

Applications and Environments

Welcome to Applications and Environments. Here we will plan out an MFC application, determine whether it will have a Dialog, Single Document (SDI), or Multiple Document (MDI) interface, and help distinguish it from all the other MFC applications generated by the Developer Studio by giving it some pizzazz.

Example 1 Planning Your MFC Application We will devise a strategy for using the Developer Studio, et al., to turn your application idea into an actual application.

Example 2 Creating an MFC Application Using the AppWizard We will use the Application Wizard to generate a set of classes and resources that will become the foundation of your MFC application.

Example 3 Creating a Class Using the ClassWizard We will use the Class Wizard to add classes to your application.

Example 4 Initializing the Application Screen We will take control of the initial size and placement of your application's window.

Example 5 Saving the Application Screen We will save the size and position of your application's window for its next execution.

Example 6 Processing Command Line Options We will convert command line flags into Boolean variables we can use in our application.

Example 7 Dynamically Changing Application Icons We will change your application's icon, which not only appears in the upper left corner of the application, but also appears in your system's task bar.

Example 8 Prompting the User for Preferences We will prompt our user for your application's options.

Example 9 Saving and Restoring User Preferences We will save your application's options in the system registry.

Example 10 Terminating Your Application We will look at a way to control how your application exits.

Example 11 Creating a Splash Screen We will create an initial screen for your application that presents its name and affiliation.

Example 1 Planning Your MFC Application

Objective

You would like to create an application using Visual C++, the MFC libraries, and the Developer Studio's wizards and editors.

Strategy

We will start by deciding what type of MFC application would best suit your needs: Dialog, SDI, MDI, or none of the above. We will then pick the best view and document for that application. Next, we'll review other ways your user can interact with your MFC application and what Developer Studio editors you can use to add those interfaces. If the functionality in your application can be shared with other applications, we'll look at your library

choices. And finally, since every application that the Developer Studio creates looks almost alike, we'll explore several features you can add to your application to make it stand out.

Steps

Pick an Application Type

1. There are three types of MFC applications you can create automatically with the Developer Studio's AppWizard utility (as seen in Example 2): Dialog, SDI, or MDI. Manually, you can create any type of hybrid application. For more details on deciding just what type of application to choose, see Chapter 2. To make a quick decision now, try following these guidelines.
 - If you are creating an application with a limited need for a user interface, or if you want your interface to be totally unique, then create a Dialog Application. Typical Dialog Applications include applications that configure hardware devices, screen savers, and game programs.
 - If your application will be editing a document, you should pick one of the other two application types. In this context, "editing a document" is meant in the broadest possible sense. The document we refer to here can be a text file, a spread sheet file, one or more tables in a third-party database, or your own proprietary binary file. It can even be the stored settings of a vast array of hardware devices. Editing simply means adding, deleting, or modifying the data in any one of these types of documents.
 - A Single Document Interface (SDI) Application allows you to edit only one document at a time. If your application can only physically edit one document at a time, as is the case of an application that monitors an array of hardware devices, then you should pick an SDI interface. Otherwise, you should create an MDI application — even if there doesn't initially appear to be any benefit to editing more than one file at a time.
 - A Multiple Document Interface (MDI) Application allows you to edit several documents at once. An MDI application isn't that much more complicated than an SDI application, and your user gets the added convenience of being able to at least view more than one document at a time.

2. If you have decided to create a Dialog application, you're done with this example. A Dialog application has no view or document, and there's really not much more to the interface. However, you might look at the rest of this example to see some of the ways to add pizzazz to your application.

Pick a Type of View

1. If you picked an SDI or MDI application, you must choose the type of view you want before you exit the AppWizard. In the last step of the AppWizard, you can choose one of the following view classes.
 - For a simple text editor application, pick `CEditView`.
 - For an application that can edit rich text format (RTF) files, pick `CRichEditView`. (This selection will also cause your application to pick `CRichEditDoc` for your document class.)
 - For a graphic application, pick `CScrollView`.
 - For a simple monitoring or accounting application, pick `CListView` (Example 36).
 - To start the creation of an Explorer-type interface, pick `CTreeView`. (You can manually add a `CListView` in a later step.)
 - To create a view out of a dialog box template, pick `CFormView`. (A dialog box is a window inhabited by several other Control windows, such as buttons and edit boxes. See Chapter 1 for more on this topic.)
2. You can also indirectly pick either the `CRecordView` or `CDaoRecordView` class for your view in an earlier step of the AppWizard, in which you decide what database support to add to your application. If you pick either of the "Database View" options in the second step, this view is added to allow you to easily access the records in an ODBC or DAO database.

Pick a Document Type

1. There are three basic types of documents your application can interact with: flat files, serialized files, or databases. The choice is usually made for you by the type of application you intend to write. Let's review the choices.

- A flat binary or text file is the simplest of documents your application can support and can usually follow any storage method you can dream up. To see what MFC classes support flat files, see Example 63.
- Serialized files represent MFC's organized way of storing binary files so that they can be easily retrieved, even when they were made by a previous version of your application. Any type of document, whether it's a text file, a spreadsheet file, or database data, can be stored this way (Examples 66 to 70).
- Your application can also work with the Microsoft Jet Engine Database Management System (DBMS) and any other third-party ODBC-compliant DBMS (Examples 72 and 73).

Other Considerations

1. The view represents the primary method your user has to interact with their document. However, there are several other features you can add to your application to allow them to interact.
 - You can add commands to the main menu using the Menu Editor. (See Examples 12 and 13 for more on this topic. For popup menus, see Example 21.)
 - You can add toolbars and buttons using the Toolbar Editor (Example 22).
 - You can add modal dialog boxes, which allow your user to enter detailed information while suspending your application (Example 40).
 - You can add modeless dialog boxes, which are dialog boxes that don't suspend your application (Example 41).
 - You can add dialog bars, which are a hybrid of toolbar and modeless dialog box, allowing your user to dock this type of dialog box (Example 45).
 - You can add Property Sheets, which are Window's conventional way to allow a user to enter and save their preferences (Example 8).
2. As you start to add layer upon layer of functionality to your MFC application, you may start to wonder which class should contain what functionality. For example, which class should process what command message and what window message? You can easily access the data and functionality of one class from another, as seen in Appendix D, but where should that functionality initially reside? Here's a rough guide to help you decide.

- The Application Class, derived from `CWinApp` and controlling no window, should have precious little additional functionality itself, other than to control the creation and destruction of your application. This can include processing command line flags and providing a customized method of opening documents. The Application Class also serves some application-wide services such as background processing and subclassing.
- The Mainframe Class, derived from `CFrameWnd` and controlling your application's Main window, should be in charge of all application-wide interfaces, including the toolbars, status bar, menu, and dialog bars. However, if any of these bars has additional functionality, it should be encapsulated in its own class. Support for user preferences is also found typically in the Mainframe Class.
- The Document Class, derived from `CDocument`, should contain any data pertaining to your application's document. For true C++ encapsulation, the Document Class shouldn't allow direct access to its data — not even from the View Class — but instead should contain wrapper functions to access its data. The Document Class should also contain all of the functionality necessary to load and save a document, from simple binary files to ODBC databases. If your application does nothing but access an ODBC database, your Document Class may contain nothing more than the logic necessary to open and close that database, since the database is the main repository of your data. The Document Class should be an island unto itself, getting its information from a storage device and handing it out to the view, but rarely storing information in another class.
- The View Class, derived from `CView`, should contain all of the logic necessary to view and edit the data in the Document Class. Any menu or toolbar commands that operate specifically on the document, such as cut or paste, should be processed here. All mouse messages affecting the view should be processed here. All drawing, reporting, editing, selecting, and printing should be done here. All dialog boxes and popup menus should be spawned here. If this or any class starts to become massive, you should factor any common functionality into a new base class. Create a new `CMyBaseView` class and stuff some basic functionality there. Or you can encapsulate some functionality into its own class. The functions that select, cut, and paste in the view make a good candidate for its own class.
- Other Classes should encapsulate as much of their own functionality as possible. Dialog classes should contain everything they need to prompt

the user, as should dialog bars, toolbars, and status bar classes. A self-drawn control should be drawn from within its own class.

- Any time you factor out common functionality from your application into a base class, you can put the new base class into an MFC Extension Class. You can then move the new MFC Extension Class into a Dynamically Linked Library, from which applications can share this functionality.

NOTE: As another rule of thumb, if you find that one class is constantly accessing the functions and data of another class, it would probably behoove you to move that functionality to the other class.

3. And finally, there are several ways to snazz up your application to help distinguish it from the 3,700 other applications created every day using the Developer Studio.
 - Add a splash screen that appears when your application first starts up (Example 11).
 - Show an animation of progress rather than a “Please Wait” message (Example 43).
 - Make your toolbar buttons larger and stick words in them (Examples 24 and 25).
 - Put other types of status messages in the status bar (Example 31).
 - Add controls other than buttons to your toolbar or use a dialog bar, instead (Examples 28 and 45).

Notes

- Writing an application using MFC is a constant battle to decide whether to use an existing MFC feature or to write one yourself. Knowing the basics from Section I, you could, for example, easily write your own button control window. You would then have total control over the final product — there’s nothing undocumented. If it doesn’t perform as expected, you can follow the debugger right down into it to find out why and change it, if desired. However, the work of creating a button control has already been done and tested forever. It’s also more universally found and understood and allows someone else to more easily pick up your code where you left off. I highly recommend that if the MFC method of

providing a function isn't readily apparent, you should continue the search until you either find one or can verify that one does not yet exist. An application that's written with MFC but looks nothing like it on the inside is a waste of the technology that went into it. When you use MFC and the Developer Studio the way it was intended, you'll find yourself creating applications faster than you ever thought you could. Thank you for your support.

CD Notes

There is no accompanying project on the CD for this example.

Example 2 Creating an MFC Application Using the AppWizard

Objective

You would like to create a Dialog, an SDI, or an MDI application using the Developer Studio's AppWizard.

Steps

Create an Application Using the AppWizard

1. Click on the Developer Studio's File/New menu commands to bring up the New dialog box. Select MFC AppWizard (exe), then enter the name and the directory in which you want to create your project (Figure 5.1). Be careful — this name is used externally and internally in almost all of your project files, so any typos will be difficult to fix later.

Figure 5.1 Specify your application's filename and location.



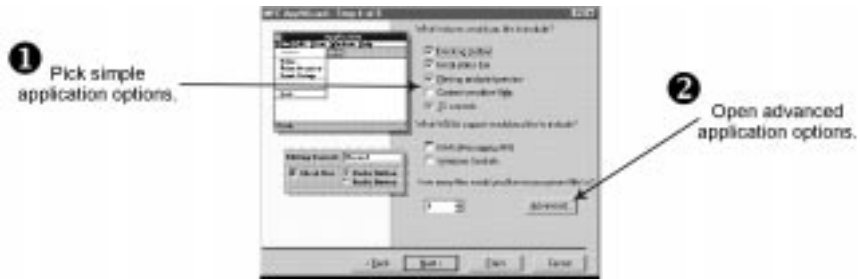
2. Pick the application type in the first step of the AppWizard (Figure 5.2). If you don't already know, see the previous example for an idea of application type on which to base your project. The rest of this example assumes you have selected an SDI or MDI application.

Figure 5.2 Pick your application type.



3. The next AppWizard step allows you to specify what type of database support you want in your application. Picking Header Files Only causes the AppWizard to simply add the MFC classes that support database access. (You can then use Example 72 or 72 to access an ODBC or DAO database.) Picking Database View without File Support or Database View with File Support causes the AppWizard to create a simple database editor, with a special View and Document class. If you select Database View without File Support, the AppWizard won't add the standard file open commands to your application's menu (i.e., File/New, File/Open, etc.). In theory, if you're only accessing a database, you won't need these commands anyway — the correct database will be opened automatically when the application starts. However, if your application will be accessing both flat files and database files, you should select Database View with File Support.
4. The next AppWizard step allows you to specify what type of COM support you want in your application. For the examples in this book, simply take the default options.
5. The next AppWizard step allows you to pick several general application options (Figure 5.3). You can choose whether or not your application will have a toolbar or status bar, will add print commands to your menu, and will include support for e-mail or network communications. The Recent File List is a list of the last n files your application opened, which can be automatically maintained by your application. You get to determine what n is here. Click on the Advanced button for more advanced options.

Figure 5.3 Pick your application's options.



6. The first page of the advanced options allows you to pick the title that will appear in your application's caption bar. If you are creating an application that will be serializing its documents to disk, you can pick the file extension that your application appends to those documents. You can then edit the text that will appear in the filter field of the File Dialog that appears when opening or saving documents (Figure 5.4).

Figure 5.4 Specify your application's title, default file extension, and File Dialog text.



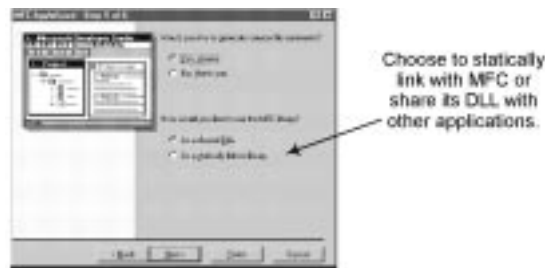
7. The second page of the advanced options allows you to add view splitting capabilities to your application, which will provide your user with a menu command that will allow them to dynamically split their view (Example 37). You can also determine whether your application's Main window or Child windows are initially maximized or minimized and whether your user can resize them (Figure 5.5 and Example 4).

Figure 5.5 Specify your application's Frame Window Options.



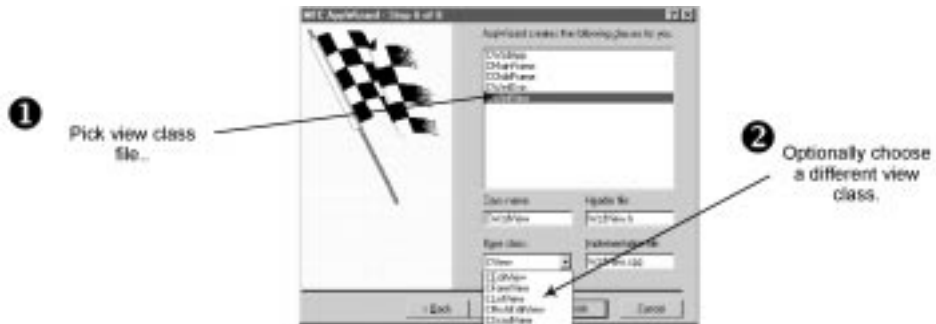
8. In the next step of the AppWizard, you must decide whether to statically link with the MFC library or to link with a shared MFC DLL, instead (Figure 5.6). Statically linking to MFC makes your application much larger, but you would never have to make sure the correct version of the MFC DLL is currently installed on your user's system. If you plan to create your own DLL's that also use the MFC library, you must link to the MFC DLL.

Figure 5.6 Choose how to link to MFC.



9. In the final AppWizard step, you are allowed to change your application's View Class. For a description of the choices, see the last example. For any View Classes not listed, pick the default `CView` class — you can edit the name later (Figure 5.7).

Figure 5.7 Pick a View Class.



10. The AppWizard will now proceed to create all the classes you need to create a fully-executable, albeit feature-poor, application. Simply click on the Studio's Build/Build All menu commands to create your executable.

Notes

- If you forget to add something to your application using the AppWizard, you can still add it manually later on, although it isn't nearly as easy. To find out what you have to add to your application, start by creating two new projects, one with the desired feature and one without. Then do a difference on the source code generated for these two projects. (A version control package would help.) Simply incorporate the differences into your application.
- Except where noted, most of the examples in this book are based on creating an MDI application using all of the defaults.
- If you do want to change the name of your application later, start by deleting all of the binary files in your directory. Then use a search and replace utility that can span multiple files to replace the old name with the new name. Make sure you replace all forms of the project name, including all-capitals, all-lowercase, and mixed case. (Search and replace utilities are available on the Internet.)

CD Notes

- There is no accompanying project on the CD for this example.

Example 3 Creating a Class Using the ClassWizard

Objective

You would like to add a class to your MFC application to either extend an MFC class or to exist on its own.

Steps

Extend an Existing MFC Class

1. Click on your Developer Studio's View/ClassWizard menu commands to open the MFC ClassWizard dialog. Then click on the Add Class button (Figure 5.8). A drop-down menu will appear from which you should select New... to open the New Class dialog box.

Figure 5.8 Create a new Class with the ClassWizard.



2. Enter the name of your new class. Add a “C” to the start of your class name. (The ClassWizard will omit this “C” when creating your class’s .h and .cpp files.) Then select the base class from the list of available MFC classes (Figure 5.9). If you choose `CRecordSet`, the ClassWizard will also lead you through the steps necessary to bind your class with a database table. To derive from the `CWnd` class, choose “generic `CWnd`.” To derive from `CSpinnerWnd`, choose “splitter.” If the MFC class from which you want to derive is not listed (as is the case with `CToolBar`), pick a similar name (such as `CToolBarCtrl`) and then edit the resulting files.

Notes

- To incorporate a class from another project, copy the appropriate files from that project's directory. The ClassWizard won't recognize this new class until you do one more step: delete the .clw file in your project and reinvoke the ClassWizard. When the ClassWizard can't find its .clw file, it will ask you if you want to recreate it. Answer Yes.
- The ClassWizard in version 6.0 of the Developer Studio automatically updates its .clw file.

CD Notes

- There is no accompanying project on the CD for this example.

Example 4 Initializing the Application Screen

Objective

You would like to position and size your application's initial screen.

Strategy

We have two options. The first is to make the appropriate choices in the AppWizard's advanced options when creating the application. If, however, you want to change your selection in an existing application, we will add code to the CMainFrame's PreCreateWindow() to control the initial position and size of our application's main window.

Steps

Use the AppWizard

1. Refer to Example 2, the 7th step, on page 136. Click on the Advanced button and select the Window Styles tab. Picking a Thick Frame allows your user to resize your application's window by dragging the lower-right corner. Picking Minimized or Maximized forces your application's window to be initially minimized or maximized.

If you want to change your choices later, you will need to edit the `CMainFrame::PreCreateWindow()` function directly. Direct editing also allows you to make some additional changes to your application's initial appearance.

Edit `CMainFrame::PreCreateWindow()`

1. To center your application and make it fill only ninety percent of your screen, you can use the following code.

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // center window at 90% of full screen
    int xSize = ::GetSystemMetrics (SM_CXSCREEN);
    int ySize = ::GetSystemMetrics (SM_CYSCREEN);
    cs.cx = xSize*9/10;
    cs.cy = ySize*9/10;
    cs.x = (xSize-cs.cx)/2;
    cs.y = (ySize-cs.cy)/2;
    return CMDIFrameWnd::PreCreateWindow(cs);
}
```

2. If you would also like to eliminate the document's title from your application's title bar, add the following to `PreCreateWindow()`.

```
cs.style &= ~ FWS_ADDTOTITLE;
```

3. If you would also like to remove the minimize and maximize buttons from your application's title bar, add

```
cs.style &= ~(WS_MAXIMIZEBOX|WS_MINIMIZEBOX);
```

4. If you would like to make the size of your application fixed, such that dragging the lower-right corner of the window would have no effect, then add

```
cs.style &= ~WS_THICKFRAME;
```

5. If you would like your application to be maximized when initially executed, then locate `ShowWindow()` in your application class and change it to use the `SW_SHOWMAXIMIZED` flag instead of `m_nCmdShow`.

```
pMainFrame->ShowWindow(SW_SHOWMAXIMIZED); //or SW_SHOWMINIMIZED
pMainFrame->UpdateWindow();
```

6. If you would like to initially maximize a child window in an MDI application, add `PreCreateWindow()` to your `CChildFrm` class and add the following to it.

```
BOOL CChildFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.style = WS_CHILD | WS_VISIBLE | WS_OVERLAPPED |
        WS_CAPTION | WS_SYSMENU | FWS_ADDTOTITLE |
        WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX |
        WS_MAXIMIZE;
    return CMDIChildWnd::PreCreateWindow(cs);
}
```

Notes

- If you initially maximize your application's window, you should also set an initial size for it in `CMainFrame::PreCreateWindow()`. When your user clicks on your application's Restore button, your application's window will snap down to whatever size you set in `PreCreateWindow()`.
- This example always sets the initialize size of your application's window to a fixed size and position. Any changes your user makes to the window size or position are not saved. To save the window size and position, see the next example. If you use the next example, however, you should still use this example. The first time your application runs on a system, it won't have any saved settings, so it will need to use these initial settings.
- If you don't set an initial size and position for your window, the Windows operating system will pick one itself based on a cascading algorithm. Each new application's window is created just to the right and below the last application as shown in Chapter 1.

CD Notes

- When executing the project on the CD, the application will be initially maximized (fill the screen) and unsizeable.

Example 5 Saving the Application Screen

Objective

You would like to save the size, position, and status of your application's screen, including the location and size of any toolbars or dialog bars so that they can be restored the next time your application runs.

Strategy

When our application closes, we will save the size and position of our main window as well as the status of the toolbars and status bar to a location in the system registry. Then, when our application is executed again, we will retrieve this information and restore our window and toolbars, et al.

Steps

Save the Settings

1. First, we will define the location in the system registry where we will be saving this information in a global include file. "Company" would be your company's name.

```
#define COMPANY_KEY "Company"  
#define SETTINGS_KEY "Settings"  
#define WINDOWPLACEMENT_KEY "Window Placement"
```

2. In the `InitInstance()` member function of our application class, add `COMPANY_KEY` to `SetRegistryKey()`.

```
SetRegistryKey(COMPANY_KEY);
```

3. Use the ClassWizard to add a `WM_CLOSE` message handler to your `CMainFrame` class. There you can save your bar positions and sizes using `SaveBarState()`. To get your application's current size and position, use

`GetWindowPlacement()` and save its results to the system registry using `WriteProfileBinary()`.

```
void CMainFrame::OnClose()
{
    // save state of control bars
    SaveBarState("Control Bar States");
    // save size of screen
    WINDOWPLACEMENT wp;
    GetWindowPlacement(&wp);
    AfxGetApp()->WriteProfileBinary(SETTINGS_KEY,
        WINDOWPLACEMENT_KEY, (BYTE*)&wp,
        sizeof(WINDOWPLACEMENT));
    CMDIFrameWnd::OnClose();
}
```

Restore the Settings

1. To restore your toolbars to their original state after reexecuting your application, add the following to the start of the `OnCreate()` member function in your `CMainFrame` class.

```
LoadBarState("Control Bar States");
```

2. To restore your application's main window from the system registry, locate `ShowWindow()` in your Application Class and replace it with the following code. Note that we now use `SetWindowPlacement()` to restore the main window to its original size and position.

```
BYTE *p;
UINT size;
WINDOWPLACEMENT *pWP;
if (GetProfileBinary(SETTINGS_KEY, WINDOWPLACEMENT_KEY, pWP, &size))
{
    pMainFrame->SetWindowPlacement(pWP);
    delete []pWP;
}
else
```

```
{  
    pMainFrame->ShowWindow(m_nCmdShow);  
}  
pMainFrame->UpdateWindow();
```

Notes

- To save other options to the system registry, as well as for more on accessing the system registry from your application, see Example 9.
- This example will only position your application's main window after your user has executed your application once. To initialize your application window for the first time, please see the previous example.

CD Notes

- When executing the project on the CD, you can reposition and resize the main window, then exit the application. When you reexecute the application, whatever size or position it had on terminating will be restored.

Example 6 Processing Command Line Options

Objective

You would like your application to process command line flags as seen here.

```
>myapp /c /d
```

Strategy

An MFC application already processes several standard flags using the `ParseParam()` member function of the `CCommandLineInfo` class. To add our own flags while still supporting these other flags, we will derive our own class from `CCommandLineInfo` and then override `ParseParam()`.

Steps

Create a New CCommandLineInfo Class

1. Use the ClassWizard to create a new class derived from CCommandLineInfo. Add a Boolean or string member variable to this new class for each new flag your application will process.

```
class CWzdCommandLineInfo : public CCommandLineInfo
{
public:
    BOOL m_bAFlag;
    BOOL m_bCFlag;
    BOOL m_bDAFlag;
    CString m_sArg;
```

2. Also add a ParseParam() function to override the base class's ParseParam() function.

```
// Operations
public:
    void ParseParam(const TCHAR* pszParam,BOOL bFlag,BOOL bLast);
};
```

3. Implement ParseParam() as follows.

```
void CWzdCommandLineInfo::ParseParam(const TCHAR* pszParam,
    BOOL bFlag,BOOL bLast)
{
    CString sArg(pszParam);
    if (bFlag)
    {
        m_bAFlag = !sArg.CompareNoCase("a");
        m_bCFlag = !sArg.CompareNoCase("c");
        m_bDAFlag = !sArg.CompareNoCase("da");
    }
}
```

```

        // m_strFileName gets the first nonflag name
        else if (m_strFileName.IsEmpty())
        {
            m_sArg=sArg;
        }
        CCommandLineInfo::ParseParam(pszParam,bFlag,bLast);
    }

```

Note that the `pszParam` argument contains the next item on the command line. The `bFlag` argument is `TRUE` if `pszParam` was preceded by a - (hyphen) or / (forward slash) character, which has since been removed. The `bLast` argument is `TRUE` if this is the last argument on the line. Make sure to call the base class's `ParseParam()` at the end or the standard flags won't be processed.

4. For a complete listing of the Command Line Info class, please see “Listings — Command Line Info Class” on page 150.

Incorporate the New Command Line Info Class in Application Class

1. Locate the `ParseCommandLine()` function in your Application Class and substitute this new class for the `CCommandLineInfo` class.

```

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

```

2. Your command line options are now available as member variables of the `cmdInfo` variable.

```

if (cmdInfo.m_bAFlag)
{
    :    :    :
}

```

3. To make these options available throughout your application, embed the `cmdInfo` variable in your Application Class and access its member variables.

```

AfxGetApp()->m_cmdInfo.m_bAFlag;

```

Notes

- The standard MFC flags are as follows. The actual processing of these standard command line flags takes place in `ProcessShellCommand(cmdInfo)` just after the `ParseCommandLine()` function in your Application Class.

nothing	Causes your application to try to open a new document.
filename	Causes your application to try open to open the filename as a document.
/p filename	Causes your application to open and print the given filename to the default printer.
/pt filename printer driver port	Same as above but to the specified printer
/dde	Causes your application to startup and wait for DDE commands.
/Automation /Embedding /Unregister /Unregserver	COM flags

- Processing nonflags, such as names, can be a bit tricky. The first nonflag to come along is assumed to be a document filename. Once a filename has been found, however, you can grab any additional nonflags for your own purposes. That is, unless a `/pt` flag is encountered, in which case the next three nonflag arguments are used to initialize printing. To simplify things you may want to disable the `/pt` flag by not passing it on to `ParseParam()` in the base class.
- Of course, if you don't want to continue to support the standard MFC flags shown previously, you have a much freer hand. Any flag or nonflag option can be used, as long as you don't call the base class's `ParseParam()`. However, don't give up the functionality these standard flags provide just because you can.

CD Notes

- When executing the project on the CD, set a breakpoint on the `ParseCmdLine()` function in `Wzd.cpp` and watch as the new `CWzdCommandLineInfo` converts command line arguments into flags that can be used later in the application.

Listings — Command Line Info Class

```
#if !defined WZDCOMMANDLINEINFO_H
#define WZDCOMMANDLINEINFO_H

// WzdCommandLineInfo.h : header file
//

////////////////////////////////////
// CWzdCommandLineInfo window

class CWzdCommandLineInfo : public CCommandLineInfo
{

// Construction
public:
    CWzdCommandLineInfo();

// Attributes
public:
    BOOL m_bAFlag;
    BOOL m_bCFlag;
    BOOL m_bDAFlag;
    CString m_sArg;

// Operations
public:
    void ParseParam(const TCHAR* pszParam,BOOL bFlag, BOOL bLast);

// Overrides

// Implementation
public:
    virtual ~CWzdCommandLineInfo();
};

////////////////////////////////////

#endif
```

```

// WzdCommandLineInfo.cpp : implementation file
//

#include "stdafx.h"
#include "wzd.h"
#include "WzdCommandLineInfo.h"

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

////////////////////////////////////
// CWzdCommandLineInfo

CWzdCommandLineInfo::CWzdCommandLineInfo()
{
    m_bAFlag = FALSE;
    m_bCFlag = FALSE;
    m_bDFlag = FALSE;
    m_sArg=_T("");
}

CWzdCommandLineInfo::~CWzdCommandLineInfo()
{
}

////////////////////////////////////
void CWzdCommandLineInfo::ParseParam(const TCHAR* pszParam, BOOL bFlag,
    BOOL bLast)
{
    CString sArg(pszParam);
    if (bFlag)
    {
        m_bAFlag = !sArg.CompareNoCase("a");
        m_bCFlag = !sArg.CompareNoCase("c");
        m_bDFlag = !sArg.CompareNoCase("da");
    }
}

```

```
// m_strFileName gets the first nonflag name
else if (m_strFileName.IsEmpty())
{
    m_sArg=sArg;
}
CCommandLineInfo::ParseParam(pszParam,bFlag,bLast);
}
```

Example 7 Dynamically Changing Application Icons

Objective

You would like to change or even animate your application's icon (Figure 5.10). This icon also appears in your system's task bar and allows you to show your application's progress, even when it is minimized.

Figure 5.10 Your application's Icon can be changed in your main window and the system's task bar.



Strategy

We will use MFC's `CWnd::SetIcon()` function to change the icon.

Steps

Change the Application's Icon

1. Use the following to change your application's icon dynamically. As seen here, you can also use `CWinApp::LoadIcon()` to load the icon first from your application's resources.

```
AfxGetMainWnd()->SetIcon(  
    AfxGetApp()->LoadIcon(IDI_STATUS_ICON), // icon handle  
    TRUE); // FALSE=16x16 bit icon
```

Note that we are using `CWinApp::LoadIcon()` to load the icon from our application's resources

Notes

- Not only will `SetIcon()` change an application's icon, but it will also change the icon of any application window that has a system menu, including Child Frame windows and dialog boxes. For example, to change the icon of a Child Frame window from a View Class, use the following.

```
GetParentFrame()->SetIcon(  
    AfxGetApp()->LoadIcon(IDI_STATUS_ICON),  
    TRUE);
```

- Use `SetIcon()` only when you need to change an icon at run time. Otherwise, you should change your icon using the Developer Studio's Icon Editor. Edit the `IDR_MAINFRAME` icon under the Icon folder in your application's resources.

NOTE: make sure you edit both the 32 · 32-bit icon and the 16 · 16-bit icon. Many a novice has edited only the 32 · 32-bit icon and spent hours trying to figure out why their icon hasn't changed. When creating an MDI application, make sure to also edit the other icon in the Icon folder to change the Child Frames icon, too.

- As mentioned previously, you can use this method to change your application's icon when minimized to indicate to your user that it's still processing a command. Just check `AfxGetMainWnd()->IsIconic()` and, if TRUE, update the icon using `SetIcon()`.
- The `CButton` class also has a `SetIcon()` function. However, in this case, it's used to set the icon that will appear on the face of a button. Make sure, however, you also use the `BS_ICON` button style when creating the button.

CD Notes

- When executing the project on the CD, click on the Test/Wzd commands to change the icon of the application.

Example 8 Prompting the User for Preferences

Objective

You would like to maintain your user's preferences (Figure 5.11).

Figure 5.11 Use Property Sheets and Pages to maintain your user's preferences.



Strategy

We will use Property Sheets and Property Pages to prompt our user for their preferences. First, we will use the Menu Editor to add an Options menu to the main menu. Then, we will use the ClassWizard to handle this command by creating a Property Sheet. To this Property Sheet, we will add Property Pages for the options our application will support.

Steps

Create the Property Pages

1. Use the Dialog Editor to create one or more dialog templates (Example 38). These templates should contain all the preferences your application will support. Each template style should be a Child with a Thin border and no system menu. Whatever title you choose for this template will become the name seen on the tab for this preference page. Try to keep the pages about the same size. However, the size of the entire Property Sheet will be based on the size of the largest dialog template added to it.
2. Use the ClassWizard to create a class for each of the dialog templates. Derive them from `CPropertyPage`. Create a member variable for each of the controls in your dialog box (Example 39), but also add a message handler for each control that indicates that the control has been modified. In that handler, call `SetModified(TRUE)`. This will tell the Property Sheet to enable the Apply button.

```
void CFirstPage::OnChange()
{
    SetModified(TRUE);
}
```

3. In the class of the first Property Page, use the ClassWizard to override `OnOK()`. Also in that class, send our own user-defined windows message called `WM_APPLY` to the `CMainFrame` class. Also, call `SetModified(FALSE)` to turn off the Apply button. The `OnOK()` function of each page is called whenever the OK or Apply buttons are pressed, but we only need to process it in one page. We will see the effect of the `WM_APPLY` message later.

```
#define WM_APPLY WM_USER+1
: : :
void CFirstPage::OnOK()
{
    AfxGetMainWnd()->SendMessage(WM_APPLY);
    SetModified(FALSE);

    CPropertyPage::OnOK();
}
```

4. For a complete listing of a Property Page class, see “Listings — Property Page Class” on page 158.

Create the Property Sheet

1. Use the Menu Editor to create a new main menu item, Options, with a single submenu item, Preferences.
2. Use the ClassWizard to process the Preferences submenu item in your CMainFrame class.
3. Process this Preferences command by first creating a Property Sheet with the CPropertySheet class and then adding the classes you created previously as the pages to this Property Sheet using AddPage(). Initialize the member variables of the pages with any current settings, and then display the Property Sheet using DoModal().

```
void CMainFrame::OnOptionsPreferences()
{
    CPropertySheet sheet(_T("Preferences"),this);
    m_pFirstPage=new CFirstPage;
    m_pSecondPage=new CSecondPage;

    sheet.AddPage(m_pFirstPage);
    sheet.AddPage(m_pSecondPage);

    m_pFirstPage->m_bOption1 = m_bFirstOption1;
    m_pFirstPage->m_sOption2 = m_sFirstOption2;
    m_pSecondPage->m_nOption1 = m_nSecondOption1;
    m_pSecondPage->m_sOption2 = m_sSecondOption2;

    sheet.DoModal();
    delete m_pFirstPage;
    delete m_pSecondPage;
}
```

Notice that in the last step, the values from our Property Pages are never stored in the application from which they came. That's where that WM_APPLY message comes in.

4. You will need to manually add a message handler for your `WM_APPLY` window message by adding the following item to the message map in `MainFrm.cpp`. Make sure to put it below the `//}AFX_MSG_MAP` notation or the ClassWizard may delete it.

```
ON_MESSAGE_VOID(WM_APPLY, OnApply)
```

Define `OnApply()` in `MainFrm.h` as

```
afx_msg void OnApply();
```

5. Now implement `OnApply()` in `CMainFrame` so that it stores the member variables from your Property Pages back into the application. As mentioned previously, this message will be sent anytime the Apply or OK buttons are clicked on the Property Sheet.

```
void CMainFrame::OnApply()
{
    m_bFirstOption1 = m_pFirstPage->m_bOption1;
    m_sFirstOption2 = m_pFirstPage->m_sOption2;
    m_nSecondOption1 = m_pSecondPage->m_nOption1;
    m_sSecondOption2 = m_pSecondPage->m_sOption2;
}
```

Notes

- Application preferences have, by convention, been handled in the `CMainFrame` class, although where the actual option is stored varies depending on where it's used.
- If you would like to add your own buttons to the Property Sheet, derive your own class from `CPropertySheet` and use that derived class instead in the previous example. To add your own buttons, you must first make the Property Sheet big enough to handle them. Use the ClassWizard to add a message handler to your new class that handles `WM_CREATE`. Use `MoveWindow()` there to make the sheet the size you want. To create your own buttons, use the method shown in Example 46 on creating control windows anywhere.

- One of the first things you assume when you see the tab control tool in the Dialog Editor is that it somehow has the same functionality as the Property Sheet. In fact, it has almost none. The tab control is more akin to the list control, in that it simply keeps track of what page you're on and it's up to you to fill a page with a dialog box. If at all possible, try to avoid using a tab control alone.

CD Notes

When executing the project on the CD, click on the Options/Preferences menu commands to open a Property Sheet.

Listings — Property Page Class

```
#if !defined AFX_FIRSTPAGE_H
#define AFX_FIRSTPAGE_H

// FirstPage.h : header file
//
////////////////////////////////////

// CFirstPage dialog

class CFirstPage : public CPropertyPage
{
    DECLARE_DYNCREATE(CFirstPage)

// Construction
public:
    CFirstPage();
    ~CFirstPage();

// Dialog Data
    //{AFX_DATA(CFirstPage)
    enum {IDD = IDD_FIRST_PAGE};
    BOOL m_bOption1;
    CString m_sOption2;
    //}}AFX_DATA
```

```

// Overrides
// ClassWizard generate virtual function overrides
//{{AFX_VIRTUAL(CFirstPage)
public:
    virtual void OnOK();
protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
//}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{AFX_MSG(CFirstPage)
    afx_msg void OnChange();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

////////////////////////////////////
// FirstPage.cpp : implementation file
//

#include "stdafx.h"
#include "wzd.h"
#include "FirstPage.h"
#include "WzdProject.h"

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

```

```
////////////////////////////////////
// CFirstPage property page

IMPLEMENT_DYNCREATE(CFirstPage, CPropertyPage)

CFirstPage::CFirstPage() : CPropertyPage(CFirstPage::IDD)
{
    //{AFX_DATA_INIT(CFirstPage)
    m_bOption1 = FALSE;
    m_sOption2 = _T("");
    //}}AFX_DATA_INIT
}

CFirstPage::~CFirstPage()
{
}

void CFirstPage::DoDataExchange(CDataExchange* pDX)
{
    CPropertyPage::DoDataExchange(pDX);
    //{AFX_DATA_MAP(CFirstPage)
    DDX_Check(pDX, IDC_CHECK, m_bOption1);
    DDX_Text(pDX, IDC_EDIT, m_sOption2);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CFirstPage, CPropertyPage)
    //{AFX_MSG_MAP(CFirstPage)
    ON_BN_CLICKED(IDC_CHECK, OnChange)
    ON_EN_CHANGE(IDC_EDIT, OnChange)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CFirstPage message handlers

void CFirstPage::OnChange()
{
    SetModified(TRUE);
}
```

```
// only needed on one page!  
void CFirstPage::OnOK()  
{  
    AfxGetMainWnd()->SendMessage(WM_APPLY);  
    SetModified(FALSE);  
  
    CPropertyPage::OnOK();  
}
```

Example 9 Saving and Restoring User Preferences

II 5

Objective

You would like to save and restore the options your user picked in the previous example.

Strategy

We will be using six `CWinApp` functions to load and save values to the system registry.

Steps

Configure Your Application

1. Add the following `#defines` to your Mainframe Class's definition file. Substitute "Company" for your company name and "Optionx" for descriptive names of your application's options.

```
#define COMPANY_KEY "Company"  
#define SETTINGS_KEY "Settings"  
#define OPTION1_KEY "Option1"  
#define OPTION2_KEY "Option2"  
#define OPTION3_KEY "Option3"  
#define OPTION4_KEY "Option4"
```

2. In your `CMainFrame` class, modify the `SetRegistryKey()` call in the `InitInstance()` function to use your company's system registry key.

```
SetRegistryKey(COMPANY_KEY);
```

Save Application Options

1. Add a new member function to your `CMainFrame` class that will save your options. There, you can use up to three different `CWinApp` functions to save integer, string, or binary options.

```
void CMainFrame::SaveOptions()
{
    // integer options
    AfxGetApp()->WriteProfileInt(SETTINGS_KEY,OPTION2_KEY,
        m_nOption2);
    // string options
    AfxGetApp()->
        WriteProfileString(SETTINGS_KEY,OPTION1_KEY,m_sOption1);
    // binary options
    AfxGetApp()->WriteProfileBinary(SETTINGS_KEY, OPTION3_KEY,
        (BYTE*)&m_dOption3, sizeof(m_dOption3));
}
```

2. Use the **ClassWizard** to add a `WM_CLOSE` message handler to your `CMainFrame` class and call `SaveOptions()` from there.

Restore Application Options

1. Add another member function to your `CMainFrame` class that will load your options. There you can use three other `CWinApp` function to load integer, string, and binary options from the system registry.

```
void CMainFrame::LoadOptions()
{
    // integer options
    m_nOption2=AfxGetApp()->
        GetProfileInt(SETTINGS_KEY,OPTION2_KEY, 3);
```

```

// string options
m_sOption1=AfxGetApp()->
    GetProfileString(SETTINGS_KEY,OPTION1_KEY,"Default");
// binary options
BYTE *p;
UINT size;
m_dOption3=33.3;
if (AfxGetApp()->GetProfileBinary(SETTINGS_KEY,OPTION3_KEY,
    &p, &size))
{
    memcpy(&m_dOption3,p,size);
    delete []p;
}
}

```

2. Call `LoadOptions()` at the start of the `OnCreate()` function in your `CMainFrame` class.

`GetProfileBinary()` and `WriteProfileBinary()` are undocumented and may not be available on all versions of MFC. If it isn't on yours, the following steps can be used to create alternative `LoadOptions()` and `SaveOptions()`. The following methods also allow you to save and load options from anywhere in the system registry.

Alternative `LoadOptions()` and `SaveOptions()`

1. Add one more item to your project include file and substitute your application's name for "Wzd".

```
#define APPLICATION_KEY "Software\\Company\\Wzd\\Settings"
```

2. Then use the following for your `SaveOptions()` function. This function opens the system registry, writes your options to it, and then closes it using the Window API directly.

```

void CMainFrame::SaveOptions()
{
    // opens system registry for writing
    HKEY key;
    DWORD size, type, disposition;

```

```
if (RegOpenKeyEx(HKEY_CURRENT_USER,APPLICATION_KEY,0,
    KEY_WRITE,&key)!=ERROR_SUCCESS)
{
    RegCreateKeyEx(HKEY_CURRENT_USER,APPLICATION_KEY,0,"",
        REG_OPTION_NON_VOLATILE,KEY_ALL_ACCESS,NULL,
        &key,&disposition);
}

// writes an option that's a string
type = REG_SZ;
size = m_sOption1.GetLength();
RegSetValueEx(key,OPTION1_KEY,0,type,
    (LPBYTE)LPCSTR(m_sOption1),size);

// writes an option that's an integer
type = REG_DWORD;
size = 4;
RegSetValueEx(
    key,OPTION2_KEY,0,type,(LPBYTE)&m_nOption2,size);

// writes all other options
type = REG_BINARY;
size = sizeof(DOUBLE);
RegSetValueEx(
    key,OPTION3_KEY,0,type,(LPBYTE)&m_dOption3,size);

// closes system registry
RegCloseKey(key);

}
```

3. Use the following for your `LoadOptions()`. It will open the system registry, read your options back in, and then close the system registry.

```
void CMainFrame::LoadOptions()
{
    HKEY key;
    DWORD size, type;
    if (RegOpenKeyEx(
        HKEY_CURRENT_USER,APPLICATION_KEY,0,KEY_READ,&key) ==
        ERROR_SUCCESS)
    {
        // read string options
        size=128;
        type = REG_SZ;
        LPSTR psz = m_sOption1.GetBuffer(size);
        RegQueryValueEx(
            key,OPTION1_KEY,0,&type,(LPBYTE)psz,&size);
        m_sOption1.ReleaseBuffer();

        // read integer options
        size=4;
        type = REG_DWORD;
        RegQueryValueEx(key,OPTION2_KEY,0,&type,
            (LPBYTE)&m_nOption2,&size);

        // read all other options as binary bytes
        size=sizeof(DOUBLE);
        type = REG_BINARY;
        RegQueryValueEx(key,OPTION3_KEY,0,&type,
            (LPBYTE)&m_dOption3,&size);

        RegCloseKey(key);
    }
}
```

Notes

- You can edit the options you write to the system registry by using the `REGEDIT.EXE` application found in your Windows directory. You'll find your application's entries under `HKEY_CURRENT_USER/Software/Company`.
- You'll notice that you can save an option as an integer, a string, or a binary. In fact, all options can be saved as binary values. Whenever possible, however, use an integer or string because these values can be more easily edited using the `REGEDIT.EXE` utility. Strings will appear as strings and integers will appear as integers. Binary values simply appear as a string of bytes.
- If you comment out `SetRegistryKey()` in step 2 of "Configure Your Application" on page 161, your options will be saved to a file instead of the system registry. This file will be in your `\Windows` directory and be named `app.ini`, where `app` is the name of your application. Older windows applications used to save their options this way.

CD Notes

- When executing the project on the CD, set a break point on `LoadOptions1()` and `SaveOptions1()` in `Mainfrm.cpp`. Then run the application and terminate it to watch options being loaded and saved to the system registry.

Example 10 Terminating Your Application

Objective

You would like to ask your user if they're sure they want to terminate your application.

Strategy

We'll use the ClassWizard to add a message handler to the `Mainframe Class` to handle a `WM_CLOSE` message in which we will ask our question and then conditionally continue to terminate the application.

Steps

Ask the Question

1. Use the ClassWizard to add a `WM_CLOSE` message handler to the `CMainFrame` class.
2. Add the following code to this message handler.

```
void CMainFrame::OnClose()
{
    if (AfxMessageBox("Do you really want to exit?",MB_YESNO)
        == IDYES)
    {
        CMDIFrameWnd::OnClose();
    }
}
```

Notes

- When the user clicks the Close button in the upper-right corner of your application — or when they select Exit in the File menu — the system sends a `WM_CLOSE` message to your application. By intercepting the message here, we can prevent the default action of this command, which is to send `WM_DESTROY` messages to all of the Child windows of this application.
- In a standard MFC application, you would never need to use this example. Since MFC applications are document-centric, closing your application should be based solely on whether or not your document(s) has been modified. When you modify a document, you should use the `SetModified(TRUE)` member function of the `CDocument` class to mark your document as modified. Then, when your user goes to terminate your application, MFC will automatically check each document to see if they've been modified, and if so, will ask your user if they want to save their changes. If yes, MFC will automatically call your `OnSaveModified()` routine. If your user refuses to save a document, the close is canceled. If no document has been modified, your application will terminate without a peep.

- For an example of using the ClassWizard to add a message handler, see Example 59.

CD Notes

- When executing the project on the CD, you can then click the Close button to cause a dialog prompt asking if you really want to terminate. Answering No will cause the application to continue to run.

Example 11 Creating a Splash Screen

Objective

You would like to create a splash screen to display a company logo and copyright (Figure 5.12).

Figure 5.12 Your application Splash Screen can show your company logo and copyright.



Strategy

We will create our splash screen at the earliest point we can, which is in the `InitInstance()` function of our Application Class. Our splash screen will appear in a plain window using one of the bitmap classes we will create in Example 57 so that our window can have all the colors in the rainbow.

Steps

Create a Splash Screen Window Class

1. Use the ClassWizard to create a plain window class derived from generic `CWnd`.
2. Add your own `Create()` member function to this class. There you will load the bitmap that our splash screen will display and create the window

centered in the screen. Use the bitmap class in Example 57 to load the bitmap — it will allow your bitmap to retain its palette when it paints to the screen.

```
void CWzdSplash::Create(UINT nID)
{
    m_bitmap.LoadBitmapEx(nID,FALSE);
    int x = (::GetSystemMetrics (SM_CXSCREEN)-
            m_bitmap.m_Width)/2;
    int y = (::GetSystemMetrics (SM_CYSCREEN)-
            m_bitmap.m_Height)/2;
    CRect rect(x,y,x+m_bitmap.m_Width,y+m_bitmap.m_Height);
    CreateEx(0,AfxRegisterWndClass(0),"",
            WS_POPUP|WS_VISIBLE|WS_BORDER,rect,NULL,0);
}
```

3. Use the ClassWizard to add a WM_PAINT message handler to this window class. There you will draw the bitmap to the screen using BitBlt().

```
void CWzdSplash::OnPaint()
{
    CPaintDC dc(this); // device context for painting
    // get bitmap colors
    CPalette *pOldPal =
        dc.SelectPalette(m_bitmap.GetPalette(),FALSE);
    dc.RealizePalette();
    // get device context to select bitmap into
    CDC dcComp;
    dcComp.CreateCompatibleDC(&dc);
    dcComp.SelectObject(&m_bitmap);
    // draw bitmap
    dc.BitBlt(0,0,m_bitmap.m_Width,m_bitmap.m_Height, &dcComp,
            0,0,SRCCOPY);
    // reselect old palette
    dc.SelectPalette(pOldPal,FALSE);
}
```

4. For a complete listing of this splash window class, please refer to “Listings—Splash Window Class” on page 170.

Incorporate the Splash Screen Class into `InitInstance()`

1. Create an instance of this splash window class at the start of the `InitInstance()` function in your Application Class. Call its `Create()` and force it to paint.

```
CWzdSplash wndSplash;
wndSplash.Create(IDB_WZDSPLASH);
wndSplash.UpdateWindow(); //send WM_PAINT
```

2. Since we created the splash window class on the stack, this window will be automatically destroyed once `InitInstance()` returns. Therefore, if your application takes a lot of time to initialize, you won't have to put any delays into your application to ensure that the splash screen is visible long enough to read. On the other hand, if your application takes little time to initialize or if you're worried that faster machines will turn your splash screen into a blip, add the following lines somewhere in your `InitInstance()` to delay your application.

```
// add if splash screen too short
Sleep(2000); <<<<<<
```

Notes

- To create an animated splash screen, derive your splash screen window from `CAnimateCtrl`. In `Create()`, load an `.avi` file instead of a bitmap file, and create the window centered. See Example 43 for more on the `CAnimateCtrl` class.

CD Notes

- When executing the project on the CD, a splash screen will initially appear and then disappear before the application window appears.

Listings—Splash Window Class

```
#if !defined WZDSPLASH_H
#define WZDSPLASH_H

// WzdSplash.h : header file
//
```

```
#include "WzdBitmap.h"

/////////////////////////////////////////////////////////////////
// CWzdSplash window

class CWzdSplash : public CWnd
{

// Construction
public:
    CWzdSplash();

// Attributes
public:

// Operations
public:
    void Create(UINT nBitmapID);

// Overrides
    // ClassWizard generated virtual function overrides
    //{AFX_VIRTUAL(CWzdSplash)
    //}AFX_VIRTUAL

// Implementation
public:
    virtual ~CWzdSplash();

    // Generated message map functions
protected:
    //{AFX_MSG(CWzdSplash)
    afx_msg void OnPaint();
    //}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
    CWzdBitmap m_bitmap;
};

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



```
#endif

// WzdSplash.cpp : implementation file
//

#include "stdafx.h"
#include "WzdSplash.h"

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

////////////////////////////////////

// CWzdSplash

CWzdSplash::CWzdSplash()
{
}

CWzdSplash::~CWzdSplash()
{
}
```

```

BEGIN_MESSAGE_MAP(CWzdSplash, CWnd)
    //{AFX_MSG_MAP(CWzdSplash)
    ON_WM_PAINT()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CWzdSplash message handlers

void CWzdSplash::OnPaint()
{
    CPaintDC dc(this);    // device context for painting
    // get bitmap colors
    CPalette *pOldPal = dc.SelectPalette(m_bitmap.GetPalette(),FALSE);
    dc.RealizePalette();

    // get device context to select bitmap into
    CDC dcComp;
    dcComp.CreateCompatibleDC(&dc);
    dcComp.SelectObject(&m_bitmap);

    // draw bitmap
    dc.BitBlt(0,0,m_bitmap.m_Width,m_bitmap.m_Height, &dcComp, 0,0,SRCCOPY);

    // reselect old palette
    dc.SelectPalette(pOldPal,FALSE);
}

void CWzdSplash::Create(UINT nID)
{
    m_bitmap.LoadBitmapEx(nID,FALSE);

    int x = (::GetSystemMetrics(SM_CXSCREEN)- m_bitmap.m_Width)/2;
    int y = (::GetSystemMetrics(SM_CYSCREEN)- m_bitmap.m_Height)/2;
    CRect rect(x,y,x+m_bitmap.m_Width,y+m_bitmap.m_Height);
    CreateEx(0,AfxRegisterWndClass(0),"",
        WS_POPUP|WS_VISIBLE|WS_BORDER,rect,NULL,0);
}

```