tools, people can use the Internet for good or ill. The bulk of the Web sites that I see are not particularly good or evil—I'm not sure how I feel about knowing the name of every record album someone owns. On the other hand, there are sites that help find missing kids and support cancer victims. There are also a few sites that most people would agree don't have wholesome goals (or audiences).

Love it or hate it, the Internet is here to stay. Just as early television was very different from today's television, tomorrow's Internet will be very different from today's. But just as television persisted (despite many predictions to the contrary), so will the Internet survive.

For a while, Microsoft appeared to be uninterested in the Internet. However, once Microsoft got hooked on the Internet's merits, they were in with a vengeance. Now, everything Microsoft does seems to factor into the Internet—and MFC is no exception. Recent versions of MFC include support for the Internet in various ways.

An Internet Primer

Before you dive into MFC's Internet support, you need to look at how the Internet works under the hood. Although most people know how to use the Internet, programmers have to understand how it works, which is a different proposition. Luckily, with MFC, you don't have to know as much about how it works as you used to. But you still need to understand a bit about why things work on the Net.

TCP/IP

It is common to think of networking software as a stack of layers. At the top layer is the user, and at the bottom layer is your network interface card (NIC). In between, there are multiple layers that handle most of the work. The advantage to this scheme is that each layer only needs to know how to talk to the layer below it and the layer above it. For example, the user layer probably doesn't care which NIC you are using.

TCP/IP (which stands for Transport Control Protocol/Internetworking Protocol) are low-level layers that define how Internet programs communicate.

IP defines very low-level protocols for addressing (an IP address often looks like a dotted set of numbers, such as 255.0.0.0). TCP handles some higher-level constructs, such as error control.

From an MFC programmer's point of view, you don't need to know much about TCP/IP directly. However, when you use sockets, you'll want to understand some of the TCP/IP nomenclature found in the socket documentation.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Sockets

Sockets originated at Berkeley as part of their Unix operating system. However, sockets worked so well that they became somewhat of a standard. Windows supports sockets via WinSock. WinSock will compile practically any properly written Unix socket code. By properly written, I mean code that avoids the use of “magic numbers.” For instance, because many Unix socket programmers know that an invalid socket is 0, they use the number 0. They should use **INVALID_SOCKET** from the header file (WINSOCK.H or sys/sockets.h). Under Windows, an invalid socket is -1, not 0.

In addition to these small pitfalls, Windows adds some socket functions that begin with the letters **WSA_**. These functions help you write programs under Windows, but they are not portable to other platforms.

Sockets come in several flavors. You can use sockets with no connection to pull in data from the network and send data out. You have no assurance that you’ll see all of the data, or that all of your data will get to where it’s going. You also can’t be sure that the data will appear in the same order in which you sent it. This is useful when the upper layers of the stack are performing error detection, correction, and sequencing. These sockets are often known as datagram sockets.

Another type of socket forms a connection between two computers. These sockets assure correct delivery of data (at the expense of some overhead, of course). One computer plays the role of a server and creates a socket with a port number that both computers agree on. Common servers (Web servers, for instance) have well-known port numbers that everyone uses. Here’s what the

server does:

- Creates a socket on the well-known port number
- Waits for a connection
- When a connection request occurs, the server accepts it by creating a different socket (with an arbitrary port number) to handle the connection

Using a different socket leaves the original one free to handle more connections. You'll see more about this when you use MFC to write a socket-based server later in this chapter.

Clients create a socket on any port number. Then they connect to the server using the server's address and the port number.

MFC programmers won't often use sockets directly because there are nice wrapper classes that represent sockets. At the highest level, these objects look like **CArchive** objects (the same ones you use to save and load files).

Protocols

There are many protocols that implement standard ways to do things with a socket. Sockets by themselves just provide a way to ship bytes between computers. Protocols allow you to do something useful with those bytes. Here are the major protocols:

- FTP (File Transfer Protocol)—a special protocol designed to transfer files between computers.
- Telnet—allows a remote user to log in and use a computer as if from a terminal.
- NNTP (Network News Transfer Protocol)—allows computers to handle news (as in the news groups popular with USENET).
- POP3/IMAP—two protocols used for electronic mail. IMAP is newer and more capable, but the majority of mail servers and clients still only support POP3.
- HTTP (Hypertext Transfer Protocol)—the protocol behind the World Wide Web. HTTP allows clients to request data from an HTTP server and send data to that same server. HTTP is flexible and open-ended.

Inside HTTP And URLs

Because HTTP is the business end of the Web, you should understand it before attempting to write any Web-based software. Even advanced Web page authors need to understand the protocol.

Here's the idea: A client Web browser issues a request using HTTP. This request includes a uniform resource locator (URL) that identifies the machine and the document. The document might be a text file, a binary file, or a Web page (a HTML document). It also might be a program of some sort that the client wishes the server to run.

Along with the request, the client can send data in several ways to the server. For example, when you fill out a form on the Web and click on the Submit

button, you are passing data from your machine to the server. There are other ways you can send data, too.

When the client requests a program to run, it is usually a server-side script (written in VBScript or JavaScript), or it can be a Common Gateway Interface (CGI) program. These programs can process the data from the client as if it were normal input. The program's output becomes the data sent to the client. Although you can write a CGI program in nearly any language, many people use a scripting language like Perl. C and C++ are also popular choices.

The URL is the key to understanding HTTP. Here is a complete URL to my Web site:

<http://www.al-williams.com:80/awc/index.html?MAINT=TRUE&ID=X10SE>.

This probably looks a bit different than most URLs you see. That's because I threw in several optional parts that you don't see very often. Let's look at each field in the example URL:

- [http](#):—The [http](#): specification says that you are making an HTTP request. Other valid types include FTP, TELNET, or FILE (local file).
- [www.al-williams.com](#):—This identifies my server. In particular, there is a machine named [www](#) at the [al-williams.com](#) domain.
- [80](#):—The 80 after the server name specifies the port (or socket number) to use. As it turns out, 80 is a well-known socket number used by all ordinary HTTP servers. However, some sites run multiple Web sites from one machine by assigning different port numbers to each server.
- [awc](#):—This is a directory name, much as you would expect to see in an ordinary file name. Notice that the directory is not usually relative to the root directory of the machine. Instead, the Web server equates this name to some actual directory on the host machine. For example, in this case, [awc](#) is really `/users/home/alw/web`.
- [index.html](#):—The document you want to load in this case is `index.html`.
- [MAINT=TRUE&ID=X10SE](#):—These interesting-looking assignments set two query variables that the server can read. If the strings contain special characters (& or a blank, for example) they must be encoded using a percent sign and the hex equivalent of the character. For example, if you want to set **GREETING** to "Hello Al", you have to use a query string like this: `GREETING=Hello%20Al`.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Of course, all you really needed was:

<http://www.al-williams.com/awc/index.html>. However, the optional parts become more important when you want to write programs that extract data from the client. You should notice that the query string is not where form data appears. The browser sends form data separately, not as part of the URL.

Another important part to the HTTP protocol is the HTTP headers. Headers are just variables (not unlike query string variables) that tell the receiving computer information about the request or reply that goes with the header. These headers tell the client information such as the type of data (HTML, GIF, JPEG, and others), the last modified date of a document, or the length of the form data attached.

ISAPI

Instead of CGI, some servers (including Microsoft's Internet Information Server, or IIS) prefer that you place your programs in a DLL that conforms to the ISAPI specification. This is much more efficient than writing a normal CGI program. ISAPI programs can filter data as the server sends it, or they can create data to send. MFC has full support for both forms of ISAPI.

ActiveX And Java

Two major Internet players that receive a great deal of press are ActiveX and Java. Whereas these two technologies often have similar goals, they have distinctly different ways of achieving them.

As you read in Chapter 8, ActiveX is a standard that allows any language to

provide objects to other programs. Java is actually an object-oriented programming language that is based (somewhat) on C++. A Web browser or server can provide objects that allow other ActiveX objects to interact with it. It can also provide a Java runtime system that allows Java programs to interact with it.

Both Java and ActiveX have strong and weak points. Microsoft’s Visual J++ product supports Java. Although a Java program can use code created in Visual C++, that defeats most of the good points to Java.

MFC has complete support for ActiveX that simplifies ActiveX programming, as you saw in Chapter 8. You can use ActiveX controls with Internet Explorer (and some other browsers, if you have the right software). You can also use ActiveX objects with IIS 3.0 and above. For example, you could write an ActiveX object that manages a database on the server and allows users to view records from their Web browser (in fact, Microsoft supplies such a component with IIS).

MFC Sockets

MFC’s easiest-to-use socket class is the **CSocket** class. This class wraps up all the messy details inherent in using sockets. You don’t need to know about network byte ordering, host name resolution, or address families. Hosts and clients use **CSockets** differently (see Tables 9.1 and 9.2). If you want to act as a client, you need only to create the **CSocket** and then call **Connect**, passing the host name and port number.

Table 9.1 Using CSocket as a client.

Step	Code	Description
1	CSocket socket;	Create an empty, uninitialized socket
2	socket.Create();	Initialize socket; use any port
3	socket.Connect(“ahost”,100);	Connect to host “ahost” on port 100

Table 9.2 Using CSocket as a host.

Step	Code	Description
1	CSocket hostsocket;	Create a socket to listen with
2	hostsocket.Create(100);	Initialize socket and use port 100
3	hostsocket.Listen();	Open up for connections
4	CSocket socket;	Create a socket to use for connection
5	hostsocket.Accept(socket);	Handle incoming connection with socket

Servers are slightly more complicated. First, you create the socket with the

port number you want. Then you call **Listen**. Finally, you call **Accept** to wait for a connection. There's one catch: You pass **Accept** an uninitialized **CSocket** object. This is the object that you use to carry on the conversation with the client.

Why do you need two sockets? The answer lies in the way most servers operate. Consider a Web server. By convention, the server listens on port 80. That way, when a client wishes to connect, it knows to use port 80. But what if it really connected with port 80? Then other clients couldn't connect until the first client closed its connection, which wouldn't be a good idea. So, servers listen using a socket with a well-known port number (like 80). When a client tries to connect to the port, the server actually redirects the request to some other random port and assigns a new socket. This leaves the well-known socket free to make more connections.

At this point, you can use a pair of **CSockets** directly to pass data back and forth between the two computers (using the **Send** and **Receive** member functions). These sockets are reliable. Assuming the network is up, the data you put in one socket will appear at the other socket, correctly and in the proper order. You can also create connectionless sockets, but then you can't be sure the data will appear without errors and in the correct order. Why use sockets that aren't reliable? Perhaps you are writing a program that has its own protocol for correcting errors and ordering data. In that case, duplicating the effort at the socket level is a waste and an unreliable socket is more efficient.

Although you can use the sockets directly at this point, all they will do is exchange raw data. What happens if you want to pass C++ objects from one side of the connection to the other? MFC allows you to serialize objects across a socket connection. Just as you serialize objects to a file and load them later, you can write your objects across the network and reconstitute them on the other side.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Using Archives With CSocket

The trick to transporting objects across the network is to attach your **CSocket** object to a **CSocketFile**. By itself, this object isn't very useful. However, you can attach the **CSocketFile** to a **CArchive** object. Then you can serialize objects using the **CArchive** object. You'll find the exact steps in Table 9.3.

Table 9.3 Connecting a CArchive and a CSocket.

Step	Code	Description
1	<code>CSocketFile file(&socket);</code>	Create CSocketFile connected to socket
2	<code>CArchive input(&file,CArchive::load);</code>	Attach socket to inbound archive
3	<code>CArchive output(&file,CArchive::store);</code>	Attach socket to outbound archive

Remember, **CArchive** objects can read objects or write objects, but not both at the same time. However, it is perfectly permissible to create two **CArchive** objects that refer to the same socket. Then you can read objects from one archive and write them to the other.

One other thing: You can't use unreliable sockets in a **CArchive**. The **CArchive** object treats the socket as a file, and files are reliable.

Going Deeper: CAsyncSocket

If you are familiar with socket programming, you might not like MFC doing so much work for you. You can use **CAsyncSocket** instead (see your online help for details about **CAsyncSocket**). This class (which is the base class for **CSocket**, by the way) allows you direct access to sockets. However, if you use **CAsyncSocket**, you'll have to do a lot more work. For example, using a **CAsyncSocket** requires that you order your bytes in network order. Also, you can't use these sockets in archives. In nearly every case, you'll want to use **CSocket** instead of **CAsyncSocket**.

Still, there are times when you'll want to use **CAsyncSocket**. In particular, **CSocket** can sometimes block and cause your program to apparently lock up while waiting for a network event to call.

Blocking Calls

One big difference between using a file-based archive and one that uses sockets is the potential for latency. If you read an object from a file, it is either in the file or it isn't. However, with a socket, you can't be sure if the data is available. If you try to read an object, your program may come to an abrupt halt until the object is available.

What's the answer? You could make these blocking calls in a thread. However, you can, in some cases, use special virtual functions to take care of this problem.

For example, the **OnReceive** function executes when the socket has data available. Of course, to override this function, you have to derive your own class from **CSocket**. **CAsyncSocket** also provides overrides for connections, but you can't use these with **CSocket**. **CSocket** connections either fail or succeed. If the connection fails, it is up to you to try again later.

The Example

Now that you know about **CSockets** and how to attach them to **CArchives**, think of how this can apply to a real program. I decided to try my hand at writing a simple game (see Figure 9.1). The game pits two army commanders against one another. In the first phase of the game, you place your pieces (for instance, a tank or a missile carrier) on the board. Once the board is full, one player enters the host mode and the other connects to the host. (To test it out, you can run both sides on one machine). Next, the players take turns bombing the board with the mouse. The computer reports all hits and misses.



Figure 9.1 The battlefield game.

It is easy to create a class that contains information to send across the network (see Table 9.4). You can easily write a serialize function for this simple packet. Although it isn't strictly necessary in this case, you should derive classes you plan to serialize from **CObject**.

Table 9.4 CPacket.

Member	Description
cmd	Integer describing command (CMD_CLICK, CMD_REPLY, CMD_RESIGN)
x	X coordinate (not used for CMD_RESIGN)
y	Y coordinate (not used for CMD_RESIGN)
data	Character representing state of cell at (x,y) (not used for CMD_CLICK or CMD_RESIGN)
Serialize	Function writes packet to CArchive

In the case of battlefield, communication follows a strict protocol. The host sends a **CMD_CLICK** packet. The client then replies with a **CMD_REPLY**. Then it is the client's turn to send a **CMD_CLICK** and receive a **CMD_REPLY** packet. Instead of a **CMD_CLICK** packet, the program can send a **CMD_RESIGN** command to end the game.

Having the packet, of course, isn't enough to play the game. You also need a user interface. That shouldn't be a big deal, but blocking network calls can require a bit of special attention.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

[Previous](#) [Table of Contents](#) [Next](#)

The Basic Framework

The program itself is an ordinary MFC document/view program that uses **CScrollView**. Nothing special there. The document object handles most of the work (see Listing 9.1). It contains a three-dimensional array (**grid**). This is really two 10×10 two-dimensional arrays. The first one contains the local computer's board. The other 2D array holds what you know about your opponent's board.

Listing 9.1 Battlefield document object.

```
// Doc.cpp : implementation of the CBfieldDoc class
// Battlefield document -- most real work is here
```

```
#include "stdafx.h"
#include "bfield.h"
#include "insert.h"
#include "bsocket.h"
#include "Doc.h"
#include "ConnDlg.h"
#include "HostDlg.h"
#include "packet.h"
#include "mainfrm.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

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

```

// CBfieldDoc

IMPLEMENT_DYNCREATE(CBfieldDoc, CDocument)
BEGIN_MESSAGE_MAP(CBfieldDoc, CDocument)
    //{AFX_MSG_MAP(CBfieldDoc)
    ON_COMMAND(IDM_CONNECT, OnConnect)
    ON_COMMAND(IDM_HOST, OnHost)
    ON_UPDATE_COMMAND_UI(IDM_CONNECT, OnUpdateConnect)
    ON_UPDATE_COMMAND_UI(ID_TURN, OnUpdateTurn)
    ON_UPDATE_COMMAND_UI(IDM_HOST, OnUpdateHost)
    ON_COMMAND(IDM_RESIGN, OnResign)
    ON_UPDATE_COMMAND_UI(IDM_RESIGN, OnUpdateResign)
    //}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////
// CBfieldDoc construction/destruction

CBfieldDoc::CBfieldDoc()
{
    sockfile=NULL;
    xmit=NULL;
    rcv=NULL;
    MyTurn=FALSE;
}

CBfieldDoc::~CBfieldDoc()
{
    NetClose();
}

BOOL CBfieldDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
    NetClose();
    // empty grids
    for (int i=0;i<2;i++)
        for (int x=0;x<10;x++)
            for (int y=0;y<10;y++)
                grid[i][x][y]='.';

    mode=0;
    placed=0;
    HitCt[0]=0;
    HitCt[1]=0;
    MyTurn=FALSE;
    return TRUE;
}

////////////////////
// CBfieldDoc serialization -- not used
void CBfieldDoc::Serialize(CArchive& ar)
{

```

```

        if (ar.IsStoring())
        {
            // TODO: add storing code here
        }
        else
        {
            // TODO: add loading code here
        }
    }

    //////////////////////////////////////
    // CBfieldDoc diagnostics

#ifdef _DEBUG
void CBfieldDoc::AssertValid() const
{
    CDocument::AssertValid();
}

void CBfieldDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}
#endif // _DEBUG

    //////////////////////////////////////
    // CBfieldDoc commands

    // Get state based on current mode
char CBfieldDoc::GetState(int x,int y)
{
    return grid[mode][x][y];
}

    // Handle mouse click from view
void CBfieldDoc::OnClick(int x,int y)
{
    if (grid[mode][x][y]!='.' || x>9 || y>9)
    {
        AfxMessageBox("Please click a blank square");
        return;
    }
    if (mode==0) // setup
    {
        int q,step=1;
        CInsert dlg;
        if (placed==0x1F)
        {
            AfxMessageBox("You have placed all your pieces");
            return;
        }
    }
    // set up placement dialog
    dlg.m_placed=placed;

```

```

        dlg.m_x=x;
        dlg.m_y=y;
// call dialog
        int res=dlg.DoModal();
        if (res==-1) return;
        if (res=='.') return; // shouldn't happen
// check for piece sticking over edge
        if ((dlg.m_Dir==DIR_WEST && x-dlg.m_Len<-1)||
            (dlg.m_Dir==DIR_EAST && x+dlg.m_Len>10)||
            (dlg.m_Dir==DIR_NORTH && y-dlg.m_Len<-1)||
            (dlg.m_Dir==DIR_SOUTH && y+dlg.m_Len>10))
        {
            AfxMessageBox(
                "The piece must completely fit on the board");
            return;
        }
// Now check for overlapping pieces
        if (dlg.m_Dir==DIR_WEST||dlg.m_Dir==DIR_NORTH) step=-1;
        if (dlg.m_Dir==DIR_EAST||dlg.m_Dir==DIR_WEST)
        {
            for (q=0;q<dlg.m_Len;q++)
                if (grid[0][x+step*q][y]!='.')
                {
                    AfxMessageBox(
                        "The piece overlaps another piece!");
                    return;
                }
// set it up
                for (q=0;q<dlg.m_Len;q++)
                    grid[0][x+step*q][y]=res;
        }
        else
        {
            for (q=0;q<dlg.m_Len;q++)
                if (grid[0][x][y+step*q]!='.')
                {
                    AfxMessageBox(
                        "The piece overlaps another piece!");
                    return;
                }
            for (q=0;q<dlg.m_Len;q++)
                grid[0][x][y+step*q]=res;
        }
// mark it as placed and redraw
        placed|=dlg.m_Piece;
        UpdateAllViews(NULL);
    }
else // not in setup mode -- hostile click!
    {
        CPacket pkt;
        if (!MyTurn) return; // no cheating
// build outbound packet
        pkt.cmd=CMD_CLICK;

```

```

        pkt.x=x;
        pkt.y=y;
        pkt.Serialize(*xmit);
        xmit->Flush(); // make sure to send it
// get reply -- might block for a bit
// but assuming everything works, no big deal
        pkt.Serialize(*rcv);
        if (pkt.cmd!=CMD_REPLY)
        {
            AfxMessageBox("Bad reply");
            return;
        }
// update board
        grid[1][pkt.x][pkt.y]=pkt.data;
// count our hits
        if (pkt.data!='X') HitCt[0]++;
// check for win!
        if (HitCt[0]==17)
        {
            AfxMessageBox("You win!");
            NetClose();
            OnNewDocument();
            UpdateAllViews(NULL);
        }
        UpdateAllViews(NULL);
        MyTurn=FALSE;
    }
}

int CBfieldDoc::GetMode(void)
{
    return mode;
}
// Come here to connect to host
// Host must be ready
void CBfieldDoc::OnConnect()
{
    CConnDlg dlg;
    if (dlg.DoModal()==IDCANCEL) return;
    mode=-1;
    UpdateStatus();
    if (!socket.Create())
    {
        err(socket.GetLastError());
        return;
    }

    if (!socket.Connect(dlg.m_Host,dlg.m_Port))
        if (socket.GetLastError()!=WSAECONNREFUSED)
        {
            mode=0;
            err(socket.GetLastError());
            socket.Close();
        }
    }

```

```

        return;
    }
    else
    {
        // Host not ready yet
        mode=0;
        AfxMessageBox("Host not available");
        socket.Close();
        return;
    }
    sockfile=new CSocketFile(&socket);
    xmit=new CArchive(sockfile,CArchive::store);
    rcv=new CArchive(sockfile,CArchive::load);
    mode=1;
    UpdateStatus();
    UpdateAllViews(NULL);
}

//Here to set up as host
void CBfieldDoc::OnHost()
{
    CHostDlg dlg;
    if (dlg.DoModal()==IDCANCEL) return;
    mode=-1;
    UpdateStatus();
    if (!hostsocket.Create(dlg.m_Port))
    {
        err(GetLastError());
        return;
    }
    if (!hostsocket.Listen())
    {
        err(GetLastError());
        return;
    }
    // wait for connection or error -- blocks
    while (!hostsocket.Accept(socket) &&
        !hostsocket.GetLastError());
    if (hostsocket.GetLastError())
    {
        err(hostsocket.GetLastError());
        return;
    }
    hostsocket.Close(); // no more connections
    sockfile=new CSocketFile(&socket);
    xmit=new CArchive(sockfile,CArchive::store);
    rcv=new CArchive(sockfile,CArchive::load);
    mode=1;
    UpdateStatus();
    UpdateAllViews(NULL);
    MyTurn=TRUE;
    AfxMessageBox("Connected: It is your turn");
}

```

```

// Enable connect menu
void CBfieldDoc::OnUpdateConnect(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(mode==0&&placed==0x1F);
}

// Enable host menu
void CBfieldDoc::OnUpdateHost(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(mode==0&&placed==0x1F);
}

// Resign game
void CBfieldDoc::OnResign()
{
    CPacket pkt;
    if (AfxMessageBox(
        "Do you really want to quit?",MB_YESNO)==IDNO)
        return;
    pkt.cmd=CMD_RESIGN;
    pkt.Serialize(*xmit);
    NetClose();
    OnNewDocument();
    UpdateAllViews(NULL);
}

// Enable Resign
void CBfieldDoc::OnUpdateResign(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(mode==1&&MyTurn);
}

// Very bad general error routine
void CBfieldDoc::err(int ern)
{
    CString s;
    s.Format("%x-%x",ern,GetLastError());
    AfxMessageBox(s);
}

// Close network connection
void CBfieldDoc::NetClose()
{
    if (xmit) delete xmit;
    if (rcv) delete rcv;
    if (sockfile) delete sockfile;
    socket.Close();
    sockfile=NULL;
    xmit=NULL;
    rcv=NULL;
    mode=0;
    placed=0;
}

```

```

}

void CBFldDoc::OnCloseDocument()
{
    NetClose();
    CDocument::OnCloseDocument();
}

// CBattleSocket calls this when data is available
void CBFldDoc::GoAhead(void)
{
    CPacket pkt;
    CPacket outgoing;
    UpdateStatus();
    pkt.Serialize(*rcv); // get packet
    if (pkt.cmd==CMD_RESIGN) // quitter?
    {
        AfxMessageBox("Your opponent resigned");
        NetClose();
        OnNewDocument();
        UpdateAllViews(NULL);
        return;
    }
    if (pkt.cmd!=CMD_CLICK)
    {
        AfxMessageBox("Unexpected error");
        return;
    }
    // prepare reply
    outgoing.data=grid[0][pkt.x][pkt.y];
    outgoing.cmd=CMD_REPLY;
    outgoing.x=pkt.x;
    outgoing.y=pkt.y;
    if (outgoing.data=='.') // convert our . to outbound X
        outgoing.data='X';
    else
        HitCt[1]++; // He hit us!
    // send it
    outgoing.Serialize(*xmit);
    xmit->Flush();
    // check for loss
    if (HitCt[1]==17)
    {
        AfxMessageBox("You lose!");
        NetClose();
        OnNewDocument();
        UpdateAllViews(NULL);
    }
    MyTurn=TRUE;
    CString msg;
    if (outgoing.data!='X')
        msg.Format("Opponent hit: %c. ",outgoing.data);
    msg=msg+"It is your turn";
}

```

```

    AfxMessageBox(msg);
}

BOOL CBfieldDoc::GetTurn()
{
    return MyTurn;
}

// Helper function to update status bar
void CBfieldDoc::UpdateStatus(void)
{
    CMainFrame *fw=(CMainFrame *)AfxGetMainWnd();
    fw->UpdateStatus();
}

// We are ready if all pieces are placed (0x1F)
BOOL CBfieldDoc::GetReady(void)
{
    return placed==0x1F;
}

// This updates the status bar pane
void CBfieldDoc::OnUpdateTurn(CCmdUI* pCmdUI)
{
    int mode=GetMode();
    BOOL turn=GetTurn();
    CString s;
    if (!mode)
    {
        if (placed==0x1F)
            s="Ready to connect";
        else
            s="Setting up...";
    }
    else
    {
        if (mode==-1) s="Connecting...";
        if (turn)
            s="Your turn";
        else
            s="Waiting...";
    }
    CString counts;
    counts.Format("(%d/%d)",HitCt[0],HitCt[1]);
    s+=counts;
    pCmdUI->SetText(s);
    pCmdUI->Enable(TRUE);
}

```

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

The only functions in the view are the ones that draw the board and handle mouse clicks. The **OnDraw** function (see Listing 9.2) queries the document object to learn the contents of each cell. All the logic that colors different squares and spaces them in the scrolling view is in this function.

Listing 9.2 Battlefield view.

```
// View.cpp : implementation of the CBfieldView class
//
```

```
#include "stdafx.h"
#include "bfield.h"
```

```
#include "bsocket.h"
#include "Doc.h"
#include "View.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
////////////////////////////////////
// CBfieldView
```

```
IMPLEMENT_DYNCREATE(CBfieldView, CScrollView)
```

```
BEGIN_MESSAGE_MAP(CBfieldView, CScrollView)
    //{{AFX_MSG_MAP(CBfieldView)
    ON_WM_LBUTTONDOWN()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```

//////////
// CBfieldView construction/destruction

CBfieldView::CBfieldView()
{
    // TODO: add construction code here

}

CBfieldView::~CBfieldView()
{
}

BOOL CBfieldView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CScrollView::PreCreateWindow(cs);
}

//////////
// CBfieldView drawing

void CBfieldView::OnDraw(CDC* pDC)
{
    CBfieldDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    TEXTMETRIC tm;
    COLORREF clr, curcolor;
    pDC->SelectStockObject(ANSI_FIXED_FONT);
    pDC->GetTextMetrics(&tm);
    pDC->SetBkColor(::GetSysColor(COLOR_WINDOW));
    // Change default color based on mode (setup/hits)
    clr=pDoc->GetMode()?RGB(0xFF,0,0):RGB(0,0x80,0);
    for (int x=0;x<10;x++)
        for (int y=0;y<10;y++)
        {
            char piece[2];
            piece[1]='\0';
            piece[0]=pDoc->GetState(x,y);
            curcolor=clr;
            // custom colors
            if (piece[0]=='.') curcolor=RGB(0,0,0);
            if (piece[0]=='X') curcolor=RGB(0,0,0xFF);
            pDC->SetTextColor(curcolor);
            pDC->TextOut(
                x*tm.tmAveCharWidth*3+
                tm.tmAveCharWidth,
                y*tm.tmHeight*3+
                tm.tmAveCharWidth, piece);
        }
}

void CBfieldView::OnInitialUpdate()

```

```

{
    CScrollView::OnInitialUpdate();
    CSize sizeTotal;
    // TODO: calculate the total size of this view
    CDC *dc=GetDC();
    TEXTMETRIC tm;
    dc->SelectStockObject(ANSI_FIXED_FONT);
    dc->GetTextMetrics(&tm);
    // allow for 3 characters per cell plus a border character
    sizeTotal.cx = 31*tm.tmAveCharWidth;
    sizeTotal.cy = 31*tm.tmHeight;
    SetScrollSizes(MM_TEXT, sizeTotal);
    // While this code should work, it has problems with the
    // new style windows
    //     GetParentFrame()->RecalcLayout(TRUE);
    //     ResizeParentToFit();

}

////////////////////////////////////
// CBfieldView diagnostics

#ifdef _DEBUG
void CBfieldView::AssertValid() const
{
    CScrollView::AssertValid();
}

void CBfieldView::Dump(CDumpContext& dc) const
{
    CScrollView::Dump(dc);
}

CBfieldDoc* CBfieldView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CBfieldDoc)));
    return (CBfieldDoc*)m_pDocument;
}
#endif // _DEBUG

////////////////////////////////////
// CBfieldView message handlers

void CBfieldView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CDC *dc;
    CPoint scrollpos;
    int x,y;
    CBfieldDoc *doc=GetDocument();
    // convert to X,Y
    dc=GetDC();
    TEXTMETRIC tm;
    dc->SelectStockObject(ANSI_FIXED_FONT);
    dc->GetTextMetrics(&tm);
    // Account for scrolling
    scrollpos=GetDeviceScrollPosition();

```

```
        point+=scrollpos;  
        x=(point.x-tm.tmAveCharWidth)/(3*tm.tmAveCharWidth);  
        y=(point.y-tm.tmHeight)/(3*tm.tmHeight);  
    // Pass to Document  
        doc->OnClick(x,y);  
}
```

The mouse event handler computes the x- and y-coordinates in terms of the 10×10 playing grid. It then passes this information directly to the document.

The document not only tracks the playing board contents, but it also knows the current mode. Mode 0 is when the player is setting up the board. Later, this becomes mode -1, when the game attempts to connect to its counterpart. When the connection is successful, the game is in progress and the document records this as mode 1.

In mode 0, the document tracks what pieces are already on the board. This serves two purposes: First, you can't place a piece twice, and second, you can't connect until your pieces are all on the board.

When your pieces are in place, you can set up as a host or connect to a host. The **OnHost** function handles the first case. **OnConnect** takes care of the second. You can find both functions in Listing 9.1. If you want to be a host, you need only specify the port number you want to use. Clients must specify the host name and the port. If you want to run both sides on the same machine, use **localhost** as the host name (or use TCP/IP address 127.0.0.1).

While your host waits for a connection, your application appears locked up. The code is at fault here because MFC will return an error if the call would block. However, battlefield simply keeps trying until the call is successful. It wouldn't be hard to put in a timeout mechanism here. It would be a bit more difficult to have a timer event try the connection until it is successful. More difficult still, you could handle the connection sequence in a separate thread. For a simple game like battlefield, I decided that it was not a problem if the program freezes while waiting for a connection.

Once the game is underway, the document keeps track of whose turn it is using the **MyTurn** variable. The host player goes first. Nothing really happens until the host player clicks on an empty square. This causes the document to send a packet to the other player.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

The program resulting from my first attempt at writing battlefield simply tried to read from the socket archive each time it wanted data. Unfortunately, if your computer was waiting for a response, your program would appear to lock up. The advantage of this approach is that you can use the standard **CSocket** class with no modifications. However, it was too annoying to lock up during 50 percent of the game, so I decided to fix things by using a custom socket class.

Adding A Custom Socket

I had to derive a new class from **CSocket** so that I could override the **OnReceive** function. **CSocket** calls this function when data is available. Therefore, you can issue a read and be certain it will succeed without delay from this routine. Class Wizard won't help you create a **CSocket** subclass; you'll have to do it the old fashioned way. You can find the **OnReceive** function (the only significant part to **CBattleSocket**) in Listing 9.3. Notice that I tried to avoid putting any code that knows about the game in **CBattleSocket**. Instead, it simply finds the current document and calls the document's **GoAhead** function. This function reads the data and processes it.

Listing 9.3 CBattleSocket.

```
// Battlefield Socket
// Simple override of CSocket
// Delegates receive to document->GoAhead function
#include "stdafx.h"
#include "bfield.h"
#include "bsocket.h"
#include "mainfrm.h"
```

```
#include "doc.h"

void CBattleSocket::OnReceive(int n)
{
    if (!n)
    {
        CMainFrame *frm=(CMainFrame *)
            AfxGetMainWnd();
        CBfieldDoc *doc=(CBfieldDoc *)
            frm->GetActiveDocument();
        doc->GoAhead();
    }
}
```

Although **CAsyncSocket** provides overrideable functions for other events, **CSocket** doesn't appear to support these. Also, **CSocket** supposedly dispatches **WM_PAINT** messages while blocking, but I could find no evidence that this works. The MFC source code contains code that clearly thinks it is doing something (see **OnMessagePending** in the MFC source file **SOCKCORE.CPP**). However, if you set a breakpoint in **OnMessagePending**, **CSocket** apparently never calls it.

The document object has two member variables for sockets. The first (**socket**) is for the active connection. This has to use **CBattleSocket**. The other socket is **hostsocket**. The document only uses this socket to listen for requests. Therefore, this can remain an ordinary **CSocket**. The program never reads data from it, so there is no need for the special code.

Other Considerations

The only other tricky part is displaying messages in the status bar. Ordinarily, this isn't a big deal. You simply write an **ON_UPDATE_COMMAND_UI** handler in your message map. In the associated function, set the text on the status bar's pane and then enable it. MFC routes the associated message just as it routes menu messages, so that you can put the handler in any message map.

Class Wizard won't write these handlers for the status bar. However, you can fool it into doing so. The key is to realize that Class Wizard will write **ON_UPDATE_COMMAND_UI** handlers for menu items. Simply create a handler for a menu item that you don't already have an **ON_UPDATE_COMMAND_UI** handler for—**IDM_ABOUT**, perhaps. Be sure to select the class you want to handle the status message. When Class Wizard prompts you to name the function **OnUpdateAbout**, pick something else that describes your real intentions (maybe **OnUpdateStatus**). Finally, edit the class that contains the handler. You'll find an entry in the message map that looks like this:

```
ON_UPDATE_COMMAND_UI(IDM_ABOUT, OnUpdateStatus)
```

Change this line to use the ID of your status bar pane instead of **IDM_ABOUT**.

The only problem is that when your socket is blocking, the update messages cease. One possible answer is to update the status bar manually just before you expect the socket to block. That's what battlefield does. The main frame class contains a simple function to force the status bar to repaint. Here's the code:

```
void CMainFrame::UpdateStatus(void)
{
    CWnd *sb=GetMessageBar();
    sb->SendMessage(WM_IDLEUPDATECMDUI);
    sb->UpdateWindow();
}
```

The code finds the **CWnd** that corresponds to the status bar, and sends it a **WM_IDLEUPDATECMDUI** message. It also calls **UpdateWindow**. Of course, **WM_IDLEUPDATECMDUI** is a private MFC message, so you need to include **AFXPRIV.H** to use it.

Socket Wrap Up

So what's the verdict on MFC sockets? That depends. Using **CSocket** is certainly much easier than directly calling the WINSOCK API. Combined with MFC's archive facility, **CSocket** allows you to write some very powerful network programs. However, **CSocket** falls short when it comes to notifying you asynchronously about events. To do that, you'll have to use **CAsyncSocket**. Frankly, using **CAsyncSocket** may be a step above calling WINSOCK, but not by much. If you are an experienced socket programmer, you may want to forego **CAsyncSocket** and just work with sockets directly.

Of course, many of **CSocket**'s shortcomings go away if you apply multithreading (see Chapter 11). By performing socket operations in a separate thread, you can avoid the hassle of polling sockets or missing events.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Higher-Level Protocols

Windows provides a library for interfacing directly with the Internet without using sockets. This library is known as WinInet.

MFC supports two different methods for using WinInet to access the Internet directly. There is a simple model for times when you may only want to read data, and a more complex model if you want to write more sophisticated clients (for example, if you want to submit a form to a Web server). Both models start with **CInternetSession** (see Table 9.5). This object represents your connection to the Internet. By using this object, you don't need to care about the dialer, using proxies, or any of the other mundane details required to connect to the Internet. The user's normal preferences will control the process.

Table 9.5 CInternetSession.

Member	Description
QueryOption	Reads current options (such as asynchronous operation, extended errors, time outs, and cache control)
SetOption	Sets current options
OpenURL	Opens a URL for reading
GetFTPConnection	Opens an FTP connection allowing you to enumerate available files, read, write, and command the remote FTP host

GetHTTPConnection	Opens an HTTP connection that allows you to post data and read files, among other things
GetGopherConnection	Opens a Gopher connection
EnableStatusCallback	Turns on asynchronous notifications
Close	Closes the session
OnStatusCallback	Handles asynchronous notifications, if enabled

Note: The documentation and header files list `ServiceTypeFromHandle` as a member, even though it doesn't actually appear in the `CInternetSession` source code. Using it will cause a link error.

If you simply want to read data from the host, you can call the **CInternetSession's OpenURL** method. This call parses a URL and returns a file object. The file object is always a **CStdioFile** or derived from **CStdioFile**. If the URL starts with **file://**, the object is actually **CStdioFile**. If the URL begins with **http://**, MFC returns a **CHttpFile** object. FTP transfers return a **CInternetFile** object and Gopher files use **CGopherFile**. Each file type is similar to **CStdioFile** (which, in turn, bases itself on **CFile**). However, each object also has its own special calls.

More sophisticated programs might want to interact with the server instead of just pulling data from it. You would need a connection object to manage such an interaction. For HTTP, the object is **CHttpConnection**. You can create the object by calling **CInternetSession::GetHttpConnection**. For FTP and Gopher, you then call **OpenFile**. For HTTP, you call **OpenRequest** followed by **SendRequest**. In either event, you'll now have a **CStdioFile**-derived class, just as before. The difference is that you can manipulate the object to interact with the server. For example, a **CHttpFile** (see Table 9.6) obtained in this way can alter the headers sent to the server (by calling **AddHeaders** between the **OpenRequest** and **SendRequest** calls).

Table 9.6 CHttpFile.

Member	Description
AddRequestHeaders	Add headers to an outbound request
SendRequest	Send request to server
QueryInfo	Read headers from server
QueryInfoStatusCode	Read status code from server
GetVerb	Determine method of transmission
GetObject	Get object name
GetFileURL	Get full URL of file
Close	Close file
Read	Read data
ReadString	Read an entire string
Write	Write data
WriteString	Write an entire string

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

[Previous](#) [Table of Contents](#) [Next](#)

The Link Checker

As an example, consider a program that checks for broken links (Listing 9.4). Just for checking links, it is sufficient to simply read an HTTP file and parse it for further **HREF** and **SRC** links to other files. As a practical matter, you don't want to do this endlessly because you would forever walk links between Web sites (like a Web spider program). Instead, LINKCK only parses the initial file for further links.

Listing 9.4 The link checker.

```
// linkckView.cpp : implementation of the CLinkckView class
//
```

```
#include "stdafx.h"
#include "linkck.h"
#include <afxinet.h>
#include "urlinput.h"
```

```
#include "linkckDoc.h"
#include "linkckView.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
////////////////////////////////////
// CLinkckView
IMPLEMENT_DYNCREATE(CLinkckView, CFormView)
```



```

BOOL CLinkckView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}

void CLinkckView::OnBeginPrinting(CDC* /*pDC*/,
    CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CLinkckView::OnEndPrinting(CDC* /*pDC*/,
    CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}

// Simple code to dump the results out to a printed page
void CLinkckView::OnPrint(CDC* pDC, CPrintInfo*)
{
    int i;
    int y=0, texthi;
    TEXTMETRIC tm;
    pDC->GetTextMetrics(&tm);
    texthi=tm.tmHeight+tm.tmExternalLeading;
    for (i=1;i<m_lb.GetCount();i++)
    {
        CString item;
        m_lb.GetText(i, item);
        pDC->TabbedTextOut(5, y, item, 0, NULL, 0);
        y+=texthi;
    }
}

////////////////////////
// CLinkckView diagnostics

#ifdef _DEBUG
void CLinkckView::AssertValid() const
{
    CFormView::AssertValid();
}
void CLinkckView::Dump(CDumpContext& dc) const
{
    CFormView::Dump(dc);
}

CLinkckDoc* CLinkckView::GetDocument()
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CLinkckDoc)));
    return (CLinkckDoc*)m_pDocument;
}

```

```

}
#endif // _DEBUG

////////////////////////
// CLinkckView message handlers

// Do a scan from user interface
void CLinkckView::OnScan()
{
    CString URL;
    CString err;
    int sel=m_lb.GetCurSel();
    if (sel==LB_ERR||sel==0||m_lb.GetItemData(sel)==-1)
    {
        // prompt if no good selection in the list box
        CUrlInput indlg;
        if (indlg.DoModal()==IDCANCEL) return;
        URL=indlg.m_url;
    }
    else
    {
        m_lb.GetText(sel,URL);
    }
    // clear list box
    m_lb.ResetContent();
    m_lb.AddString("<<New URL>>");
    // do real work
    DoScan(URL);
    // enable menu items
    PrintEnable=TRUE;
}

// This routine really does the work
void CLinkckView::DoScan(CString & URL, BOOL recurse)
{
    CString line;
    int item;
    CInternetSession session("Commando Link Check V1.0");
    CStdioFile *f;
    try
    {
        f=session.OpenURL(URL);
    }
    catch (...)
    {
        f=NULL;
    }
    // apparently the MFC Dlls don't export
    // CHttpFile's runtime class? (see text)
    if (f && f->IsKindOf(RUNTIME_CLASS(CHttpFile)))
    // if (f && !strnicmp(URL,"http",4))
    {
        DWORD stat; // http status

```

```

        CHttpFile *hf=(CHttpFile *)f;
        hf->QueryInfoStatusCode(stat);
// only codes 200-299 are good
        if (stat<200||stat>299)
        {
            f->Close();
            f=NULL;
        }
    }
    if (!f) // some problem
    {
        CString err;
        err=URL;
        err+="\tError!";
        item=m_lb.AddString(err);
        m_lb.SetItemData(item,-1);
        return;
    }
// go!
    item=m_lb.AddString(URL);
    m_lb.SetItemData(item,0);
// if recurse is true, then parse file
    while (recurse && f->ReadString(line))
    {
        int n=0;
        char *p, *e;
        p=(char *)(LPCSTR)line;
        do {
            p=strchr(p,'='); // search for =
            if (p) // backward search for HREF/SRC
            {
                if (toupper(p[-1])!='F' ||
                    toupper(p[-2])!='E' ||
                    toupper(p[-3])!='R' ||
                    toupper(p[-4])!='H')
                {
                    if (toupper(p[-1])!='C' ||
                        toupper(p[-2])!='R' ||
                        toupper(p[-3])!='S')
                    {
                        p++;
                        continue;
                    }
                }
                if (p[1]=='"') // skip quotes
                    p++;
                e=p+1+strcspn(p+1," \t\n>\"#");
                *e='\0';
// don't recurse empty URLs (such as HREF=#anchor)
                if (e!=p+1)
                {
                    CString newURL;
                    ExpandURL(newURL,p+1,URL);
                    DoScan(newURL,FALSE);
                }
            }
        } while (p);
    }

```

```

                                *e=' ';
                                p=e+1;
                                }
                                } while (p && *p);
        } // end while
        f->Close();
    }

void CLinkckView::OnInitialUpdate()
{
    CFormView::OnInitialUpdate();
    GetParentFrame()->RecalcLayout();
    ResizeParentToFit(FALSE);
}

void CLinkckView::ExpandURL(CString &result,
    CString source,CString current)
{
    CString proto,host,path,file,chost,cpath,cproto;
    // break up both URLs
    ParseURL(source,&proto,&host,&path,&file);
    ParseURL(current,&cproto,&chost,&cpath,NULL);
    // copy empty parts from current URL
    if (proto.IsEmpty()) proto=cproto;
    if (host.IsEmpty()) host=chost;
    if (path.IsEmpty())
        path=cpath;
    else if (path[0]!='/'&&path[0]!='\\')&&!cpath.IsEmpty())
        path=cpath+"/"+path;
    result=proto + host ;
    // if path is relative, prepend current path
    if (!path.IsEmpty())
    {
        if (path[0]!='/' && path[0]!='\\')
            result+="/";
        result+= path;
    }
    if (!file.IsEmpty()) result+="/" + file;
}

// Homebrew URL parser AfxParseURL is too lazy to
// decide what is a file and what is a directory, so we
// do it ourselves (sigh)
void CLinkckView::ParseURL(CString url,CString *proto,
    CString *host, CString *path,
    CString *fileplus)
{
    CString _proto, _host, _path, _fileplus;
    int n;
    n=url.Find("://");
    if (n==-1) n=url.Find(":\\\\");
    // get protocol

```

```

        if (n!=-1)
        {
            _proto=url.Left(n+3);
            url=url.Right(url.GetLength()-(n+3));
            n=url.FindOneOf("/\\");
            if (n==-1)
            { // get host
                _host=url;
                url="";
            }
            else
            {
                _host=url.Left(n);
                url=url.Right(url.GetLength()-n);
            }
        }
    // find path or file name
    n=url.ReverseFind('/');
    if (n==-1) n=url.ReverseFind('\\');
    if (n!=-1)
    {
        _fileplus=url.Right(url.GetLength()-n-1);
        _path=n==-1?url.Left(n);
        if (!_path.IsEmpty()&&(_path[0]=='/'||
            _path[0]=='\\'))
            _path=_path.Right(_path.GetLength()-1);
    }
    else
    {
        _fileplus=url;
    }
    // Special case heuristic
    // if host name and file provided, but no path, then file
    // is probably a path if it doesn't contain a period
    if (!_host.IsEmpty() && _path.IsEmpty() &&
        !_fileplus.IsEmpty() && _fileplus.Find('.')==-1)
    {
        _path="/" + _fileplus;
        _fileplus="";
    }
    if (proto) *proto=_proto;
    if (host) *host=_host;
    if (path) *path=_path;
    if (fileplus) *fileplus=_fileplus;
}

void CLinkckView::OnEndPrintPreview(CDC* pDC, CPrintInfo* pInfo,
    POINT point, CPreviewView* pView)
{
    AfxGetMainWnd()->ShowWindow(SW_NORMAL);
    CFormView::OnEndPrintPreview(pDC, pInfo, point, pView);
}

```

```
void CLinkckView::OnUpdateFilePrint(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(PrintEnable);
}
```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

The parsing is somewhat tricky and not terribly robust. For example, the link checker will pick up any **SRC=** or **HREF=** line, even if it's in a comment. Also, there are a lot of special cases to handle relative links, links with anchors, and all the other myriad ways to express URLs. I even try to allow for forward slashes or backslashes. Although my algorithm handles all the cases I could think of, I'm sure someone will find a URL that breaks it.

The key parts to the parser are the **ExpandURL** and **ParseURL** functions. **ExpandURL** takes two URLs: the URL you want to expand and the current URL. It then calls **ParseURL** twice to break the URL down into its constituent parts. Finally, the routine completes the URL by filling in any missing pieces (using the original URL as a guide). These routines are very general-purpose, and you might find a use for them in your own code. MFC also provides **AfxParseURL**, but because the output format isn't what I needed, I decided to write my own version.

The **OnScan** routine starts the show when you select the scan button, a menu item, or double-click on a selection in the list box. If the list box has a selection, and the selection's extra data is zero, the **OnScan** uses the list box's entry as the URL to scan. Otherwise, the routine displays a simple dialog box to request a URL for scanning. The user must enter a complete URL.

The real work occurs in **DoScan**. This code opens the given URL (using **OpenURL**), checks for a valid response, and, if required, parses the file for links. Note that the code assumes you are scanning an HTML file. If you point at an FTP file, for example, **DoScan** still attempts to find links in the file.

That brings up another interesting problem. Checking to see if the file exists is tricky. Suppose you request a file from a legitimate server, but the file doesn't

exist. You would expect an error or an exception, right? Wrong. **OpenURL** succeeds, but the contents of the **CHttpFile** is an error message from the server! To correctly check the result, you'd have to examine the return code. Normal return codes are in the range of 200 to 299. Missing files usually show a return of 404.

This should be easy, right? **CHttpFile** has a member function named **QueryInfoStatus-Code** to find the return code. The problem is that **OpenURL** returns a pointer to a **CStdioFile** that may (or may not) point to a **CHttpFile**. Casting is dangerous because you don't know for sure what type of file object **OpenURL** returns.

This is the perfect job for **IsKindOf**. Because **CFile** derives from **CObject**, why not use **IsKindOf** to detect a **CHttpFile**? This sounds good in theory, but in practice, it looks as though Microsoft forgot to export the class information for **CHttpFile** from the MFC DLLs. So, if you write

```
if (f && f->IsKindOf(RUNTIME_CLASS(CHttpFile))) . . .
```

the code won't even link with the DLL version of the MFC library. If you turn on the static library option (under Project|Settings), everything works fine.

I left the code the way I wanted it to be and assumed the project would link with the static library. However, if you prefer, you can use the commented-out code as a workaround. The only way to get a **CHttpFile** is if the URL begins with "http", so the workaround code looks like this:

```
if (f && !strnicmp(URL, "http", 4)) . . .
```

This has the same effect in practice as the if statement above that uses **IsKindOf**.

Beyond that, the program isn't very interesting. **DoScan** adds all URLs it finds to the list box. Good items appear by themselves and have an extra data word of zero. Bad URLs have a tab and the word "Error!" appended to them. They also have an extra word of -1. This allows **OnScan** to identify the URL as bad so it won't try to scan it later.

The parsing is a mishmash of **CString** code and good old-fashioned C library string calls. Most of the **CString** manipulation functions refuse to work on portions of strings, and that makes it easier to use the ordinary calls in certain cases.

Other Ideas

If you want to submit data to the server, write to an FTP host, or do anything more sophisticated than simply reading, you'll need to resort to the more sophisticated usage of the WinInet wrappers. Still, even just reading data can be useful. Here are some ideas:

- A Web spider to build a local search database on your favorite keywords
- A program that monitors URLs and alerts you when they change

- For security, you could write a program that periodically scans FTP files, checksums them, and informs you if there were unauthorized changes

ActiveX Internet Support

Another way to access the Internet is with an ActiveX control. If you opt for this approach, you have several choices. First, Microsoft's Internet Explorer is really just an ActiveX control (Shell.Explorer in SHDOCVW.DLL). If you examine it with an object browser or look in the ActiveX documentation, you can learn to use it to embed a browser right in your application.

Microsoft also provides an ActiveX control that you can use, called the Microsoft Internet Transfer Control. It can read Web pages (and their headers), submit data to Web servers (as a form submits data), and do FTP transactions. When the control submits data, it can use standard HTTP or a secure protocol (HTTPS).

If you want to use this control, you'll need to add the component to your project like any other ActiveX control (it is in the MSINET.OCX file). If you just want basic transfers, you'll find that the control is very easy to use. If you need to submit data or monitor the transfer's progress, you'll still find it relatively easy, but there is a bit more to do.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

The Transfer Control

The transfer control offers you a vast array of properties, a few methods, and only one event. For many simple uses, you won't use the event; you'll only use one or two key methods.

Table 9.7 shows the members the control exposes to the outside world. If you only want to read a URL, **OpenURL** is all you will really need. This method takes a URL as an argument; you can also specify if you want the data returned as a string or as a byte array. The default is to return the data as a string. This call literally does all the work.

Many of the members you see in Table 9.7 aren't used very often. For example, **hInternet** is only useful if you plan to make WinInet calls directly. You usually don't want to specify if the control should connect directly or via a proxy, either. Instead, you'll want it to use the default for the current system. Luckily, that is exactly what the control will do if you don't change the **AccessType** property.

Another oddity about some properties is their relationship with the **URL** property. In particular, the **Document**, **Protocol**, **RemoteHost**, and **RemotePort** properties change to match what you put in the **URL** property. Conversely, if you change those properties, the control alters the **URL** property. The **OpenURL** and **Execute** methods also change these properties. In practice, you won't really use the properties as much as you'll make calls to **OpenURL** and, perhaps, **Execute**.

You can access HTTP headers using the **GetHeader** method (see Table 9.8 for typical header values). If this method receives a header name as an argument,

it returns the value of that header. If you don't pass any arguments, the function returns all the available headers. You can read the HTTP response code from the **ResponseCode** property (see Tables 9.7 and 9.9).

Table 9.7 Internet transfer control members.

Member	Type	Description
AccessType	Property	Indicates if Internet connection is direct or via a proxy
Document	Property	Sets the file name to be used with the Execute method
hInternet	Property	The underlying HINTERNET handle for the connection (used to directly call WININET functions only)
Password	Property	Users password for login, if any (see UserName)
Protocol	Property	Selects HTTP, HTTPS, or FTP protocol
Proxy	Property	Returns or sets the name of the proxy used to access the Internet (see also AccessType)
RemoteHost	Property	Selects remote host computer
RemotePort	Property	Selects port to connect to on host computer
RequestTimeout	Property	Sets timeout in seconds (0 means no timeout)
ResponseCode	Property	Error code (when state is icError; see Table 9.10)
ResponseInfo	Property	Description of error (when state is icError; see Table 9.10)
StillExecuting	Property	Boolean flag that is True while a transfer is in progress
URL	Property	Current URL; changing this property may change other properties (such as Protocols, RemoteHost, RemotePort, and Document); changing other properties may also affect URL
UserName	Property	The user's name, if any (see Password)
Cancel	Method	Cancel current transaction
Execute	Method	Perform FTP command or HTTP Get/Post
GetChunk	Method	Retrieve data when state is icResponseReceived or icResponseCompleted (see Table 9.10)
GetHeader	Method	Get specific header or all headers
OpenURL	Method	Completely retrieves an HTTP document or FTP file
StateChanged	Event	Indicates a change in the download status (see Table 9.10)

Table 9.8 Typical HTTP headers.

Header	Description
Date	Returns the time and date of the document's transmission; the format of the returned data is Wednesday, 27-April-96 19:34:15 GMT
MIME-version	Returns the MIME protocol version (currently 1.00)
Server	Returns the name of the server
Content-length	Returns the length in bytes of the data
Content-type	Returns the MIME Content-type of the data
Last-modified	Returns the date and time of the document's last modification; the format of the returned data is Wednesday, 27-April-96 19:34:15 GMT

Table 9.9 HTTP response codes.

Value	Keyword	Description
200	OK	Everything is OK
201	Created	Successful POST
202	Accepted	Request accepted, but not completed
203	Partial Information	Information returned may not be complete
204	No Response	Don't display any data; stay in current document
301	Moved	Requested document moved
302	Found	Document redirected
303	Method	Try alternate URL
304	Not Modified	Requested document has not changed
400	Bad Request	Improper client request
401	Unauthorized	Document protected
402	Payment Required	Client needs ChargeTo: header
403	Forbidden	No one can access this document
404	Not Found	Document not found
500	Internal Error	Server blew up
501	Not Implemented	Server doesn't do this
502	Service Overloaded	Too busy to service request
503	Gateway Timeout	A gateway (such as a CGI script) did not respond

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

For HTTP and anonymous FTP, you don't need to bother about the **User** and **Password** properties. However, if you want to connect to a host that requires you to log in, you may need these. When you don't specify a user name or password, the control will use anonymous FTP. Of course, you could also set up anonymous FTP by setting the **UserName** property to anonymous and the **Password** property to the user's email address.

If you want to execute FTP commands or submit data to a Web server, you'll need to use the control's **Execute** method. You can actually use **Execute** instead of **OpenURL**. **Execute** can do everything **OpenURL** does, and then some. However, **Execute** is a bit trickier to use, so you'll only want to resort to it when you can't use **OpenURL**.

The control's sole event, **StateChanged**, passes a number to your code to indicate what is happening (see Table 9.10). If you use **OpenURL**, you won't get many events because the component handles them internally. However, when you use **Execute**, you'll get enough events that you can track every step of the download process. The **icResponseReceived** indicates that some data is available, and you can read it with **GetChunk**. If you prefer to wait until all data is available, you can wait for the **icResponseComplete** code.

Table 9.10 StateChanged constants.

Numeric Value	value	Description
icNone	0	No state to report
icHostResolvingHost	1	The control is looking up the IP address of the specified host computer

icHostResolved	2	The control successfully found the IP address of the specified host computer
icConnecting	3	The control is connecting to the host computer
icConnected	4	The control successfully connected to the host computer
icRequesting	5	The control is sending a request to the host computer
icRequestSent	6	The control successfully sent the request
icReceivingResponse	7	The control is receiving a response from the host computer
icResponseReceived	8	The control successfully received a response from the host computer
icDisconnecting	9	The control is disconnecting from the host computer
icDisconnected	10	The control successfully disconnected from the host computer
icError	11	An error occurred in communicating with the host computer
icResponseCompleted	12	The request has completed and all data has been received

Most of the other members are either self-explanatory or obscure. If you are really trying to write a sophisticated application, you can control almost everything. You can even get the underlying handle that represents the Internet connection and use it, if you like. Otherwise, this is just an ordinary ActiveX control (see Chapter 8 for more about ActiveX).

ISAPI Support

If you want to create active Web pages, the traditional answer is to write a CGI program. A CGI program can accept input (from a form or URL) and send output to a Web browser. For example, a CGI program might accept your name and email address, look up your account in a database, and display your current bill.

If you use Microsoft's IIS server, you can write CGI programs, but there is another way: ISAPI. An ISAPI DLL actually becomes part of the server and is generally more efficient than a classic CGI program. I've written ISAPI DLLs in C and using MFC. Neither way is that difficult. Once, a friend asked me if he could write ISAPI extensions in VB. My first answer was no, because VB can't make traditional DLLs (only ActiveX DLLs).

I fired up a Web search engine, looked around, and found that Microsoft has a sample, OLEISAPI, that allows VB ISAPI. However, judging from the traffic on the Web, it didn't work well. Sure enough, after trying to make it work for two days, we gave up. Besides, OLEISAPI didn't encapsulate ISAPI in an

object-oriented way. One of the advantages to ActiveX is OOP. It also didn't allow you to fully access ISAPI features. A few days later, I finished CBISAPI, an ISAPI module that allows you to write ActiveX ISAPI extensions. Although I wrote it with VB5 in mind, you can use it with any ActiveX-capable language. CBISAPI itself uses MFC to form a bridge between IIS and an ActiveX object.

Avoiding ISAPI

Sometimes it's better to avoid ISAPI completely. If you use Microsoft's IIS, you can do a great deal of work with Active Server Pages (ASP) instead of resorting to CGI or ISAPI.

ASP files allow you to embed VBScript or JavaScript functions in your HTML file that the server interprets and uses to generate the outgoing HTML. By combining this script with ActiveX objects, you can access databases, store user preferences, rotate advertisements, and much more.

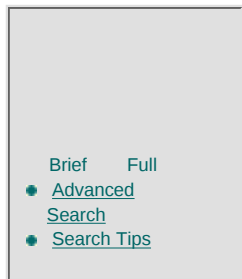
ASP files are well outside the scope of this book. If you'd like to learn more about them, check out *Active Server Pages Black Book*, also published by The Coriolis Group.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97



Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

The Plan

My idea was simple: Write an ISAPI extension DLL that calls an ActiveX server. The DLL passes the server an ActiveX object that it uses to read the HTTP information and manipulate the HTTP output (usually an HTML file). This is a bit odd: The DLL is both an ISAPI DLL and an ActiveX server. It, in turn, serves an object to another ActiveX server (the VB ISAPI extension). In truth, the VB ISAPI extension doesn't really have to be an OLE server, but that is the only kind of DLL VB can make.

Table 9.11 shows the members of the object the VB server uses to interact with IIS. The VB server's SUB may have any name, but it must take an object as an argument:

```
Public Sub VBISAPI(server as Object)
```

```
· · ·  
End Sub
```

Table 9.11 CBISAPI object members.

Member	Type	Description
RetVal	Property	Sets ISAPI return value
StatCode	Property	Sets HTTP response code
Method	Property (R/O)	Method ("GET" or "POST", for example)
QueryString	Property (R/O)	Entire query string
PathInfo	Property (R/O)	Virtual path to script
PathTranslated	Property (R/O)	Translated path to script
Content	Property (R/O)	Data submitted by client
ContentType	Property (R/O)	Type of data in Content property
ContentLength	Property (R/O)	Length of data in Content property
Write	Method	Write a string to the client (terminates on NULL byte)
WriteLine	Method	Write a string terminated by a new line (not a <P>)
WriteByte	Method	Write a single byte to the client
ServerVariable	Method	Retrieves a standard CGI server variable (see Table 9.3)
ServerDoneSession	Method	Indicates extension is complete during asynchronous operations

Redirect	Method	Redirects browser to a different URL
SendUrl	Method	Sends an alternate URL
SendHeaders	Method	Sends HTTP headers
MapURL2Path	Method	Maps a local URL to a full path name

You use the object (**server** in this case) in the VB code by using the members in Table 9.11.

A May-December Marriage

Although VB5 makes it easy to create an ActiveX DLL, it doesn't have a good way of making an ISAPI DLL. Therefore, the main ISAPI DLL uses MFC. (ISERVER.CPP; see Listing 9.5. Please note that due to page width constraints, there are some code lines that begin **DISP_...** shown here broken in two lines, but should be actually one.) Although MFC has special provisions for creating ISAPI DLLs, this server doesn't use them because they add another layer on top of ISAPI. Instead, the DLL is just an ordinary MFC DLL with the correct ISAPI entry points. You can find the supporting DLL code (CBISAPI.CPP and some header files) online.

Listing 9.5 ISERVER.CPP.

```
// IServer.cpp : implementation file
//

#include "stdafx.h"
#include "cbisapi.h"
#include "IServer.h"
#include <malloc.h>
#include <afxconv.h> // BSTR conversions

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CIsapiServer

IMPLEMENT_DYNCREATE(CIsapiServer, CCmdTarget)

CIsapiServer::CIsapiServer()
{
    EnableAutomation();
}

CIsapiServer::~CIsapiServer()
{
}

void CIsapiServer::OnFinalRelease()
{
    CCmdTarget::OnFinalRelease();
}

BEGIN_MESSAGE_MAP(CIsapiServer, CCmdTarget)
    //{AFX_MSG_MAP(CIsapiServer)
    // NOTE - the Class Wizard will add and remove mapping macros here.
    //}AFX_MSG_MAP
END_MESSAGE_MAP()
BEGIN_DISPATCH_MAP(CIsapiServer, CCmdTarget)
    //{AFX_DISPATCH_MAP(CIsapiServer)
```

```

DISP_PROPERTY(CIsapiServer, "RetVal", m_retVal, VT_I4)
DISP_PROPERTY(CIsapiServer, "StatCode", m_statCode, VT_I4)
DISP_PROPERTY_EX(CIsapiServer, "Method", GetMethod, SetNotSupported,
    VT_BSTR)
DISP_PROPERTY_EX(CIsapiServer, "QueryString", GetQueryString,
    SetNotSupported, VT_BSTR)
DISP_PROPERTY_EX(CIsapiServer, "PathInfo", GetPathInfo,
    SetNotSupported, VT_BSTR)
DISP_PROPERTY_EX(CIsapiServer, "PathTranslated", GetPathTranslated,
    SetNotSupported, VT_BSTR)
DISP_PROPERTY_EX(CIsapiServer, "ContentLength", GetContentLength,
    SetNotSupported, VT_I4)
DISP_PROPERTY_EX(CIsapiServer, "Content", GetContent,
    SetNotSupported, VT_BSTR)
DISP_PROPERTY_EX(CIsapiServer, "ContentType", GetContentType,
    SetNotSupported, VT_BSTR)
DISP_FUNCTION(CIsapiServer, "Write", Write, VT_BOOL, VTS_VARIANT)
DISP_FUNCTION(CIsapiServer, "ServerVariable", ServerVariable, VT_BOOL,
    VTS_VARIANT VTS_PVARIANT)
DISP_FUNCTION(CIsapiServer, "WriteLine", WriteLine, VT_BOOL, VTS_VARIANT)
DISP_FUNCTION(CIsapiServer, "WriteByte", WriteByte, VT_BOOL, VTS_VARIANT)
DISP_FUNCTION(CIsapiServer, "ServerDoneSession", ServerDoneSession,
    VT_BOOL, VTS_NONE)
DISP_FUNCTION(CIsapiServer, "Redirect", Redirect, VT_BOOL, VTS_VARIANT)
DISP_FUNCTION(CIsapiServer, "SendURL", SendURL, VT_BOOL, VTS_VARIANT)
DISP_FUNCTION(CIsapiServer, "SendHeaders", SendHeaders, VT_BOOL,
    VTS_VARIANT VTS_VARIANT)
DISP_FUNCTION(CIsapiServer, "MapURL2Path", MapURL2Path, VT_BOOL,
    VTS_PVARIANT)
//}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()

// Note: We add support for IID_IIsapiServer to support typesafe binding
// from VBA. This IID must match the GUID that is attached to the
// dispinterface in the .ODL file.
// Not really used in any meaningful way, but the wiz puts it here.

// {A3B7D305-647C-11D0-A7B2-444553540000}
static const IID IID_IIsapiServer =
{ 0xa3b7d305, 0x647c, 0x11d0, { 0xa7, 0xb2, 0x44, 0x45, 0x53, 0x54, 0x0,
    0x0 } };

BEGIN_INTERFACE_MAP(CIsapiServer, CCmdTarget)
    INTERFACE_PART(CIsapiServer, IID_IIsapiServer, Dispatch)
END_INTERFACE_MAP()

////////////////////////////////////
// CIsapiServer message handlers
// Write to client
BOOL CIsapiServer::Write(const VARIANT FAR& idata)
{
    COleVariant data=idata;
    USES_CONVERSION;
    data.ChangeType(VT_BSTR); // Force to BSTR
    if (data.vt!=VT_BSTR)
        return FALSE;
    char *s=W2A(data.bstrVal); // switch to ANSI
    DWORD siz=strlen(s);
    return ecb->WriteClient(ecb->ConnID,s,&siz,0); // out!
}

```

```

// This fetches a Server Variable into a VARIANT
// Be careful. Since the second argument is a variant
// by reference, the formal argument must really be
// a variant. In other words, NO:
//     dim x as string
//     server.ServerVariable "SCRIPT_NAME",x
// YES:
//     dim x as variant
//     server.ServerVariable "SCRIPT_NAME",x
// Probably should have been a function returning VARIANT, but then
// again...
BOOL CIsapiServer::ServerVariable(const VARIANT FAR& Variable,
                                   VARIANT FAR* Result)
{
    COleVariant var;
    var=Variable;
    var.ChangeType(VT_BSTR);
    if (var.vt!=VT_BSTR) return FALSE;

    USES_CONVERSION;
    char *v=W2A(var.bstrVal);
    CString res;
    DWORD siz=1024;
    BOOL rv;
    rv=ecb->GetServerVariable(ecb->ConnID,v,
        (char *)res.GetBufferSetLength(siz),&siz);
    res.ReleaseBuffer(siz-1);
    VariantClear(Result);
    Result->vt=VT_BSTR;
    Result->bstrVal=res.AllocSysString();
    return rv;
}

// R/O Property -- these all look the same
BSTR CIsapiServer::GetMethod()
{
    CString strResult=ecb->lpszMethod;
    BSTR rv;
    rv=strResult.AllocSysString();
    return rv;
}

// Another R/O Property
BSTR CIsapiServer::GetQueryString()
{
    CString strResult=ecb->lpszQueryString;
    BSTR rv;
    rv=strResult.AllocSysString();
    return rv;
}

// R/O Property
BSTR CIsapiServer::GetPathInfo()
{
    CString strResult=ecb->lpszPathInfo;
    BSTR rv;
    rv=strResult.AllocSysString();
    return rv;
}

```

```

// R/O Property
BSTR CIsapiServer::GetPathTranslated()
{
    CString strResult=ecb->lpszPathTranslated;
    BSTR rv;
    rv=strResult.AllocSysString();
    return rv;
}

// R/O Property
long CIsapiServer::GetContentLength()
{
    return ecb->cbTotalBytes;
}

// R/O Property with a twist
// Apparently, sometimes the server calls the
// extension without having all the content
// data (does this really happen?)
// This function reads it all so it is available
// BTW, the docs say that if the count is
// 0xFFFFFFFF, then MORE than 4G of data
// is forthcoming and you should call ReadClient
// until it is empty.
// NEWS BULLETIN: If you expect 4G or more in
// a request, don't use these functions!
BSTR CIsapiServer::GetContent()
{
    CString strResult;
    char *p=strResult.GetBufferSetLength(ecb->cbTotalBytes);
    // put available bytes in CString
    memcpy(p,ecb->lpbData,ecb->cbAvailable);
    // Read excess
    if (ecb->cbAvailable!=ecb->cbTotalBytes)
    {
        DWORD siz=ecb->cbTotalBytes-ecb->cbAvailable;
        ecb->ReadClient(ecb->ConnID,p+ecb->cbAvailable,&siz);
    }
    strResult.ReleaseBuffer(ecb->cbTotalBytes);
    BSTR rv;
    rv=strResult.AllocSysString();
    return rv;
}

// Another R/O
BSTR CIsapiServer::GetContentType()
{
    CString strResult=ecb->lpszContentType;
    BSTR rv;
    rv=strResult.AllocSysString();
    return rv;
}

// Simple Method to write a line
// Note that HTML doesn't care one
// whit about the \r\n -- it just
// makes the HTML source nicer.
// Use <P> or <BR> to get a newline in HTML.
BOOL CIsapiServer::WriteLine(const VARIANT FAR& idata)

```

```

{
    BOOL rv=Write(idata);
    DWORD siz=2;
    if (rv) rv=ecb->WriteClient(ecb->ConnID, "\r\n", &siz, 0);
    return rv;
}

// Write a byte out
BOOL CIsapiServer::WriteByte(const VARIANT FAR& byte)
{
    COleVariant num=byte;
    num.ChangeType(VT_UI1);
    if (num.vt!=VT_UI1)
        return FALSE;
    char s=num.bVal;
    DWORD siz=1;
    return ecb->WriteClient(ecb->ConnID, &s, &siz, 0);
}

// Wrap ServerSupportFunction Done with Session
BOOL CIsapiServer::ServerDoneSession()
{
    return ecb->ServerSupportFunction(ecb->ConnID,
        HSE_REQ_DONE_WITH_SESSION, NULL, NULL, NULL);
}

// Redirect to another URL (wrap ServerSupportFunction)
BOOL CIsapiServer::Redirect(const VARIANT FAR& url)
{
    COleVariant var;
    var=url;
    var.ChangeType(VT_BSTR);
    if (var.vt!=VT_BSTR) return FALSE;

    USES_CONVERSION;
    char *v=W2A(var.bstrVal);
    DWORD siz=strlen(v);
    return ecb->ServerSupportFunction(ecb->ConnID,
        HSE_REQ_SEND_URL_REDIRECT_RESP, v, &siz, NULL);
}

// Send alternate URL (wrap ServerSupportFunction)
BOOL CIsapiServer::SendURL(const VARIANT FAR& url)
{
    COleVariant var;
    var=url;
    var.ChangeType(VT_BSTR);
    if (var.vt!=VT_BSTR) return FALSE;

    USES_CONVERSION;
    char *v=W2A(var.bstrVal);
    DWORD siz=strlen(v);
    return ecb->ServerSupportFunction(ecb->ConnID,
        HSE_REQ_SEND_URL, v, &siz, NULL);
}

// Send headers (wrap ServerSupport Function)
BOOL CIsapiServer::SendHeaders(const VARIANT FAR& Status,
    const VARIANT FAR& Headers)
{

```

```

COleVariant var,var2;
var=Status;
var2=Headers;
var.ChangeType(VT_BSTR);
if (var.vt!=VT_BSTR) return FALSE;
var2.ChangeType(VT_BSTR);
if (var.vt!=VT_BSTR) return FALSE;

USES_CONVERSION;
char *status=W2A(var.bstrVal);
char *hdr=W2A(var.bstrVal);
return ecb->ServerSupportFunction(ecb->ConnID,
    HSE_REQ_SEND_RESPONSE_HEADER, status, NULL, (DWORD *)hdr);
}

// Map Virtual Path to Real Path (wrap ServerSupportFunction)
BOOL CIsapiServer::MapURL2Path(VARIANT FAR* urlpath)
{
    BOOL rv;
    COleVariant var,var2;
    var=urlpath;
    var.ChangeType(VT_BSTR);
    if (var.vt!=VT_BSTR) return FALSE;
    USES_CONVERSION;
    char *varin=W2A(var.bstrVal);
    DWORD siz=1024;
    CString url(varin);
    rv=ecb->ServerSupportFunction(ecb->ConnID,
        HSE_REQ_MAP_URL_TO_PATH,
        url.GetBufferSetLength(siz),&siz,NULL);
    url.ReleaseBuffer(siz-1);
    // set up return value
    VariantClear(urlpath);
    urlpath->vt=VT_BSTR;
    urlpath->bstrVal=url.AllocSysString();
    return rv;
}

```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

• [Advanced](#)[Search](#)• [Search Tips](#)BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

My example ISAPI extension uses VB, but really any language that can create a comparable OLE server will work. In this chapter, however, I'll call the OLE extension server a VB server to distinguish it from the C++ ISAPI extension DLL. How do you write a URL that calls your code? It is a two-step process. First, you have to get ISAPI to call the C++ DLL (CBISAPI.DLL). Then, you name the VB server in the URL's query string (the part following a question mark). Finally, you add a colon and the name of the OLE method you want to call. For example:

```
<A HREF=http://www.al-williams.com/awc/scripts/cbisapi.dll?hilo.dll:
Guess+newgame>
Click to begin
</A>
```

What does all of that mean? The part before the question mark invokes the ISAPI DLL. When you create an OLE server with VB, the server's name will contain the project name, a period, and the name of the VB Class module that contains the object you want to work with. In this example, the HILO project (I'll show it to you later in this chapter) has all of its code in a class module named DLL. Inside that module is a SUB named **Guess**. The plus sign signifies the end of the OLE server name. Everything after that is part of the query string the server sends to the program. Notice that the C++ server doesn't change the query string—it is up to your code to skip the first part, parse HTTP escape sequences, and otherwise process the query string.

A Quick Look At ISAPI

ISAPI programs are not too difficult to write in C or C++. There are two types. The extension DLL (the one you'll see in this chapter) generates output dynamically. Another type of extension, a filter, can handle certain requests for data. You can find out more about writing filters in the IIS and MFC documentation.

MFC provides a wrapper for ISAPI extensions that handles many common cases. However, for the ActiveX/ISAPI bridge, the wrappers got in the way. Therefore, I built the ISAPI support by hand, but I used MFC for ActiveX support. Later in this chapter, you'll see a more traditional MFC ISAPI program.

For this type of ISAPI DLL, you need two functions: **GetExtensionVersion** and

HttpExtensionProc. The **GetExtensionVersion** informs the server what version of IIS your DLL expects and provides a description string. You can copy this code directly from help—it is trivial code and you just need to change the string to reflect your extension. The **HttpExtensionProc** function is where all the work occurs. It receives a single argument, but that argument is a pointer to an **EXTENSION_CONTROL_BLOCK** (Table 9.12) that contains quite a bit of data. Compare Table 9.11 and Table 9.12. You'll notice that the members in Table 9.11 generally encapsulate the **EXTENSION_CONTROL_BLOCK** in an object-oriented way.

Table 9.12 Extension control block members.

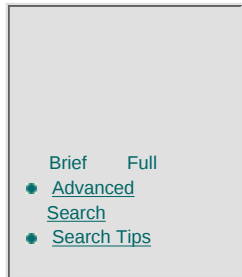
Member	Description
cbSize	Size of structure
dwVersion	Version number of structure
ConnID	Connection ID identifies particular request (passed back to many IIS functions)
dwHttpStatusCode	HTTP result code
lpszLogData	Extension-specific log data
lpszMethod	Method of request ("POST" or "GET", for example)
lpszQueryString	Query String
lpszPathInfo	Path to script
lpszPathTranslated	Translated path
cbTotalBytes	Total amount of content
cbAvailable	Amount of content already read
lpbData	Content (cbAvailable bytes)
lpszContentType	Data type of content
GetServerVariable	Function pointer used to retrieve server variables (see Table 9.3)
WriteClient	Function pointer used to write data to client
ReadClient	Function pointer used to read excess data from client (cbTotalBytes-cbAvailable bytes)
ServerSupport	Function pointer used to call special functions used to redirect, send URLs, map virtual paths, and more

You have to be careful when writing an extension DLL. Just as the DLLs that you use in a program become part of your process, your extension DLL will become part of IIS. If your DLL crashes, you could crash the server. If you throw an exception, the server will quietly continue and terminate your DLL. CBISAPI is careful to run the VB code inside a **try** block. It reports any exceptions it finds to the HTML stream.

[Previous](#) [Table of Contents](#) [Next](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97



Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Writing The HILO.DLL Server

VB5 makes writing an ActiveX DLL easy. From the starting screen, select ActiveX DLL. This correctly sets up the project and starts you with a class module to contain your properties and code. You'll want to be sure to rename the project and the class module—these will make up the name of the server.

For a CBISAPI server, you only need to define a **Public Sub** that takes an **Object** as an argument. The HILO.DLL class module (Listing 9.6) has several private **Subs** to handle internal processing. These are strictly for the benefit of the **Guess** subroutine—the one the HTML code calls.

Listing 9.6 The Visual Basic HILO.DLL.

```
VERSION 1.0 CLASS
BEGIN
    MultiUse = -1    'True
END
Attribute VB_Name = "DLL"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = True
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = True
Option Explicit
Private Sub svrerr(server As Object, errstr As String)
server.WriteLine "Error: " & errstr
server.statcode = 400
server.retval = 4
End Sub

Private Sub Win(server As Object)
server.WriteLine "<HTML><HEAD><TITLE>I Win</TITLE></HEAD><BODY>"
server.WriteLine "I got it right!</BODY></HTML>"
End Sub

Private Sub GuessAgain(server As Object, Hi As Long, Lo As Long)
Dim servername As Variant
Dim script As Variant
server.WriteLine "<HTML><HEAD><TITLE>HiLo!</TITLE></HEAD><BODY>"
server.WriteLine "My guess is " & CInt((Hi + Lo) / 2) & "<P>"
```

```

server.ServerVariable "SERVER_NAME", servername
server.ServerVariable "SCRIPT_NAME", script
server.WriteLine "Is my guess:<P>"
server.Write "<FORM ACTION=http://" & servername
server.Write "/" & script
server.WriteLine "?HILO.DLL:Guess+HI=" & Hi & "+LO=" & Lo & " METHOD=POST>"
server.WriteLine "High <INPUT TYPE=RADIO NAME=ANSWER VALUE=HI><P>"
server.WriteLine "Correct <INPUT TYPE=RADIO NAME=ANSWER VALUE=OK><P>"
server.WriteLine "Low <INPUT TYPE=RADIO NAME=ANSWER VALUE=LO><P>"
server.WriteLine "<INPUT TYPE=SUBMIT>"
server.WriteLine "</FORM>"
server.WriteLine "</BODY></HTML>"
End Sub

```

```

Public Sub Guess(server As Object)
    Dim Guess As Long
    Dim Hi As Long
    Dim Lo As Long
    Dim pos As Long
    Dim ans As String
    pos = InStr(1, server.QueryString, "HI=", vbTextCompare)
    If pos = 0 Then
        svrerr server, "Can't find HI"
        Exit Sub
    End If
    Hi = Val(Mid(server.QueryString, pos + 3))
    pos = InStr(1, server.QueryString, "LO=", vbTextCompare)
    If pos = 0 Then
        svrerr server, "Can't find LO"
        Exit Sub
    End If
    Lo = Val(Mid(server.QueryString, pos + 3))
    If server.ContentLength = 0 Then
        GuessAgain server, Hi, Lo
    Else
        Guess = (Hi + Lo) / 2
        pos = InStr(1, server.Content, "ANSWER=", vbTextCompare)
        If pos = 0 Then
            svrerr server, "Form error"
            Exit Sub
        End If
        ans = Mid(server.Content, pos + 7, 2)
        If ans = "OK" Then Win server
        If ans = "LO" Then GuessAgain server, Hi, Guess
        If ans = "HI" Then GuessAgain server, Guess, Lo
        If ans <> "OK" And ans <> "LO" And ans <> "HI" Then
            svrerr server, "Unknown Response: " & server.Content
        End IF
    End If
End Sub

```

Although this class `server` only has one public entry point, there is no reason you couldn't have multiple ones in a server. You can add more class modules, too. This would allow you to group related functions together in each class and group related classes in one DLL.

You can see the finished product in Figure 9.2. The game is actually a simple binary search. It will always guess correctly within 10 tries. Of course, you could always cheat. If the content (the data submitted by a form) is empty, the program assumes you are just starting the game. It expects the query string to have two variables: **HI** and **LO** (which specify the range of numbers). The program just calls the private **GuessAgain** routine to generate a form that displays the current guess and offers three radio buttons that specify if the guess is high, low, or correct. The form submits back to the same **Guess** method via CBISAPI. The program sets the query

string to reflect the current high and low values.



Figure 9.2 HILO on the Web.

On subsequent calls, the content will contain the status of the form buttons. The code detects that content is present and recalculates the high and low limits. It then calls **GuessAgain** to generate a new form. Of course, if the guess is correct, the code doesn't generate the form. Instead, it calls the **Win** routine to generate an appropriate message.

Listing 9.7 shows the HTML file that starts the whole thing running. Of course, you could embellish this if you like. Also, the guessing form in Figure 9.2 could be fancier. For example, you might put some JavaScript in the form so that when you click on a radio button, it automatically submits the form. Still, the existing code does the job and is enough to show how the ISAPI interface works.

Listing 9.7 The HILO Web page.

```
<HTML>
<HEAD>
<TITLE>Play Hi-Lo!</TITLE>
</HEAD>
<BODY>
I'll guess your number.<BR>
Think of a number between 1 and 1024 and I'll guess it.<BR>
Think of your number and
<A HREF=http://www/scripts/cbisapi.dll?HILO.DLL:GUESS+HI=1024+LO=1>
click here to play</A>
</BODY>
</HTML>
```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Inside The C++ DLL

The C++ DLL is a fairly straightforward MFC OLE DLL. The CBISAPI.CPP file (see the online listings) contains entry points required for ActiveX (MFC's App Wizard) and the ISAPI entry points.

The standard HTTPEXT.H prototypes the functions as C entry points so that the C++ compiler won't alter the names. However, this same header doesn't declare the functions as exportable (using **`__declspec(dllexport)`**). Therefore, you must mention the functions in the **EXPORTS** section of the DEF file so that IIS can locate them.

The **HttpExtensionProc** routine isn't very sophisticated. It parses the query string to find the name of the server and the method name. This name must be the first thing in the query string. The parsing ends at the end of the query string or at the first plus sign the code encounters. If you omit the method name, CBISAPI tries to call the **ISAPI** method.

Notice the use of UNICODE characters for the member name. If you pass in a name, the program uses the **A2W** function to convert the string to UNICODE. Otherwise, it uses the string literal **L"ISAPI"** (the L indicates a UNICODE constant). This is the first (but not the last) place this problem rears its ugly head. Of course, IIS supports HTTP. HTTP uses ANSI characters (the normal characters we all know and love). However, ActiveX uses UNICODE (two-byte character) strings for everything. In theory, the whole world will eventually switch to UNICODE. In theory, the U.S. will switch to the metric system—someday. In the meantime, you'll have to resort to conversions.

There are many ways to convert ANSI and UNICODE characters. I elected to

use the MFC functions from AFXCONV.H. See MFC's Technical Note 59 for more about these macros. The note, by the way, inaccurately states that you need AFXPRIV.H for the macros; this used to be true, but now you should use AFXCONV.H.

Once the code knows what object to create, it uses the MFC **CDispatchDriver** class to represent it. A call to **CreateDispatch** will create the object. Next, a call to **GetIDsOfNames** converts the member name to a dispatch ID (DISPID). DISPIDs are function codes that ActiveX automation objects use to identify members (and properties, too). Armed with the DISPID, a call to **InvokeHelper** calls the VB code. A previous call to **GetIDDispatch** retrieves the pointer you need to pass to the VB code so that it can access the server object.

Notice that CBISAPI protects the **InvokeHelper** call with a **try** statement. This ensures that any exceptions return to CBISAPI. CBISAPI reports errors by printing to the HTML stream. This works as long as the VB extension hasn't started writing some non-HTML data type before causing the exception.

The server object (see Listing 9.5) is where all the real work occurs. This class is easier to construct than you might expect. First, I used Class Wizard to create a **CCmdTarget**-derived class. All ActiveX automation objects in MFC derive from **CCmdTarget**. Then I used Class Wizard's Automation tab to add the properties and methods. The only hard part is writing the code.

The only odd part about the code is the conversion between **BSTRs** (ActiveX UNICODE strings) and **char *** (IIS ANSI strings). In several places, I used the MFC conversion functions I mentioned earlier. However, in several cases, I had the data in an MFC **CString** and decided to use **CString::AllocSysString()** to create a **BSTR**. That's where I ran into a little trouble.

BSTRs are not C (or C++) strings. They may contain embedded '\0' characters. To facilitate this, each **BSTR** has a count of characters. Usually, the string ends in a '\0' out of consideration for C/C++ programmers (and the Windows API), but the terminal NULL isn't usually part of the string. For example, a **BSTR** that contains the string "Coriolis\0" should have a count of 8 unless your intent is to embed the '\0' character inside the string. However, the size returned by ISAPI includes the '\0' character. Who cares? Well, VB cares. Consider the **ServerVariable** property in the CBISAPI server object. If it creates a **BSTR** that contains the NULL, what happens? Consider this VB code:

```
Dim x as Variant
server.ServerVariable "HOST_NAME",x
server.Write x
server.ServerVariable "SCRIPT_NAME",x
server.Write x
```

In this code, everything works fine, and the trailing zero byte is innocuous. However, consider this:

```
Dim x as Variant
Dim y as Variant
server.ServerVariable "HOST_NAME",x
server.ServerVariable "SCRIPT_NAME",y
server.Write x & y
```

Suppose the value of **HOST_NAME** is www.al-williams.com\0 and the value of **SCRIPT_NAME** is ztest.dll\0. VB dutifully forms the string www.al-williams.com\ 0ztest.dll\0. However, the C code that drives the **Write** method stops at the first NULL.

To prevent this problem, don't forget to subtract one from the size that ISAPI returns. Alternately, you could allow **CString** to recalculate the size. Either way, you must not set the size to include the zero byte.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Installation And Distribution

The C++ DLL has to create an ActiveX object. However, it does so on the behalf of an anonymous Internet user. Therefore, the default Internet user must have privileges to create ActiveX objects. For NT 4.0, you can set this by running DCOMCNFG (a program in your SYSTEM32 directory). Select the Default Security tab and add **IUSR_xxx** (where **xxx** is your server's name) to the Default Access Permissions and Default Launch Permissions sections. When you first click on the Add button on each choice, you'll only see group names. Click on the Show Users button to show individual users (including **IUSR_xxx**).

Of course, all the DLLs required by each piece of the puzzle must be on the server. If you build CBISAPI to use the MFC DLLs, then they must be present (preferably in the \WINNT\SYSTEM32 directory). The VB portion also requires the VB runtime DLLs.

One other thing that should be obvious: You must register your VB server on the Internet server machine using REGSVR32 for DLLs or by running the executable. Registering the server on a different machine only affects that machine's registry. If you fail to do this, you'll get an exception with an error code of **REGDB_E_CLASSNOTREG (0x80040154)**.

By the way, even though CBISAPI provides an ActiveX object, it doesn't require registration. That's because no other program ever creates its object. CBISAPI creates the object itself and passes it to other ActiveX programs. Of course, that means that the object has no type information. This prevents VB from validating your calls at compile time. Instead, you'll find out any type

mismatches or misspelled names at runtime.

Debugging ISAPI Extensions

Debugging ISAPI extensions is an unpleasant business at best. To debug the C++ portion, you can fudge IIS to run as a user process and run it under a debugger. You can find complete instructions for how to do this in MFC Technical Note 63.

Debugging the VB portion is even worse. The easiest thing to do is pepper your code with temporary **WriteLine** commands so you can see things in your HTML stream. Not ideal, but it works. Luckily, CBISAPI will report any exceptions that occur, so you only need to worry about logical errors.

Another annoyance is that you usually have to shut down IIS so you can copy over (or relink) your files. You can set the **HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/W3SVC/Parameters/CacheExtensions** registry key to zero to make IIS release the files quicker. Also, each time you recreate the VB server, you must reregister it on the IIS machine.

Future Directions

There are many enhancements you could make to the C++ DLL, CBISAPI. One welcome change would be automatic parsing of the query string and of content. For example, you could create a parameterized property named **ParsedQueryString**. It could take the name of an argument and return the string value, like this:

```
Dim s as String  
s=server.ParsedQueryString("HI")
```

I also toyed with adding a debugging mode that you could turn on with a query string option. Another idea would be to make a debugging version of CBISAPI.DLL (perhaps CBISAPID.DLL). This version would print debugging information out to the HTML stream. There are two problems with this. First, if the extension you're writing wants to set headers, it must do so before any output (including your debugging output). Second, what about extensions that output something other than an HTML file (for example, a GIF file)?

Finally, it would probably be a good idea to deny remote users the ability to create arbitrary ActiveX servers on your machine. Of course, the risk is minimal because the ActiveX server would need to have an entry point that expects a single object. Still, it would be simple to make CBISAPI read a configuration file on the server that defined symbolic names for ActiveX servers. If the request specifies a name that isn't on the list, CBISAPI could reject the command.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

Traditional MFC ISAPI

You'll find a more traditional MFC ISAPI extension in Listing 9.8. You can easily generate programs like this using the MFC ISAPI Wizard (see Figure 9.3). This Wizard creates a DLL that contains functions that an HTML document or form can invoke.

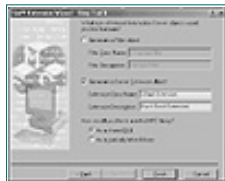


Figure 9.3 The ISAPI extension wizard.

MFC automates the entire process of parsing variables from form data by using a parse map. This is a map similar to a message map or a data map, except that Class Wizard doesn't support it. You have to add entries to the parse map by hand.

Listing 9.8 An MFC-based ISAPI extension.

```
// ISAPIMFC.CPP - Implementation file for your Internet Server
// Example Form Extension

#include "stdafx.h"
#include "isapimfc.h"

////////////////////
// The one and only CWinApp object
// NOTE: You may remove this object if you alter your project to no
// longer use MFC in a DLL.

CWinApp theApp;

////////////////////
// command-parsing map
```

```

BEGIN_PARSE_MAP(CExtension, CHttpServer)
    // TODO: insert your ON_PARSE_COMMAND() and
    // ON_PARSE_COMMAND_PARAMS() here to hook up your commands.
    // For example:

    ON_PARSE_COMMAND(Default, CExtension, ITS_EMPTY)
    ON_PARSE_COMMAND(Greet, CExtension, ITS_PSTR)
    ON_PARSE_COMMAND_PARAMS("name=~")
    DEFAULT_PARSE_COMMAND(Default, CExtension)
END_PARSE_MAP(CExtension)

//////////
// The one and only CExtension object

CExtension theExtension;
//////////
// CExtension implementation

CExtension::CExtension()
{
}

CExtension::~~CExtension()
{
}

BOOL CExtension::GetExtensionVersion(HSE_VERSION_INFO* pVer)
{
    // Call default implementation for initialization
    CHttpServer::GetExtensionVersion(pVer);

    // Load description string
    TCHAR sz[HSE_MAX_EXT_DLL_NAME_LEN+1];
    ISAPIVERIFY(::LoadString(AfxGetResourceHandle(),
        IDS_SERVER, sz, HSE_MAX_EXT_DLL_NAME_LEN));
    _tcscpy(pVer->lpszExtensionDesc, sz);
    return TRUE;
}

//////////
// CExtension command handlers

void CExtension::Default(CHttpServerContext* pCtxt)
{
    StartContent(pCtxt);
    WriteTitle(pCtxt);

    *pCtxt << _T("This default message was produced by the Internet");
    *pCtxt << _T(" Server DLL Wizard. Edit your CExtension::Default()");
    *pCtxt << _T(" implementation to change it.\r\n");

    EndContent(pCtxt);
}

// Do not edit the following lines, which are needed by Class Wizard.
#if 0

```

```

BEGIN_MESSAGE_MAP(CExtension, CHttpServer)
   //{{AFX_MSG_MAP(CExtension)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
#endif // 0

//////////
// If your extension will not use MFC, you'll need this code to make
// sure the extension objects can find the resource handle for the
// module. If you convert your extension to not be dependent on MFC,
// remove the comments around the following AfxGetResourceHandle()
// and DllMain() functions, as well as the g_hInstance global.

/****

static HINSTANCE g_hInstance;

HINSTANCE AFXISAPI AfxGetResourceHandle()
{
    return g_hInstance;
}

BOOL WINAPI DllMain(HINSTANCE hInst, ULONG ulReason,
    LPVOID lpReserved)
{
    if (ulReason == DLL_PROCESS_ATTACH)
    {
        g_hInstance = hInst;
    }

    return TRUE;
}

****/

void CExtension::Greet(CHttpServerContext * pCtxt, char *name)
{
    StartContent(pCtxt);
    WriteTitle(pCtxt);
    if (*name=='~')
    {
        *pCtxt << _T("Hello <B>");
        *pCtxt << name;
    }
    else
        *pCtxt << _T("You should have entered a name!");

    EndContent(pCtxt);
}

```

Luckily, parse maps are easy. You can use the macros you find in Table 9.13. The **ON_PARSE_COMMAND** macro tells MFC that there is a function you want to map to a particular name. In Listing 9.8, there are two such entries: **Default** and **Greet**. These functions return **void** and take arguments as specified by the third argument in the macro. Each function also takes a **CHttpServerContext** object as its first argument. This object allows the function to interact with the server.

Table 9.13 The parse map macros.

Macro	Meaning
BEGIN_PARSE_MAP	Starts parse map
ON_PARSE_COMMAND	Identifies entry point into DLL
ON_PARSE_COMMAND_PARAMS	Parameters for previous parse command
DEFAULT_PARSE_COMMAND	Entry point to use if none specified
END_PARSE_MAP	End of map

[Previous](#) [Table of Contents](#) [Next](#)

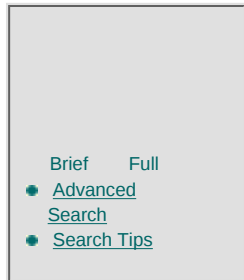
[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE



BROWSE BY TOPIC



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

If the function takes no arguments (other than the **CHttpServerContext** object), you don't need any other macros other than **ON_PARSE_COMMAND** for that function. If it does take arguments, you must immediately follow **ON_PARSE_COMMAND** with **ON_PARSE_COMMAND_PARAMS**. This macro lets you specify the form (or query string) values that correspond to the function's variables. You can also specify a default value. For instance, the example program contains the following lines in its parse map:

```
ON_PARSE_COMMAND(Greet,CExtension,ITS_PSTR)
ON_PARSE_COMMAND_PARAMS("name=~")
```

This means that the function's string argument should receive the value of the form field **name**. If there is no data for that field, MFC should supply a tilde (the ~ character).

What if your function took two pieces of input, say **FName** and **LName**? Then your map might look like this:

```
ON_PARSE_COMMAND(Greet,CExtension,ITS_PSTR)
ON_PARSE_COMMAND_PARAMS("Fname=~ LName=~")
```

Inside your ISAPI functions, you can write to the HTML stream by using the << operator with the **CHttpServerContext** object. You can also make the calls you'll find in Table 9.14 with the object to interact with the server.

Table 9.14 CHttpRequestContext members.

Member	Definition
m_pECB	Extension Control Block
m_pStream	CHtmlStream in use
GetServerVariable	Retrieve a server variable
WriteClient	Write data to client
ReadClient	Read data from client
ServerSupportFunction	Call function to (for example) redirect, map paths, and so forth
<<	Write to HTML output

When you want to call your function, you name it after the query string in your URL. So to call **Default**, you might use the URL <http://www.al-williams.com/awc/isapi/isapimfc.dll?Default>.

Because there is a **DEFAULT_PARSE_COMMAND** macro for **Default** in the parse map, MFC will call **Default** even if you omit the query string. To call **Guess** for a form, you'd write:

```
<FORM ACTION=http://www.al-williams.com/awc/isapi/isapimfc.dll?Guess
METHOD=POST>
```

You should always use **POST** because **GET** can cause some problems with certain browsers. You could also set the **Name** variable in the query string like this:

```
<A HREF=http://www.al-williams.com/awc/isapi/isapimfc.dll?Guess?name=none>
```

Writing ISAPI programs isn't overly difficult with or without MFC. Once you get used to manipulating parse maps, you'll find you can turn out ISAPI extensions very rapidly.

Summary

Although Microsoft was slow to get started in the Internet race, they have certainly done a fine job of playing catch-up. It seems like every product that Microsoft has is Internet-enabled in some way. MFC is no exception.

The MFC socket and ISAPI extensions have real value in some cases. In other cases, they may just get in your way. Using ActiveX controls to perform data transfer, however, is the kind of task that really shows off the true power of ActiveX.

Practical Guide To MFC And The Internet

- Using Sockets
- Using Sockets As Streams
- Using WinInet With MFC
- The Internet Transfer Control
- Writing ISAPI Extensions And Filters With MFC
- When Not To Use ISAPI
- CBISAPI—An Object-Oriented Approach To ISAPI

The Internet is the biggest craze the computer industry has ever seen. Although writing network software used to be the purview of a few specialists, more and more programmers must incorporate Internet functionality into their programs. Luckily, MFC gives you several options to ease Internet programming.

Using Sockets

MFC supports sockets using the **CSocket** and **CAsyncSocket** class. Usually, you'll use **CSocket**, which provides a reliable connection between two computers. Both server and client require an object. You can find the calls you'll need to make in both cases listed in Tables 9.1 and 9.2.

Once you have a connection, you can easily communicate using the **Send** and **Receive** member functions. However, MFC programs may prefer sending entire objects via serialization (see the next section).

You won't normally need to override the socket classes. The only exception is when you want to use the socket asynchronously. Then you'll need to override the notification functions (like **OnReceive**, for example). You can find an example of this in Listing 9.3.

Using Sockets As Streams

The **CSocketFile** class allows you to treat a socket as though it were an archive. Of course, via serialization, an archive can handle complete MFC objects. When you construct a **CSocketFile** object, you'll pass a pointer to an existing **CSocket** object. You can then use the socket to create one or more archives. Here is a bit of code excerpted from Listing 9.1:

```
sockfile=new CSocketFile(&socket);
```

```
xmit=new CArchive(sockfile,CArchive::store);  
rcv=new CArchive(sockfile,CArchive::load);
```

Here, the program creates two archives from the same file, one for each direction (transmit and receive). You can use this archive the same way you would use any **CArchive** object (for example, with << or the **Serialize** member function of an object).

Using WinInet With MFC

WinInet is a library within Windows that makes it easier to manipulate the Internet. MFC provides access to WinInet via the **CInternetSession** object. You can use this object to open a URL or start a session for HTTP, FTP, or Gopher (see Table 9.5).

The easiest way to use this object is to call **OpenURL**, which returns an object derived from **CStdioFile** (or, perhaps, a **CStdioFile**). There are other calls you can make (see Table 9.5) if you want more control, but **OpenURL** will serve most of your needs.

The Internet Transfer Control

The Internet Transfer Control is an ActiveX control that allows your programs to get data from the Internet. You'll see that the ActiveX object is a thin disguise over the standard WinInet library calls. You even still use **OpenURL** to access an object. You can find all of the calls in Table 9.7.

Notice that if you want to parse and display HTML, you probably will be better off using the Internet Explorer ActiveX object (Shell.Explorer in SHDOCVW.DLL) instead of the transfer control.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Writing ISAPI Extensions And Filters With MFC

ISAPI programs allow you to extend Microsoft's Internet Information Server (IIS) and a few other Web servers without writing CGI scripts. ISAPI is more powerful and more efficient than traditional methods (such as CGI).

MFC provides a wizard that helps you write ISAPI extensions and filters (see Figure 9.3). You can fill the resulting DLL with functions you want Web pages (including forms) to call. Then you have to manually construct a parse map to associate the functions with names the HTML code uses.

Your functions should return **void** and take at least one argument (a **CHttpServerContext** object). This server context object allows you to find out about the Web server and request services from it. For example, you can write data to the output stream using this object. You can find an example in Listing 9.8. You can also include other arguments that you expect the HTML code to supply.

A parse map is similar to other maps used in MFC in so much as it uses macros (see Table 9.13). The wizard will construct a skeletal map for you. You simply add an **ON_PARSE_COMMAND** macro to inform MFC about a function you wrote and how many arguments to expect. If the function requires extra arguments from HTML, you'll also need to immediately follow this macro with an **ON_PARSE_COMMAND_PARAMS** macro that describes the arguments and supplies default values. Here is a simple parse map:

```
BEGIN_PARSE_MAP(CExtension, CHttpServer)
```

```
ON_PARSE_COMMAND(Default, CExtension, ITS_EMPTY)
ON_PARSE_COMMAND(Greet, CExtension, ITS_PSTR)
ON_PARSE_COMMAND_PARAMS("name=~")
DEFAULT_PARSE_COMMAND(Default, CExtension)
END_PARSE_MAP(CExtension)
```

The first **ON_PARSE_COMMAND** defines the **Default** function that takes no arguments. The **DEFAULT_PARSE_COMMAND** also names this function. This macro informs MFC that if the HTML doesn't supply a function name, it should use this function.

The second **ON_PARSE_COMMAND** defines a function (**Greet**) that takes a single string argument. The next line informs MFC that the HTML argument **name** (from a form or query string) will contain the value of the argument. If the HTML does not supply the argument, MFC will use the tilde (~) character because this macro specifies it as the default value. You can find a complete example in Listing 9.8.

HTML usually sends the arguments via named fields in a form. Your form's action is then the name of the DLL, a question mark, and the name of the function you want to call. Of course, the Web server must have the directory that contains the DLL marked as executable.

When Not To Use ISAPI

It is worth noting that some of the traditional uses of ISAPI are better handled via server-side scripting using ASP (Active Server Page) files. These files allow you to intermix HTML and functions written in VBScript (a language similar to Visual Basic) and JavaScript (a language similar to Java). ASP files are outside the scope of this book, but you should consider them first instead of writing CGI or ISAPI programs if your server supports them.

CBISAPI—An Object-Oriented Approach To ISAPI

In this chapter, you'll find two ISAPI examples. Listing 9.8 shows a traditional MFC ISAPI extension program. However, you'll also find CBISAPI in Listing 9.5. This MFC program is really an ActiveX approach to ISAPI. It creates an ActiveX object to handle the work and supplies that object with another object that allows it to work with the server.

One advantage to this approach is that it allows you to write ISAPI extensions using any language that can support ActiveX. The example program in Listing 9.6 uses Visual Basic. Of course, MFC can create ActiveX objects, too.

You'll notice that because CBISAPI needs low-level control over ISAPI, it does not use the normal MFC ISAPI approach. Instead, it interacts directly with ISAPI. However, MFC does help out quite a bit with the ActiveX portion of the code.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Chapter 10

MFC And Databases

Because Class Wizard is able to connect database fields to fields in a record view, writing MFC database apps is easy. Whether you are writing a batch processing program or one that has a user interface, you can use Class Wizard-generated code to simplify your life.

Programmer's Notes...

Times were different before the introduction of the personal computer. Although I was more involved in the hardware end of things at that time, everyone in the business was regarded almost as a high priest, doing things far beyond the imagination of ordinary people. I miss those days. Today when you tell someone that you program computers, they usually reply, "Oh, yeah. My first-grader learned that in school last month." Sure. But can she put relational tables in normal form?

Even before the PC became popular, the average person associated databases with computers. Of course, television and the movies helped this perception. Even my favorite show, *Star Trek*, had the know-it-all computers: "Computer: Identify the name RedJack." Of course, the computer knows the answer. The robot on *Lost in Space* had the same quality. These computers had the sum-of-all-knowledge database installed.

Real databases tend to be more focused (usually answering the burning

question of “Who owes us money?”). However, it’s surprising to see how often programmers have to deal with databases in some form. Even if your primary job doesn’t involve databases, you’ll often find yourself involved with them in some capacity.

Because databases are so ubiquitous, many programming environments struggle to make them easy to use. MFC is no exception. If you can work with dialog boxes, you can easily construct powerful database programs using MFC.

At the highest level, writing database programs is nearly the same as writing a form-based program. You ask App Wizard to connect to a database table when you create your program.

The App Wizard does all of the real work. It creates a special view class and an object called a record set. This object encapsulates the connection to the database.

The App Wizard also creates a tool bar that allows navigation through the table. All you need to do is add fields to the view (which is very similar to a **CFormView**) and connect them to variables in the record set using Class Wizard. That’s all for the simplest case (see Figure 10.1).

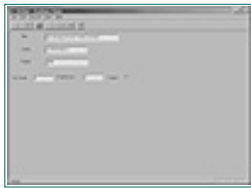


Figure 10.1 A simple application.

This requires practically no work, but it is best suited for simplistic applications. However, the basic architecture is flexible. You can use the record set in a variety of applications that require database access—even those that don’t use a conventional document and view.

The record set object is useful in many situations in which you want to manipulate data directly. There are two different types of record sets that you can create. One is for working with any ODBC (Open DataBase Connectivity) source. Of course, this implies that ODBC drivers are available for your database and that you have them installed. The other is a DAO (Data Access Object). The DAO accesses Microsoft Access databases (and some other formats that the Microsoft Jet engine understands). In certain cases, you will get better performance from DAO than ODBC. However, ODBC generally performs better for client/server transactions. Also, DAO provides more ways to manipulate table structure. DAO can read any ODBC database, but using DAO with an ODBC driver impacts performance (because there are two layers of libraries to filter through). In particular, DAO works best with Microsoft Access files.

Luckily, both the ODBC classes and the DAO classes are very similar. Because DAO does everything ODBC can do and more, it’s fairly easy to move from ODBC to DAO. It can be a bit more taxing to go the other way if you have used any special DAO features.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Table 10.1 shows the members of the ODBC record set object (**CRecordset**). You can find the same information for the DAO object (**CDaoRecordset**) in Table 10.2. You'll notice that there isn't much difference between the two.

Table 10.1 ODBC record set members.

Member	Description
m_hstmt	The ODBC statement handle for the recordset
m_nFields	The number of field data members in the recordset
m_nParams	The number of parameter data members in the recordset
m_pDatabase	Pointer to the CDatabase object through which the recordset is connected to a data source
m_strFilter	SQL WHERE clause
m_strSort	SQL ORDER BY clause
Open	Opens the recordset
Close	Closes the recordset and the ODBC HSTMT
CanAppend	Returns nonzero if new records can be added with AddNew
CanBookmark	Returns nonzero if the recordset supports bookmarks
CanRestart	Returns nonzero if Requery can be called to run the query again

CanScroll	Returns nonzero if you can scroll through the records
CanTransact	Returns nonzero if the data source supports transactions
CanUpdate	Returns nonzero if the recordset can be updated
GetODBCFieldCount	Returns the number of fields in the recordset
GetRecordCount	Returns the number of records in the recordset
GetStatus	Gets the status of the recordset (the current record's index and whether a final count of the records is available)
GetTableName	Gets the name of the table
GetSQL	Gets the SQL string
IsOpen	Returns nonzero if already open
IsBOF	Returns nonzero if the recordset's position is before the current record
IsEOF	Returns nonzero if the recordset's position is after the last record
IsDeleted	Returns nonzero if the recordset is on a deleted record
AddNew	Prepares for adding a new record (call Update to complete)
CancelUpdate	Cancels any pending updates (from an AddNew or Update)
Delete	Deletes the current record from the recordset
Edit	Prepares for changes to the current record (call Update to complete the edit)
Update	Completes an AddNew or Edit operation by saving the new or edited data on the data source
GetBookmark	Assigns the bookmark value of a record to the parameter object
Move	Positions the recordset to a specified number of records from the current record in either direction
MoveFirst	Positions the current record on the first record in the recordset
MoveLast	Positions the current record on the last record
MoveNext	Positions the current record on the next record
MovePrev	Positions the current record on the previous record
SetAbsolutePosition	Positions the recordset on the record corresponding to the specified record number
SetBookmark	Positions the recordset on the record specified by the bookmark

Cancel	Cancels an asynchronous operation or a process from a second thread
FlushResultSet	Returns nonzero if there is another result set to be retrieved (used with predefined queries)
GetFieldValue	Returns the value of a field in a recordset
GetODBCFieldInfo	Returns specific kinds of information about the fields in a recordset
GetRowsetSize	Returns the number of records you wish to retrieve during a single fetch
GetRowsFetched	Returns the actual number of rows retrieved during a fetch
GetRowStatus	Returns the status of the row after a fetch
IsFieldDirty	Returns nonzero if the specified field in the current record has been changed
IsFieldNull	Returns nonzero if the specified field in the current record is Null (has no value)
IsFieldNullable	Returns nonzero if the specified field in the current record can be set to Null
RefreshRowset	Refreshes the data and status of the specified row(s)
Requery	Runs the recordset's query again to refresh the selected records
SetFieldDirty	Marks the specified field in the current record as changed
SetFieldNull	Sets the value of the specified field in the current record to Null (having no value)
SetLockingMode	Sets the locking mode to "optimistic" locking (the default) or "pessimistic" locking (this determines how records are locked for updates)
SetParamNull	Sets the specified parameter to Null
SetRowsetCursorPosition	Positions the cursor on the specified row within the rowset
Check	Called to examine the return code from an ODBC API function
CheckRowsetError	Called to handle errors generated during record fetching
DoBulkFieldExchange	Called to exchange bulk rows of data from the data source to the recordset
DoFieldExchange	Called to exchange data (in both directions) between the field data members of the recordset and the corresponding record on the data source
GetDefaultConnect	Called to get the default connect string
GetDefaultSQL	Called to get the default SQL string to execute

OnSetOptions	Called to set options for the specified ODBC statement
SetRowsetSize	Specifies the number of records you wish to retrieve during a fetch

Table 10.2 Members for the DAO record set.

Member	Description
m_bCheckCacheForDirtyFields	Indicates if fields are automatically marked as changed
m_pDAORecord set	A pointer to the DAO interface underlying the record set object
m_nFields	Number of field data members in the record set class and the number of columns selected by the record set from the data source
m_nParams	Contains the number of parameter data members in the record set class and the number of parameters passed with the record set's query
m_pDatabase	Source database for this result set (pointer to a CDaoDatabase object)
m_strFilter	SQL WHERE statement
m_strSort	SQL ORDER BY statement
Close	Closes the record set
Open	Creates a new record set
CanAppend	Returns nonzero if new records can be added with AddNew
CanBookmark	Returns nonzero if the record set supports bookmarks
CanRestart	Returns nonzero if Requery can be called to run the record set's query again
CanScroll	Returns nonzero if you can scroll through the records
CanTransact	Returns nonzero if the data source supports transactions
CanUpdate	Returns nonzero if the record set can be updated
GetCurrentIndex	Contains the name of the index most recently used on an indexed, table-type record set
GetDateCreated	Returns the date and time the base table underlying a CDaoRecord set object was created
GetDateLastUpdated	Returns the date and time of the most recent change made to the design of a base table underlying a CDaoRecord set object

GetEditMode	Returns a value that indicates the state of editing for the current record
GetLastModifiedBookmark	Used to determine the most recently added or updated record
GetName	Contains the name of the record set
GetParamValue	Retrieves the current value of the specified parameter stored in the underlying DAOParameter object
GetRecordCount	Returns the number of records accessed in a record set object
GetSQL	Gets the SQL string used to select records for the record set
GetType	Called to determine the type of a record set: table-type, dynaset-type, or snapshot-type
GetValidationRule	Gets the rule that validates data as it is entered into a field
GetValidationText	Retrieves the text that is displayed when a validation rule is not satisfied
IsBOF	Returns nonzero if the record set has been positioned before the first record
IsDeleted	Returns nonzero if the record set is positioned on a deleted record
IsEOF	Returns nonzero if the record set has been positioned after the last record
IsFieldDirty	Returns nonzero if the specified field in the current record has been changed
IsFieldNull	Returns nonzero if the specified field in the current record is Null
IsFieldNullable	Returns nonzero if the specified field in the current record can be set to Null
IsOpen	Returns nonzero if Open has been called
SetCurrentIndex	Called to set an index on a table-type record set
SetParamValue	Sets the current value of the specified parameter stored in the underlying DAOParameter object
SetParamValueNull	Sets the current value of the specified parameter to Null
AddNew	Prepares for adding a new record (call Update to complete)
CancelUpdate	Cancels any pending updates due to an Edit or AddNew operation
Delete	Deletes the current record from the record set
Edit	Prepares for changes to the current record (call Update to complete the edit)

Update	Completes an AddNew or Edit operation by saving the new or edited data on the data source
Find	Locates the first, next, previous, or last location of a particular string in a dynaset-type record set that satisfies the specified criteria and makes that record the current record
FindFirst	Locates the first record in a dynaset-type or snapshot-type record set that satisfies the specified criteria and makes that record the current record
FindLast	Locates the last record in a dynaset-type or snapshot-type record set that satisfies the specified criteria and makes that record the current record
FindNext	Locates the next record in a dynaset-type or snapshot-type record set that satisfies the specified criteria and makes that record the current record
FindPrev	Locates the previous record in a dynaset-type or snapshot-type record set that satisfies the specified criteria and makes that record the current record
GetAbsolutePosition	Returns the record number of a record set object's current record
GetBookmark	Returns a value that represents the bookmark on a record
GetPercentPosition	Returns the position of the current record as a percentage of the total number of records
Move	Positions the record set to a specified number of records from the current record in either direction
MoveFirst	Positions the current record on the first record in the record set
MoveLast	Positions the current record on the last record in the record set
MoveNext	Positions the current record on the next record in the record set
MovePrev	Positions the current record on the previous record in the record set
Seek	Locates the record in an indexed table-type record set object that satisfies the specified criteria for the current index and makes that record the current record
SetAbsolutePosition	Sets the record number of a record set object's current record

SetBookmark	Positions the record set on a record containing the specified bookmark
SetPercentPosition	Sets the position of the current record to a location corresponding to a percentage of the total number of records in a record set
FillCache	Fills all or a part of a local cache for a record set object that contains data from an ODBC data source
GetCacheSize	Returns a value that specifies the number of records in a dynaset-type record set containing data to be locally cached from an ODBC data source
GetCacheStart	Returns a value that specifies the bookmark of the first record in the record set to be cached
GetFieldCount	Returns a value that represents the number of fields in a record set
GetFieldInfo	Returns specific kinds of information about the fields in the record set
GetFieldValue	Returns the value of a field
GetIndexCount	Retrieves the number of indexes in a table underlying a record set
GetIndexInfo	Returns various kinds of information about an index
GetLockingMode	Returns a value that indicates the type of locking that is in effect during editing
Requery	Runs the record set's query again to refresh the selected records
SetCacheSize	Sets a value that specifies the number of records in a dynaset-type record set containing data to be locally cached from an ODBC data source
SetCacheStart	Sets a value that specifies the bookmark of the first record in the record set to be cached
SetFieldDirty	Marks the specified field in the current record as changed
SetFieldNull	Sets the value of the specified field in the current record to Null
SetFieldValue	Sets the value of a field
SetFieldValueNull	Sets the value of a field to Null
SetLockingMode	Sets a value that indicates the type of locking to put into effect during editing
DoFieldExchange	Called to exchange data (in both directions) between the field data members and the corresponding record on the data source
GetDefaultDBName	Returns the name of the default data source

To work with a database, you must set up an ODBC data source. You can do this in the Control Panel (see Figure 10.2). The example program in this book uses a user data source named Books. If you create a database attached to the data set, or open an existing one, the program will automatically open that database when it starts. If you leave the file name blank, the program will prompt you when it begins to select the database file.



All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



SEARCH ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Database In Detail

To create a database program, you only need to run App Wizard. Many database programs are SDI programs because you only want to work with one record at a time. However, you can select MDI if it suits your needs. On the second page (see Figure 10.3), select Database View Without File Support. This causes the wizard to build all the support you need. You can also ask it to add provisions for serializing ordinary files (the Database View With File Support option) or to put headers in your files so that you can create and use your own database objects (the Header Files Only option). By default, the record set queries the entire table, but you can usually return a custom query in the **GetDefaultSQL** member.



Figure 10.3 App Wizard database options.

You also need to specify the data source by clicking on the Data Source button (see Figure 10.4). Here you can select ODBC and the appropriate source. You can also select one of three choices for how you want to work with the table:

- Dynaset—Allows you to dynamically work with the record set along with other users.
- Snapshot—Allows you to work with the table as it existed at the time of the query.

- Table—Works directly with a table instead of a query.



Figure 10.4 Selecting a data source.

The table option will be grayed out unless you select DAO.

Once the wizard is done, you only need to place controls on the dialog template (just like a form view). Then, you use Class Wizard's Member Variable tab to connect these controls to database fields (see Figure 10.5).



Figure 10.5 Binding fields to the record view.

If you don't need any computed fields or other special features, and you only want to browse and edit the existing records, you're done. It is that easy.

Adding More Features

One of the most common additions you'll want to make to your database program is a computed field. For example, a sales database might store a price, a quantity, and a discount. If you want to compute the total price, you'll need to calculate it. Luckily, because the record view is essentially the same as a form view, that is easy to do. Just use Class Wizard to set up a member variable and proceed as usual. You can perform the calculations when the record changes (use the view's **OnMove** event).

Adding And Deleting Records

The wizard doesn't generate code to insert or delete records because each program handles these operations differently. But it is easy to add these commands yourself. To delete the current record, just call the **Delete** member of the record set.

Inserting a new record is typically a three-step process. First, you call the record set's **AddNew** function. Then you load the form's data into the record set variables using **UpdateData** (just like a **CFormView**). Then you call **Update** to finish the transaction. You can also call **CancelUpdate** to forget the whole thing.

Not Using A View

You can create a database program without a record view if that suits your purpose. For example, you might want to write a program that changes each field in a database table automatically, or you may want one that generates a report.

For these programs, you can always use a record set without a record view. You can view the available calls in Table 10.1 (ODBC) and 10.2 (DAO).

You can see in the tables that the record set is a generally useful interface to your database. DAO even allows you to create tables and manipulate table structure. With ODBC, you'd need to resort to direct ODBC calls for this functionality.

An Example Program

Because database programming is so easily accomplished with MFC, the example program is also easy (see Figure 10.1). The program was originally an ordinary App Wizard-generated program. Later, I added code to add and delete records, as well as a computed field.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

You can find the record set in Listing 10.1. This is exactly the way App Wizard created it, with no modifications. Listing 10.2 shows the view, which has several new pieces in it.

Listing 10.1 An example record set.

```
// dbdemoSet.cpp : implementation of the CDbdemoSet class
//
```

```
#include "stdafx.h"
#include "dbdemo.h"
#include "dbdemoSet.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
////////////////////////////////////
// CDbdemoSet implementation
```

```
IMPLEMENT_DYNAMIC(CDbdemoSet, CRecordset)
```

```
CDbdemoSet::CDbdemoSet(CDatabase* pdb)
: CRecordset(pdb)
{
   //{{AFX_FIELD_INIT(CDbdemoSet)
    m_Title = _T("");
    m_ISBN = _T("");
    m_Pages = 0;
    m_warehouse = 0;
    m_shelf = 0;
    m_nFields = 5;
   //}}AFX_FIELD_INIT
```

```

    m_nDefaultType = snapshot;
}

CString CDbdemoSet::GetDefaultConnect()
{
    return _T("ODBC;DSN=Books");
}

CString CDbdemoSet::GetDefaultSQL()
{
    return _T("[Books]");
}

void CDbdemoSet::DoFieldExchange(CFieldExchange* pFX)
{
    //{AFX_FIELD_MAP(CDbdemoSet)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Text(pFX, _T("[Title]"), m_Title);
    RFX_Text(pFX, _T("[ISBN]"), m_ISBN);
    RFX_Int(pFX, _T("[Pages]"), m_Pages);
    RFX_Int(pFX, _T("[InWhse]"), m_warehouse);
    RFX_Int(pFX, _T("[OnShelf]"), m_shelf);
    //}}AFX_FIELD_MAP
}

//////////
// CDbdemoSet diagnostics

#ifdef _DEBUG
void CDbdemoSet::AssertValid() const
{
    CRecordset::AssertValid();
}

void CDbdemoSet::Dump(CDumpContext& dc) const
{
    CRecordset::Dump(dc);
}
#endif // _DEBUG

```

Listing 10.2 The example record view.

```

// dbdemoView.cpp : implementation of the CDbdemoView class
//

#include "stdafx.h"
#include "dbdemo.h"
#include "dbdemoSet.h"
#include "dbdemoDoc.h"
#include "dbdemoView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

```

```

////////////////////
// CDbdemoView

IMPLEMENT_DYNCREATE(CDbdemoView, CRecordView)

BEGIN_MESSAGE_MAP(CDbdemoView, CRecordView)
   //{{AFX_MSG_MAP(CDbdemoView)
    ON_COMMAND(IDM_DELETE, OnDelete)
    ON_COMMAND(IDM_ADD, OnAdd)
    ON_UPDATE_COMMAND_UI(IDM_ADD, OnUpdateAdd)
    ON_UPDATE_COMMAND_UI(IDM_CANCEL, OnUpdateCancel)
    ON_COMMAND(IDM_CANCEL, OnCancel)
   //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CRecordView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CRecordView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CRecordView::OnFilePrintPreview)
END_MESSAGE_MAP()

////////////////////
// CDbdemoView construction/destruction

CDbdemoView::CDbdemoView()
    : CRecordView(CDbdemoView::IDD)
{
   //{{AFX_DATA_INIT(CDbdemoView)
    m_pSet = NULL;
    m_copies = _T("");
   //}}AFX_DATA_INIT
    // TODO: add construction code here
    updateMode=FALSE;
}

CDbdemoView::~CDbdemoView()
{
}

void CDbdemoView::DoDataExchange(CDataExchange* pDX)
{
    CRecordView::DoDataExchange(pDX);
    // set copies up
    m_copies.Format("%d",m_pSet->m_warehouse+m_pSet->m_shelf);
   //{{AFX_DATA_MAP(CDbdemoView)
    DDX_Text(pDX, IDC_COPIES, m_copies);
    DDX_FieldText(pDX, IDC_TITLE, m_pSet->m_Title, m_pSet);
    DDX_FieldText(pDX, IDC_ISBN, m_pSet->m_ISBN, m_pSet);
    DDX_FieldText(pDX, IDC_PAGES, m_pSet->m_Pages, m_pSet);
    DDV_MinMaxInt(pDX, m_pSet->m_Pages, 0, 9999);
    DDX_FieldText(pDX, IDC_WHSE, m_pSet->m_warehouse, m_pSet);
    DDV_MinMaxInt(pDX, m_pSet->m_warehouse, 0, 99999);
    DDX_FieldText(pDX, IDC_SHELF, m_pSet->m_shelf, m_pSet);
    DDV_MinMaxInt(pDX, m_pSet->m_shelf, 0, 99999);
   //}}AFX_DATA_MAP
}

BOOL CDbdemoView::PreCreateWindow(CREATESTRUCT& cs)
{

```

```

    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CRecordView::PreCreateWindow(cs);
}

void CDbdemoView::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_dbdemoSet;
    CRecordView::OnInitialUpdate();
}

//////////
// CDbdemoView printing

BOOL CDbdemoView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}

void CDbdemoView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CDbdemoView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}

//////////
// CDbdemoView diagnostics

#ifdef _DEBUG
void CDbdemoView::AssertValid() const
{
    CRecordView::AssertValid();
}

void CDbdemoView::Dump(CDumpContext& dc) const
{
    CRecordView::Dump(dc);
}

CDbdemoDoc* CDbdemoView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CDbdemoDoc)));
    return (CDbdemoDoc*)m_pDocument;
}
#endif // _DEBUG

//////////
// CDbdemoView database support
CRecordset* CDbdemoView::OnGetRecordset()
{
    return m_pSet;
}

```

```

}

////////////////////////////////////
// CDbdemoView message handlers

void CDbdemoView::OnDelete()
{
    if (MessageBox("Delete Record", "Confirm", MB_YESNO)==IDYES)
    {
        m_pSet->Delete();
        m_pSet->Requery();
        UpdateData(FALSE);
    }
}

void CDbdemoView::OnAdd()
{
    m_pSet->AddNew();
    updateMode=TRUE;
    m_pSet->m_Title="";
    m_pSet->m_ISBN="";
    m_pSet->m_Pages=0;
    UpdateData(FALSE);
}

BOOL CDbdemoView::OnMove(UINT nIDMoveCommand)
{
    if (updateMode)
    {
        if (!UpdateData())
            return FALSE;
        TRY
        {
            m_pSet->Update();
        }
        CATCH(CDBException, e)
        {
            AfxMessageBox(e->m_strError);
            return FALSE;
        }
        END_CATCH

        m_pSet->Requery();
        UpdateData(FALSE);
        updateMode= FALSE;
        return TRUE;
    }
    else
    {
        return CRecordView::OnMove(nIDMoveCommand);
    }
}

void CDbdemoView::OnCancel()
{

```

```
        m_pSet->CancelUpdate();
        m_pSet->Requery();
    }

void CDbdemoView::OnUpdateAdd(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(!updateMode);
}

void CDbdemoView::OnUpdateCancel(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(updateMode);
}
```

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Examining The Example

If you open the example program and bring up Class Wizard, you'll find that the member variable associations in the view are a bit unusual looking (see Figure 10.5). Most of them have an arrow in front of them. This is because the data resides in the record set object instead of in the view. You can also select the record set and see which columns in the database are bound to variables in the record set (see Figure 10.6).



Figure 10.6 Connecting the record set.

The record set has a function named **GetDefaultConnect**. This is where the program gets the connect string that it uses to open the database. App Wizard writes this function so that it returns something appropriate for the database that you select at design time. However, you may need to alter this function before you deliver your program or if you change databases. You could also change it if you wanted to provide a user interface for selecting a database.

The code required to add a record (see Listing 10.2) is highly dependent on how you want to implement this feature. The example program uses a menu item to add a blank record and then calls **Update** when you move off the new record. Some databases won't immediately show the new record, so the code calls **Requery**. This has the side effect of resetting the display to the first

record. Not all databases require this.

You might write code to add records differently. For example, you could automatically add a record when the user moves past the end of the database (using the **OnMove** event).

The program also shows you an example of working with a computed field. The program calculates the total count of books from two different fields. The view maps an ordinary variable (**m_copies**) to a static control. Unfortunately, Class Wizard only allows you to map strings to static controls, so the data must be in string form. If you wanted to maintain the control's contents as an integer, you could consider using a read-only edit control.

The problem with a computed field is when and where to update it. A good place is the **DoDataExchange** function. The framework calls this function to make sure that the controls are up to date, so the record set data should be good at this point. Be sure to add your code before the Class Wizard comments so that the data you place in the variable will appear in the control. You can see this in Listing 10.2.

There are many ways to improve this simple application. For example, you could set it to automatically insert new records at the end or to return to the insertion spot after adding or deleting a record. Even this simple example is useful, however. It could hardly be easier to build a database program using MFC.

Summary

MFC makes it easy to access databases through the **CRecordset** object. Even if you are writing batch processing programs (that is, a program with no user interface), you may be well served by using MFC and **CRecordset** (or **CDaoRecordset**).

If you do need a form-based user interface, the **CRecordView** class is extremely useful. It works like a form view, but Class Wizard can connect database fields (from a record set) directly with fields in the view.

Like most programming jobs, you can make a database program as difficult to write as you want. You can add bells and whistles to your heart's content. However, at the core, you'll usually use the wizard-generated code and write very little yourself.

Practical Guide to MFC And Databases

- Starting A Database Application
- Selecting ODBC Or DAO
- Setting Up A Data Source
- Binding Database Fields To Recordset Variables
- Binding Recordset Variables To Controls
- Deleting Records

- Adding And Updating Records
- Working With Computed Fields

MFC makes writing database applications easy. Once you learn some of the terminology involved, you won't have much work to do—because the App Wizard does it for you!

Starting A Database Application

To start a database application, run App Wizard and select one of the database options in step two. If you want to do all of the work yourself (which might be a good idea if you are doing batch processing, for example), you can select the second option to force the App Wizard to add only headers to your projects. (The first option disables database support and is the default.)

The third and fourth options allow you to have App Wizard create a record set and a record view for you. The record set's purpose is to store a row from the database query or table. The record view (which is similar to a form view) displays the row and allows you to edit it. Select the fourth option only when you want code to handle file open, save, and so forth. Many database programs don't use these. If yours doesn't, select the third option.

If you use the third or fourth option, App Wizard also creates a navigation tool bar and menu items that allow you to move through the record set. You will also specify a data set from this screen. You can select any ODBC data set, or you can elect to use DAO (see the next topic). You can also decide to access a table directly (DAO only), a static snapshot of the data, or a dynamically updateable set of records (a dynaset).

Selecting ODBC Or DAO

MFC supports two ways to work with your database. You can use the older-style ODBC classes or the newer DAO classes. ODBC works well with any database that has an ODBC driver. DAO will also work with these same databases, but it works especially well with Microsoft Access. DAO also allows you to manipulate table structure more easily than ODBC.

Which one should you use? ODBC is probably your best bet. However, if you are only using Access, DAO will work best. If you need to perform tasks such as creating new tables, you certainly should consider DAO.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Setting Up A Data Source

To use a database, you need to configure an ODBC data source using the ODBC icon in your Control Panel. Just add a User or System DSN and follow the directions. You don't have to specify a file to use for the database. If you omit the file name, your program will prompt for a file name each time it starts. If you create a new database, or attach the data set to an existing database, your program will automatically open it without user intervention.

How does your program know which data set to open? The record set object has a member named **GetDefaultConnect**. App Wizard creates this function for you and returns the correct connection string that reflects the data source you select.

Of course, you could change this function to select different data sources using a user interface or other means. You could also have one version of this function for debugging against a local database and use another string for a remote database in the release version.

Binding Database Fields To Recordset Variables

Class Wizard understands record set objects. If you select your record set object in Class Wizard and pick the Member Variables tab, you'll see a list of all the fields in your database as well as the record set variables that represent them. This list will only show fields that were in the database when App Wizard ran. If you don't need all of the variables, you can delete the ones that you don't want. If you change your database structure, you can also refresh the field names using a special button on this screen.

Binding Recordset Variables To Controls

Class Wizard knows that record views may want to bind controls to record set variables. Therefore, you may specify record set variables when working with the view's member variables. Although you could type them in directly, it is usually easier to drop down the list box under the variable name combo box and select from the list.

Note that the association between record set variables and record view controls is very similar to the connection between dialog variables and fields. You must call **UpdateData** to transfer data in either direction. The default handlers usually do this at the appropriate times, but you may have to manually call **UpdateData** occasionally if you are trying to directly manipulate the database or the form fields.

Deleting Records

Deleting records is easy. Position the record set to the correct record, call the **Delete** function, and you're done. The record set may not actually remove the record, but it will mark it as deleted. You can verify this by calling **IsDeleted**. The record view automatically shows a deleted record with the string "-Deleted-" in all of its fields. If you want the record set to not contain the record, call **Requery** after you delete the record.

Adding And Updating Records

If you want to update a record, call **Edit** for the record set while that record is current. When using a record view, this occurs automatically. Then call **Update** after the record set's variables are correct. The new records are added to the database, but they may or may not appear in the record set itself until you call **Requery**. If you want to cancel an update, just call **CancelUpdate**.

Working With Computed Fields

If you are using a record view, you'll often want to calculate values based on other fields and work with those. A handy place to do this is in the **DoDataExchange** routine. This routine exchanges data with the fields, so you know when it executes that something has changed and that the recordset contains the new values.

Here is an example of adding two fields together to produce a third field, excerpted from the example in this chapter:

```
void CDbdemoView::DoDataExchange(CDataExchange* pDX)
{
    CRecordView::DoDataExchange(pDX);
    // set copies up
    m_copies.Format("%d",m_pSet->m_warehouse+m_pSet->m_shelf);
   //{{AFX_DATA_MAP(CDbdemoView)
    DDX_Text(pDX, IDC_COPIES, m_copies);
    .
    .
    .
}}
```

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Chapter 11 Multithreading

Although its classes aren't generally thread-safe, MFC allows you to easily create and synchronize threads. Multithreading is a powerful and sophisticated technique, allowing you to easily perform background tasks. If you are well-prepared and have a good understanding of multiprogramming and Windows, you won't have a difficult time working with threads in MFC.

Programmer's Notes...

Have you ever noticed that doctors seem to rush you in and out of their office? You wait for an hour in the waiting room, and then they finally call you inside. You think that you're all set to go, but no, now you have to wait in the little room. The little room doesn't even have any good magazines, just a copy of *People* with the cover half torn.

Just when you've finally found something to read, the doctor breezes in, thumps you, pokes you, prods you, scribbles on a piece of paper, mumbles a few phrases, and blows right out the door. Now I know that doctors are busy, but I wish that they would be more like the doctors on TV. It seems like TV doctors only have one important patient at once.

The real doctors that I know always have a long list of patients they're treating at one time (unless they aren't any good, of course). Like most people, they have to juggle many things at one time.

Computers are great at juggling multiple tasks. Although it seems as though computers run many programs at once, this is just an illusion. The programs all share the available processors (often just one processor, in fact). The computer shuffles the programs so quickly that it gives the appearance that all of the programs are running at once.

Older operating systems, such as Windows 3.1, didn't prevent programs from interacting. Programs were able to communicate with each other rapidly. However, it also meant that the system was prone to crashing when one program interfered with one or more others. More modern systems, such as Windows NT and Windows 95, attempt to make it difficult for programs to interact. This makes for a more robust system. However, if programs need to communicate, this strict isolation is a hindrance.

When you write several pieces of code that should execute independently while still interacting heavily with each other, you need to consider making each piece of code part of the same program. You simply put each piece of code in a separate thread.

Threads Vs. Processes

Many traditional operating systems schedule multiple processes. A process, in general, is an instance of a program. Like most traditional operating systems, Windows creates a process for each program that you execute. However, the process is not what executes. The process owns memory, open files, and open sockets, along with a host of other resources. It also contains a thread. This thread is essentially an image of the processor registers. Therefore, the thread contains the information about where the program is executing code (the program counter) and the thread's local variables (through the stack pointer and general purpose registers). Windows schedules threads, not processes, to execute.

As long as there is one thread per process, it's hard to see the significance of this. However, when the main thread creates other threads, things become interesting. You can distribute your program's processing among different threads. Of course, any new threads can also create more new threads. The threads share global variables and resources in the process, but they have their own execution point and local variables.

Problems With Threads

Processes are isolated from one another by the operating system. However, within one process, threads can freely interact with each other. While this is a great strength, it also makes it easy for one thread to crash another thread in the same process. None of the protection afforded to processes applies to threads.

Another problem with threads arises with the need to share resources. Suppose you have two threads that want to write to an error file on the disk. You can't have them both write at the same time because the output will become jumbled. Somehow, you have to arbitrate their access to the shared resource.

Your first thought might be to use a global variable as a flag. Each thread would test the flag to determine whether or not it was clear. If the flag is clear, the thread sets it and proceeds to access the file. This sounds good in theory, but there's a problem. On a single-processor system, Windows may preempt the threads at any time to allow other threads to execute. Imagine that the first thread reads the flag and is then preempted. Then imagine that while the first thread is waiting to execute, the second thread takes control and also reads the flag. Noticing that the flag is clear, the second thread sets it and starts writing to the file. Eventually, Windows preempts the second thread and returns to the first one. Because the first thread already read the flag, it also notices that the flag is clear (it isn't, but it was when the thread examined it). So it sets the flag and starts writing to the file.

The result, of course, is chaos. Even if the threads are executing on multiple processors, you can have the same problem depending on timing. The answer is to have some way to test and set a flag so that Windows can't preempt the thread in the middle of the operation. Fortunately, Windows and MFC provide such operations.

Another problem that's common to multitasking and multithreading systems is deadlock (or deadly embrace). Suppose that two threads want to open two files (such as A.DAT and B.DAT) for exclusive access. If the first thread has file A.DAT open, and the second file has file B.DAT open, then neither thread can acquire the resources it needs. If both continue trying to open the file they need and never release the other file, the threads will be deadlocked.

There are a variety of well-understood solutions to this problem. For example, if the threads would release the file they have open if they can't acquire the other one, one would eventually get both files. Of course, if both threads opened the files in the same order (A.DAT first, then B.DAT, for example), there wouldn't be a problem. An access flag (similar to the one used to protect the error log in the previous example) could also provide a solution.

None of these problems are unique to Windows. You can learn more about them in any standard operating systems textbook. However, because many Windows programs don't exploit multithreading (or even multitasking), some programmers are not accustomed to keeping these problems in mind.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Threads And MFC

MFC uses the **CWinThread** object to represent all threads. It even uses an object derived from **CWinThread** to represent your program's main thread. If you look closely, you'll see that **CWinApp** (the class that represents your program), is derived from **CWinThread**.

If you need a thread that has its own event loop, you'll also have to derive a class from **CWinThread**. You can override **InitInstance** and **ExitInstance** to handle the thread's initialization and termination. If you don't need an event loop (that is, you won't use any windows except, perhaps, modal dialogs and dialog boxes), then you can use **CWinThread** just as it is. In either case, you don't directly create a **CWinThread** object, you use **AfxBeginThread** instead.

Creating An MFC Worker Thread

The simplest form of thread is a worker thread. These threads execute an ordinary global (or static member) function independently. The **AfxBeginThread** call takes several arguments in this case:

- The name of a function that returns a **UINT** and takes an **LPVOID** argument. Even if you don't care about the return value or argument, you still have to write the function as though you did. The function can't be a proper member variable. It can be a static member of a class, or you can just use a global function.
- A **void** pointer argument to pass to the function.
- The priority level for the thread.

- The stack size for the thread.
- A zero if you want the thread to run right away, or use **CREATE_SUSPENDED** to begin the thread in a suspended state.
- The security attributes for the thread. This is only useful for Windows NT (Windows 95 doesn't support security). The default is **NULL**, which is normally what you want anyway.

You must supply the first two parameters, but the other four are optional.

Once you call **AfxBeginThread**, it will return a pointer to the **CWinThread** object that it creates. You can use this object to manipulate the thread, as you will see shortly.

Your thread will exit when you return from the function or when you call **AfxEndThread**. Other threads can read the return value (or the argument to **AfxEndThread**) if they know how.

Creating An MFC User-Interface Thread

AfxBeginThread comes in two flavors. The first one creates worker threads (as you already know). The other version creates a class that you derive from **CWinThread**. This version is useful when you want to create a thread that has a user interface. In other words, if your thread requires an event loop, you need to derive a new class from **CWinThread**.

The arguments for this version of **AfxBeginThread** are similar to the worker thread version. However, instead of a function name and an argument, you supply the runtime class of your custom thread object (usually using the **RUNTIME_CLASS** macro). All of the other arguments remain the same.

A good place to find an example of deriving a class from **CWinThread** is the MFC source code. After all, **CWinApp** is just a thread (sure, it's the first thread, but it's still just a thread).

At the very least, you'll need to override **InitInstance** to create your main window. Don't forget to set **m_pMainWnd** to point to the thread's main window. App Wizard takes care of that automatically when it creates your application object, but when you do your own class, you need to be aware of these details.

You can derive your new class using Class Wizard. **AfxBeginThread** returns a pointer to a **CWinThread** object, but the object is actually whatever type you specified. If you need to work with any custom members, you'll need to cast the pointer to the appropriate type.

To terminate a thread with an event loop, call **PostQuitMessage**. The argument to this call becomes the return value of the thread that other threads can read.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Manipulating Threads

CWinThread has several useful members (see Table 11.1). By default, the **m_bAutoDelete** flag is **TRUE**. That means that when your thread terminates, MFC automatically destroys the object. This is handy if you are just creating a thread to do independent work. However, there is a problem if you want to interact with the object.

Table 11.1 CWinThread members.

Member	Definition
CreateThread	Creates the thread; use AfxCreateThread instead
m_pMainWnd	Main window for thread
m_pActiveWnd	Current active window
m_bAutoDelete	If TRUE, delete thread object when thread terminates; TRUE is the default
m_hThread	Windows thread handle (can also cast CWinThread to HTHREAD)
m_nThreadID	Thread ID
GetThreadPriority	Gets thread's priority level
SetThreadPriority	Sets thread's priority level
SuspendThread	Suspends thread
ResumeThread	Resumes thread (call once for each call to SuspendThread)

PostThreadMessage	Posts a message to the thread's event queue
InitInstance	Override to provide initialization code for the thread
Run	Override to customize the thread's message loop
PreTranslateMessage	Filters messages and processes accelerator keys
PumpMessage	Low-level message pump
OnIdle	Called when no messages are pending
IsIdleMessage	Checks for special MFC idle messages
ExitInstance	Override to provide code on thread termination
ProcessWndProcException	Processes unhandled exceptions
ProcessMessageFilter	Filters messages
GetMainWnd	Returns main window

Suppose that you have a user-interface thread object derived from **CWinThread**. The new thread's job is to read a command from a serial port. There are many ways you could communicate this command to the rest of the program. However, for the sake of illustration, assume that you decide to put a member variable in the thread object that contains the command string.

Assuming that everything goes well, this isn't a bad idea. The main program starts the thread using **AfxBeginThread**, casts the pointer returned to the correct type, and accesses the variable to find out the current command. You would need some other variables, too. For example, you'd probably want a flag to indicate a valid command and a member to clear the command and wait for the next one.

The problem occurs if the thread terminates unexpectedly. Then the object is no good and the pointer is invalid. This will cause major problems.

So the answer, obviously, is to set **m_bAutoDelete** to **FALSE**. But there's still a problem. What happens if the thread terminates almost immediately? In other words, what happens if by the time you try to set the flag, the object is already gone? You can't assume that the object will be valid for any length of time.

The solution is to create the thread in a suspended state. Remember the **CREATE_SUSPENDED** flag you can pass to **AfxCreateThread**? You can pass this flag and MFC will create the thread in a dormant condition. Then you can set the **m_bAutoDelete** flag, or do anything else you want to do, before calling **ResumeThread** to start execution.

Speaking of **ResumeThread**, you should know that **SuspendThread** and **ResumeThread** have to balance. If you call **SuspendThread** three times, the thread will not run again until you call **ResumeThread** three times also.

Learning The Return Value

Another way to communicate with the other threads is to return a value on

termination. You can set a worker thread's return value by returning it from the controlling function or passing it to **AfxEndThread**. You can set a user-interface thread's return value using **PostQuitMessage**.

Your program will need a thread's handle if it is to read another thread's return value. This is available in the **CWinThread** object (the **m_hThread** variable). You use the handle in a call to **::GetExitCodeThread**. This function requires the handle as the first argument and a pointer to a **DWORD** result as the second argument. If the thread is still active, the function places the constant **STILL_ACTIVE** in the **DWORD** result. Otherwise, the function sets the **DWORD** to reflect the exit code of the specified thread.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Synchronizing Threads

There is another problem that occurs when you try to make threads communicate. Although all of the threads in one process can access the same variables, they may conflict over the use of those variables.

For example, you may have a variable you wish to use as a flag. This flag controls access to a file so that the threads won't all write to the file at once. In theory, before each thread writes to the file, it has to make sure the flag is clear and then set the flag.

The problem here is that the threads may run at the same time (or, at least, appear to run at the same time). Now, let's say that one thread reads the flag and then Windows preempts it. While it is waiting to run again, another thread reads the flag, finds it clear, and then sets it. When the first thread resumes execution, it also sees the flag was clear, sets it, and begins to write to the file. Now both threads think they have the right to use the file.

To solve this problem, you need some way to test and set the flag in one uninterruptible operation. Luckily, you can do this in Windows with two primitive calls: **InterlockedIncrement** and **InterlockedDecrement**. However, you won't use these calls too frequently because Windows provides a higher-level method in which to accomplish the same tasks. In particular, Windows provides synchronization objects that MFC wraps using objects derived from **CSyncObject**.

In traditional Windows programming, you use the synchronization objects directly in order to arbitrate resource access among threads. However, with

C++, it is a better design to create thread-safe classes. Thinking again about the case where two threads want to share a file, it would be better to have a class that has a member function called **WriteString** (for example). This function would write to the file, but only after making sure no other threads were writing at the same time. Then the threads don't really need to understand the synchronization objects directly.

Synchronization objects can exist in one of two states: signaled and unsignaled. Exactly what this means depends upon the object, but in general, it refers to the availability of a resource. When the object is in the signaled state, the resource is available. If it's not signaled, the resource isn't available. Threads can find the state of an object. If the object is not signaled, the thread can choose to wait for a certain period of time, or indefinitely, to determine whether the object transitions to the signaled state.

Most synchronization objects can have names. You use the name when you want to share an object between multiple threads in different processes. Of course, you can name the object and use it from within one process, but in this case, it's easier to use a single variable that all of the threads share.

If you attempt to create an MFC synchronization object using a name that is already in use, you will simply open the existing object. Of course, creating an object with no name always constructs a new object.

Types Of Synchronization Objects

Windows provides several different synchronization objects. Each of the following is useful in certain circumstances:

- **CEvent**—This is a flag that you can set and test in a thread-safe fashion. The event can be of several types (you specify the type during creation). A manual event allows you to set and reset the flag at will. An automatic event resets automatically after one thread determines it is signaled.
- **CMutex**—A mutex is a special kind of flag that implies ownership of a resource. Only one thread can own the mutex at one time. Once a thread notices the mutex is signaled, the mutex will appear unsignaled for all other threads. However, the mutex continues to show a signaled state to the original thread. This allows the thread to acquire the mutex multiple times. It also means the thread has to release the mutex as many times as it acquired it.
- **CSemaphore**—A semaphore is a shared counter. As long as the counter is greater than zero, the semaphore is signaled. Each time a thread asks to acquire the semaphore, the count decreases by one. Once the count drops to zero, the semaphore becomes unsignaled until one or more threads release it and the count rises again. This is useful in cases where you want to limit the number of threads doing something at a particular time (perhaps accessing a database).
- **CCriticalSection**—A critical section is very similar to a mutex except that it is only usable with threads that belong to the same process. This simplified mutex is useful when you don't need cross-process

capabilities and you want better efficiency than a mutex affords.

In addition to the synchronization objects, MFC uses two special objects, **CSingleLock** and **CMultiLock**, to check the synchronization object's status. These correspond to the Windows API calls **WaitForSingleObject** and **WaitForMultipleObjects**. You can use one of the lock objects to determine if a synchronization object is signaled.

As an example, consider a mutex. You create a **CMutex** object in your code and a **CSingleLock** object. You'll pass a pointer to the **CMutex** into the **CSingleLock** constructor. When you want to acquire the mutex, you call **CSingleLock::Lock**. When the **CSingleLock** object goes out of scope (or is otherwise destroyed), it automatically calls **CSingleLock::Unlock**; this releases the mutex. Of course, you could call **Unlock** directly, if you needed to do so. **CMultiLock** works the same way, but you pass an array of objects into the constructor, not just one. Then the lock will lock one object that is available.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Alternatives To Threading

Although threading is very useful for background processing, it isn't the only option. If you have operations that seldom interact, you could always write them as separate programs. Windows provides many methods for such programs to communicate, although not at the high bandwidth afforded to threads.

Another option you should consider in some cases is idle processing. Remember, your application object has an **OnIdle** member that MFC calls when there are no messages pending for the thread. Of course, you have to be careful not to do too much during idle time processing. It wouldn't be a good idea to wait for serial port input during **OnIdle**, for example. However, for background printing or similar tasks, it is a viable option. It also has the advantage of working on 16-bit Windows, which doesn't support threads.

Another way to avoid multithreading is by using overlapped I/O. MFC does not directly support overlapped I/O. Also, under Windows 95, overlapped I/O is severely limited. The idea behind overlapped I/O is that you can request input or output on a file (or device) and not have to wait for it to complete.

In order for this to work, you must use **::CreateFile** with the **FILE_FLAG_OVERLAPPED** bit set. It is worth noting that although the call is **::CreateFile**, it is possible to simply open an existing file with it. A better name for the function would be **::CreateFileHandle**, since it always creates a file handle.

Once the file is open for overlapped I/O, you can make the usual Windows

API calls (like **::ReadFile** and **::WriteFile**). Each of these functions takes a pointer to an **OVERLAPPED** structure. This pointer contains the file offset you wish to work with, and possibly an event handle (described soon). If the pointer is **NULL**, then the functions work as they always do. However, if you pass a pointer to an **OVERLAPPED** structure, one of two things can occur.

The first case is that the I/O can complete immediately. Then the call returns success and the operation is complete. However, if the call would cause the thread to block (in other words, wait for the I/O), Windows returns an error. A call to **::GetLastError** will return **ERROR_IO_PENDING**. This isn't really an error. It only means that the I/O is completing asynchronously, allowing the thread to continue processing.

To determine when the I/O completes, you have several options. First, you could call **::GetOverlappedResult**. You can vary the parameters to this function so that it causes the thread to wait indefinitely for the I/O to complete, or you can simply test to see if it did. If you are only testing,

::GetOverlappedResult may return an error and a subsequent call to **::GetLastError** will return **ERROR_IO_INCOMPLETE**. That means that you will have to try again later. Obviously, if **::GetLastError** returns some other error after **::ReadFile**, **::WriteFile**, or **::GetOverlappedResult**, you have a genuine error.

A better idea is to specify an event synchronization object in the **OVERLAPPED** structure. Then Windows will signal the event when the I/O completes. You can test for this in all the usual ways you work with events. For example, suppose that you needed to read data from two serial ports and display the data in a window. Your program might start two overlapped reads, one on each port and each with a separate event. Then it could use a **CMultiLock** object to wait until one or both ports signal that they have data.

Another way to deal with overlapped I/O is to use **::ReadFileEx** and **::WriteFileEx** (only useable on Windows NT). These calls take a pointer to an **OVERLAPPED** structure (which can't be **NULL**) and a pointer to a function. When the I/O completes, Windows calls the function to process the data. This is ideal for schemes in which data mysteriously appears in a circular buffer for the rest of the program to use. However, it is useless under Windows 95. A general solution is to perform normal I/O in another thread and then have that thread fill in the circular buffer. Which approach is best? It depends. The overlapped I/O method is probably a bit more efficient. On the other hand, if you need to run on Windows 95, you'll have to use a separate thread.

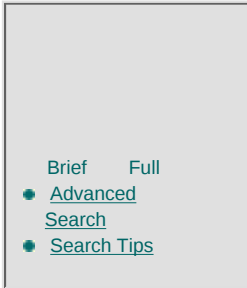
Windows 95 puts other limitations on overlapped I/O. For example, Windows 95 can only perform overlapped I/O on serial devices and certain files opened with the **::DeviceIoControl** call (which opens devices directly).

So while threads can do many things, there are often alternatives to consider. Still, threads are a powerful mechanism, and you should certainly use them where appropriate. Just don't forget that there is more than one way to achieve the same effect, and you should carefully consider your options.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

An Example Program

Have you ever wanted to call an MFC user interface from some other program? It is very tempting to design a DLL that has a single entry point that starts what is essentially an entire MFC program.

There are several ways you might accomplish this, but using a thread is an easy and useful method. Your DLL simply spawns a new thread, and this thread corresponds closely with an ordinary MFC application.

In Figure 11.1, you'll see a console application that computes a large quantity of even numbers. Because this may take a while, the application places a dialog box on the screen that allows the user to cancel the operation. Obviously, the console application can't use a modal dialog box because it must continue processing. However, because the console application doesn't have an event loop, it can't use a modeless dialog, either.

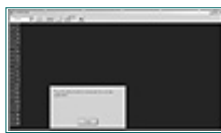


Figure 11.1 The example program.

The solution is to define an MFC DLL that handles the dialog box. This presents two challenges. First, the console application must have some way to start the MFC portion. Second, the console application and the MFC abort dialog must communicate. In particular, the console program must detect that the user wants to abort and the MFC dialog has to close down when the console program no longer needs it.

Making an exported C function is not hard (see Chapter 7 for more on DLL functions). This function can create a **CWinThread**-derived class that starts the dialog box. Listing 11.1 shows the main DLL code (including the exported function **AbortBox**). Listing 11.2 contains the thread code, and Listing 11.3 shows the dialog box. Listing 11.4 shows the simple console program that uses the MFC DLL.

Listing 11.1 The Abort DLL.

```
// abortdll.cpp : Defines the initialization routines for the DLL.
//
```

```
#include "stdafx.h"
```

```

#include "abortdll.h"
#include "abortthread.h"
#include "abortdlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//
// Note!
//
// If this DLL is dynamically linked against the MFC
// DLLs, any functions exported from this DLL which
// call into MFC must have the AFX_MANAGE_STATE macro
// added at the very beginning of the function.
//
// For example:
//
// extern "C" BOOL PASCAL EXPORT ExportedFunction()
// {
//     AFX_MANAGE_STATE(AfxGetStaticModuleState());
//     // normal function body here
// }
//
// It is very important that this macro appear in each
// function, prior to any calls into MFC. This means that
// it must appear as the first statement within the
// function, even before any object variable declarations,
// as their constructors may generate calls into the MFC
// DLL.
//
// Please see MFC Technical Notes 33 and 58 for additional
// details.
//

////////////////////////////////////
// CAbortdllApp

BEGIN_MESSAGE_MAP(CAbortdllApp, CWinApp)
//{{AFX_MSG_MAP(CAbortdllApp)
// NOTE - the Class Wizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CAbortdllApp construction

CAbortdllApp::CAbortdllApp()
{
    // TODO: add construction code here
    // Place all significant initialization in InitInstance
}

////////////////////////////////////

```

```

// The one and only CAabortdllApp object

CAabortdllApp theApp;

extern "C" {
    __declspec(dllexport) BOOL AbortBox(char *text,char *event);
}

__declspec(dllexport) BOOL AbortBox(char *text, char *event)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
    CAabortThread *t=(CAabortThread *)
        AfxBeginThread(RUNTIME_CLASS(CAabortThread),THREAD_PRIORITY_NORMAL,
            0,CREATE_SUSPENDED);

    if (!t)
    {
        ::MessageBox(NULL,"Can't spawn thread",NULL,MB_OK);
        return FALSE;
    }
    // Set up parameters
    t->text=text;
    t->eventname=event;
    t->ResumeThread();
    return TRUE;
}

```

Listing 11.2 The Abort thread.

```

// AbortThread.cpp : implementation file
//

#include "stdafx.h"
#include "abortdll.h"
#include "AbortThread.h"
#include "Abortedlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CAabortThread

IMPLEMENT_DYNCREATE(CAabortThread, CWinThread)

CAabortThread::CAabortThread()
{
}

CAabortThread::~CAabortThread()
{
}

BOOL CAabortThread::InitInstance()

```

```

{
    CAbortDlg dlg;
    m_pMainWnd=&dlg;
    event=new CEvent(FALSE, TRUE, eventname);
    dlg.m_text=text;
    dlg.Thread=this;
    dlg.DoModal();
    delete event;
    return FALSE;
}

int CAbortThread::ExitInstance()
{
    // TODO: perform any per-thread cleanup here
    return CWinThread::ExitInstance();
}

BEGIN_MESSAGE_MAP(CAbortThread, CWinThread)
    //{AFX_MSG_MAP(CAbortThread)
        // NOTE - the Class Wizard will add and remove mapping macros here.
    //}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////
// CAbortThread message handlers

```

If you study the code, it is amazingly unremarkable. The dialog is just a regular dialog box. The **CAbortThread** class (created with Class Wizard) is very closely related to an ordinary **CWinApp** object and serves the same purpose. The DLL code is certainly ordinary.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

BROWSE
BY TOPIC

The only special part of the **AbortBox** code is the communications between the main program and the abort thread. The calling program creates an event using any name it chooses. It then passes this name as one of the arguments to **AbortBox**. The abort thread also opens the event.

The event serves two purposes. First, if the user aborts, the MFC code signals the event. This allows the creating thread to know when an abort occurs. Second, if the original code signals the event, the MFC code takes that as a sign to shut down.

Listing 11.3 The Abort dialog box.

```
// AbortDlg.cpp : implementation file
//

#include "stdafx.h"
#include "abortdll.h"
#include "AbortThread.h"
#include "AbortDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////
// CAbortDlg dialog

CAbortDlg::CAbortDlg(CWnd* pParent /*=NULL*/)
: CDialog(CAbortDlg::IDD, pParent)
{
   //{{AFX_DATA_INIT(CAbortDlg)
    m_text = _T("");
   //}}AFX_DATA_INIT
}
```

```

void CAbortDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAbortDlg)
    DDX_Text(pDX, IDC_TEXT, m_text);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAbortDlg, CDialog)
    //{{AFX_MSG_MAP(CAbortDlg)
    ON_WM_TIMER()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////
// CAbortDlg message handlers

void CAbortDlg::OnOK()
{
    Thread->event->SetEvent();
    CDialog::OnOK();
}

BOOL CAbortDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    SetTimer(1,500,NULL); // Timer every 1/2 second
    return TRUE; // return TRUE unless you set the focus to a control
                // EXCEPTION: OCX Property Pages should return FALSE
}

void CAbortDlg::OnTimer(UINT nIDEvent)
{
    CSingleLock lock(Thread->event,FALSE);
    if (lock.Lock(0))
    {
        CDialog::OnOK();
    }
}

```

Other than the event and the custom **CWinThread** object, this could be any MFC program. Because **AbortBox** is just an ordinary exported function, you could call it from almost any program. Of course, **AbortBox** is overly simple, but you can easily apply the same concepts to more difficult tasks.

Listing 11.4 A console program to test the DLL.

```

#include <stdio.h>
#include <windows.h>

// DLL Function
extern "C" {
    BOOL AbortBox(char *string,char *eventname); // DLL function
}

```

```

void main()
{
    HANDLE event;
    unsigned int ct=0;
    event=CreateEvent(NULL,TRUE,FALSE,"CONABORTEVENT");
    AbortBox("Press the Abort button to terminate the console application"
        ,"CONABORTEVENT");
    while (WaitForSingleObject(event,0)==WAIT_TIMEOUT&&ct<100000)
    {
        printf("%d\n",ct);
        ct+=2;
    }
    if (WaitForSingleObject(event,0)==WAIT_TIMEOUT)
    {
        SetEvent(event); // make sure Abort box gets the message
        ::MessageBox(NULL,"Done","Operation Complete",
            MB_OK|MB_SETFOREGROUND);
    }
    // No message if aborted by abort box
    CloseHandle(event);
}

```

Summary

Multithreading is a powerful, but sophisticated, technique. While it isn't for every program you write, when you need it, there is nothing else like it. Threaded programs can easily perform tasks and do I/O seamlessly in the background. Threads also find wide application in simulation, where each thread represents a simulated entity.

Although MFC makes it fairly easy to create and synchronize threads, it also presents some problems. Most MFC classes are not thread-safe. If you want more than one thread to manipulate the same object, you'll have to arbitrate that access yourself. Worse, objects that contain maps (like **CWnd** objects, for instance) use thread local storage. This means that objects with maps don't work properly except in the thread that created them. Be prepared to deal with these problems when you start adding threads to your program.

Practical Guide to Multithreading

- Creating A Worker Thread
- Creating A User-Interface Thread
- Terminating A Thread
- Making Windows Appear On Top
- Making Message Boxes Appear On Top
- Preventing Autodestruction Of Threads
- Creating A Suspended Thread
- Learning The Return Value
- Types Of Synchronization Objects
- Waiting For A Synchronization Object
- Waiting For Multiple Synchronization Objects
- **Using OnIdle**

Threads are not difficult to work with in MFC. The difficulty usually comes from designing a single coherent program that uses multiple threads. Armed with a good understanding of multiprogramming and the arsenal of Windows synchronization objects, you can get multithreaded programs to work.

Creating A Worker Thread

An MFC thread that does not require a message loop is a worker thread. A worker thread can do any type of processing, and it may use modal dialogs and dialog boxes. However, without a message loop, you can't create modeless dialogs or regular windows.

Worker threads are easy to create. You only need to write a global or class static function that returns a **UINT** and takes a **void** pointer as an argument. You call **AfxBeginThread** with the name of the function and the argument you want to pass. You may optionally supply the priority level, the stack size, a creation flag, and a security attribute. The creation flag is 0 if you want the thread to start right away or **CREATE_SUSPENDED** if you want the thread not to run until you call **ResumeThread**.

MFC will create a **CWinThread** object and return a pointer to it as the return value to **AfxBeginThread**. Your function will execute in its own thread. It will have unique local variables, but it will share global variables with other threads in the same process.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Creating A User-Interface Thread

If your thread requires a message loop (that is, it will have ordinary windows or modeless dialogs), you will need to use Class Wizard to derive a new class from **CWinThread**. In this class, you'll provide an override version of **InitInstance** that creates your main window (**m_pMainWnd**) and do other initialization (just like you would in a **CWinApp** object).

Terminating A Thread

To terminate a worker thread, you can return from the controlling function or call **AfxEndThread**. You might find **AfxEndThread** useful if you want to exit after calling other functions. In either event, you can specify the thread's exit value, which other threads can read. Notice that you must make the call to **AfxEndThread** from within the thread you wish to terminate.

You can terminate a user-interface thread by calling **PostQuitMessage**. The argument to this call becomes the exit code that other threads can determine.

Making Windows Appear On Top

When you are working with threads, you will find that the thread that owns the focus window for a process has a special status. It is sometimes referred to as the foreground thread. When this thread creates a window, it will appear on top, whereas other threads in the same process don't normally create windows at the top of the window stack. Instead, the windows appear below other windows.

If you create windows with MFC, MFC normally calls **SetForegroundWindow** to make certain that the window pops to the top. However, if you create a window on your own, you'll need to call **SetForegroundWindow** yourself.

Making Message Boxes Appear On Top

Placing a message box in the foreground presents a special problem. MFC attempts to make message boxes always pop to the top, but what about ordinary non-MFC boxes? You can't call **SetForegroundWindow** because your code is not running while the message box is valid. To solve this problem, Microsoft provides the **MB_SETFOREGROUND** flag. If you pass this flag to a message box call, it will force the message box to the top (probably by calling **SetForegroundWindow** internally).

Preventing Autodestruction Of Threads

By default, when an MFC thread terminates, it destroys its own thread object. This is a handy way to make sure that threads don't leak memory. However, if you need to obtain data from the thread object when the thread is no longer executing, you have a problem.

The solution is simple. Just be sure that the **m_bAutoDelete** flag in the thread object is set to **FALSE**. This prevents the automatic destruction. Of course, that means that you will have to free the memory yourself when you are done with the object.

There is one pitfall to setting this flag. If the thread terminates quickly, the object may already be gone by the time you try to set the flag. The answer is to create the thread in suspension, set the flag to **FALSE**, and then start the thread's execution. See the next topic for details on how to do this.

Creating A Suspended Thread

The creation flag you send to **AfxBeginThread** is usually set to zero. This causes MFC to begin executing the thread immediately. However, there are times when you might wish to delay the start of a thread. In this case, you should specify **CREATE_SUSPENDED** as the flag argument.

There are several reasons why you might want to create a thread in the suspended state. The most common reason is the need to set initial conditions in the thread object (see the example in Listing 11.2, for instance). You will also want to start a thread later if you don't want it to delete its own thread object (see the previous topic).

To start a thread that was created dormant, call **ResumeThread**. The thread then begins executing the function you provided (worker threads) by calling the appropriate functions in your **CWinThread**-derived object (user-interface threads).

You can always suspend a running thread by calling **SuspendThread**. If you call **SuspendThread** multiple times, you'll need to call **ResumeThread** the

same number of times before the thread will resume execution.

Learning The Return Value

To learn the return value of a thread, you need its thread handle (the **m_hThread** member of the thread object). You also need to be sure that the object is valid when you attempt to read this member.

Armed with the thread handle, you can call the Windows API function **::GetExitCodeThread** with a pointer to a **DWORD**. This call will set the integer to the exit code (as returned from the worker function, or passed to **AfxEndThread** or **PostQuitMessage**) or **STILL_ACTIVE** if the thread is still running. Sometimes, the fact that the thread is still running is all the information you need.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Types Of Synchronization Objects

There are several types of synchronization objects that Windows provides. You can use these to coordinate multiple threads so that they cooperate with each other. Most of the synchronization objects (all of them but the critical section) will work with threads in the same process or in different processes. MFC provides wrappers (derived from **CSyncObject**) for each type.

The **CEvent** object is little more than a flag that your multiple threads can share. If the flag is set, the event is signaled. Otherwise, it is left unsignaled. You can create a manual event, which requires you to explicitly set and reset the flag, or you can create an automatic event. Automatic events reset themselves when one thread determines the flag is set. There is also a call (**PulseEvent**) that allows any thread waiting for a manual event to proceed, and then resets the event.

A **CMutex** object is a flag that indicates ownership of a resource. Once a thread perceives that the mutex is signaled, all other threads will see it as unsignaled until the first thread releases it.

The **CSemaphore** manages a count. When a certain number of threads have acquired the semaphore (that is, found it in the signaled state), the semaphore becomes unsignaled. This is useful if you want to restrict an operation (such as a database query or network request) to a certain number of threads at one time.

The last type of synchronization object is the **CCriticalSection**. A critical section is similar to a mutex, but not as flexible. The advantage to a critical

section is that it is more efficient than a mutex. However, the critical section is only usable in the threads of a single process. Two threads in different processes can't use a critical section. They can use a mutex, of course.

If you are using a synchronization object in the threads of one process, there is no need to name the object. However, if you are using them between processes, it is easiest to name the objects. All of the objects share a common name space (in other words, you can't name a mutex and a semaphore the same thing). All threads using the same name for an object are sharing the actual object.

None of these objects are very difficult to use. Often, the major obstacle is deciding which one to use and how to design an architecture that produces the desired result.

Waiting For A Synchronization Object

Synchronization objects may be in the signaled or unsignaled state. MFC programs use a **CSingleLock** object to determine the state of a synchronization object.

To find out if an object is signaled, create a **CSingleLock** object, passing the synchronization object of interest to the constructor. Then you can call **Lock** to find if the object is signaled.

When calling **Lock**, you can specify that you want to wait for the object to become signaled for a certain period of time, or you can wait forever. You can also simply determine the current state.

If the **Lock** call finds a mutex or semaphore signaled, that also changes the state of the mutex or semaphore. To reverse that change, call **Unlock**. This has the effect of releasing the mutex or increasing the count of the semaphore. When the lock object destructor executes, it automatically calls **Unlock**. That means you can often avoid calling it yourself.

Waiting For Multiple Synchronization Objects

Occasionally, you will want to work with more than one synchronization object. For example, you might want to know when any one of four events occur. Or you might want to wait for any one of three mutexes to become available.

In these cases, simply use **CMultiLock** instead of **CSingleLock**. The operation is the same, but you pass an array of objects to the constructor.

Using OnIdle

Sometimes you don't need to use multithreading at all. You can often achieve the same effect by overriding **OnIdle** in your application object (just use Class Wizard). MFC calls this function when there is spare time during message processing.

Of course, you don't want to do anything that will take a long time, or cause

your thread to wait during **OnIdle**. For example, it would probably be a bad idea to read serial port input during **OnIdle**. That task would be a good candidate for multithreading (or overlapped I/O, a sophisticated Windows technique mostly useful for Windows NT). However, for background printing, spell checking, and other similar tasks, idle processing is a viable alternative. You may need to split the work into chunks, doing a small amount on each idle time cycle.

It is very important that you call the base class version of **OnIdle**. MFC uses idle time processing to do many different things that you wouldn't suspect. For example, tool bar states update during idle time. MFC also deletes temporary objects during idle time. If you forget to call the base class, odd things will start happening. It isn't intuitively obvious, for example, why adding idle-time processing stops your tool bars from working properly or causes memory leaks in other parts of your program.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)

Chapter 12

The End Of The Road

The future is like a Valentine's Day candy: The only way to know what it holds is to bite into it—and by then, if you don't like it, it's too late.

Still, with a little forethought, you can bite into the future without getting bitten back.

Programmer's Notes...

Do you have kids? If you do, you know there isn't anything quite like the experience of parenthood. Just getting there is a major ordeal. Takes nine months. It's great fun at the very beginning. Then it isn't so much fun, but it gets more exciting as you anticipate the new arrival. That is, until reality sets in and you are gripped with sheer panic. Eventually, you come to terms with the panic and accept the daily discomfort and indignities. Finally, after a lot of pain, you have a baby. But that's really just the beginning, isn't it?

Writing a book is very much like having kids. It takes roughly the same amount of time. It is also great fun at the beginning, but you go through very similar stages as the project progresses. The intense pain at the end is especially apparent.

Getting back to kids, any parent will tell you that birth is just the beginning. There are diapers and feedings to take care of. You worry that they are sick. You worry that they aren't growing fast enough. You worry that they are

growing too fast. There aren't many logical processes involved in parenthood.

The operative word in the last paragraph is *worry*. If I had to pick one word that describes parenthood, that would be the word. You worry about them all through their life. You worry that they aren't happy. You worry that they aren't doing well in school. You worry that you aren't raising them to have your values. You worry that you worry so much. Sometimes, you even worry that you aren't worrying enough.

Most young people who have small children think that they can't wait for the child to turn some magic age (usually 18 or 21) so they can quit worrying. Well, I have been through all of this (we only have one of our three left at home) and I can tell you that the worry never seems to end. When they are older, you might even worry more, because you don't know where they are or what they are doing most of the time. You worry that they aren't doing well at their jobs. You worry that their marriages aren't going well. Worry, worry, worry.

There are many milestones you mark as a parent, and most of them have to do with decreasing control and increasing worry. Your child's first day of school is always a big one. That first sleepover party also comes to mind. Watching a kid drive off in a car alone for the first time is particularly agonizing (especially if it's your car). Helping a child pack to go away to school is another of these times. I can assure you, it is hard to walk a daughter down the aisle and then sit down to start a whole new set of worries.

Well, books are a little different at this point. Once your book is born, there isn't a whole lot you can do to affect it. But you still worry. You worry that the bookstores won't stock it. You worry that the technology will change radically. You mostly worry that people won't find it useful or enjoyable. But there isn't much you can do. So it's more like worrying about an adult child (if that isn't an oxymoron).

Sure, you can answer questions from readers. With the Web, you can even post updates to your book and make them widely available, which was unheard of not long ago. But for the most part, your book is on its own. Some great books flounder because they were published at the wrong time or for some other seemingly insignificant reason. Some bad books prosper because they just happen to be at the right place at the right time. But usually there is justice—the good books do well and the bad books do poorly.

What about this book? That's for you to decide, and me to worry about. This book is more or less a dump of everything I have learned about MFC over the years. My hope is that you found something useful here, even if it is only a different way of looking at MFC.

The End Of The Road?

Like having kids, programming is a job that never ends. Sometimes people come to me and ask if they (or their kids) should pursue a job in computing. My one question to them is always: "Do you like to read?" I haven't found any other question that better indicates success in programming.

The reason is simple. Programming changes constantly. You can't just go to school, have someone teach you everything you need to know, and then say, "Well, that's that. I'm done learning to program."

Every time I get comfortable with the world, somebody changes everything. I made the transition between Univac mainframes and Data General minicomputers. Then I had to learn about microprocessors. Then I learned to love Unix. Then Windows (perhaps the word love is a bit strong here). Somewhere in between, I gave up C for C++. I learned VB, Delphi, and a few other languages. I switched from the SDK to MFC.

What will catch our interest tomorrow? I don't know. If I did, I'd probably be writing about it instead. Many people are excited about Java. Java is certainly interesting. But the real interesting things are the things we don't even know about yet. Of course, that's something you can't predict. However, I think we can look at how other technology has progressed and see what the future might hold for software development.

Previous	Table of Contents	Next
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Things To Come

Think about the innovations that have transformed our lives. In 1910, who could have predicted the transistor or the personal computer? Even if you did predict their existence, would you have predicted their impact? It is a well-known (and probably true) story that IBM once thought there was a world market for a handful of (six or seven) computers.

A lot of changes, however, are more evolutionary than revolutionary. Look at the ATM machine, for example. One of the biggest problems with travelling used to be making sure you had enough money for your trip. This was especially a problem when your trip lasted longer than planned. No one would take your ordinary check. In those days, the best perk to having an American Express card was that most hotels would cash a small check for you with the card. You could even cash bigger checks at American Express offices.

These days, with ATM machines, who cares about that? I don't even carry checks anymore. I can go nearly anywhere, insert my card into the machine, and draw out large sums of cash. Who could foresee this basic change in life? The ATM itself isn't revolutionary. It is an evolution of the small, inexpensive computer and telecommunications networks. These things are also evolved from other technologies.

ICs Vs. Core Memory

In my mind, the biggest transformation the world has ever seen is the introduction of the IC (integrated circuit), sometimes called a microchip. Mind you, a great part of the population only has a vague idea what an IC is. But it

effects everyone's life every day.

By themselves, ICs are not revolutionary. There are only a handful of things that ICs can do that would be difficult to do without them (and most of those things can be done some other way, anyway). What ICs do is make reproducing circuitry a cheap reproducible process.

Back when core memory was in vogue, computer companies hired people with small hands to thread tiny ferrite cores on wires. There were thousands of these cores packed into a very small area (perhaps 50 square centimeters or so). This is an expensive way to build memory.

Today, a crew of technicians performs a few steps that are very similar to developing a roll of film. When they are done, they have a large silicon disk that will contain hundreds of memory circuits. Each circuit can hold 10 times the data the old core memory could, or maybe even more.

Because it is cheap to make ICs, companies can sell them cheaply. Because of the low price, they can also sell more of them. That means that you can amortize the development costs of an IC over millions, or even billions, of units sold. The upshot to this is that companies can spend man-years of development effort on ICs.

Another interesting thing about ICs is that the companies that manufacture them make sure they are compatible. Usually you can mix Intel microprocessors with Hitachi memory and I/O chips from Texas Instruments. If Hitachi is out of memory chips for some reason, you can go to NEC and get the same part. The NEC part (or a newer Hitachi part) might even be better (but still compatible).

Imagine what this could do for the software industry. What if you could spend three man-years developing a software component knowing that you could sell millions of them? What if you could buy components that had that much effort put into them? Of course, the components would have to easily work with other components from different vendors. With that much development effort involved, you'd expect the components to work flawlessly and be very powerful. And, of course, they would be cheap.

Fantasy? No, you can already see this starting. In one way or another, Visual Basic Extension (VBX) controls, ActiveX controls (OCX), and JavaBeans all strive to provide a framework for this sort of thing. Is it here now? No. Today's components don't work well with other components. They also don't have the standard interfaces required to allow different vendors to build compatible components. But it is a start.

It isn't hard to imagine the programming environment of tomorrow as being little more than a container to place controls. In fact, Visual Basic, Delphi, and Internet Explorer are just about that already.

If you are like me, you are cringing at the idea of point and click programming. Don't! We can all go off to our corners and write the components everyone else will use. Besides, it's almost surely going to happen, so you might as well make the best of it.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Other Resources

Because learning to program is a journey and not a destination, it is important to keep abreast of the changing landscape. The traditional methods are still good: Read books, subscribe to journals, take classes, and go to trade shows. However, the Internet is now the premiere place to get up-to-the-minute information about programming (and pretty much everything else, too, I suppose).

Of course, there are two problems with the Internet. First, there is a lot of stuff to sift through. Unless you are looking for something very specific, it is easy to get overwhelmed. Searching through the newsgroups is particularly frustrating because of all the unrelated spam postings. Second, you can't believe everything you read on the Net. The information in books or magazines has been checked by several people. The publisher must make sure it is factual. Sure, printed matter is sometimes wrong, but at least there has been significant effort put forth to avoid errors. Many Web sites are at least as accurate as most printed matter. But some are not. It is as easy for one person to publish a Web site as it is for a big publishing group (perhaps easier, in some ways). Sometimes things are accidentally wrong. There is also a certain amount of misinformation deliberately floating around on the Web.

However, the fact that anyone can publish on the Web is also one of its strong points. The Gutenberg printing press made books available for everyone to read, starting the Age of Enlightenment. The Web makes publishing available to everyone and is surely starting an age that has no name yet. Perhaps the Age of Free Information.

So how do you find what you want? I've listed some places to start, both electronically and in print, that you will find useful as you continue to work with MFC:

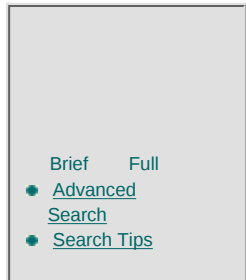
- www.al-williams.com—This is my Web site, where you can find the tip of the month (which is nearly always about MFC), information on books (including this one), updates, and lots of other things. You can also find out about other things my company does, including seminars I teach on MFC and other programming topics.
- *Visual Developer Magazine*—This magazine (from The Coriolis Group) features my “Windows Commando” column in each issue. Although the column is officially about Windows programming in C/C++, probably 9 out of 10 articles deal with MFC. The magazine is full of other articles and columns, too. You can find out more online at www.coriolis.com.
- www.program.com—This is an excellent site for finding other programming sites. Because so many sites are not worthwhile or accurate, it is useful to refer to meta sites like this one to separate the wheat from the chaff.
- www.r2m.com/windev—Another outstanding site of programming links.
- www.microsoft.com—It probably goes without saying that you can find a lot of MFC information on the Microsoft Web site.
- www.stars.com—This site isn't about MFC, but it is very useful if you are involved in any Internet development.
- The MFC FAQ—Scot Wingo maintains an excellent FAQ for MFC. You can easily find the most recent version easily by searching for MFC FAQ on any decent Web search engine.
- The MFC TechNotes—Although it might seem obvious, the tech notes in the MFC help are quite valuable. They often offer insights to why MFC works the way that it does. You can find these by searching the help for either TechNotes or Technical Notes (depending on which version of Visual C++ you are using).
- www.reference.com—One way to avoid the meaningless noise on the newsgroups is to search for articles that interest you. There are several sites that can do this, but www.reference.com does them all one better. Not only can this site search newsgroups, it can also remember your searches, run them automatically at a certain time interval, and mail you the results. Now that's a great idea.

There is plenty of MFC material out there, so I'm sure you could add to this list. But this will at least get you started. If you know of a really outstanding site that I missed, let me know and I might even put a link to it on my Web site.

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97



Search this book:

[Table of Contents](#)

Appendix A About The Shell Icon Handler

One of my oldest friends, Scott Anderson, taught me a lesson that I haven't forgotten. We were in high school (I told you we were old friends) and I was doing miserably in drafting class. Our final project was to produce a complete set of house plans for a house of our own design. As the teacher told us what we had to do, Scott kept raising his hand. He'd ask the instructor, "Can we include a deck?" If the teacher hesitated, Scott would press him until he would agree to allow the deck (or whatever feature Scott wanted). The list grew: Jacuzzis, atriums, skylights, storm cellars. The rest of us grew annoyed.

After class, I asked Scott, "Why did you keep pressing for all that stuff? This is going to be hard enough." Scott replied, "I probably won't do any of those things. I just wanted to be sure that if I did want to, I could."

To me, this was a revelation. I'd remember it as I went on to design a great deal of hardware and software. Scott went on to become a top-notch marketing executive. I should have expected that. After all, he did persuade our drafting teacher to allow all of those crazy things.

I think that the Windows 95 (and NT 4.0) shell designers learned this same lesson. That's why they allow you to extend the shell through the registry and OLE COM (or ActiveX) objects. You may never need to do this, but if you want the shell to work directly with your file types, it will allow you to write the code that you need.

In Chapter 6, you saw a custom App Wizard that created an icon handler (one of the types of shell extensions). If you want to know more about how icon handlers (and shell extensions) work, you'll want to read this appendix.

Types Of Shell Extensions

There are several different types of shell extensions (see Table A.1). A shell extension is, technically, an inproc OLE server that implements objects used by the shell. In plain English, a shell extension is a DLL with a few special functions. The DLL also provides a particular set of functions that the shell will call to perform certain operations.

Table A.1 Shell extensions.

Name	Description	Interfaces
Icon	Supplies dynamic icon for file type	IExtractIcon, IPersistFile

Context menu	Adds context menus or drag and drop menus (resulting from right-drag)	IcontextMenu, IShellExtInit
Property sheet	Adds custom property sheets	IShellPropSheetExt, IShellExtInit
Copy hook	Manages file operations	ICopyHook
Drop target	Reacts when an object is dropped on another	IDropTarget, IPersistFile
Data object	Supplies a data object for drag and drop or copy and paste	IDataObject, IPersistFile
Quick view	Parses file for Quick Viewer	IFileViewer, IPersistFile
Briefcase reconcile	Combines multiple versions of a file into one file	IReconcilableObject, IPersistFile

MFC provides some support for COM objects (but not as much as it does for other ActiveX or OLE programs like clients and servers). Still, some support is better than none—especially when it comes to ActiveX. Before you start, you need to ask yourself, “Is this trip really necessary?”

When Not To Use Shell Extensions

Shell extensions are powerful and, as you might expect, not trivial to write. Often, you can obtain the same effect without resorting to a true extension. The example program (which is the project the icon handler wizard is based on) provides a custom icon for a special file type. The icon is different depending on the number of bytes in the file. This is a good example of a case in which you must use a shell extension. If your goal is simply to provide a custom icon that doesn’t change, forget shell extensions. You need only to add a DefaultIcon key in the registry for the given file type.

About COM Objects

You may have heard a lot about COM objects and OLE. It seems confusing, mainly because the OLE documentation uses a lot of new terms. Here it is without the jargon: A COM object is a table of function pointers. That’s it. The program providing the function pointers (the server) and the program using the pointers (the client) must agree on what functions are in which slot of the table. They also need to agree on what arguments the functions take, return values, and so forth. They do not have to agree on the function’s name, nor do they need to agree on the language used.

Technically, the table of function pointers is an interface. Some COM objects support more than one interface and, therefore, contain more than one table of function pointers. All COM objects have at least three function table entries and these must be the same. These three functions make up **IUnknown** interface.

Table A.2 shows the three **IUnknown** interface functions. Because these three functions are the first three that all COM objects use in the table, you can treat any interface as an **IUnknown** interface. Then you can use **QueryInterface** (see Table A.2) to learn which interfaces the object supports.

Table A.2 The IUnknown interface.

Function	Description
AddRef	Increments the object’s in-use counter
Release	Decrements the in-use counter; when the counter reaches zero, the system will free the object
QueryInterface	Requests a pointer to a specific interface

The system doesn’t require it, but many COM functions return an **HRESULT**. This is a 32-bit word that has bit fields that can return information about a function’s success or failure. For simple COM programs, you can usually return **S_OK** (good return), **E_NOTIMPL** (not implemented), or **E_FAIL** (failure) with the **ResultFromScode** macro.

When you check the return code, don’t assume that zero means success—a successful **HRESULT** may not always equal zero. Instead, use the **SUCCEEDED** macro to test an **HRESULT**.

When a DLL provides a COM object, it also provides a class factory. A class factory is a special COM object

that handles the creation of a specific type of object. Objects and interfaces have a GUID (globally unique identifier) that identifies its type. You use this GUID in calls to the class factory, when using **QueryInterface**, and in the system registry to identify the object. The UUIDGEN program will create GUIDs for you, but MFC can create them also.

MFC Support For COM

Writing a COM inproc server (just a DLL, remember) isn't hard, but there is a lot of details to handle. Luckily, MFC will do some of this for you if you ask. If you search for COM support in App Wizard, however, you won't find it directly. The trick is to create an OLE automation object (which is a special form of COM object) and cut away everything you don't need.

Before you try this, however, let's look at some of the OLE support MFC provides that your COM object will need. MFC supports objects with multiple interfaces in an odd way. If you think that a COM interface sounds like a C++ VTBL, you're right. MFC uses virtual functions to create interfaces. The VTBL then acts as a COM interface. To support multiple interfaces, MFC creates nested objects (by default, their names start with an X). For example, here's what a class for an OLE object with two interfaces, I1 and I2, might look like:

```
class myobject: public CCmdTarget
{
    .
    .
    class XI1
    {
        // functions for I1
        // (including AddRef,
        // Release, and
        //QueryInterface)
        .
        .
        .
        STDMETHODIMP f1(void);
    };
    class XI2
    {
        // functions for I2
        .
        .
        .
        STDMETHODIMP f2(void);
    };
} // end of myobject
```

All COM objects derive from **CCmdTarget** (the base class for any MFC object that has maps). This base class provides default implementations for **AddRef**, **Release**, and **QueryInterface**. Good thing, too. **AddRef** and **Release** are more complex than they appear because they must be thread safe. Here's an example of what I1's **AddRef** function might look like:

```
STDMETHODIMP_(ULONG) myobject::XI1::AddRef()
{
    METHOD_PROLOGUE(myobject, I1)
    return pThis->ExternalAddRef();
}
```

Notice that you use the X in the class name (as in **XI1**) for the function declaration but you don't use it in the **METHOD_PROLOGUE** macro (where you just use **I1**). This macro allows you to access the outer class (**myobject**) using the predefined **pThis** pointer.

You don't write the nested classes directly. Instead, you use the **BEGIN_INTERFACE_PART** and **END_INTERFACE_PART** macros in the header file. Omit the X from the nested class names when using these macros. Between the two macros, you'll name the functions that the interface supports (the **IUnknown**

triplets—**AddRef**, **Release**, and **QueryInterface**—appear automatically when you use the macros). The actual definition for **myobject** could look like this:

```
class myobject : public CCmdTarget
{
    DECLARE_DYNCREATE(myobject, CCmdTarget);
// class factory
    DECLARE_OLECREATE(myobject);
    myobject();
    DECLARE_MESSAGE_MAP()
    BEGIN_INTERFACE_PART(myobject, I1)
        STDMETHOD (f1)();
    END_INTERFACE_PART(I1)
    BEGIN_INTERFACE_PART(myobject, I2)
        STDMETHOD (f2)();
    END_INTERFACE_PART(I2)
};
```

MFC's support for OLE is mainly in the form of macros (like **BEGIN_INTERFACE_PART**). Here are the most important ones:

- **DECLARE_INTERFACE_MAP**—This macro appears in your class' header file. It creates an interface map. This map designates which nested class corresponds to a particular COM interface.
- **BEGIN_INTERFACE_MAP**—This macro creates the interface map in the class' CPP file.
- **INTERFACE_PART**—Use this macro after the **BEGIN_INTERFACE_MAP** macro to define the particular map entries. The macro takes three arguments: the main class name, the interface IID, and the nested class name (without the leading X).
- **END_INTERFACE_MAP**—This macro ends the interface map.
- **DECLARE_OLE_CREATE**—Use this macro in your H file to declare a class factory.
- **IMPLEMENT_OLECREATE**—This corresponds to **DECLARE_OLE_CREATE**, but it appears in the CPP file. It takes the class name, an external name, and the object's CLSID.
- **STDMETHOD**—Declares a COM interface function that returns an **HRESULT**.
- **STDMETHODIMP**—Used to implement a COM function that returns an **HRESULT**.
- **STDMETHOD_**—Declares a COM interface function that returns something other than an **HRESULT**.
- **STDMETHODIMP_**—Implements a COM function that doesn't return an **HRESULT**. See the **AddRef** code in the section "MFC Support For COM" for an example of this.
- **METHOD_PROLOGUE**—This macro appears in member functions of the nested classes that implement each interface. It creates a variable, **pThis**, that points to the main class. See the **AddRef** code in the section "MFC Support For COM" for an example of this.

A COM Object Step-By-Step

To get MFC to create a COM object, you'll need to create an automation DLL and then remove the portions that relate to automation. Here are the steps you'll need to take, to coax MFC to generate an inproc COM server:

1. Use App Wizard to create an OLE automation DLL (see Figure A.1).

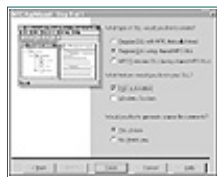


Figure A.1 App Wizard.

2. Use Class Wizard to derive a class from **CCmdTarget** that uses OLE automation (see Figure A.2).

Note: All interfaces include the IUnknown functions (see Table A.2).

Table A.4 The IExtractIcon interface.

Function	Description
GetIconLocation	Returns the name of the file that contains the icon and its index in that file
ExtractIcon	Returns icon by handle

Note: All interfaces include the IUnknown functions (see Table A.2).

The two functions that provide an icon (**IExtractIcon::GetIconLocation** and **IExtractIcon::Extract**) are complementary. You usually only implement one. **GetIconLocation** allows you to return a file name and an icon number. It's often convenient to use **GetModuleFileName** to learn the name of your DLL. Then you can use the icons inside the DLL. You can also use **Extract** to return an **HICON** for the file. This is useful if you want to draw the icon (using an image list, perhaps) from scratch. The arguments to **Extract** include the file name and icon number set in **GetIconLocation** (if any).

Listings A.1 and A.2 show the main portion of the example icon handler. This icon handler supplies two icons: one for files less than 1,024 bytes long, and another for larger files. Both icons are in the DLL, so the handler only needs to determine which icon to use.

Providing the icon handler isn't enough—you also need to tell the shell about it. Suppose you wanted to use this icon handler for .VIZ files. You'd need the following registry entries:

```
[HKEY_CLASSES_ROOT\.VIZ]="VIZFile"

[HKEY_CLASSES_ROOT\VIZFile]="Visual Developer File"

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{your CLSID here}]="VIZFILE
Icon Ext"

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{your CLSID here}\
InProcServer32]
    ="vizicon.dll"
"ThreadingModel"="Apartment"

[HKEY_CLASSES_ROOT\VIZFile\shellex\IconHandler]="{your CLSID here}"

[HKEY_CLASSES_ROOT\VIZFile\DefaultIcon]="%1"
```

The End Of Shell Extensions?

Most of the shell extensions work like the icon extension handler. Some use the **IShellExtInit** instead of the **IPersist** interface to accept the file name. The difference is trivial. The good news is that if you know how to do one, the rest are easy.

(Please note that due to page width constraints, the line that begins **IMPLEMENT_OLECREATE...** is shown in Listing A.1 broken into two lines, but it should actually be one).

Listing A.1 The icon handler's main file.

```
// IconHandler.cpp : implementation file
//

#include "stdafx.h"
#include "vizicon.h"
#include "winnetwk.h"
#include "shlobj.h"
```

```

#include "winnls.h"
#include "IconHandler.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////
// CIconHandler

IMPLEMENT_DYNCREATE(CIconHandler, CCmdTarget)

CIconHandler::CIconHandler()
{
}

CIconHandler::~CIconHandler()
{
}

BEGIN_MESSAGE_MAP(CIconHandler, CCmdTarget)
    //{AFX_MSG_MAP(CIconHandler)
    // NOTE - the Class Wizard will add and remove mapping macros
    // here.
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// Note: This is the GUID used for your icon handler
// {D9FB7760-5CAC-11CF-A7B2-444553540000}
static const IID IID_IIconHandler =
{ 0xD9FB7760, 0x5CAC, 0x11CF, { 0xA7,0xB2,0x44,0x45,0x53,0x54,0x00,0x00
} };

IMPLEMENT_OLECREATE(CIconHandler, "AIconHandler", 0xD9FB7760, 0x5CAC, 0x11CF,
    0xA7, 0xB2, 0x44, 0x45, 0x53, 0x54, 0x00, 0x00);
BEGIN_INTERFACE_MAP(CIconHandler, CCmdTarget)
    INTERFACE_PART(CIconHandler, IID_IPersistFile, PersistFile)
    INTERFACE_PART(CIconHandler, IID_IExtractIcon, Icon)
END_INTERFACE_MAP()

STDMETHODIMP_(ULONG) CIconHandler::XPersistFile::AddRef()
{
    METHOD_PROLOGUE(CIconHandler, PersistFile)
    return pThis->ExternalAddRef();
}

STDMETHODIMP_(ULONG) CIconHandler::XPersistFile::Release()
{
    METHOD_PROLOGUE(CIconHandler, PersistFile)
    return pThis->ExternalRelease();
}

STDMETHODIMP CIconHandler::XPersistFile::QueryInterface(
    REFIID iid, void FAR * FAR *pObj)
{
    METHOD_PROLOGUE(CIconHandler, PersistFile)
    return (HRESULT)pThis->ExternalQueryInterface(&iid, pObj);
}

```

```

}

STDMETHODIMP CIconHandler::XPersistFile::IsDirty()
{
    return E_NOTIMPL;
}

STDMETHODIMP CIconHandler::XPersistFile::Load(LPCOLESTR pszFileName, DWORD)
{
    METHOD_PROLOGUE(CIconHandler, PersistFile);
    WideCharToMultiByte(CP_ACP, 0, (LPCWSTR)pszFileName, -1,
        pThis->m_szFile, sizeof(pThis->m_szFile),
        NULL, NULL);
    return NOERROR;
}

STDMETHODIMP CIconHandler::XPersistFile::Save(LPCOLESTR, BOOL)
{
    return E_NOTIMPL;
}

STDMETHODIMP CIconHandler::XPersistFile::SaveCompleted(LPCOLESTR)
{
    return E_NOTIMPL;
}

STDMETHODIMP CIconHandler::XPersistFile::GetCurFile(LPOLESTR FAR *)
{
    return E_NOTIMPL;
}

STDMETHODIMP CIconHandler::XPersistFile::GetClassID(LPCLSID)
{
    return E_NOTIMPL;
}

STDMETHODIMP_(ULONG) CIconHandler::XIcon::AddRef()
{
    METHOD_PROLOGUE(CIconHandler, Icon)
    return pThis->ExternalAddRef();
}

STDMETHODIMP_(ULONG) CIconHandler::XIcon::Release()
{
    METHOD_PROLOGUE(CIconHandler, Icon)
    return pThis->ExternalRelease();
}

STDMETHODIMP CIconHandler::XIcon::QueryInterface(REFIID iid, void
FAR * FAR *pObj)
{
    METHOD_PROLOGUE(CIconHandler, Icon)
    return (HRESULT)pThis->ExternalQueryInterface(&iid, pObj);
}

```

Listing A.2 The header file.

```

// IconHandler.h : header file
//

```

```

////////////////////////////////////
// CIconHandler command target

class CIconHandler : public CCmdTarget
{
    DECLARE_DYNCREATE(CIconHandler)
    DECLARE_OLECREATE(CIconHandler)
    CIconHandler(); // protected constructor used by dynamic creation

// Attributes
public:
    char m_szFile[256];
// Operations
public:

// Overrides
    // Class Wizard generated virtual function overrides
    //{{AFX_VIRTUAL(CIconHandler)
    public:
    //}}AFX_VIRTUAL

// Implementation
protected:
    virtual ~CIconHandler();

    // Generated message map functions
    //{{AFX_MSG(CIconHandler)
    // NOTE - the Class Wizard will add and remove member functions
    // here.
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
    DECLARE_INTERFACE_MAP()
    BEGIN_INTERFACE_PART(PersistFile, IPersistFile)
        STDMETHOD (GetClassID)(LPCLSID);
        STDMETHOD (IsDirty)();
        STDMETHOD (Load)(LPCOLESTR, DWORD);
        STDMETHOD (Save)(LPCOLESTR, BOOL);
        STDMETHOD (SaveCompleted)(LPCOLESTR);
        STDMETHOD (GetCurFile)(LPOLESTR FAR *);
    END_INTERFACE_PART(PersistFile)

    BEGIN_INTERFACE_PART(Icon, IExtractIcon)
        STDMETHOD (GetIconLocation)(UINT, LPSTR, UINT, int *, UINT *);
        STDMETHOD (Extract)(LPCSTR, UINT, HICON *, HICON *, UINT);
    END_INTERFACE_PART(Icon)

};

////////////////////////////////////

```

Table of Contents

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

[Bookmark It](#)

Search this book:

[Table of Contents](#)

Appendix B Field Guide To The MFC Source Code

Having MFC's source code is both a boon and a bane. It is certainly useful to be able to delve into the most private details of the MFC implementation. On the other hand, it can also be frustrating because you wish there were more documentation than the often cryptic source code comments.

Another thing that leads to confusion is that MFC doesn't always declare one class in one source file. Some source files have multiple classes, while some classes are spread out among multiple files.

To help a bit, you'll find two lists at the end of this appendix. The first list is an alphabetical list of MFC classes and the files that define them. The second list shows each file with the classes it contains.

Appropriate Use Of The Source Code

It is possible to modify the MFC source code and build your own custom version of the library. However, this is an extraordinarily bad idea. Doing this effectively branches away from the mainstream MFC product. Integrating future changes into your custom library will be a specialized task each time. Depending on your changes (and Microsoft's) this may be fairly simple, or could be a veritable nightmare.

Does that mean you have to take the library as-is? Not at all. This book is full of examples where I derived a new class from existing classes in order to

change behavior or fix bugs. Look at the splitter window and list control classes in Chapter 4, for example. Using your own classes gives you a bit of distance from MFC. If you like, you can even use the techniques in Chapter 7 to add your classes to MFC at runtime. There isn't anything wrong with that.

Perhaps the best use of the source code, however, is simply reading it. No matter what the documentation says (or doesn't say), the answers to all your questions are in the source code if you can find them. Reading it can be very educational.

I wouldn't model my code on MFC's internal structure. Some of it is very clever, but some of it isn't. Much of it makes object-oriented designers wince. Still more of it goes to great lengths to accommodate something that was a problem when it was written, but hasn't been a problem for a long time. I don't mean to knock MFC, but like many things that evolve, it has its share of warts that you have to overlook.

Don't forget about the MFC technical notes in the online help. These gems have some of the best documentation about why the library does the things that it does, and will help you in your quest through the source code.

The Longest Journey...

The old saying goes that the longest journey begins with the first step. Hopefully the information in this appendix will help you take the first step on a very long journey indeed. Even if you don't read every line of the MFC source, you are sure to find something useful on each trip you take through the source code.

MFC Source By Class Name

AFX_CLASSINIT: objcore.cpp
AFX_COM: filecore.cpp
AFX_DDPDATA: ctlppg.cpp
AFX_EXCEPTION_LINK: except.cpp
AFX_MAINTAIN_STATE: afxstate.cpp
AFX_MODULE_STATE: afxstate.cpp
AFX_MODULE_THREAD_STATE: afxstate.cpp
AFX_PARSEMAP: isapi.cpp
AFX_PARSEMAP_ENTRY_PARAMS: isapi.cpp
AUX_DATA: auxdata.cpp
CAfxOleSymbolTable: olereg.cpp
CAnimateCtrl: winctrl2.cpp
CArchive: arccore.cpp, arcobj.cpp
CArchiveException: arcex.cpp
CArchivePropExchange: ctlpropx.cpp
CArchiveStream: arcstrm.cpp
CAsyncMonikerFile: oleasmon.cpp
CAsyncPropExchange: ctlpropx.cpp

CAsyncSocket: sockcore.cpp
CBitmap: wingdi.cpp
CBitmapButton: winbtn.cpp
CBlobProperty: ctlpbag.cpp
CBrush: wingdi.cpp
CButton: winctrl1.cpp
CByteArray: array_b.cpp
CCachedDataPathProperty: ctlprop.cpp
CCheckListBox: winctrl3.cpp
CClientDC: wingdi.cpp
CCmdTarget: cmdtarg.cpp, occevent.cpp, oleconn.cpp, oledisp1.cpp, oletyp1b.cpp, oleunk.cpp, oleverb.cpp
CCmdUI: cmdtarg.cpp
CColorButton: ppgcolor.cpp
CColorDialog: dlgclr.cpp
CColorPropPage: ppgcolor.cpp
CComboBox: winctrl1.cpp
CCommandLineInfo: appcore.cpp
CCommonDialog: dlgcomm.cpp
CConnectionPoint: oleconn.cpp
CControlBar: barcore.cpp, bardock.cpp, dockstat.cpp, winfrm.cpp
CControlBarInfo: dockstat.cpp
CControlFrameWnd: ctlframe.cpp
CCtrlView: viewcore.cpp
CDBByteArray: dbrfx.cpp
CDBException: dbcore.cpp
CDBVariant: dbvar.cpp
CDC: wingdi.cpp, wingdix.cpp
CDWordArray: array_d.cpp
CDaoDatabase: daocore.cpp
CDaoDatabaseInfo: daocore.cpp
CDaoErrorInfo: daocore.cpp
CDaoException: daocore.cpp
CDaoFieldExchange: daodfx.cpp
CDaoFieldInfo: daocore.cpp
CDaoIndexFieldInfo: daocore.cpp
CDaoIndexInfo: daocore.cpp
CDaoParameterInfo: daocore.cpp
CDaoQueryDef: daocore.cpp
CDaoQueryDefInfo: daocore.cpp

CDaoRecordView: daoview.cpp
CDaoRecordset: daocore.cpp
CDaoRelationFieldInfo: daocore.cpp
CDaoRelationInfo: daocore.cpp
CDaoTableDef: daocore.cpp
CDaoTableDefInfo: daocore.cpp
CDaoWorkspace: daocore.cpp
CDaoWorkspaceInfo: daocore.cpp
CDataBoundProperty: occsite.cpp
CDataExchange: dlgdata.cpp, occmgr.cpp, wincore.cpp
CDataPathProperty: ctlprop.cpp
CDataSourceControl: occsite.cpp
CDatabase: dbcore.cpp
CDialog: dlgcore.cpp
CDialogBar: bardlg.cpp
CDialogTemplate: dlgtempl.cpp
CDocItem: oledoc1.cpp
CDocManager: docmgr.cpp
CDocObjectServer: oledocob.cpp, oledocvw.cpp
CDocObjectServerItem: oledocob.cpp
CDocTemplate: doctempl.cpp
CDockBar: bardock.cpp, dockstat.cpp
CDockContext: dockcont.cpp
CDockState: dockstat.cpp
CDocument: doccore.cpp, docmapi.cpp
CDragListBox: winctrl2.cpp
CDumpContext: dumpcont.cpp, dumpflt.cpp
CDynLinkLibrary: dllinit.cpp
CEdit: winctrl1.cpp
CEditView: viewedit.cpp
CEnumArray: oleenum.cpp
CEnumConnPoints: oleconn.cpp
CEnumConnections: oleconn.cpp
CEnumFormatEtc: oledobj2.cpp
CEnumOleVerb: oleverb.cpp
CEnumUnknown: occcont.cpp
CEvent: mtex.cpp
CException: except.cpp
CFieldExchange: dbrfx.cpp
CFile: filecore.cpp, filest.cpp

CFileDialog: dlgfile.cpp
CFileException: filex.cpp
CFileFind: filefind.cpp
CFileStatus: filest.cpp
CFindReplaceDialog: dlgfr.cpp
CFont: wingdi.cpp, wingdix.cpp
CFontComboBox: ppgfont.cpp
CFontDialog: dlgfnt.cpp
CFontHolder: ctlfont.cpp
CFontPropPage: ppgfont.cpp
CFormView: viewform.cpp
CFrameWnd: dockstat.cpp, wincore.cpp, winfrm.cpp, winfrm2.cpp, winfrm3.cpp
CFtpConnection: inet.cpp
CFtpFileFind: inet.cpp
CGdiObject: wingdi.cpp
CGopherConnection: inet.cpp
CGopherFile: inet.cpp
CGopherFileFind: inet.cpp
CGopherLocator: inet.cpp
CHandleMap: winhand.cpp
CHeaderCtrl: winctrl2.cpp
CHotKeyCtrl: winctrl2.cpp
CHtmlStream: isapi.cpp, isapimix.cpp
CHttpConnection: inet.cpp
CHttpFile: inet.cpp
CHttpFilter: isapi.cpp
CHttpServer: isapi.cpp
CHttpServerContext: isapi.cpp, isapimix.cpp
CImageList: winctrl2.cpp
CInnerUnknown: oleunk.cpp
CInternetConnection: inet.cpp
CInternetException: inet.cpp
CInternetFile: inet.cpp
CInternetSession: inet.cpp
CListBox: winctrl1.cpp
CListCtrl: winctrl2.cpp
CListView: viewcmn.cpp
CLongBinary: dblong.cpp
CMDIChildWnd: winmdi.cpp

CMDIFrameWnd: winmdi.cpp
CMapPtrToPtr: map_pp.cpp
CMapPtrToWord: map_pw.cpp
CMapStringToOb: map_so.cpp
CMapStringToPtr: map_sp.cpp
CMapStringToString: map_ss.cpp
CMapWordToOb: map_wo.cpp
CMapWordToPtr: map_wp.cpp
CMemFile: filemem.cpp, filest.cpp
CMemoryState: afxmem.cpp
CMenu: wincore.cpp, winmenu.cpp
CMetaFileDC: dcmeta.cpp
CMiniDockFrameWnd: bardock.cpp
CMiniFrameWnd: dockcont.cpp, winmini.cpp
CMirrorFile: doccore.cpp
CMonikerFile: olemon.cpp
CMultiDocTemplate: docmulti.cpp
CMultiLock: mtex.cpp
CMutex: mtex.cpp
CNewTypeDlg: docmgr.cpp
CNoTrackObject: afxtls.cpp
CObArray: array_o.cpp
CObList: list_o.cpp
CObject: afxmem.cpp, objcore.cpp
COccManager: occdlg.cpp, occmgr.cpp
COleBusyDialog: oledlgs2.cpp
COleChangeIconDialog: oledlgs1.cpp
COleChangeSourceDialog: oledlgs3.cpp
COleClientItem: olecli1.cpp, olecli2.cpp, olecli3.cpp, oleui1.cpp
COleCmdUI: oledoctg.cpp
COleCntrFrameWnd: oleipfrm.cpp
COleConnPtContainer: oleconn.cpp
COleControl: ctlconn.cpp, ctlcore.cpp, ctldata.cpp, ctlevvent.cpp, ctlinplc.cpp, ctlnownd.cpp, ctlobj.cpp, ctlprop.cpp, ctlpropx.cpp, ctlpset.cpp, ctltrack.cpp, ctlview.cpp
COleControlContainer: occcont.cpp
COleControlLock: occlock.cpp
COleControlModule: ctlmodul.cpp
COleControlSite: occsite.cpp
COleConvertDialog: oledlgs1.cpp
COleCurrency: olevar.cpp

COleDataObject: oledobj1.cpp
COleDataSource: oledobj2.cpp, oledrop1.cpp
COleDateTime: olevar.cpp
COleDateTimeSpan: olevar.cpp
COleDialog: oledlgs2.cpp
COleDispatchDriver: oledisp2.cpp
COleDispatchException: oledisp1.cpp
COleDispatchImpl: oledisp1.cpp
COleDocIPFrameWnd: oledocip.cpp
COleDocument: docmapi.cpp, oledoc1.cpp, oledoc2.cpp, oleui1.cpp, oleui2.cpp
COleDropSource: oledrop1.cpp
COleDropTarget: oledrop2.cpp
COleException: olemisc.cpp
COleFrameHook: olecli2.cpp
COleIPFrameWnd: oleipfrm.cpp
COleInsertDialog: oledlgs1.cpp
COleLinkingDoc: olelink.cpp
COleLinksDialog: oledlgs1.cpp
COleMessageFilter: olemsgf.cpp
COleObjectFactory: olefact.cpp
COlePasteSpecialDialog: oledlgs1.cpp
COlePropertiesDialog: oledlgs3.cpp
COlePropertyPage: ctlppg.cpp
COleResizeBar: olebar.cpp
COleSafeArray: olevar.cpp
COleServerDoc: olesvr1.cpp
COleServerItem: olesvr1.cpp, olesvr2.cpp
COleStreamFile: olestrm.cpp
COleTemplateServer: oletsvr.cpp
COleUILinkInfo: oledlgs1.cpp
COleUpdateDialog: oledlgs1.cpp
COleVariant: olevar.cpp
CPageSetupDialog: dlgprnt.cpp
CPaintDC: wingdi.cpp
CParkingWnd: ctlrefl.cpp
CPen: wingdi.cpp
CPictureHolder: ctlpict.cpp
CPicturePropPage: ppgpict.cpp
CPlex: plex.cpp

CPoint: wincore.cpp
CPreviewDC: dcprev.cpp
CPreviewView: viewprev.cpp
CPrintDialog: dlgprnt.cpp
CPrintInfo: viewprnt.cpp
CPrintPreviewState: viewprev.cpp
CPrintingDialog: viewprnt.cpp
CProcessLocalObject: afxtls.cpp
CProgressCtrl: winctrl2.cpp
CPropExchange: ctlpropx.cpp
CPropbagPropExchange: ctlpbag.cpp
CProperty: olepset.cpp
CPropertyPage: dlgprop.cpp
CPropertySection: olepset.cpp
CPropertySet: olepset.cpp
CPropertySheet: dlgprop.cpp
CPropsetPropExchange: ctplset.cpp
CPtrArray: array_p.cpp
CPtrList: list_p.cpp
CReObject: viewrich.cpp
CRecentFileList: filelist.cpp
CRecordView: dbview.cpp
CRecordset: dbcore.cpp
CRect: wingdix.cpp
CRectTracker: trckrect.cpp
CReflectorWnd: ctlrefl.cpp
CResetPropExchange: ctlpropx.cpp
CRichEditCntrlItem: viewrich.cpp
CRichEditCtrl: winctrl2.cpp, winctrl4.cpp
CRichEditDoc: viewrich.cpp
CRichEditView: viewrich.cpp
CRuntimeClass: arccore.cpp, objcore.cpp
CScrollBar: winctrl1.cpp
CScrollView: viewscl.cpp
CSemaphore: mtex.cpp
CSharedFile: fileshrd.cpp
CSimpleException: except.cpp
CSimpleList: afxtls.cpp
CSingleDocTemplate: docsingl.cpp
CSingleLock: mtex.cpp

CSizeComboBox: ppgfont.cpp
CSliderCtrl: winctrl2.cpp
CSocket: sockcore.cpp
CSocketFile: sockcore.cpp
CSocketWnd: sockcore.cpp
CSpinButtonCtrl: winctrl2.cpp
CSplitterWnd: winsplit.cpp
CStatic: winctrl1.cpp
CStatusBar: barstat.cpp
CStatusBarCtrl: winctrl2.cpp
CStatusCmdUI: barstat.cpp
CStdioFile: filetxt.cpp
CStockPropPage: ppgstock.cpp
CString: oledisp1.cpp, strcore.cpp, strex.cpp, winstr.cpp
CStringArray: array_s.cpp
CStringList: list_s.cpp
CSyncObject: mtcore.cpp
CTabCtrl: winctrl2.cpp
CTestCmdUI: wincore.cpp
CThreadLocalObject: afxtls.cpp
CThreadSlotData: afxtls.cpp
CTime: timecore.cpp
CTimeSpan: timecore.cpp
CToolBar: bartool.cpp
CToolBarCtrl: winctrl2.cpp
CToolCmdUI: bartool.cpp
CToolTipCtrl: tooltip.cpp
CTreeCtrl: winctrl2.cpp
CTreeView: viewcmn.cpp
CTypeLibCache: afxstate.cpp, oletyplb.cpp
CUIntArray: array_u.cpp
CView: viewcore.cpp, viewprev.cpp, viewprnt.cpp
CWinApp: app3d.cpp, app3ds.cpp, appcore.cpp, appdlg.cpp, appgray.cpp, apphelpx.cpp, appinit.cpp, appprnt.cpp, appui.cpp, appui2.cpp, appui3.cpp, oleinit.cpp
CWinThread: thrdcore.cpp
CWindowDC: wingdi.cpp
CWindowlessDC: ctlnownd.cpp
CWnd: appui.cpp, dlgcore.cpp, occcont.cpp, wincore.cpp, winctrl1.cpp, winfrmxx.cpp, winocc.cpp
CWordArray: array_w.cpp

OLE_DATA: oleinit.cpp
_AFXCTL_AMBIENT_CACHE: ctlquick.cpp
_AFXCTL_UIACTIVE_INFO: ctlinplc.cpp
_AFX_CHECKLIST_STATE: winctrl3.cpp
_AFX_COLOR_STATE: dlgclr.cpp
_AFX_CTL3D_STATE: app3d.cpp
_AFX_CTL3D_THREAD: app3d.cpp
_AFX_DAO_STATE: daocore.cpp
_AFX_DB_STATE: dbcore.cpp
_AFX_DEBUG_STATE: dumpinit.cpp
_AFX_EDIT_STATE: viewedit.cpp
_AFX_EXTDLL_STATE: ccdata.cpp
_AFX_INIT_OLE_DLL_STATE: dllole.cpp
_AFX_MAIL_STATE: docmapl.cpp
_AFX_OLE_STATE: oledata.cpp
_AFX_RICHEDIT_STATE: winctrl4.cpp
_AFX_SOCKET_STATE: sockcore.cpp
_AFX_TERM_APP_STATE: appmodul.cpp
_AFX_TERM_DLL_STATE: dllmodul.cpp
_AFX_THREAD_STATE: afxstate.cpp
_AFX_WIN_STATE: appcore.cpp

MFC Source By File Name

afxstate.cpp: CObject, CMemoryStateafxtls.cpp:
AFX_MAINTAIN_STATE, _AFX_THREAD_STATE,
AFX_MODULE_STATE, AFX_MODULE_THREAD_STATE,
CTypeLibCache
app3d.cpp: CSimpleList, CNoTrackObject, CThreadSlotData,
CThreadLocalObject, CProcessLocalObject
app3ds.cpp: _AFX_CTL3D_STATE, _AFX_CTL3D_THREAD,
CWinApp
appcore.cpp: CWinApp
appdlg.cpp: _AFX_WIN_STATE, CWinApp, CCommandLineInfo
appgray.cpp: CWinApp
apphelpx.cpp: CWinApp
appinit.cpp: CWinApp
appmodul.cpp: CWinApp
apprnt.cpp: _AFX_TERM_APP_STATE
appui.cpp: CWinApp
appui2.cpp: CWinApp, CWnd
appui3.cpp: CWinApp
arccore.cpp: CWinApp

arcex.cpp: CArchive, CRuntimeClass
arcobj.cpp: CArchiveException
arcstrm.cpp: CArchive
array_b.cpp: CArchiveStream
array_d.cpp: CByteArray
array_o.cpp: CDWordArray
array_p.cpp: CObArray
array_s.cpp: CPtrArray
array_u.cpp: CStringArray
array_w.cpp: CUIntArray
auxdata.cpp: CWordArray
barcore.cpp: AUX_DATA
bardlg.cpp: CControlBar
bardock.cpp: CDialogBar
barstat.cpp: CDockBar, CControlBar, CMiniDockFrameWnd
bartool.cpp: CStatusBar, CStatusCmdUI
ccdata.cpp: CToolBar, CToolCmdUI
cmdtarg.cpp: _AFX_EXTDLL_STATE
ctlconn.cpp: CCmdTarget, CCmdUI
ctlcore.cpp: COleControl
ctldata.cpp: COleControl
ctlevent.cpp: COleControl
ctlfont.cpp: COleControl
ctlframe.cpp: CFontHolder
ctlinplc.cpp: CControlFrameWnd
ctlmodul.cpp: _AFXCTL_UIACTIVE_INFO, COleControl
ctlnownd.cpp: COleControlModule
ctlobj.cpp: CWindowlessDC, COleControl
ctlpbag.cpp: COleControl
ctlpict.cpp: CBlobProperty, CPropbagPropExchange
ctlppg.cpp: CPictureHolder
ctlprop.cpp: AFX_DDPDATA, COlePropertyPage
ctlpropx.cpp: COleControl, CDataPathProperty, CCachedDataPathProperty
ctlpset.cpp: COleControl, CPropExchange, CArchivePropExchange, CResetPropExchange, CAsyncPropExchange
ctlquick.cpp: COleControl, CPropsetPropExchange
ctlrefl.cpp: _AFXCTL_AMBIENT_CACHE
ctltrack.cpp: CReflectorWnd, CParkingWnd
ctlview.cpp: COleControl

daocore.cpp: COleControl
daodfx.cpp: _AFX_DAO_STATE, CDaoErrorInfo,
CDaoWorkspaceInfo, CDaoDatabaseInfo, CDaoTableDefInfo,
CDaoFieldInfo, CDaoIndexFieldInfo, CDaoIndexInfo,
CDaoRelationFieldInfo, CDaoRelationInfo, CDaoQueryDefInfo,
CDaoParameterInfo, CDaoException, CDaoWorkspace, CDaoDatabase,
CDaoTableDef, CDaoQueryDef, CDaoRecordset
daoview.cpp: CDaoFieldExchange
dbcore.cpp: CDaoRecordView
dblong.cpp: _AFX_DB_STATE, CDBException, CDatabase,
CRecordset
dbrfx.cpp: CLongBinary
dbvar.cpp: CDBByteArray, CFieldExchange
dbview.cpp: CDBVariant
dcmeta.cpp: CRecordView
dcprev.cpp: CMetaFileDC
dlgclr.cpp: CPreviewDC
dlgcomm.cpp: _AFX_COLOR_STATE, CColorDialog
dlgcore.cpp: CCommonDialog
dlgdata.cpp: CDialog, CWnd
dlgfile.cpp: CDataExchange
dlgfmt.cpp: CFileDialog
dlgfr.cpp: CFontDialog
dlgprnt.cpp: CFindReplaceDialog
dlgprop.cpp: CPageSetupDialog, CPrintDialog
dlgtempl.cpp: CPropertyPage, CPropertySheet
dllinit.cpp: CDialogTemplate
dllmodul.cpp: CDynLinkLibrary
dllole.cpp: _AFX_TERM_DLL_STATE
doccore.cpp: _AFX_INIT_OLE_DLL_STATE
dockcont.cpp: CDocument, CMirrorFile
dockstat.cpp: CDockContext, CMiniFrameWnd
docmapi.cpp: CControlBarInfo, CDockState, CFrameWnd,
CControlBar, CDockBar
docmgr.cpp: _AFX_MAIL_STATE, CDocument, COleDocument
docmulti.cpp: CDocManager, CNewTypeDlg
docsingl.cpp: CMultiDocTemplate
doctempl.cpp: CSingleDocTemplate
dumpcont.cpp: CDocTemplate
dumpflt.cpp: CDumpContext
dumpinit.cpp: CDumpContext
except.cpp: _AFX_DEBUG_STATE

filecore.cpp: CException, AFX_EXCEPTION_LINK, CSimpleException
filefind.cpp: CFile, AFX_COM
filelist.cpp: CFileFind
filemem.cpp: CRecentFileList
fileshrd.cpp: CMemFile
filest.cpp: CSharedFile
filetxt.cpp: CFileStatus, CFile, CMemFile
filex.cpp: CStdioFile
inet.cpp: CFileException
isapi.cpp: CHttpConnection, CInternetSession, CInternetFile, CInternetConnection, CFtpConnection, CGopherConnection, CHttpFile, CGopherFile, CFtpFileFind, CGopherFileFind, CGopherLocator, CInternetException
isapimix.cpp: CHttpServer, AFX_PARSEMAP, AFX_PARSEMAP_ENTRY_PARAMS, CHttpServerContext, CHtmlStream, CHttpFilter
list_o.cpp: CHttpServerContext, CHtmlStream
list_p.cpp: CObList
list_s.cpp: CPtrList
map_pp.cpp: CStringList
map_pw.cpp: CMapPtrToPtr
map_so.cpp: CMapPtrToWord
map_sp.cpp: CMapStringToOb
map_ss.cpp: CMapStringToPtr
map_wo.cpp: CMapStringToString
map_wp.cpp: CMapWordToOb
mtcore.cpp: CMapWordToPtr
mtx.cpp: CSyncObject
objcore.cpp: CSemaphore, CMutex, CEvent, CSingleLock, CMultiLock
occcont.cpp: CObject, CRuntimeClass, AFX_CLASSINIT
occdlg.cpp: CWnd, COleControlContainer, CEnumUnknown
occevent.cpp: COccManager
occlock.cpp: CCmdTarget
occmgr.cpp: COleControlLock
occsite.cpp: COccManager, CDataExchange
oleasmon.cpp: COleControlSite, CDataSourceControl, CDataBoundProperty
olebar.cpp: CAsyncMonikerFile
olecli1.cpp: COleResizeBar
olecli2.cpp: COleClientItem
olecli3.cpp: COleFrameHook, COleClientItem

oleconn.cpp: COleClientItem
oleconn.cpp: _AFX_OLE_STATE
oledata.cpp: CConnectionPoint, CEnumConnections,
CEnumConnPoints, COleConnPtContainer
oledisp1.cpp: CCmdTarget
oledisp2.cpp: CCmdTarget, COleDispatchImpl, CString,
COleDispatchException
oledlgs1.cpp: COleDispatchDriver
oledlgs2.cpp: COleUILinkInfo, COleInsertDialog, COleConvertDialog,
COleChangeIconDialog, COleLinksDialog, COleUpdateDialog,
COlePasteSpecialDialog
oledlgs3.cpp: COleDialog, COleBusyDialog
oledobj1.cpp: COlePropertiesDialog, COleChangeSourceDialog
oledobj2.cpp: COleDataObject
oledoc1.cpp: COleDataSource, CEnumFormatEtc
oledoc2.cpp: COleDocument, CDocItem
oledocip.cpp: COleDocument
oledocob.cpp: COleDocIPFrameWnd
oledoctg.cpp: CDocObjectServer, CDocObjectServerItem
oledocvw.cpp: COleCmdUI
oledrop1.cpp: CDocObjectServer
oledrop2.cpp: COleDropSource, COleDataSource
oleenum.cpp: COleDropTarget
olefact.cpp: CEnumArray
oleinit.cpp: COleObjectFactory
oleipfrm.cpp: OLE_DATA, CWinApp
olelink.cpp: COleCntrFrameWnd, COleIPFrameWnd
olemisc.cpp: COleLinkingDoc
olemon.cpp: COleException
olemsgf.cpp: CMonikerFile
olepset.cpp: COleMessageFilter
olereg.cpp: CProperty, CPropertySection, CPropertySet
olestrm.cpp: CAfxOleSymbolTable
olesvr1.cpp: COleStreamFile
olesvr2.cpp: COleServerDoc, COleServerItem
oletsvr.cpp: COleServerItem
oletyplb.cpp: COleTemplateServer
oleui1.cpp: CCmdTarget, CTypeLibCache
oleui2.cpp: COleClientItem, COleDocument
oleunk.cpp: COleDocument
olevar.cpp: CCmdTarget, CInnerUnknown

oleverb.cpp: COleVariant, COleCurrency, COleDateTime, COleDateTimeSpan, COleSafeArray
plex.cpp: CEnumOleVerb, CCmdTarget
ppgcolor.cpp: CPlex
ppgfont.cpp: CColorButton, CColorPropPage
ppgpict.cpp: CFontPropPage, CFontComboBox, CSizeComboBox
ppgstock.cpp: CPicturePropPage
sockcore.cpp: CStockPropPage
strcore.cpp: _AFX_SOCKET_STATE, CAsyncSocket, CSocket, CSocketFile, CSocketWnd
strex.cpp: CString
thrdcore.cpp: CString
timecore.cpp: CWinThread
tooltip.cpp: CTime, CTimeSpan
trckrect.cpp: CToolTipCtrl
viewcmn.cpp: CRectTracker
viewcore.cpp: CListView, CTreeView
viewedit.cpp: CView, CCtrlView
viewform.cpp: CEditView, _AFX_EDIT_STATE
viewprev.cpp: CFormView
viewprnt.cpp: CPrintPreviewState, CView, CPreviewView
viewrich.cpp: CPrintingDialog, CView, CPrintInfo
viewscrl.cpp: CReObject, CRichEditView, CRichEditDoc, CRichEditCntrlItem
winbtn.cpp: CScrollView
wincore.cpp: CBitmapButton
winctrl1.cpp: CWnd, CMenu, CTestCmdUI, CDataExchange, CFrameWnd, CPoint
winctrl2.cpp: CStatic, CButton, CWnd, CListBox, CComboBox, CEdit, CScrollBar
winctrl3.cpp: CDragListBox, CToolBarCtrl, CStatusBarCtrl, CListCtrl, CTreeCtrl, CSpinButtonCtrl, CSliderCtrl, CProgressCtrl, CHeaderCtrl, CHotKeyCtrl, CTabCtrl, CAnimateCtrl, CRichEditCtrl, CImageList
winctrl4.cpp: _AFX_CHECKLIST_STATE, CCheckListBox
winfrm.cpp: _AFX_RICHEDIT_STATE, CRichEditCtrl
winfrm2.cpp: CFrameWnd, CControlBar
winfrm3.cpp: CFrameWnd
wingdi.cpp: CWnd, CFrameWnd
wingdix.cpp: CDC, CClientDC, CWindowDC, CPaintDC, CGdiObject, CPen, CBrush, CFont, CBitmap
winhand.cpp: CDC, CFont, CRect
winmdi.cpp: CHandleMap

winmenu.cpp: CMDIFrameWnd, CMDIChildWnd
winmini.cpp: CMenu
winocc.cpp: CMiniFrameWnd
winsplit.cpp: CWnd
winstr.cpp: CSplitterWnd

Table of Contents

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Brief Full

- [Advanced Search](#)
- [Search Tips](#)

BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

MFC Black Book

(Publisher: The Coriolis Group)

Author(s): Al Williams

ISBN: 1576101851

Publication Date: 12/01/97

Bookmark It

Search this book:

[Table of Contents](#)

Index

A

A2W, 442

AbortDlg.cpp, 504-506

AbortDll.cpp, 501-503

AbortThread.cpp, 503-504

Active Server Pages (ASP), 428

ActiveX, 333-381, 390-391

ambient properties, 342

ATL, and, 375, 380

building IDispatch object, 342-351, 378

controls. *See* ActiveX controls.

DISIPs, 342

DLLs, 327-328

encapsulation, 339

events, 342

interfaces, 337, 341

Internet Explorer, 422, 425

IUnknown interface, 337-338

methods, 342

- objects, 337
- OLE/OCX/COM, compared, 336-337
- properties, 341
- PX_functions, 365-366, 379-380
- registration, 346, 348, 378
- reuse, 339-340
- Visual Basic, and, 375, 380

ActiveX controls, 351-352. *See also* ActiveX.

- ambient properties, 355
- Control Wizard, 352-354
- custom code, 354
- events, 356-358
- example (Bull control), 360-371
- generated CPP files, 358-359
- Internet Transfer Control, 422-426
- methods, 355-356
- properties, 354-355
- property sheets, 358, 380
- testing/debugging, 359, 379
- using, 371-374, 381

AddPage, 283

AddRef, 337-338, 525

AdjustWindowRectEx, 60, 158, 219

AfxBeginThread, 492-493

AFXCONV.H, 442

AfxEndThread, 492

AfxGetApp, 21

AfxGetResourceHandle, 324-325

AFXPRINT.RC, 127

AFXPRIV.H, 127, 132

AfxRegisterClass, 156, 219

AfxRegisterWndClass, 156, 219

AFX_EXT_CLASS, 324

AFX_MSGMAP_ENTRY, 51

AFX_WS_DEFAULT_VIEW, 158

Aggregation, 339-340

Al Williams' Web site, 521

Ambient properties, 342, 355

Ambient property functions, 356

AmbientBackColor, 365

App Wizards. *See* Custom App Wizards.
Archives, 98-99. *See also* Serialization.
ASP files, 428
ATL (Advanced Template Library), 375, 380
ATM machines, 519
AutoLoad, 198
Automation, 343
AWX file, 292
axcalView.cpp, 372-374

B

BadDlg.cpp, 208-209
BEGIN_INTERFACE_MAP, 528
BEGIN_INTERFACE_PART, 527
BEGIN_PARSE_MAP, 449
BEGIN_PROPPAGEIDS, 358, 366-367
Blocking calls, 393
BMP viewer, 85-88
bmpvidoc.cpp, 86-87
Briefcase reconcile, 524
BSTRs, 443
bull.cpp, 369-371
BullCtl.cpp, 360-364
BullPpg.cpp, 368-369
Buttons, self-draw, 199-201. *See also* Owner-draw buttons.
buttonsView.cpp, 195-198

C

C++ DLL, 442-445
CAbortThread, 504
CancelToClose, 274
CAppWizStepDlg, 299
CArchive, 67-69, 392-393
CArrays, 43-44. *See also* 2D CArray.
CAsyncSocket, 393, 411
CBattleSocket, 409
CBISAPI, 428-429, 456-457
CBitmapButton, 193-195, 198
CBrush, 38
CCmdTarget, 30, 343
CCmdUI, 31

- CCommandLineInfo, 22
- CCriticalSection, 497
- CCryptFile, 95
- CCS_ADJUSTABLE, 248
- CCustomAppWiz, 298
- CDaoRecordSet, 466-471
- CDaoRecordView, 26
- CDataExchange, 239-241, 266
- CDialog, 228
- CDialogBar, 247
- CDib, 84-85
- CDispatchDriver, 443
- CDM_GETFILEPATH, 259
- CDM_GETFOLDERPATH, 259
- CDM_GETSPEC, 259
- CDM_HIDECONTROL, 259
- CDM_SETCONTROLTEXT, 259
- CDM_SETDEFEXT, 259
- CDocTemplate, 32-34
- CDocument, 27-28
- CDynLinkLibrary, 324
- CEditView, 11, 26, 176-185, 222
- CEvent, 497
- CFile, 94, 97
- CFontDialog, 192
- CFormView, 26, 141, 371
- CFrameWnd, 29-32
- CFullList, 153
- CGI program, 390, 426-427
- CHARFORMAT, 192, 222
- ChildFrm.cpp, 184-185, 212-214
- chkpropView.cpp, 277-279
- CHOOSECOLOR, 256
- CHttpFile, 412-413, 421-422
- CHttpServerContext, 450
- CInternetSession, 411-412
- Class factory, 526
- ClickPage.cpp, 287-288
- Client window, 12-13
- CList, 43-44

- CListCtrl, 146, 207
- CListView, 26
- CLSIDs, 378
- CLW file, 244
- CMap, 44-46
- CMDIChildWnd, 29
- CMDIFrameWnd, 29
- CMemFile, 94-95, 97
- CModelessDialog, 229
- CMouseSheet, 290
- CMultiDocTemplate, 32
- CMultiLock, 497
- CMutex, 497
- CNestSplit, 212
- CObject, 38-39
- CoCreateGuid, 297
- COleControl, 358
- COleControlModule, 358
- COlePropertyPage, 359
- Collection classes, 39-46
- Collection helpers, 46-47
- Color dialog, 255-258
- COLORREF, 365
- COM, 336
 - object step-by-step, 529-530
 - support, 526-528
- Combo boxes, self-draw, 199, 202-205
- Command routing, 18-20
- Common dialogs, 254-255
- CompareElements, 47
- CompareItem, 203
- COMPAREITEMSTRUCT, 203
- Component Gallery, 247, 267, 371
- conndotView.cpp, 128-131
- Console applications, 318-319
- ConstructElements, 47
- Containment, 339
- Context menu, 524
- Continuing education, 518-519, 521-522
- Controls

- ActiveX. *See* ActiveX controls.
- defined, 351
- list, 146-153, 220
- owner-draw, 192-193. *See also* Self-draw controls.
- tree/list, in dialogs, 207-209
- Copy hook, 524
- CopyElements, 47
- Core classes, 5, 21-34
- CPacket, 394
- CPen, 38
- CPoint, 37
- CPreviewView, 127, 131-133
- CPrintInfo, 112
- CPropertyPage, 273
- CPropertyPage overrideables, 274
- CPropertySheet, 273
- CreateDispatch, 443
- CreateEx, 154
- CreateNewFrame, 34
- CREATESTRUCT, 155
- CREATE_SUSPENDED, 495, 512
- CRecordSet, 463-466
- CRecordView, 26
- CRect, 37
- CRichEditView, 26, 186-192, 222
- Critical section, 497
- CScrollView, 26, 162-164
 - horizontal scrolling, 171
 - keyboard scrolling, 164-168
 - optimizing scrolling, 168-170
 - scrolling many items, 170-176, 221-222
- CSemaphore, 497
- CShiconwzAppWiz, 303
- CSingleDocTemplate, 32
- CSingleLock, 497
- CSize, 37
- CSocket, 391-392, 409, 411
- CSocketFile, 392
- CSplitterWnd, 211-212, 214
- CStdioFile, 411

- CString, 36
- CSyncObject, 496
- CToolBar, 248, 266
- CTreeCtrl, 207
- CTreeView, 26
- custdlg.cpp, 78-82
- custdlgDoc.cpp, 74-77
- Custom App Wizards, 291-292, 308
 - creating a wizard, 292
 - creating the project, 296-297
 - customizing the customizer, 293-295
 - debugging, 303
 - listings, 299-303
 - macros, 295
 - MFC classes, 298
 - step dialog, 298-299
- Custom color dialog, 256-258
- Custom DDX/DDV routines, 235-246, 266
- Custom dialog, 73-83
- Custom file prompt, 102-103
- Custom serialization, 94-97
- Custom window classes, 156
- customco.cpp, 256-257
- CustomFile.cpp, 260-261
- Customizing common dialogs, 254-258, 267
- Customizing file open, 258-261
- CustomPreview.cpp, 134-137
- CView, 24-27
- CWinApp, 5-8, 21-24, 491
- CWinThread, 491-494
- CWnd, 12, 14, 37-38

D

- DAO, 463, 466-471
- Data maps, 231, 239, 265
- Data object, 524
- Data validation, 235-240
- Databases, 459-486
 - adding/deleting records, 474
 - binding fields to recordset variables, 485
 - binding recordset variables to controls, 485

- computed fields, 473
- DAO record set, 466-471, 484
- example program, 475-482
- ODBC record set, 463-466
- programs without record view, 474-475
- starting database application, 471-472
- dbdemoSet.cpp, 475-476
- dbdemoView.cpp, 476-480
- DCPREV.CPP, 132
- DDV_MinMaxFloat, 241
- DDX.CLW file, 244-245
- DDX/DDV, 231-246
- Deadlock (deadly embrace), 491
- Debugging ISAPI extensions, 445
- DECLARE_CYNCREATE, 275
- DECLARE_INTERFACE_MAP, 528
- DECLARE_OLECREATE, 346
- DECLARE_OLE_CREATE, 528
- DECLARE_SERIAL, 67
- DEF file, 320
- Default, 148
- DEFAULT_PARSE_COMMAND, 449-450, 456
- DeleteItem, 203
- DELETEITEMSTRUCT, 204
- DeleteRow, 184
- Derivation, 340
- Design by difference, 5
- DestructElements, 47
- Developer's Network CD, 83
- Developing ActiveX Web Controls*, 375
- Dialog bars, 246-247
- Dialogs, 225-267
 - common, 254-258
 - customizing tool bars, 248-254
 - DDX/DDV, 231-246
 - dialog bars, 246-247
 - file open, 258-261
 - implementing modeless, 229-231, 264
 - modal vs. modeless, 228
 - tabbed. *See* Property sheets.

- tree/list controls, 207-209
- Dispatcher.cpp, 346-347
- dispatch.h, 347-348
- DISPIDs, 342
- DLLs, 309-332
 - ActiveX, 327-328
 - adding a LIB file, 314
 - C++, 442-445
 - creating ordinary, 315-323
 - DLL library strategy, 316
 - exporting functions/data, 319-321, 331
 - function names/numbers, 315, 330
 - header file, 313-314
 - ISAPI, 437
 - language considerations, 313
 - link process, 313
 - main file, 316-319
 - MFC extension, 324-326, 331
 - optimizing DLL load addresses, 331-332
 - ordinary, 313-315
 - private/shared variables, 321-323
 - static copy of library, 316
- Doc.cpp, 395-404
- DoChar, 177
- Document, 10
- Document object, 9-10
- Document templates, 20-21
- Document/view architecture, 8-9
- DoDataExchange, 231, 235-236, 265, 482
- DoKeyboardSplit, 184
- DoPreparePrinting, 110, 115
- DoPrintPreview, 127
- DoPromptFileName, 70-74
- DoPropExchange, 355, 365
- DoScan, 421-422
- Drawing process, 24-25
- DrawItem, 198, 200
- DRAWITEMSTRUCT, 198-199
- Drop target, 524
- dumbdll.cpp, 317-318

- DUMBDLL.H, 318, 321
- DUMBDLLMFC.H, 318, 321
- DUMPBIN, 315, 330
- DumpElements, 47
- Dynamic creations support, 39
- Dynamic data exchange/dynamic data validation, 231-246
- Dynamic link libraries. *See* DLLs.
- Dynamic splitters, 210-211

E

- EMailDB, 92-93
- emailsDoc.cpp, 90-92
- Encapsulation, 338-339
- Encrypted file class, 95
- EndDialog, 229
- END_INTERFACE_MAP, 528
- END_INTERFACE_PART, 527
- END_PARSE_MAP, 449
- EN_KILLFOCUS, 235-236
- Events, 342
- Exchange routines, 239-240
- Exiting code, 83, 103
- ExitInstance, 5, 8
- ExpandURL, 420
- EXTENSION_CONTROL_BLOCK, 437-438
- ezwpView.cpp, 187-191

F

- File dialog, 255
- File dialog messages, 259
- FILE_FLAG_OVERLAPPED, 498
- Filters, 437
- Flush, 97
- Font dialog, 255
- Footers, 141
- Foreground thread, 511
- Frame windows, 12-14
- FTP (File Transfer Protocol), 388
- FullList.cpp, 151-153
- FWS_ADDTOTITLE, 161-162
- FWS_PREFIXTITLE, 161

G

- GetAmbientProperty, 355
- GetBackColor, 365
- GetBuffer, 37
- GetCharFormat, 192
- GetCharFormatSelection, 222
- GetClientRect, 168
- GetDefaultConnect, 481
- GetDeviceScrollPosition, 168, 171
- GetDialog, 298
- GetDocument, 26
- GetExStyle, 219
- GetExtensionVersion, 437
- GetFile, 97
- GetFirstViewPosition, 27-28
- GetForeColor, 365
- GetIDispatch, 443
- GetIDsOfNames, 345, 443
- GetKeyState, 168
- GetNetView, 27-28
- GetObjectSchema, 94
- GetProcAddress, 314-315
- GetStyle, 60, 219
- GetStyleEx, 60
- GUIDs, 341, 526

H

- HashKey, 46-47
- Headers, 141
- Helix button, 193
- Helper functions, 46-47
- HILO.DLL server, 438-442
- Hints, 25
- HKEY_CLASSES_ROOT, 378
- Horizontal scrolling, 171
- hostsocket, 409
- HTTP, 388-390
- HTTP headers, 390, 423-424
- HTTP response codes, 425-426
- HttpExtensionProc, 437, 442

I

- Icon handler, 530-536
- IconHandler.cpp, 533-535
- IconHandler.h, 535-536
- ICs (integrated circuits), 519-520
- IDispatch, 341
- Idle time processing, 498, 514
- ID_EDITMODE, 133
- ID_FILE_OPEN, 70
- IDR_MAINFRAME, 31
- IExtractIcon, 531
- IID, 341
- IMPLEMENT_DYNCREATE, 67
- IMPLEMENT_OLECREATE, 346, 528
- IMPLEMENT_SERIAL, 67, 88-89, 93
- Import libraries (IMPLIBS), 313
- InitialUpdateFrame, 34
- InitInstance, 5, 8
- Inproc ActiveX servers, 327
- INTERFACE_PART, 528
- InterlockedDecrement, 496
- InterlockedIncrement, 496
- Internet, 383-457
 - ActiveX control (Internet Transfer Control), 422-426
 - higher-level protocols, 411-422
 - HTTP, 388-390
 - ISAPI extensions, 426-451, 455-456
 - MFC sockets, 391-411
 - protocols, 388
 - sockets, 387-388
 - TCP/IP, 386-387
 - useful Web sites, 521-522
- Internet Explorer, 422, 455
- Internet Resources for Windows Developers Web site, 522
- Internet Transfer Control, 422-426
- InvokeHelper, 443
- IPersistFile, 531
- ISAPI, 426-451, 455-456
- ISAPI extension wizard, 449
- ISAPIMFC.CPP, 446-448

IServer.cpp, 430-436
IsKindOf, 421
IUnknown, 337-338, 525

J

Java, 390

K

Keyboard scrolling with CScrollView, 164-168
KeyScroll, 164, 168

L

Label editing, 207, 223
linkckView.cpp, 413-420
List boxes, self-draw, 199, 202-205
List controls, 146-153, 207-209, 220
Live data validation, 235-236, 265
LiveDialog.cpp, 233-235
LoadFrame, 31
LoadLibrary, 313-314
Lock objects, 497-498
LOGFONT, 192

M

m_bAutoDelete, 493, 495, 511
m_IpCmdLine, 21
m_nShellCommand, 23
MainFrm.cpp, 250-254
MapObject, 89
Mapping modes, 114-116, 123
MDI programs, 12-13
mdlspropView.cpp, 288-290
MeasureItem, 200, 202-203
MEASUREITEMSTRUCT, 202
Menus, self-draw, 199, 205-207. *See also* Owner-draw/menus.
Message map function signatures, 52-54
Message maps, 14-18
Message routing, 18-20
Methods, 342
METHOD_PROLOGUE, 527-528
MFC-based ISAPI extension, 446-448

- MFC extension DLLs, 324-326, 331
- MFC FAQ, 522
- MFC source code, 73
- MFC TechNotes, 522
- MFC treasure map, 35
- mfctttView.cpp, 117-122
- MFT_BITMAP, 206
- Microsoft Web site, 522
- mlessdlg.cpp, 325-326
- MM_ANISOTROPIC, 116
- MM_HIENGLISH, 116
- MM_HIMETRIC, 116, 140
- MM_ISOTROPIC, 116, 123
- MM_LOENGLISH, 116, 140
- MM_LOMETRIC, 116
- MM_TEXT, 115-116
- MM_TWIPS, 116
- Model view controller architecture, 10
- modeless.cpp, 230-231
- ModelessDlg.cpp, 326-327
- MouseSheet.cpp, 284-287
- multeditDoc.cpp, 180-183
- multeditView.cpp, 177-180
- Multiple views, 56-57
- Multithreading, 487-514
 - alternatives to threading, 498-499
 - autodestruction of threads, 493, 495, 511
 - deadlock (deadly embrace), 491
 - example program, 500-506
 - making message boxes appear on top, 511
 - making windows appear on top, 511
 - manipulating threads, 493, 495
 - problems with threads, 490-491, 507
 - return value, 495
 - suspended threads, 495, 511-512
 - synchronization objects, 496-498, 512-514
 - terminating a thread, 493, 510
 - threads vs. processes, 490
 - user-interface threads, 493
 - worker threads, 492, 510

Mutex, 497

N

Navigating Objects at runtime, 34-36

Nesting splitters, 211-214, 224

New document types, 55-56

NNTP (Network News Transfer Protocol), 388

Notepad, 12-13

O

ODBC control panel applet, 472

Object-oriented programming, 335, 338. *See also* ActiveX.

Object serialization, 88-89

OCXs, 337

ODBC, 463-466, 484

ODButton.cpp, 200-201

ODCombo.cpp, 204-205

ODStatic.cpp, 201-202

OLE, 336

OLE automation types, 341

OLE_COLOR, 365

OLE support, 526-528

OLEISAPI, 428

OLEIVERB_PROPERTIES, 358

ON_COMMAND, 16

ON_COMMAND_EX, 16

ON_COMMAND_EX_RANGE, 16

ON_COMMAND_RANGE, 16

ON_CONTROL, 18

ON_CONTROL_RANGE, 18

ON_MESSAGE, 18, 50

ON_NOTIFY, 17, 249

ON_NOTIFY_EX, 17

ON_NOTIFY_EX_RANGE, 17

ON_NOTIFY_RANGE, 17

ON_PARSE_COMMAND, 448-449, 456

ON_PARSE_COMMAND_PARAMS, 449, 456

ON_REGISTERED_MESSAGE, 18, 50

ON_UPDATE_COMMAND_UI, 16, 30, 133, 410

ON_UPDATE_COMMAND_UI_RANGE, 17

ON_WM_COMMAND, 232

ON_WM_DRAWITEM_REFLECT, 202
OnApply, 274
OnBeginPrinting, 111, 115
OnCancel, 274
OnChar, 177
OnCharEffect, 187
OnConnect, 408
OnCreate, 154
OnDraw, 24, 113-116, 365
OnDrawItem, 200
OnEndPrinting, 115-116
OnFilePrint, 115
OnFilePrintPreview, 125
OnGetMinMaxInfo, 154, 157-158
OnHost, 408
OnIdle, 498, 514
OnInitialUpdate, 219
OnKillActive, 274
OnMeasureItem, 200
OnNCCreate, 154
OnOK, 274
OnOpenDocument, 70-71
OnPaint, 24
OnPrepareDC, 24, 113, 115-116
OnPreparePrinting, 110, 115
OnPrint, 113, 115-116, 141
OnQueryCancel, 274
OnReset, 274
OnSaveDocument, 72
OnScan, 421-422
OnSetActive, 274
OnStepChanged, 366
OnUpdate, 24-25
OnUpdateCharEvent, 187
OnUpdateCmdUI, 162
OnVScroll, 171
OpenDocumentFile, 34, 70
OpenURL, 411, 421, 423, 455
OutputStream, 299
OVERLAPPED, 498-499

Overlapped I/O, 498-499

Owner-draw

 buttons, 199-201. *See also* Self-draw/buttons.

 controls, 192-193. *See also* Self-draw/controls.

 menus, 199, 205-207. *See also* Self-draw menus.

P

Page1.cpp, 280

PAGE_INFO, 132

Parse map macros, 449

Parse maps, 446, 448, 456

ParsedQueryString, 445

ParseURL, 421

Parsing command line parameters, 59

Persistence, 66, 365

Polymorphism, 338, 340

POP3/IMAP, 388

PostQuitMessage, 493, 495

PreCreateWindow, 154, 218

Print dialog, 140, 255

Print preview, 124

 advanced customization, 131

 customizing, 142

 deriving the class, 132

 editing, 133-138

 example (connect the dots), 127-131

 internals, 132-133

 stripping down, 125

 tool bar, 141-142

Printing, 107-142

 example (tick-tac-toe), 117-124

 headers/footers, 141

 mapping modes, 114-116, 123

 overview of, process, 110-114

 print dialog, 140

 print preview. *See* Print preview.

 scaling, 114, 140

 screen/printout differences, 141

Private documents, 56

Process, 490

Program.com Web site, 522

- Properties, 341
- Property sheets
 - ActiveX, 358
 - creation, 306
 - modeless, 283-290
 - overview, 272
 - single template, 274-279
 - wizard mode, 279-283
- PROPPAGEID(CLSID_CColorPropPage), 367, 380
- PROPPAGEID(CLSID_CFontPropPage), 380
- Protocols, 388
- PX_exchange functions, 365-366

Q

- QueryInterface, 337-338, 525
- Quick view, 524

R

- Record set, 462
- Reference.com Web site, 522
- Reference materials, 521-522
- Registering custom DDX/DDV, 246
- Release, 337, 525
- Requery, 481
- Resource IDs, 194
- Restricting window size, 156-161
- ResumeThread, 495, 512
- Reuse, 338-340
- Run, 23-24
- RUNTIME_CLASS, 493

S

- Saving document data, 11
- Scaling, 114, 140
- ScanForAvailableLanguages, 298
- scrollerView.cpp, 164-168, 171-176
- Scrolling. *See* CScrollView.
- SDI programs, 12
- Search order, 19
- Secondary document templates, 34
- Selective validation, 239

Self-draw

- controls, 193-198
- buttons, 199-201
- list/combo boxes, 199, 202-205
- menus, 199, 205-207
- overview, 198-199
- static controls, 199, 202-202

Semaphore, 497

Serialization, 63-105

- altering archive format, 103-104
- CArchive, 67-69
- classes, 67, 102
- custom, 94-97
- custom dialog, 73-83
- custom file prompt, 102-103
- existing (custom) code, 83, 103
- loading and saving, 72
- MFC sockets, and, 392-393, 454
- multiple versions, 89-94, 104
- objects, 88-89
- persistence vs. storage, 66-67
- portability issues, 98
- simple customization, 97-98

SerializeElements, 46-47

SetCharFormat, 222

SetCustomAppWizClass, 298

SetForegroundWindow, 511

SetIcon, 198

SetMenuItemBitmaps, 207

SetModified, 274

SetModifiedFlag, 27-28

SetNumberOfSteps, 298

SetScaleToFitSize, 164

SetScrollSizes, 162

SetSupportedLanguages, 298

SetWindowText, 161

Shell extensions, 524-525, 532

shiconwzaw.cpp, 299-301

Signaled state, 496

Simple customization, 97-98

- Simple serializable types, 99
- sizeView.cpp, 159-161
- Sizing a view, 60, 219-220
- Smalltalk, 10
- Sockets, 387-388. *See also* MFC sockets.
- SplitRow, 184
- Startup, preventing new documents, 58
- StateChanged constants, 427
- Static controls, self-draw, 199, 201-202
- Static libraries, 312
- Static splitters, 210-211
- statuspg.cpp, 276
- STDMETHOD, 528
- STDMETHODIMP, 528
- Step, 365-366
- Step1Dlg.cpp, 301-303
- Stock members, 342
- Stream, 66
- StringFormCLSID, 297
- SubclassDlgItem, 153
- Subclassing, 153
- Supporting objects, 36-39
- Suspended threads, 495, 511-512
- SuspendThread, 495, 512
- Synchronization objects, 496-498, 512-514

T

- Tabbed dialogs. *See* Property sheets.
- TBN_GETBUTTONINFO, 249, 254
- TBN_QUERYDELETE, 249, 254
- TBN_QUERYINSERT, 249, 254
- TBN_RESET, 249, 254
- TBN_TOOLBARCHANGE, 249
- TCP/IP, 386-387
- TechNotes, 522
- Telnet, 388
- Template-based collections, 43-46
- Templates, 40
- Text editor window. *See* CEditView, CRichEditView.
- theApp, 21
- Threads, 490. *See also* Multithreading.

- Tool bar notifications, 149
- Tool bars, 247-254
- TranslateColor, 365
- Treasure map, 35
- Tree controls, 207-209
- 2D CArray, 61
- Type library, 341
- typedef, 41, 61

U

- Unlock, 498
- Unsignaled state, 496
- Update, 481
- UpdateAllViews, 27-28, 233
- UpdateData, 232, 283, 485
- UpdateWindow, 410
- URL, 389
- User.cpp, 349-351
- User-interface threads, 493
- User messages, 50-55
- UUIDGEN, 378, 526
- UUIDs, 341, 378

V

- Validation routines, 235-240
- validView.cpp, 236-239, 241-243
- VERSIONABLE_SCHEMA, 93, 104
- Versions, 89-94, 104
- View.cpp, 405-407
- View size, 60
- VIEWPREV.CPP, 132
- Visual Developer Magazine*, 522
 - Web site, 522

W

- Web Developer's Virtual Library Web site, The, 522
- Web sites
 - www.al-williams.com, 521
 - www.microsoft.com, 522
 - www.program.com, 522
 - www.reference.com, 522

www.r2m.com/windev, 522

www.stars.com, 522

Window operations

class names, 219

custom window classes, 156

OnUpdateCmdUI, 162

restricting window size, 156-161

scrolling. *See* CScrollView.

setting custom icon/cursor/background, 218-219

setting styles/initial conditions, 155-157, 218

setting the title, 161

splitter windows, 209-215

window creation process, 154

WinInet, 411, 455

WinMain, 23-24

Wizard buttons, 279

Wizard-style property sheet, 274-279. *See also* Custom App Wizards.

wizView.cpp, 281-283

WM_?BUTTONDOWN, 357

WM_?BUTTONUP, 357

WM_CHAR, 357

WM_COMMAND, 14-15, 30, 232-233

WM_ERASEBKGND, 219

WM_GETMINMAXINFO, 220

WM_IDLEUPDATECMDUI, 30, 410

WM_INITDIALOG, 267

WM_KEYDOWN, 357

WM_KEYUP, 357

WM_LBUTTONDOWN, 148

WM_MOUSEMOVE, 357

WM_NCDESTROY, 229

WM_NOTIFY, 14, 29

WM_PAINT, 24-25

WM_SETCURSOR, 133, 219

WM_SIZE, 156

WM_TIMER, 193

WNDCLASS, 157

Worker threads, 492, 510

WriteBlock, 299

WriteLine, 299

WS_MAXIMIZE, 154

WS_MAXIMIZEBOX, 154

WS_THICKFRAME, 154

Table of Contents

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.