Module 16: Dynamic Link Libraries- DLL

Program examples compiled using Visual C++ 6.0 (MFC 6.0) compiler on Windows XP Pro machine with Service Pack 2. Topics and sub topics for this Tutorial are listed below. If you think the terms used in this DLL tutorial quite blur, you can try studying the Win32 DLL first.

Dynamic Link Libraries - DLL

If you want to write modular software, you'll be very interested in dynamic link libraries (DLLs). You're probably thinking that you've been writing modular software all along because C++ classes are modular. But classes are **build-time** modular, and DLLs are **runtime modular**. Instead of programming giant EXEs that you must rebuild and test each time you make a change, you can build smaller DLL modules and test them individually. You can, for example, put a C++ class in a DLL, which might be as small as 12 KB after compiling and linking. Client programs can load and link your DLL very quickly when they run. Microsoft Windows itself uses DLLs for its major functions. DLLs are getting easier to write. Win32 has greatly simplified the programming model, and there's more and better support from AppWizard and the Microsoft Foundation Class (MFC) library. This module shows you how to write DLLs in C++ and how to write client programs that use DLLs. You'll explore how Win32 maps DLLs into your processes, and you'll learn the differences between **MFC library regular DLLs** and **MFC library extension DLLs**. You'll see examples of simple DLLs of each type as well as a more complex DLL example that implements a custom control.

Fundamental DLL Theory

Before you look at the application framework's support for DLLs, you must understand how Win32 integrates DLLs into your process. You might want to review Module 20 to refresh your knowledge of processes and virtual memory. Remember that a process is a running instance of a program and that the program starts out as an EXE file on disk.

Basically, a DLL is a file on disk (usually with a DLL extension) consisting of global data, compiled functions, and resources, that becomes part of your process. **It is compiled to load at a preferred base address**, and if there's no conflict with other DLLs, the file gets mapped to the same virtual address in your process. The DLL has various **exported functions**, and the client program (the program that loaded the DLL in the first place) **imports those functions**. Windows matches up the imports and exports when it loads the DLL. Win32 DLLs allow **exported global variables** as well as **functions**.

In Win32, each process gets its own copy of the DLL's read/write global variables. If you want to share memory among processes, you must either use a memory-mapped file or declare a shared data section as described in Jeffrey Richter's Advanced Windows (Microsoft Press, 1997). Whenever your DLL requests heap memory, that memory is allocated from the client process's heap.

How Imports Are Matched to Exports

A DLL contains a **table of exported functions**. These functions are identified to the outside world by their **symbolic names** and (optionally) by integers called **ordinal numbers**. The function table also contains the **addresses of the functions** within the DLL. When the client program first loads the DLL, it doesn't know the addresses of the functions it needs to call, but it does know the symbols or ordinals. The dynamic linking process then builds a table that connects the client's calls to the function addresses in the DLL. If you edit and rebuild the DLL, you don't need to rebuild your client program unless you have changed function names or parameter sequences. In a simple world, you'd have one EXE file that imports functions from one or more DLLs. In the real world, many DLLs call functions inside other DLLs. Thus, a particular DLL can have both exports and imports. This is not a problem because the dynamic linkage process can handle cross-dependencies. In the DLL code, you must explicitly declare your exported functions like this:

```
__declspec(dllexport) int MyFunction(int n);
```

The alternative is to list your exported functions in a module-definition `[DEF]` file, but that's usually more troublesome. On the client side, you need to declare the corresponding imports like this:

```
__declspec(dllimport) int MyFunction(int n);
```

If you're using C++, the compiler generates a **decorated name** for let say `MyFunction()`, that other languages can't use. These decorated names are the long names the compiler invents based on class name, function name, and parameter types. They are listed in the project's **MAP** file. If you want to use the plain name `MyFunction()`, you have to write the declarations this way:

```
extern "C" __declspec(dllexport) int MyFunction(int n);
extern "C" __declspec(dllimport) int MyFunction(int n);
```

By default, the compiler uses the `__cdecl` argument passing convention, which means that the calling program pops the parameters off the stack. Some client languages might require the `__stdcall` convention, which replaces the Pascal calling convention, and which means that the called function pops the stack. Therefore, you might have to use the `__stdcall` modifier in your DLL export declaration. Just having import declarations isn't enough to make a client link to a DLL. The client's project must specify the **import library** (**LIB**) to the linker, and the client program must actually contain a call to at least one of the **DLL's imported functions**. That call statement must be in an executable path in the program.

Implicit Linkage vs. Explicit Linkage

The preceding section primarily describes **implicit linking**, which is what you as a C++ programmer will probably be using for your DLLs. When you build a DLL, the linker produces a companion import LIB file, which contains every DLL's exported symbols and (optionally) ordinals, but no code. The LIB file is a surrogate for the DLL that is added to the client program's project. When you build (statically link) the client, the imported symbols are matched to the exported symbols in the LIB file, and those symbols (or ordinals) are bound into the EXE file. The LIB file also contains the DLL filename (but not its full pathname), which gets stored inside the EXE file. When the client is loaded, Windows finds and loads the DLL and then dynamically links it by symbol or by ordinal.
Explicit linking is more appropriate for interpreted languages such as Microsoft Visual Basic, but you can use it from C++ if you need to. With explicit linking, you don't use an import file; instead, you call the Win32 `LoadLibrary()`

function, specifying the DLL's pathname as a parameter. `LoadLibrary()` returns an `HINSTANCE` parameter that you can use in a call to `GetProcAddress()`, which converts a symbol (or an ordinal) to an address inside the DLL. Suppose you have a DLL that exports a function such as this:

```
extern "C" __declspec(dllexport) double SquareRoot(double d);
```

Here's an example of a client's explicit linkage to the function:

```
typedef double (SQRTPROC)(double);
HINSTANCE hInstance;
SQRTPROC* pFunction;
VERIFY(hInstance = ::LoadLibrary("c:\\winnt\\system32\\mydll.dll"));
VERIFY(pFunction = (SQRTPROC*)::GetProcAddress(hInstance, "SquareRoot"));
double d = (*pFunction)(81.0); // Call the DLL function
```

With implicit linkage, all DLLs are loaded when the client is loaded, but with explicit linkage, you can determine when DLLs are loaded and unloaded. Explicit linkage allows you to determine at runtime which DLLs to load. You could, for example, have one DLL with string resources in English and another with string resources in Spanish. Your application would load the appropriate DLL after the user chose a language.

Symbolic Linkage vs. Ordinal Linkage

In Win16, the more efficient ordinal linkage was the preferred linkage option. In Win32, the symbolic linkage efficiency was improved. Microsoft now recommends **symbolic** over **ordinal linkage**. The DLL version of the MFC library, however, uses ordinal linkage. A typical MFC program might link to hundreds of functions in the MFC DLL. Ordinal linkage permits that program's EXE file to be smaller because it does not have to contain the long symbolic names of its imports. If you build your own DLL with ordinal linkage, you must specify the ordinals in the project's DEF file, which doesn't have too many other uses in the Win32 environment. If your exports are C++ functions, you must use decorated names in the DEF file (or declare your functions with `extern "C"`). Here's a short extract from one of the MFC library DEF files:

```
?ReadList@CRecentFileList@@UAEXXZ @ 5458 NONAME
?ReadNameDictFromStream@CPropertySection@@QAEHPAUIStream@@@Z @ 5459 NONAME
?ReadObject@CArchive@@QAEPAVCObject@@PBUCRuntimeClass@@@Z @ 5460 NONAME
?ReadString@CArchive@@QAEHAAVCString@@@Z @ 5461 NONAME
?ReadString@CArchive@@QAEPADPADI@Z @ 5462 NONAME
?ReadString@CInternetFile@@UAEHAAVCString@@@Z @ 5463 NONAME
?ReadString@CInternetFile@@UAEPADPADI@Z @ 5464 NONAME
```

The numbers after the at (@) symbols are the ordinals. Kind of makes you want to use symbolic linkage instead, doesn't it?

The DLL Entry Point: `DllMain()`

By default, the linker assigns the main entry point `_DllMainCRTStartup()` to your DLL. When Windows loads the DLL, it calls this function, which first calls the constructors for global objects and then calls the global function `DllMain()`, which you're supposed to write. `DllMain()` is called not only when the DLL is attached to the process but also when it is detached (and at other times as well). Here is a skeleton `DllMain()` function:

```
HINSTANCE g_hInstance;
extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        TRACE0("MYMFC22A.DLL Initializing!\n");
        // Do initialization here
    }
    else if (dwReason == DLL_PROCESS_DETACH)
    {
        TRACE0("MYMFC22A.DLL Terminating!\n");
```

```
            // Do cleanup here
        }
        return 1;   // ok
    }
```

If you don't write a `DllMain()` function for your DLL, a do-nothing version is brought in from the runtime library. The `DllMain()` function is also called when individual threads are started and terminated, as indicated by the dwReason parameter. Richter's book tells you all you need to know about this complex subject.

Instance Handles: Loading Resources

Each DLL in a process is identified by a unique 32-bit `HINSTANCE` value. In addition, the process itself has an `HINSTANCE` value. All these instance handles are valid only within a particular process, and they represent the starting virtual address of the DLL or EXE. In Win32, the `HINSTANCE` and `HMODULE` values are the same and the types can be used interchangeably. The process (EXE) instance handle is almost always **`0x400000`**, and the handle for a DLL loaded at the default base address is **`0x10000000`**. If your program uses several DLLs, each will have a different `HINSTANCE` value, either because the DLLs had different base addresses specified at build time or because the loader copied and relocated the DLL code.
Instance handles are particularly important for loading resources. The Win32 `FindResource()` function takes an `HINSTANCE` parameter. EXEs and DLLs can each have their own resources. If you want a resource from the DLL, you specify the DLL's instance handle. If you want a resource from the EXE file, you specify the EXE's instance handle. How do you get an instance handle? If you want the EXE's handle, you call the Win32 `GetModuleHandle()` function with a NULL parameter. If you want the DLL's handle, you call the Win32 `GetModuleHandle()` function with the DLL name as a parameter. Later you'll see that the MFC library has its own method of loading resources by searching various modules in sequence.

How the Client Program Finds a DLL

If you link explicitly with `LoadLibrary()`, you can specify the DLL's full pathname. If you don't specify the pathname, or if you link implicitly, Windows follows this search sequence to locate your DLL:

1. The directory containing the EXE file.
2. The process's current directory.
3. The Windows system directory.
4. The Windows directory.
5. The directories listed in the `Path` environment variable.

Here's a trap you can easily fall into. You build a DLL as one project, copy the DLL file to the system directory, and then run the DLL from a client program. So far, so good. Next you rebuild the DLL with some changes, but you forget to copy the DLL file to the system directory. The next time you run the client program, it loads the old version of the DLL. Be careful!

Debugging a DLL

Visual C++ makes debugging a DLL easy. Just run the debugger from the DLL project. The first time you do this, the debugger asks for the **pathname of the client EXE file**. Every time you "run" the DLL from the debugger after this, the debugger loads the EXE, but the EXE uses the search sequence to find the DLL. This means that you must either set the `Path` environment variable to point to the DLL or copy the DLL to a directory in the search sequence.

MFC DLLs: Extension vs. Regular

We've been looking at Win32 DLLs that have a `DllMain()` function and some exported functions. Now we'll move into the world of the MFC application framework, which adds its own support layer on top of the Win32 basics. AppWizard lets you build **two kinds of DLLs** with MFC library support: **extension DLLs** and **regular DLLs**. You must understand the differences between these two types before you decide which one is best for your needs.
Of course, Visual C++ lets you build a pure Win32 DLL without the MFC library, just as it lets you build a Windows program without the MFC library. This is an MFC-oriented book, however, so we'll ignore the Win32 option here.

An extension DLL supports a **C++ interface**. In other words, the DLL can export whole classes and the client can construct objects of those classes or derive classes from them. An extension DLL dynamically links to the code in the DLL version of the MFC library. Therefore, an extension DLL requires that your client program be **dynamically linked** to the MFC library (the AppWizard default) and that both the client program and the extension DLL be synchronized to the same version of the MFC DLLs (**mfc42.dll**, **mfc42d.dll**, and so on). Extension DLLs are quite small; you can build a simple extension DLL with a size of 10 KB, which loads quickly.
If you need a DLL that can be loaded by any Win32 programming environment (including Visual Basic version 6.0), you should use a **regular DLL**. A big restriction here is that the regular DLL can **export only C-style functions**. It can't export C++ classes, member functions, or overloaded functions because every C++ compiler has its own method of decorating names. You can, however, use C++ classes (and MFC library classes, in particular) inside your regular DLL. When you build an MFC regular DLL, you can choose to statically link or dynamically link to the MFC library. If you choose static linking, your DLL will include a copy of all the MFC library code it needs and will thus be self-contained. A typical **Release-build statically linked** regular DLL is about 144 KB in size. If you choose **dynamic linking**, the size drops to about 17 KB but you'll have to ensure that the proper MFC DLLs are present on the target machine. That's no problem if the client program is already dynamically linked to the same version of the MFC library. When you tell AppWizard what kind of DLL or EXE you want, compiler `#define` constants are set as shown in the following table.

	Dynamically Linked to Shared MFC Library	Statically Linked* to MFC Library
Regular DLL	`_AFXDLL`, `_USRDLL`	`_USRDLL`
Extension DLL	`_AFXEXT`, `_AFXDLL`	unsupported option
Client EXE	`_AFXDLL`	no constants defined

Table 1

* Visual C++ Learning Edition does not support the static linking option.

If you look inside the MFC source code and header files, you'll see a ton of `#ifdef` statements for these constants. This means that the library code is compiled quite differently depending on the kind of project you're producing.

The Shared MFC DLLs and the Windows DLLs

If you build a Windows Debug target with the shared MFC DLL option, your program is dynamically linked to one or more of these (ANSI) MFC DLLs:

DLL	Description
mfc42d.dll	Core MFC classes.
mfco42d.dll	ActiveX (OLE) classes.
mfcd42d.dll	Database classes (ODBC and DAO).
mfcn42d.dll	Winsock, `WinInet` classes.

Table 2.

When you build a **Release target**, your program is dynamically linked to **mfc42.dll** only. Linkage to these MFC DLLs is implicit via import libraries. You might assume implicit linkage to the ActiveX and ODBC DLLs in Windows, in which case you would expect all these DLLs to be linked to your Release-build client when it loads, regardless of whether it uses ActiveX or ODBC features. However, this is not what happens. Through some creative thinking, MFC loads the ActiveX and ODBC DLLs explicitly (by calling `LoadLibrary()`) when one of their functions is first called. Your client application thus loads only the DLLs it needs.

MFC Extension DLLs: Exporting Classes

If your extension DLL contains only exported C++ classes, you'll have an easy time building and using it. The steps for building the MYMFC22A example show you how to tell AppWizard that you're building an extension DLL skeleton. That skeleton has only the `DllMain()` function. You simply add your own C++ classes to the project. There's only one special thing you must do. You must add the macro `AFX_EXT_CLASS` to the class declaration, as shown here:

```
class AFX_EXT_CLASS CStudent : public CObject
```

This modification goes into the **H file** that's part of the DLL project, and it also goes into the H file that client programs use. In other words, the H files are exactly the same for both client and DLL. The macro generates different code depending on the situation, it exports the class in the DLL and imports the class in the client.

The MFC Extension DLL Resource Search Sequence

If you build a dynamically linked MFC client application, many of the MFC library's standard resources (error message strings, print preview dialog templates, and so on) are stored in the MFC DLLs (**mfc42.dll**, **mfco42.dll**, and so on), but your application has its own resources too. When you call an MFC function such as CString::LoadString or `CBitmap::LoadBitmap`, the framework steps in and searches first the EXE file's resources and then the MFC DLL's resources.
If your program includes an extension DLL and your EXE needs a resource, the search sequence is first the EXE file, then the extension DLL, and then the MFC DLLs. If you have a string resource ID, for example, that is unique among all resources, the MFC library will find it. If you have duplicate string IDs in your EXE file and your extension DLL file, the MFC library loads the string in the EXE file.
If the extension DLL loads a resource, the sequence is first the extension DLL, then the MFC DLLs, and then the EXE. You can change the search sequence if you need to. Suppose you want your EXE code to search the extension DLL's resources first. Use code such as this:

```
HINSTANCE hInstResourceClient = AfxGetResourceHandle();
// Use DLL's instance handle
AfxSetResourceHandle(::GetModuleHandle("my_dll_file_name.dll"));
CString strRes;
strRes.LoadString(IDS_MYSTRING);
// Restore client's instance handle
AfxSetResourceHandle(hInstResourceClient);
```

You can't use `AfxGetInstanceHandle()` instead of `::GetModuleHandle()`. In an extension DLL, `AfxGetInstanceHandle()` returns the EXE's instance handle, not the DLL's handle.

The MYMFC22A Example: An MFC Extension DLL

This example makes an extension DLL out of the `CPersistentFrame` class you saw in Module 9. First you'll build the **mymfc22A.dll** file, and then you'll use it in a test client program, MYMFC22B.

Here are the steps for building the MYMFC22A example:

Run AppWizard to produce \mfcproject\mymfc22A. Choose **New** from Visual C++'s **File** menu, and then click on the **Projects** tab as usual. Instead of selecting **MFC AppWizard (exe)**, choose **MFC AppWizard (dll)**, as shown here.

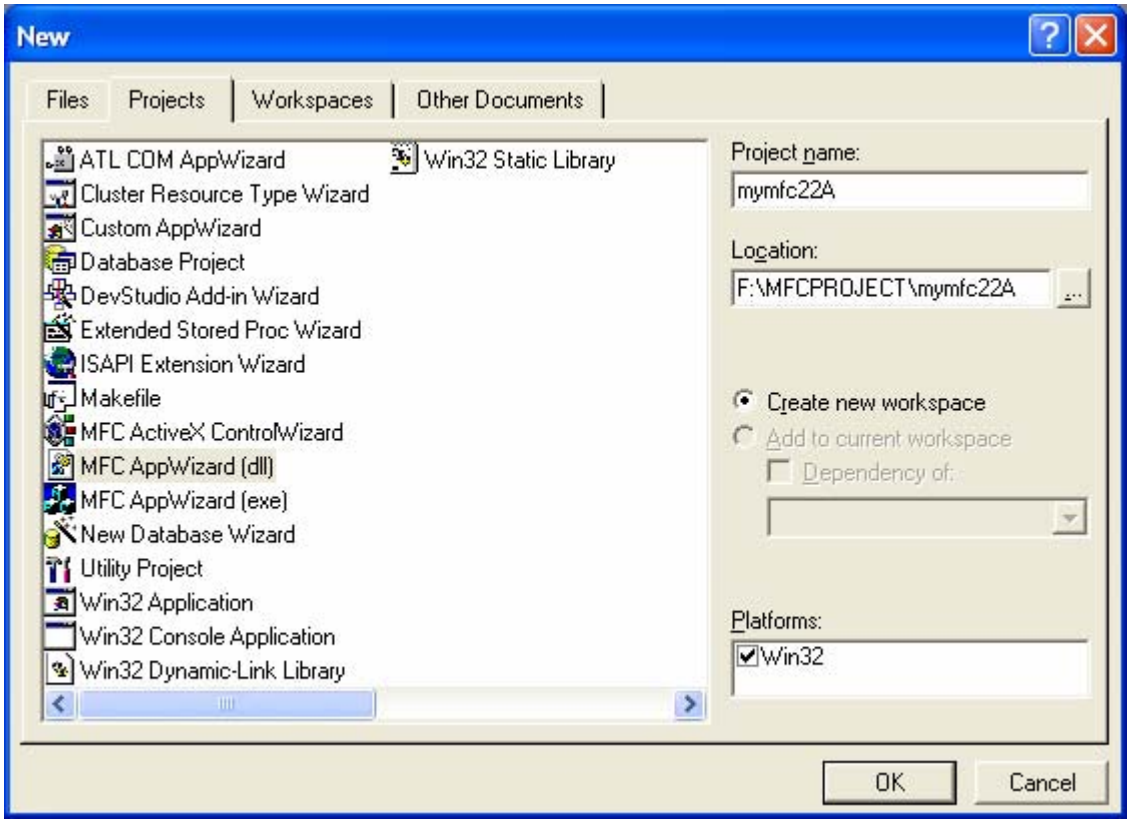

Figure 1: AppWizard new DLL project creation dialog.

In this example, only one AppWizard screen appears. Choose **MFC Extension DLL**, as shown here.

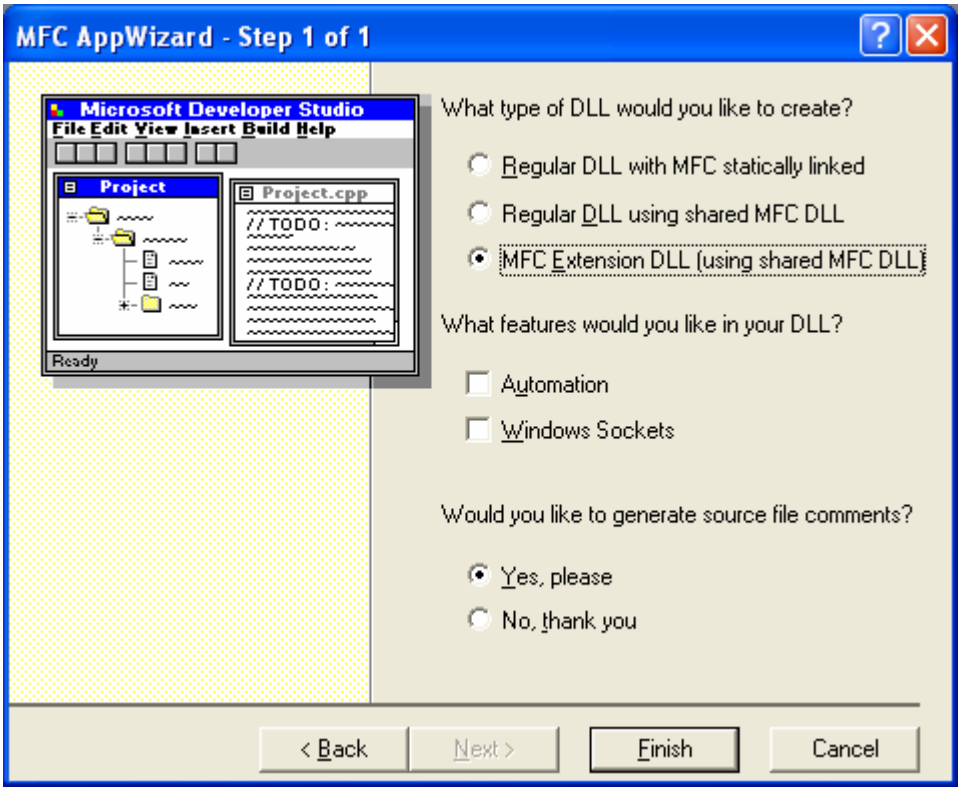

Figure 2: The only step 1 of 1 DLL project.

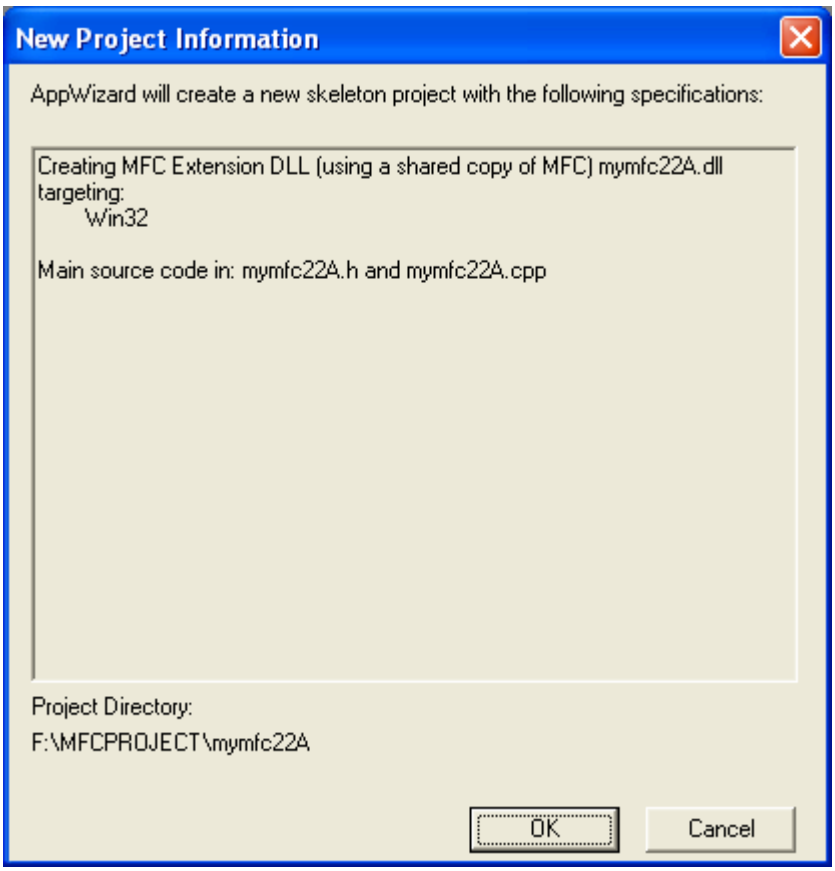

Figure 3: MYMFC22A DLL project summary.

Examine the **mymfc22A.cpp** file. AppWizard generates the following code, which includes the `DllMain()` function:

```
// mymfc22A.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 Mymfc22ADLL = { 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("MYMFC22A.DLL Initializing!\n");

		// Extension DLL one-time initialization
```

```
		if (!AfxInitExtensionModule(Mymfc22ADLL, 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(Mymfc22ADLL);
	}
	else if (dwReason == DLL_PROCESS_DETACH)
	{
		TRACE0("MYMFC22A.DLL Terminating!\n");
		// Terminate the library before destructors are called
		AfxTermExtensionModule(Mymfc22ADLL);
	}
	return 1;   // ok
}
```

Insert the `CPersistentFrame` class into the project. Choose **Add To Project** from the **Project** menu, and then choose **Components And Controls** from the submenu.

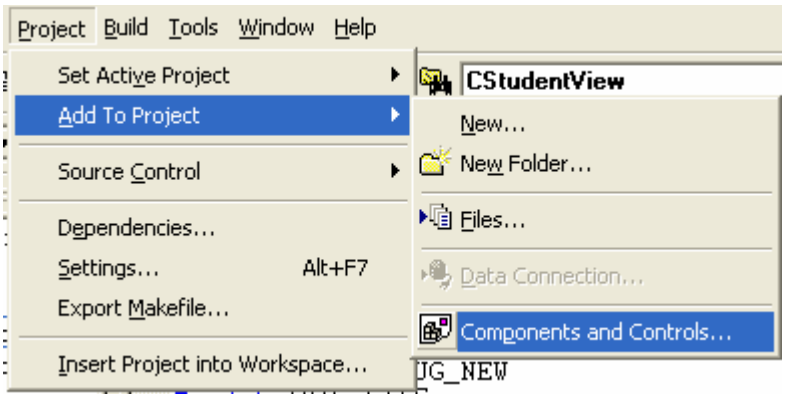

Figure 4: Inserting `CPersistentFrame` class into the MYMFC22A project.

Locate the file **Persistent Frame.ogx** that you created in Module 9. Click the **Insert** button to insert the class into the current project.

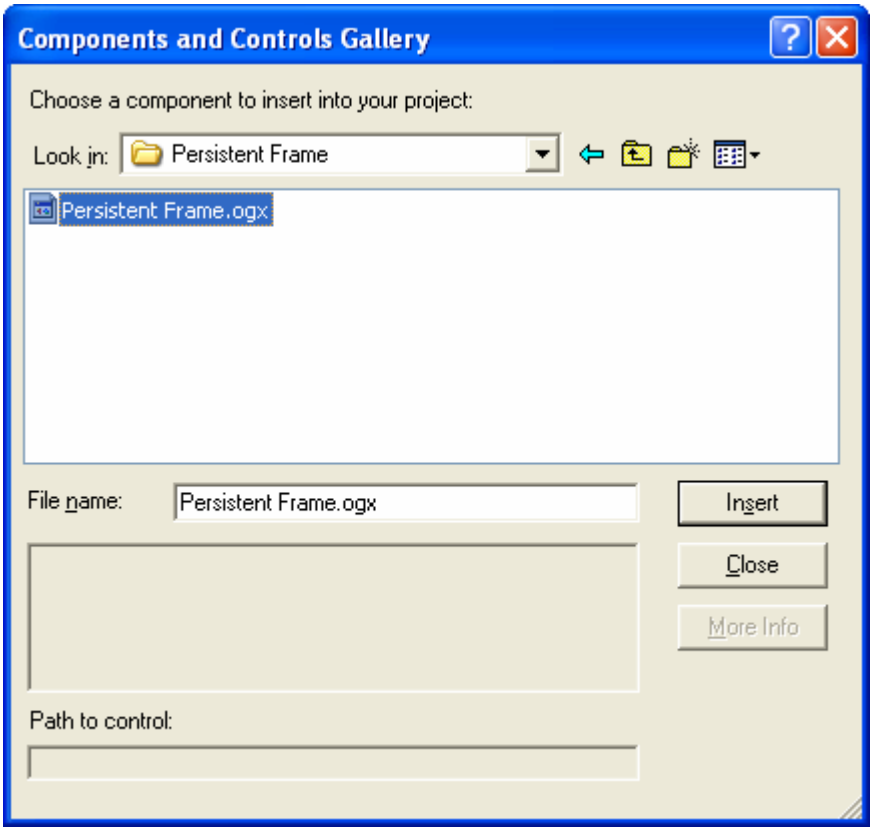

Figure 5: Our previous `CPersistentFrame` class that we stored in the gallery.

If you don't want to use the **OGX** component, you can copy the files **Persist.h** and **Persist.cpp** into your project directory and add them to the project by choosing **Add To Project** from the **Visual C++ Project** menu.

Figure 6: Adding **Persist.h** and **Persist.cpp** files manually to the MYMFC22A project.

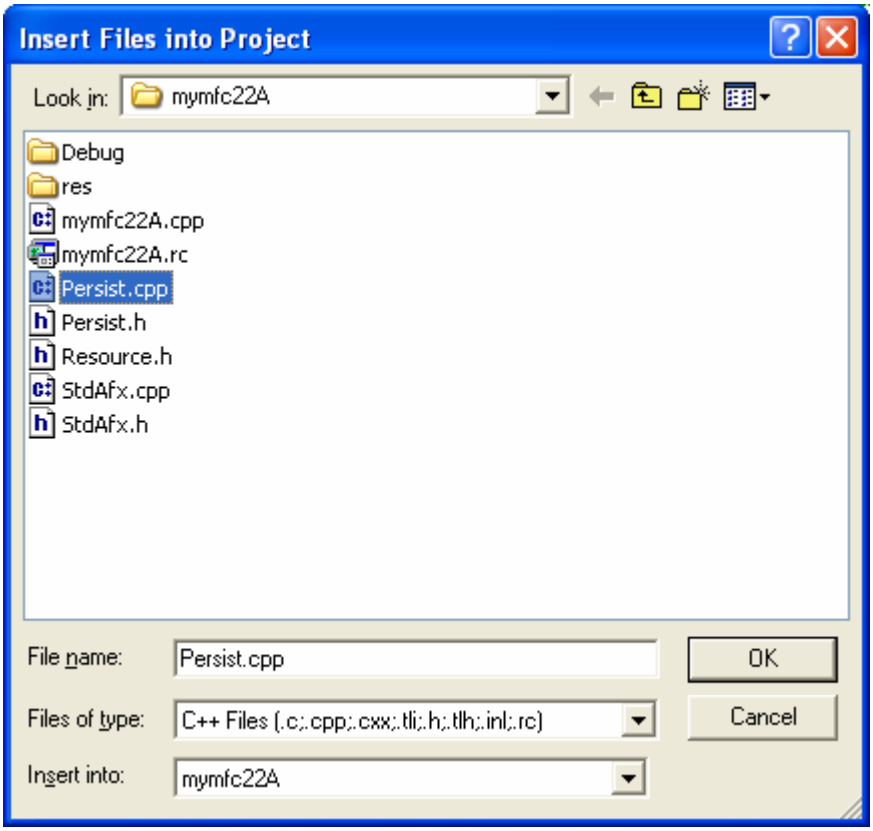

Figure 7: Selecting **Persist.h** and **Persist.cpp** files.

Edit the **persist.h** file. Modify the line:

```
class CPersistentFrame : public CFrameWnd
```

to read:

```
class AFX_EXT_CLASS CPersistentFrame : public CFrameWnd
```

```
// Persist.h

#ifndef _INSIDE_VISUAL_CPP_PERSISTENT_FRAME
#define _INSIDE_VISUAL_CPP_PERSISTENT_FRAME

class AFX_EXT_CLASS CPersistentFrame : public CFrameWnd
{ // remembers where it was on the desktop
    DECLARE_DYNAMIC(CPersistentFrame)
private:
```

Listing 1.

Build the project and **copy the DLL file**. Copy the file **mymfc22A.dll** from the \myproject\mymfc22A**Debug** directory to your system directory (\Windows**System** or \Winnt**System32**).

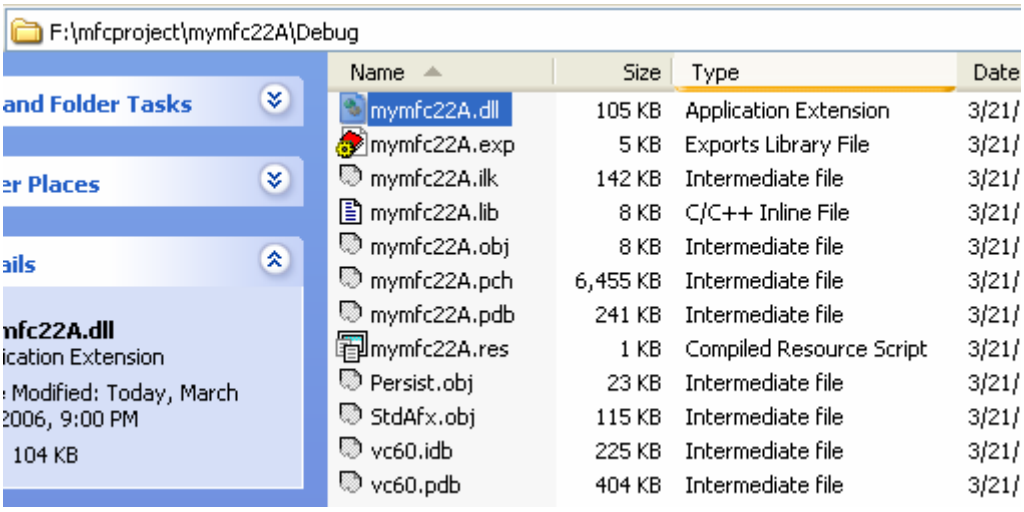

Figure 8: The generated DLL file, copied to the Windows system directory.

System directory for Win Xp Pro is shown below (or C:\WINDOWS**system32**).

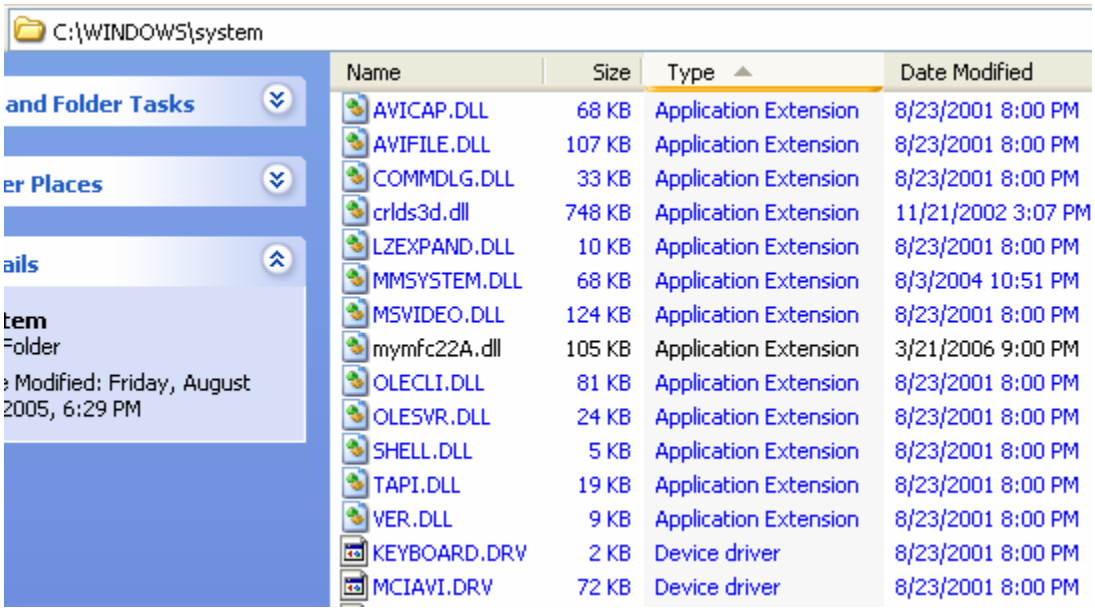

Figure 9: Copy the previous generated DLL file to the Windows directory, so that it can be found by applications from any path.

The MYMFC22B Example: A DLL Test Client Program

This example starts off as a client for **mymfc22A.dll**. It imports the `CPersistentFrame` class from the DLL and uses it as a base class for the SDI frame window. Later you'll add code to load and test the other sample DLLs in this module. Here are the steps for building the MYMFC22B example:

Run AppWizard to produce \mfcproject**mymfc22B**. This is an **ordinary MFC EXE program**. Select **Single Document**. Otherwise, accept the default settings. Be absolutely sure that in Step 5 you accept the **As A Shared DLL** option.

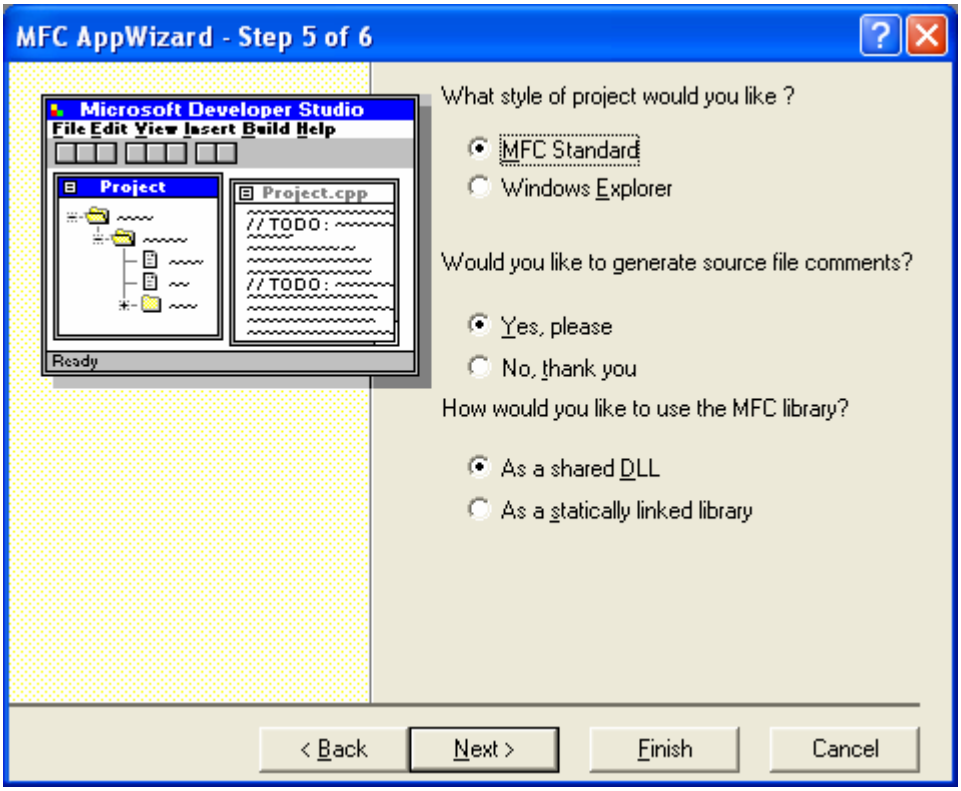

Figure 10: AppWizard step 5 of 6, selecting **As a shared DLL** option.

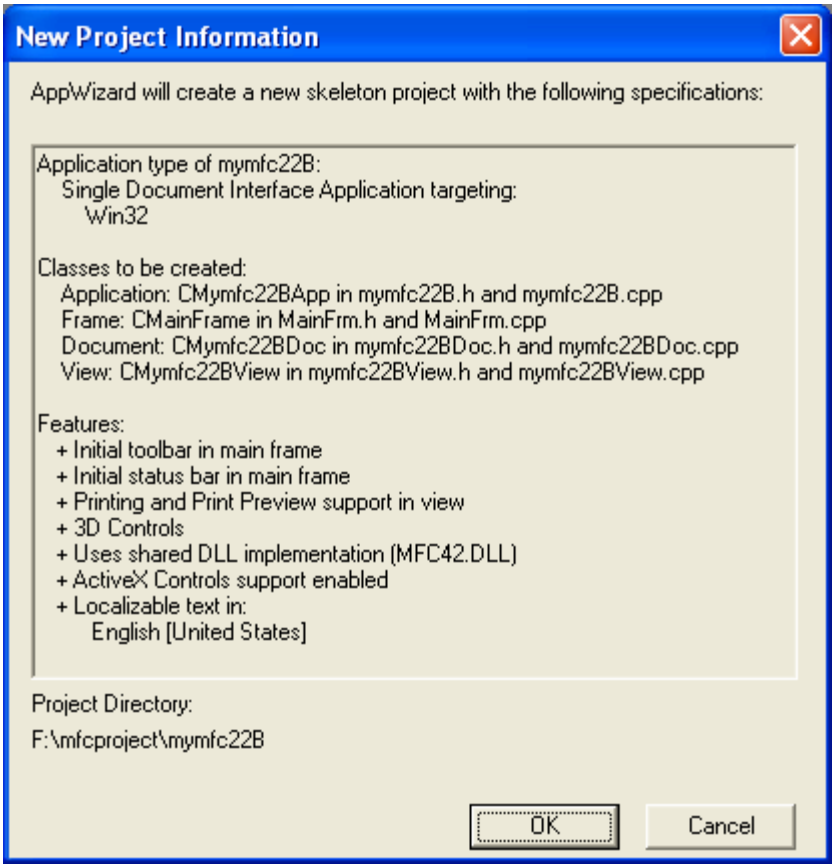

Figure 11: MYMFC22A SDI project summary.

Copy the file **persist.h** from the \mfcproject**mymfc22A** directory to \mfcproject**mymfc22B**. Note that you're copying the **header file**, not the source file, **persist.cpp**.

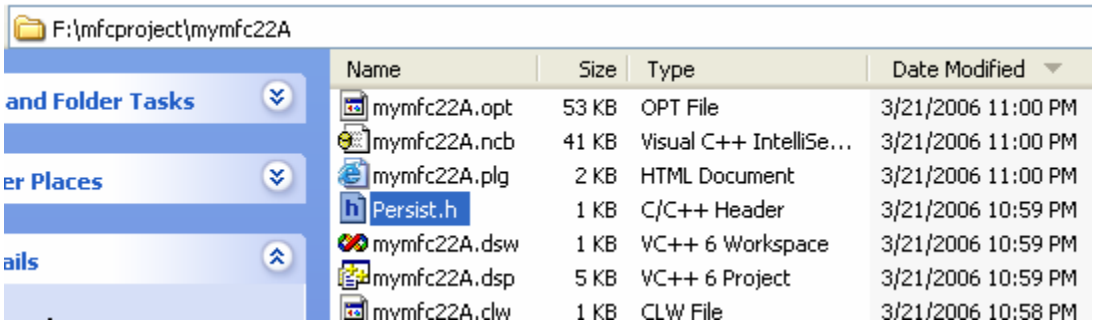

Figure 12: Copying the **Persist.h** header file from the MYMFC22A project directory.

To **mymfc22B** directory.

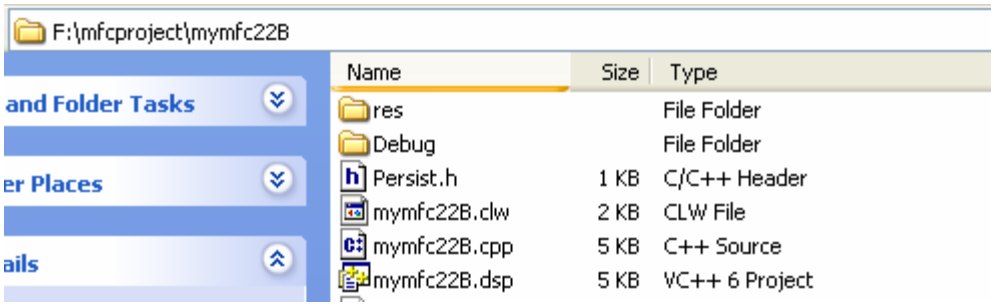

Figure 13: The MYMFC22B project directory.

Also insert the following line into **MainFrm.h**:

```
#include "persist.h"
```

```
// MainFrm.h : interface of the CMainF
//
/////////////////////////////////////////

#include "persist.h"

#if !defined(AFX_MAINFRM_H__68A0B1AF_7
#define AFX_MAINFRM_H__68A0B1AF_707A_4

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
```

Listing 2.

Change the `CFrameWnd` base class to `CPersistentFrame` as you did in MYMFC14. Replace all occurrences of `CFrameWnd` with `CPersistentFrame` in both **MainFrm.h** and **MainFrm.cpp**.

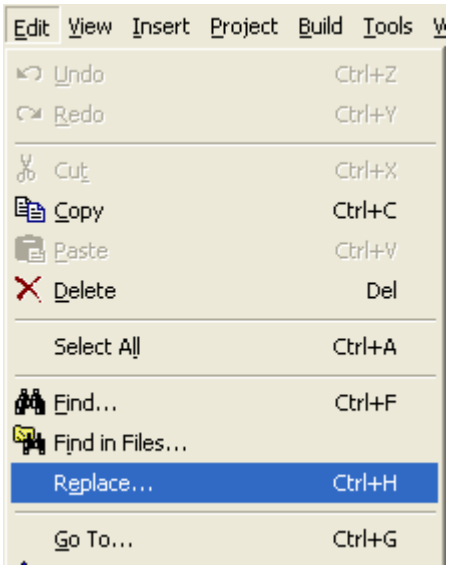

Figure 14: Invoking the find and replace menu.

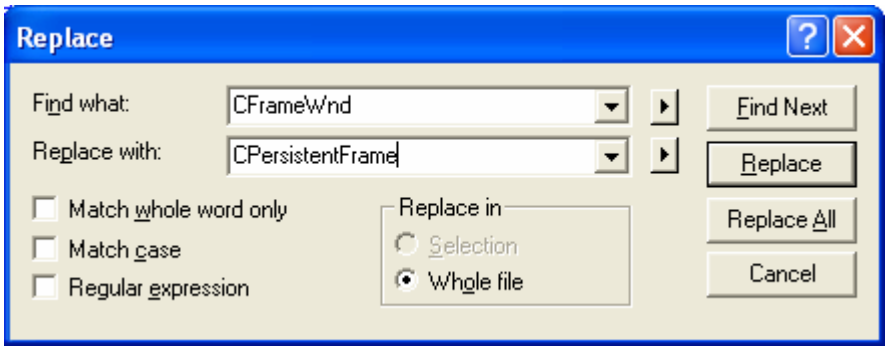

Figure 15: Replacing all the `CFrameWnd` with `CPersistentFrame` in **MainFrm.h** and **MainFrm.cpp** files.

Add the **mymfc22A** import library to the linker's input library list. Choose **Settings** from Visual C++'s **Project** menu. Select **All Configurations** in the **Settings For** drop-down list. Then fill in the **Object/Library Modules** control on the **Link** page as shown below.
You must specify the full pathname for the **mymfc22A.lib** file unless you have a copy of that file in your project directory.

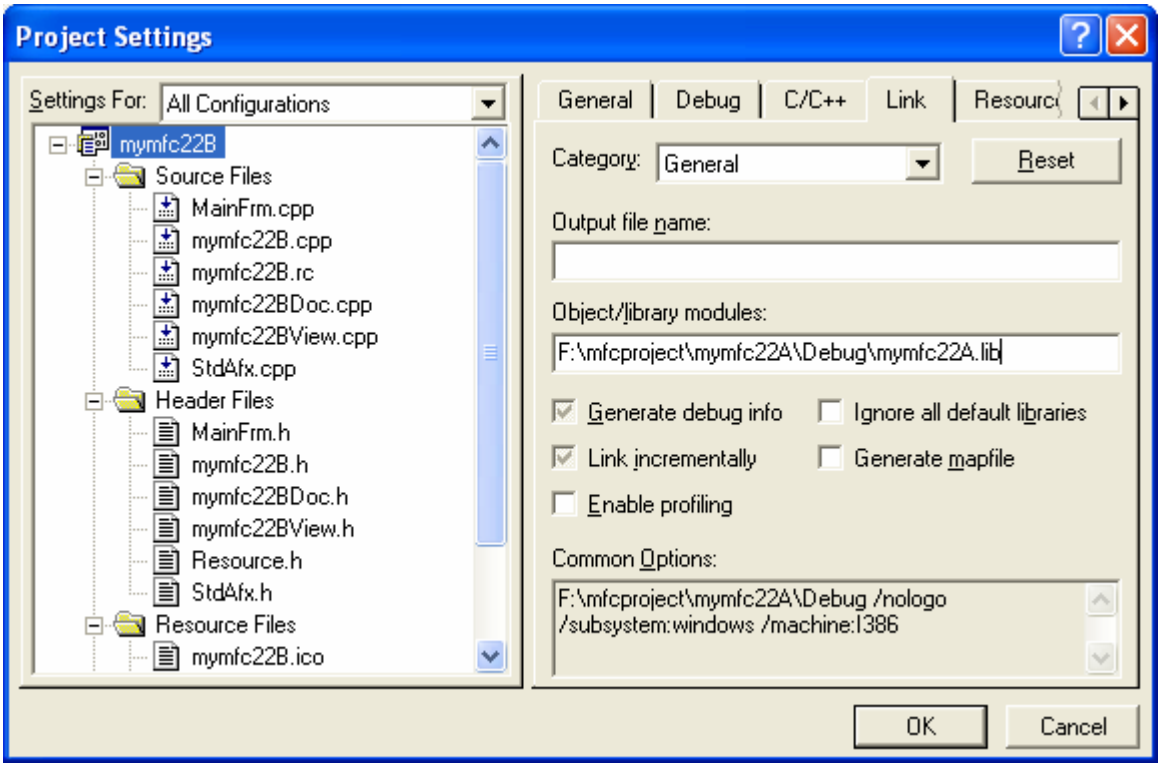

Figure 16: Adding the **mymfc22A.lib** (import) library to the linker's input library list.

Build and test the MYMFC22B program. If you run the program from the debugger and Windows can't find the **mymfc22A.dll**, Windows displays a message box when MYMFC22B starts. If all goes well, you should have a persistent frame application that works exactly like the one in EX15A. The only difference is that the `CPersistentFrame` code (**Persist.h** and **Persist.cpp**) is in an extension DLL.

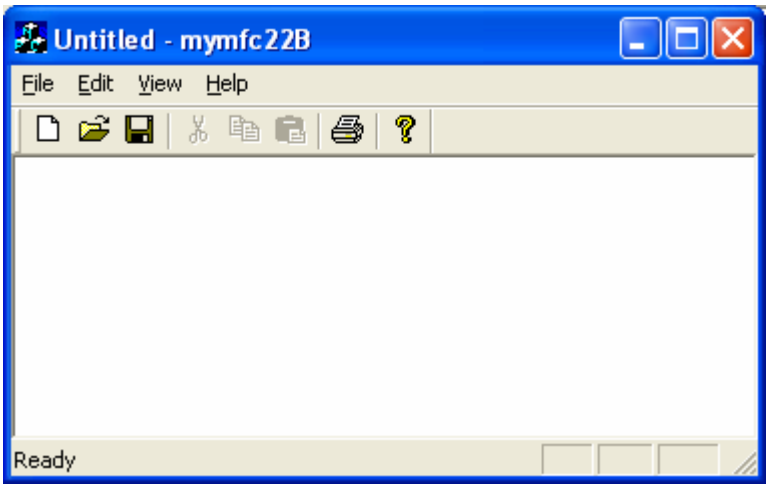

Figure 17: MYMFC22B program output, using the `CPersistentFrame` class through the DLL.

MFC Regular DLLs: The `CWinApp` Derived Class

When AppWizard generates a regular DLL, the `DllMain()` function is inside the framework and you end up with a class derived from `CWinApp` (and a global object of that class), just as you would with an EXE program. You can get control by overriding `CWinApp::InitInstance` and `CWinApp::ExitInstance`. Most of the time, you don't bother overriding those functions, though. You simply write the C functions and then export them with the `__declspec(dllexport)` modifier (or with entries in the project's **DEF** file).

Using the AFX_MANAGE_STATE Macro

When **mfc42.dll** is loaded as part of a process, it stores data in some truly global variables. If you call MFC functions from an MFC program or extension DLL, **mfc42.dll** knows how to set these global variables on behalf of the calling process. If you call into **mfc42.dll** from a regular MFC DLL, however, the global variables are not synchronized and the effects will be unpredictable. To solve this problem, insert the line:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState());
```

at the start of all exported functions in your regular DLL. If the MFC code is statically linked, the macro will have no effect.

The MFC Regular DLL Resource Search Sequence

When an EXE links to a regular DLL, resource loading functions inside the EXE will load the EXE's own resources. Resource loading functions inside the regular DLL will load the DLL's own resources. If you want your EXE code to load resources from the DLL, you can use AfxSetResourceHandle to temporarily change the resource handle. The code will be nearly the same as that shown in "**The MFC Extension DLL Resource Search Sequence**" topic. If you're writing an application that needs to be localized, you can put language-specific strings, dialogs, menus, and so forth in an MFC regular DLL. You might, for example, include the modules English.dll, German.dll, and French.dll. Your client program would explicitly load the correct DLL and use code such as that in "**The MFC Extension DLL Resource Search Sequence**" topic to load the resources, which would have the same IDs in all the DLLs.

The MYMFC22C Example: An MFC Regular DLL

This example creates a regular DLL that exports a single square root function. First you'll build the **mymfc22C.dll** file, and then you'll modify the test client program, MYMFC22B, to test the new DLL.

Here are the steps for building the MYMFC22C example:

Run AppWizard to produce \mfcproject**mymfc22C**. Proceed as you did for MYMFC22A, but accept **Regular DLL Using Shared MFC DLL** (instead of choosing **MFC Extension DLL**) from the one and only AppWizard page.

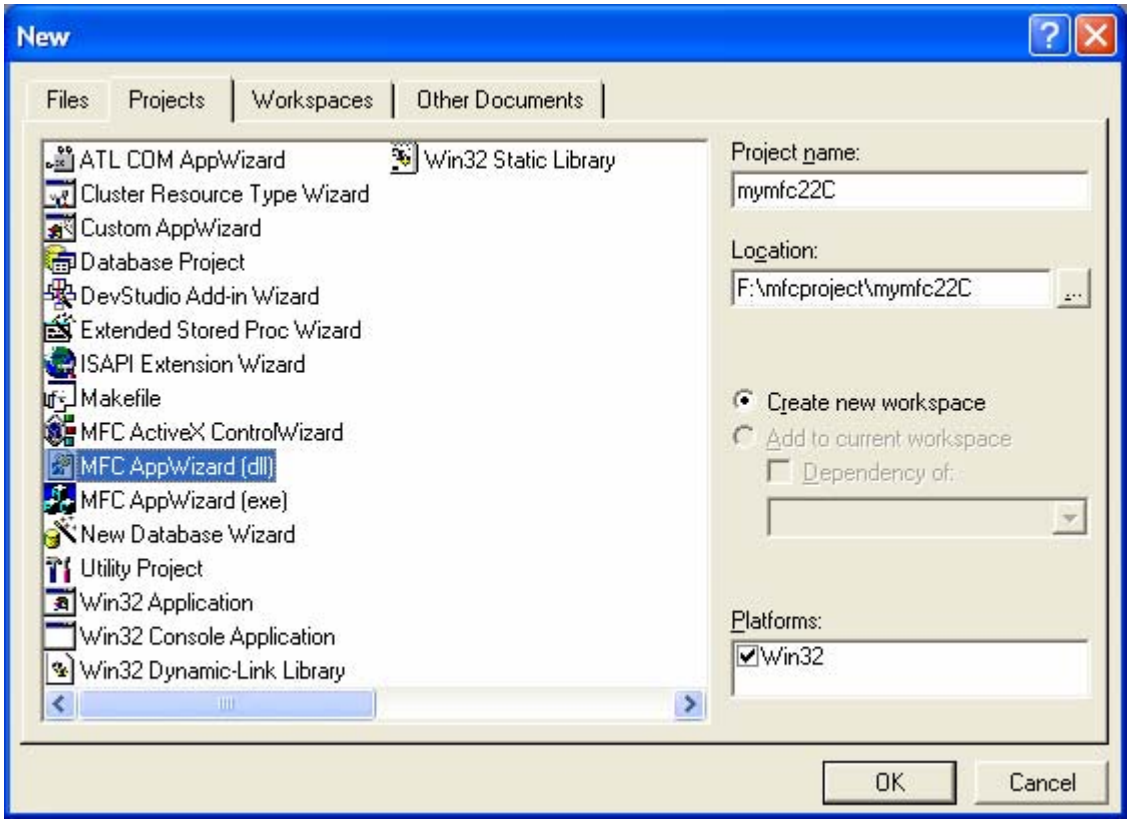

Figure 18: MYMFC22C, new DLL project dialog.

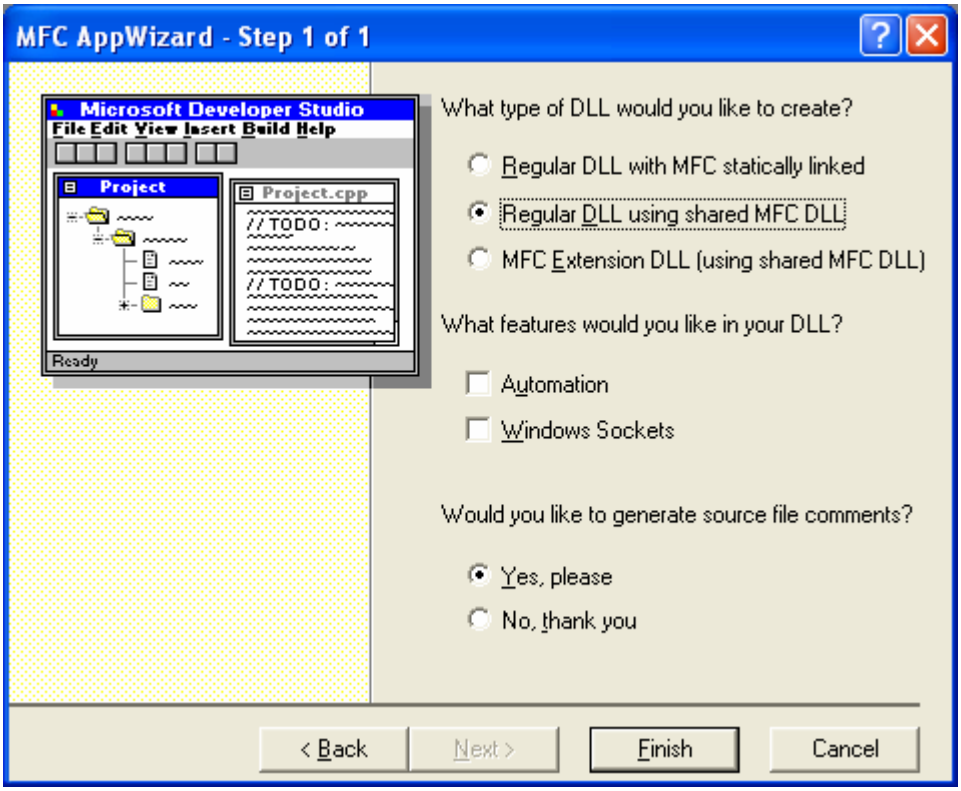

Figure 19: The only step 1 of 1 AppWizard for MYMFC22C, a **Regular DLL using shared MFC DLL**.

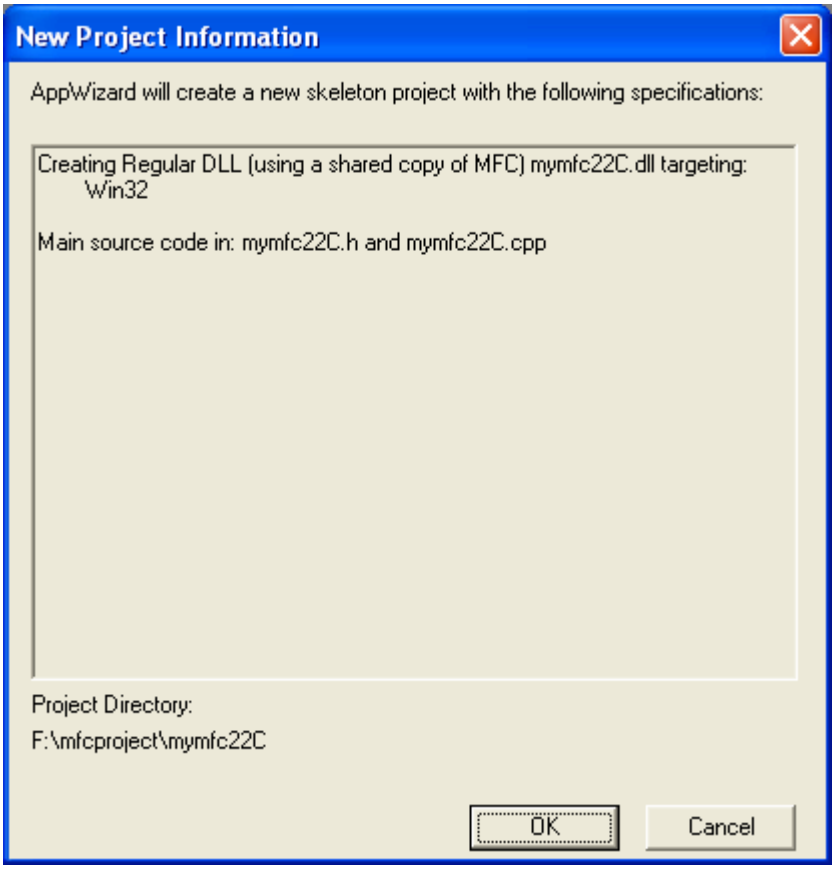

Figure 20: MYMFC22C DLL project summary.

Examine the **mymfc22C.cpp** file. AppWizard generates the following code, which includes a derived `CWinApp` class:

```
// mymfc22C.cpp : Defines the initialization routines for the DLL.
//

#include "stdafx.h"
#include "mymfc22C.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.
//

/////////////////////////////////////////////////////////////////////////////
// CMymfc22CApp

BEGIN_MESSAGE_MAP(CMymfc22CApp, CWinApp)
       //{{AFX_MSG_MAP(CMymfc22CApp)
              // 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()

/////////////////////////////////////////////////////////////////////////////
// CMymfc22CApp construction

CMymfc22CApp::CMymfc22CApp()
{
       // TODO: add construction code here,
       // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CMymfc22CApp object

CMymfc22CApp theApp;
```

Add the code for the exported `Mymfc22CSquareRoot()` function. It's okay to add this code in the **mymfc22C.cpp** file, although you can use a new file if you want to:

```
extern "C" __declspec(dllexport) double Mymfc22CSquareRoot(double d)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    TRACE("Entering Mymfc22CSquareRoot\n");
    if (d >= 0.0)
    {
        return sqrt(d);
    }
    AfxMessageBox("Can't take square root of a negative number.");
    return 0.0;
}
```

```
//////////////////////////////////////////////////////////////////////////
// The one and only CMymfc22CApp object

CMymfc22CApp theApp;

extern "C" __declspec(dllexport) double Mymfc22CSquareRoot(double d)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    TRACE("Entering Mymfc22CSquareRoot\n");
    if (d >= 0.0)
    {
        return sqrt(d);
    }
    AfxMessageBox("Can't take square root of a negative number.");
    return 0.0;
}
```

Listing 3.

You can see that there's no problem with the DLL displaying a message box or another modal dialog. You'll need to include `math.h` in the file containing this code because we are going to use the `sqrt()` pre-defined function.

```
// mymfc22C.cpp : Defines the init
//

#include "stdafx.h"
#include "mymfc22C.h"
#include "math.h"

#ifdef _DEBUG
#define new DEBUG_NEW
```

Listing 4.

Build the project and copy the DLL file. Copy the file **mymfc22C.dll** from the \mfcproject\mymfc22C**Debug** directory to your system directory.

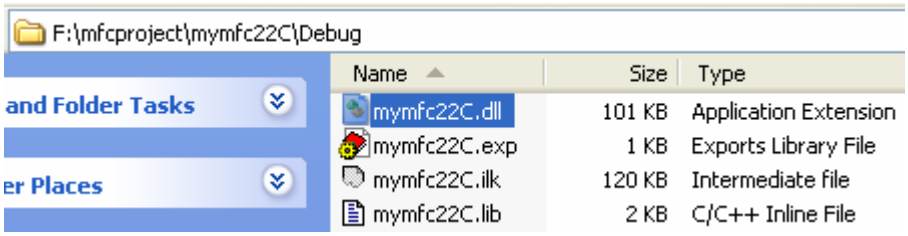

Figure 21: Generated DLL file of MYMFC22C program.

Updating the MYMFC22B Example: Adding Code to Test `mymfc22C.dll`

When you first built the MYMFC22B program, it linked dynamically to the MYMFC22A MFC extension DLL. Now you'll update the project to implicitly link to the MYMFC22C MFC regular DLL and to call the DLL's square root function.

Following are the steps for updating the MYMFC22B example.

Add a new dialog resource and class to \mfcproject **mymfc22B**. Use the dialog editor to create the `IDD_MYMFC22C` template, as shown here.

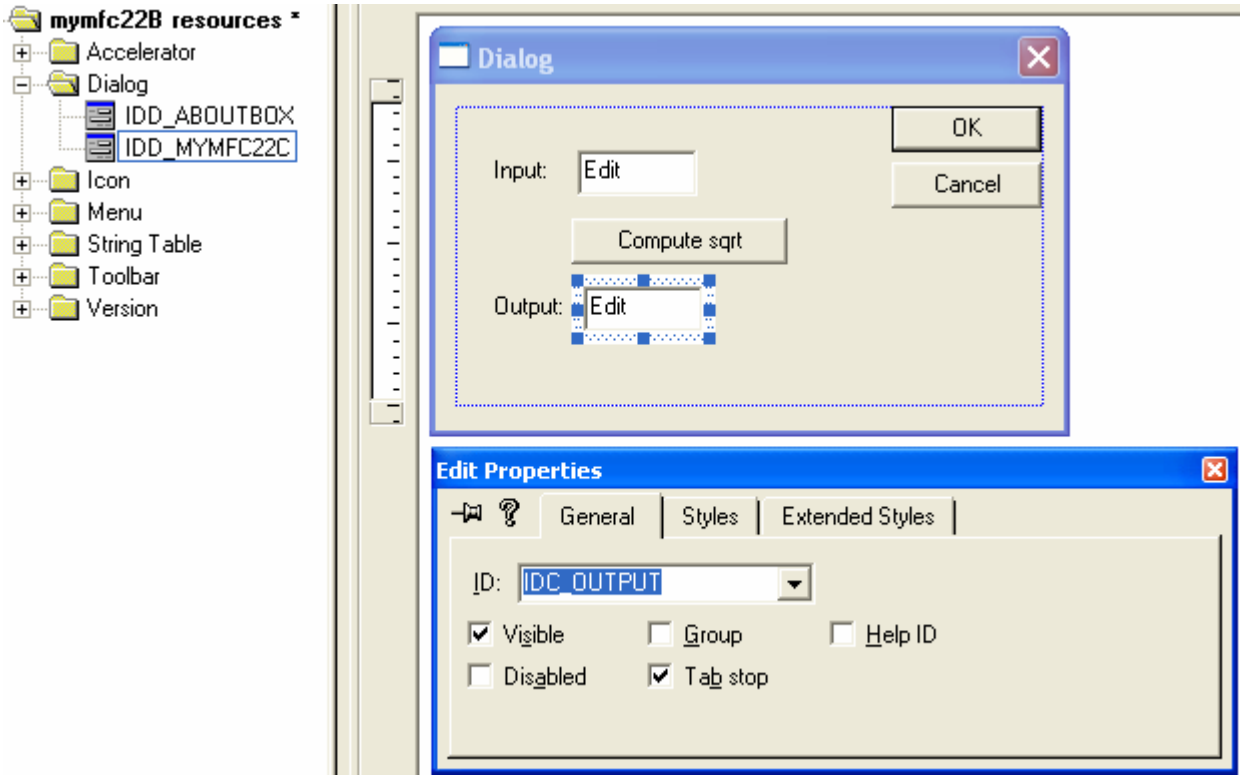

Figure 22: Adding and modifying a new dialog resource to MYMFC22B project.

Then use ClassWizard to generate a class `CTest22cDialog`, derived from `CDialog`.

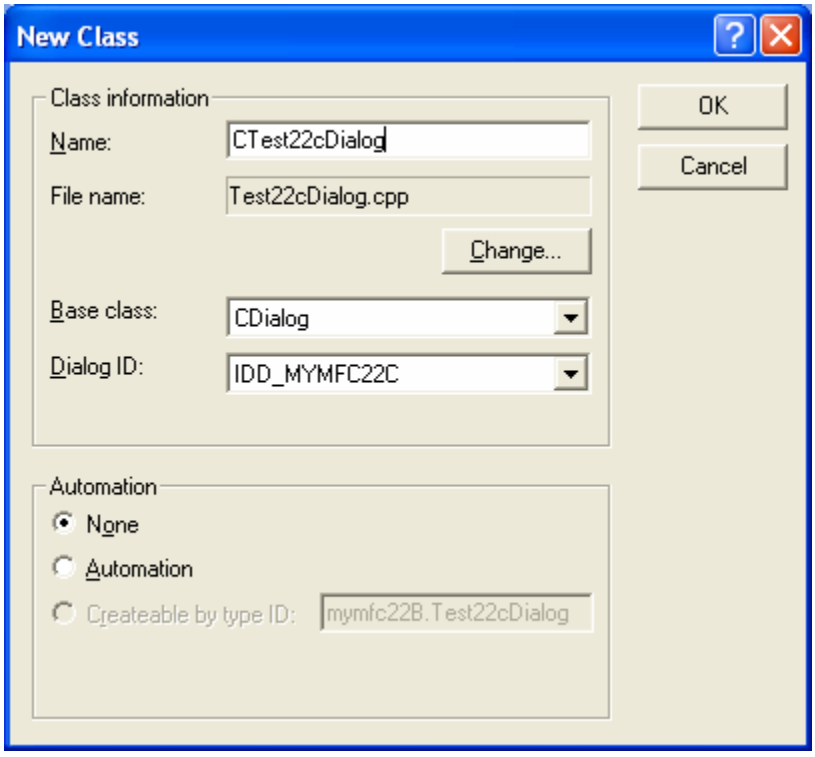

Figure 23: Adding and modifying a new class to MYMFC22B project.

The controls, data members, and message map function are shown in the following table.

Control ID	Type	Data Member	Message Map Function
IDD_MYMFC22C	Dialog template	-	-
IDC_INPUT	edit	m_dInput (double)	-
IDC_OUTPUT	edit	m_dOutput (double)	-
IDC_COMPUTE	button	-	OnCompute()

Table 3.

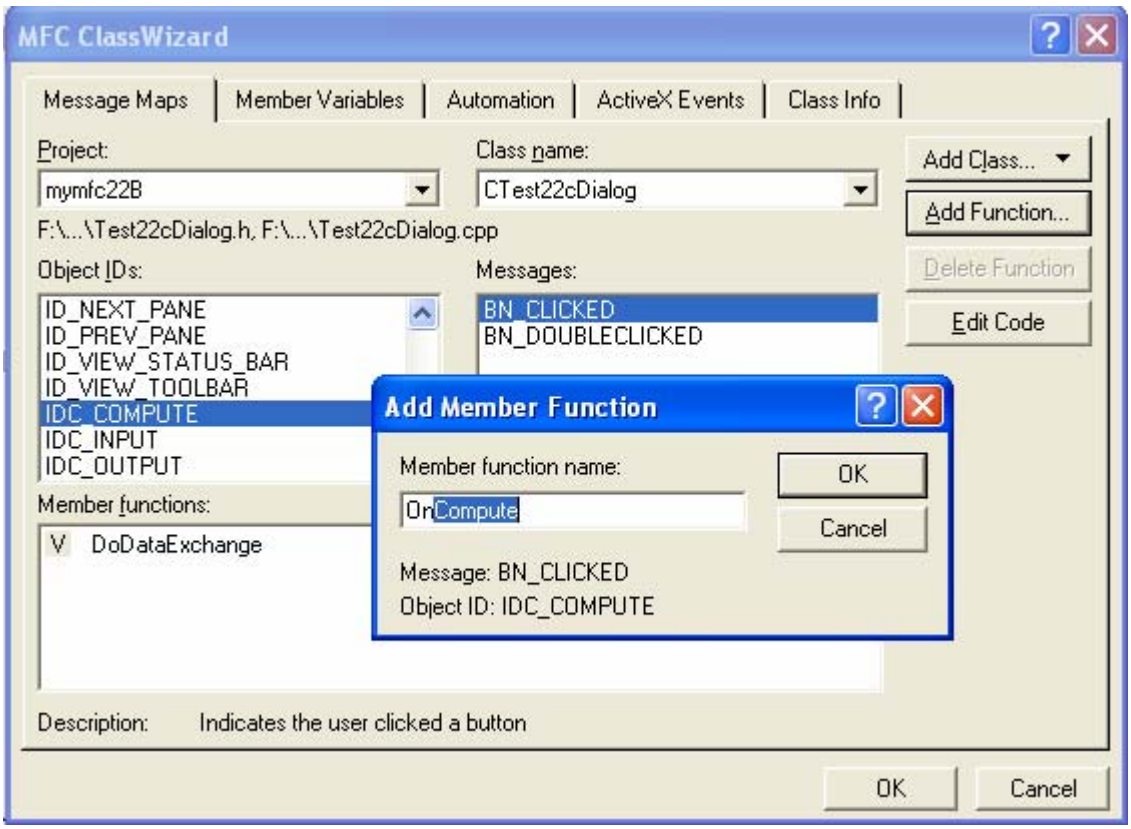

Figure 24: Adding IDC_COMPUTE message handler function.

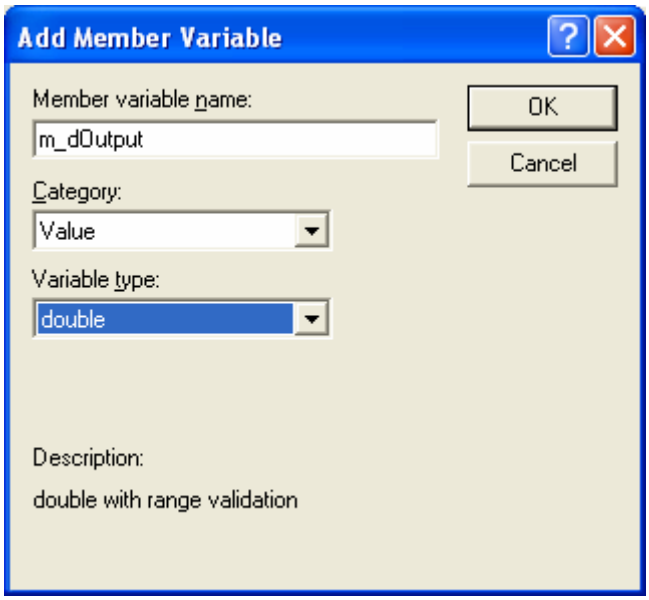

Figure 25: Adding data members/member variables using ClassView.

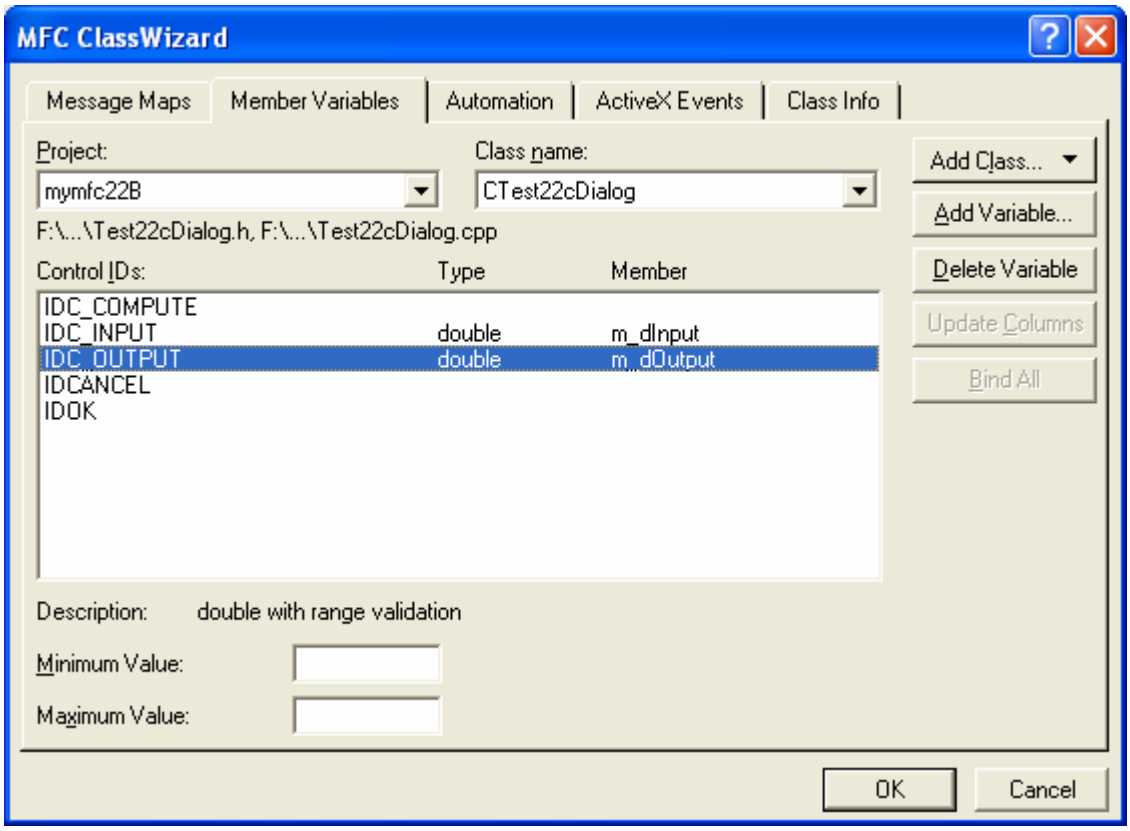

Figure 26: Adding data members/member variables using ClassWizard.

Code the `OnCompute()` function to call the DLL's exported function. Edit the ClassWizard-generated function in **Test22cDialog.cpp** as shown here:

```
void CTest22cDialog::OnCompute()
{
    UpdateData(TRUE);
    m_dOutput = Mymfc22CSquareRoot(m_dInput);
    UpdateData(FALSE);
}
```

```
// CTest22cDialog message handlers

void CTest22cDialog::OnCompute()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    m_dOutput = Mymfc22CSquareRoot(m_dInput);
    UpdateData(FALSE);
}
```

Listing 5.

You'll have to declare the `Mymfc22CSquareRoot()` function as an imported function. Add the following line to the **Test22cDialog.h** file:

```
extern "C" __declspec(dllimport) double Mymfc22CSquareRoot(double d);
```

```
// CTest22cDialog dialog

extern "C" __declspec(dllimport) double Mymfc22CSquareRoot(double d);

class CTest22cDialog : public CDialog
```

Listing 6.

Integrate the `CTest22cDialog` class into the MYMFC22B application. You'll need to add a top-level menu, **Test**, and a **Mymfc22C DLL** option with the ID `ID_TEST_MYMFC22CDLL`.

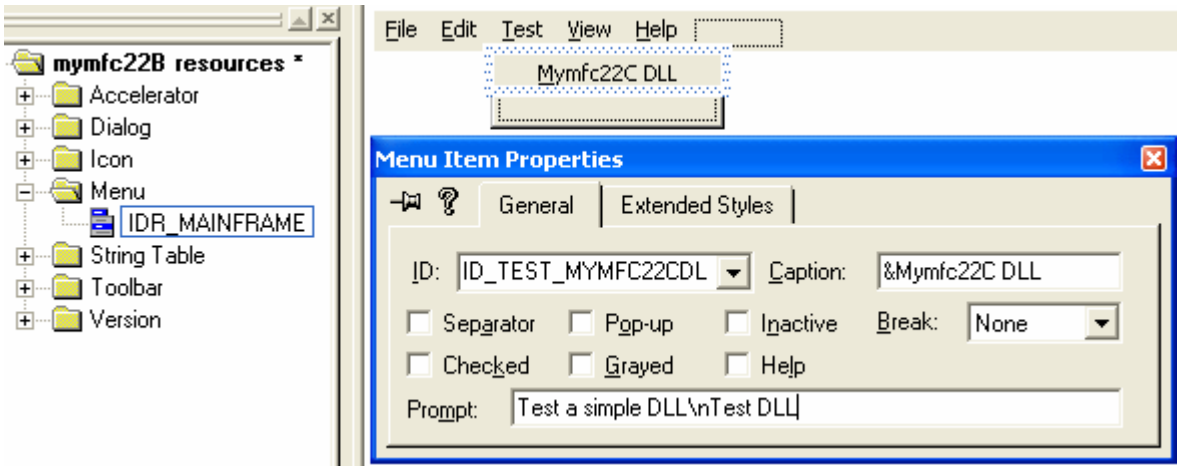

Figure 27: Adding and modifying menu and its item to the menu resource.

Use ClassWizard to map this option to a member function in the `CMymfc22BView` class, and then code the handler in `Mymfc22BView.cpp` as follows:

```
      void CMymfc22BView::OnTestMymfc22Cdll()
      {
          CTest22cDialog dlg;
          dlg.DoModal();
      }

// CMymfc22BView message handlers

void CMymfc22BView::OnTestMymfc22cdll()
{
    // TODO: Add your command handler code here
    CTest22cDialog dlg;
    dlg.DoModal();
}
```

Listing 7.

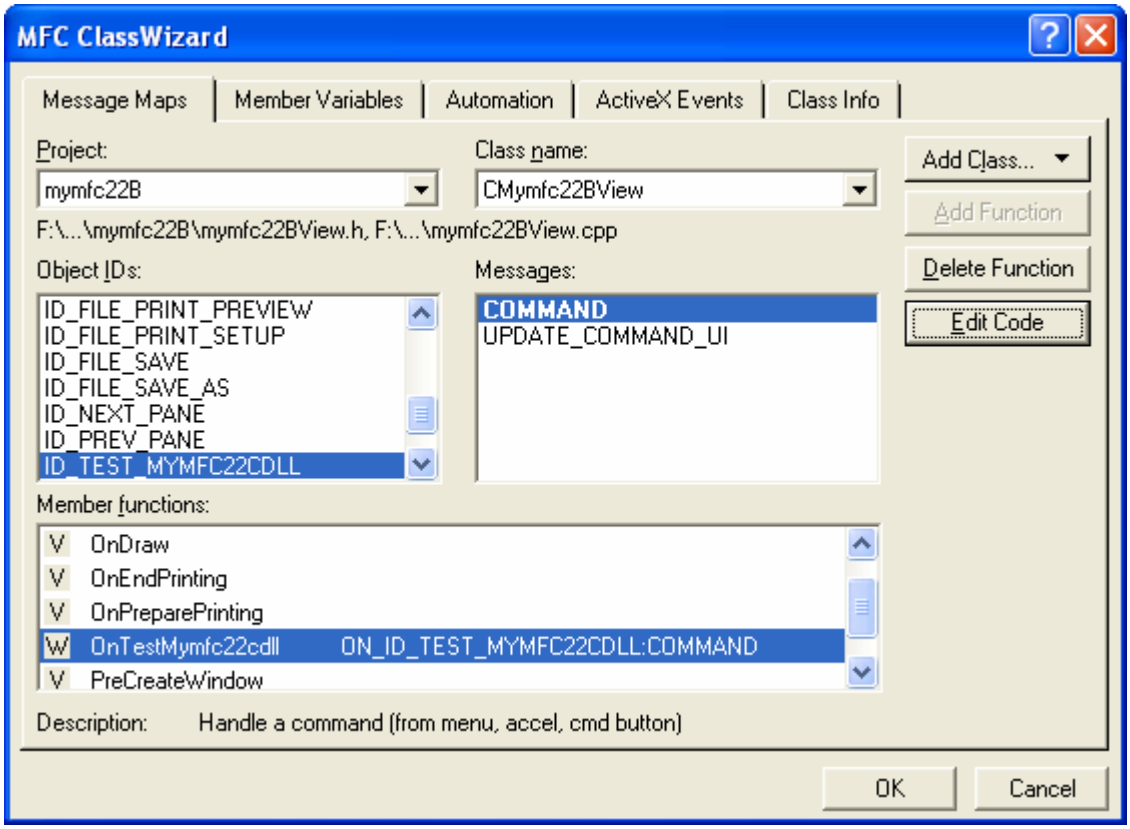

Figure 28: Using ClassWizard to map the menu item to a member function in the `CMymfc22BView` class.

Of course, you'll have to add this `#include` line to the **Mymfc22BView.cpp** file:

```
   #include "Test22cDialog.h"
```

```
// mymfc22BView.cpp : imple
//

#include "stdafx.h"
#include "mymfc22B.h"

#include "mymfc22BDoc.h"
#include "mymfc22BView.h"

#include "Test22cDialog.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
```

Listing 8.

Add the MYMFC22C import library to the linker's input library list. Choose **Settings** from Visual C++'s **Project** menu, and then add **\mfcproject\mymfc22C\Debug\mymfc22C.lib** to the **Object/Library modules** control on the **Link** page. Use a space to separate the new entry from the existing entry. In this example a full path is used as shown in the following Figure.

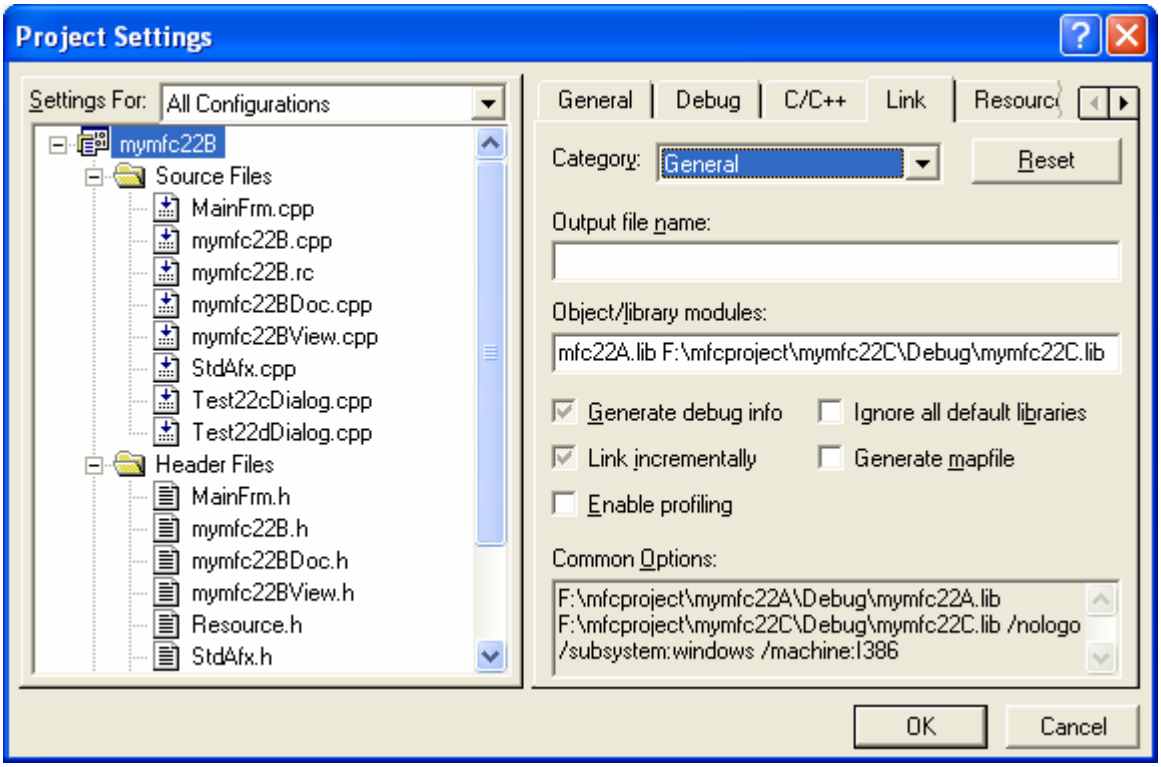

Figure 29: Adding the **mymfc22C.lib** (import) library to the linker's input library list.

Now the program should **implicitly link** to both the **MYMFC22A DLL** and the **MYMFC22C DLL**. As you can see, the client doesn't care whether the DLL is a regular DLL or an extension DLL. You just specify the **LIB name** to the linker.

Build and test the updated MYMFC22B application. Choose **Mymfc22C DLL** from the **Test** menu. Type a number in the **Input** edit control, and then click the **Compute Sqrt** button. The result should appear in the **Output** control.

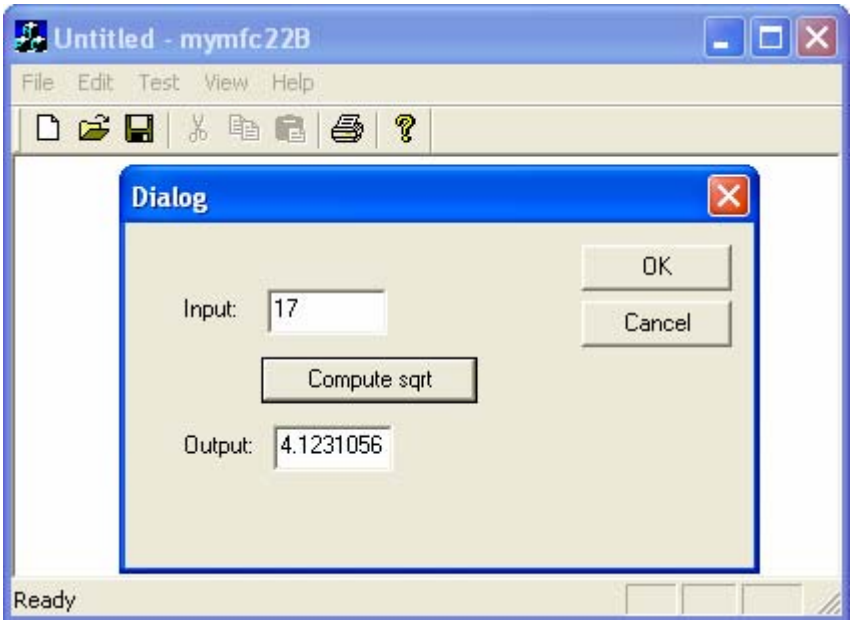

Figure 30: MYMFC22B program output with new DLL that used to compute a square root of a given number.

A Custom Control DLL

Programmers have been using DLLs for custom controls since the early days of Windows because custom controls are neatly self-contained. The original custom controls were written in pure C and configured as stand-alone DLLs. Today you can use the features of the MFC library in your custom controls, and you can use the wizards to make coding easier. A regular DLL is the best choice for a custom control because the control doesn't need a C++ interface and because it can be used by any development system that accepts custom controls (such as the Borland C++ compiler). You'll probably want to use the MFC dynamic linking option because the resulting DLL will be small and quick to load.

What Is a Custom Control?

You've seen ordinary controls and Microsoft Windows common controls in Module 5, and you've seen ActiveX controls in Module 18. The custom control acts like an ordinary control, such as the edit control, in that it sends `WM_COMMAND` notification messages to its parent window and receives user-defined messages. The dialog editor lets you position custom controls in dialog templates. That's what the "head" control palette item, shown here, is for.

Figure 31: Custom control in control palette.

You have a lot of freedom in designing your custom control. You can paint anything you want in its window (which is managed by the client application) and you can define any notification and inbound messages you need. You can use ClassWizard to map normal Windows messages in the control (`WM_LBUTTONDOWN`, for example), but you must manually map the user-defined messages and manually map the notification messages in the parent window class.

A Custom Control's Window Class

A dialog resource template specifies its custom controls by their symbolic window class names. Don't confuse the **Win32 window** class with the C++ class; the only similarity is the name. A window class is defined by a structure that contains the following:

- The name of the class.
- A pointer to the `WndProc()` function that receives messages sent to windows of the class.
- Miscellaneous attributes, such as the background brush.

The Win32 `RegisterClass()` function copies the structure into process memory so that any function in the process can use the class to create a window. When the dialog window is initialized, Windows creates the custom control child windows from the window class names stored in the template. Suppose now that the control's `WndProc()` function is inside a DLL. When the DLL is initialized (by a call to `DllMain()`), it can call `RegisterClass()` for the control. Because the DLL is part of the process, the client program can create child windows of the custom control class. To summarize, the client knows the name string of a control window class and it uses that class name to construct the child window. All the code for the control, including the `WndProc()` function, is inside the DLL. All that's necessary is that the client loads the DLL prior to creating the child window.

The MFC Library and the `WndProc()` Function

Okay, so Windows calls the control's `WndProc()` function for each message sent to that window. But you really don't want to write an old-fashioned switch-case statement, you want to map those messages to C++ member functions, as you've been doing all along. Now, in the DLL, you must rig up a C++ class that corresponds to the control's window class. Once you've done that, you can happily use ClassWizard to map messages.
The obvious part is the writing of the C++ class for the control. You simply use ClassWizard to create a new class derived from `CWnd`. The tricky part is wiring the C++ class to the `WndProc()` function and to the application framework's message pump. You'll see a real `WndProc()` in the MYMFC22D example, but here's the pseudocode for a typical control `WndProc()` function:

```
LRESULT MyControlWndProc(HWND hWnd, UINT message WPARAM wParam, LPARAM lParam)
{
    if (this is the first message for this window)
    {
        CWnd* pWnd = new CMyControlWindowClass();
        attach pWnd to hWnd
    }
    return AfxCallWndProc(pWnd, hWnd, message, WParam, lParam);
}
```

The MFC `AfxCallWndProc()` function passes messages to the framework, which dispatches them to the member functions mapped in `CMyControlWindowClass`.

Custom Control Notification Messages

The control communicates with its parent window by sending it special `WM_COMMAND` notification messages with parameters, as shown here.

Parameter	Usage
`(HIWORD) wParam`	Notification code
`(LOWORD) wParam`	Child window ID
`lParam`	Child window handle

Table 4

The meaning of the notification code is arbitrary and depends on the control. The parent window must interpret the code based on its knowledge of the control. For example, the code **77** might mean that the user typed a character while positioned on the control.

The control might send a notification message such as this:

```
GetParent()->SendMessage(WM_COMMAND, GetDlgCtrlID() | ID_NOTIFYCODE << 16, (LONG)
GetSafeHwnd());
```

On the client side, you map the message with the MFC `ON_CONTROL` macro like this:

```
ON_CONTROL(ID_NOTIFYCODE, IDC_MYCONTROL, OnClickedMyControl)
```

Then you declare the handler function like this:

```
afx_msg void OnClickedMyControl();
```

User-Defined Messages Sent to the Control

You have already seen user-defined messages in Module 6. This is the means by which the client program communicates with the control. Because a standard message returns a 32-bit value if it is sent rather than posted, the client can obtain information from the control.

The MYMFC22D Example: A Custom Control

The MYMFC22D program is an **MFC regular DLL** that implements a traffic light control indicating off, red, yellow, and green states. When clicked with the left mouse button, the DLL sends a clicked notification message to its parent and responds to two user-defined messages, `RYG_SETSTATE` and `RYG_GETSTATE`. The state is an integer that represents the color. Credit goes to Richard Wilton, who included the original C-language version of this control in his book Windows 3 Developer's Workshop (Microsoft Press, 1991). The MYMFC22D project was originally generated using AppWizard, with linkage to the shared MFC DLL, just like MYMFC22C.

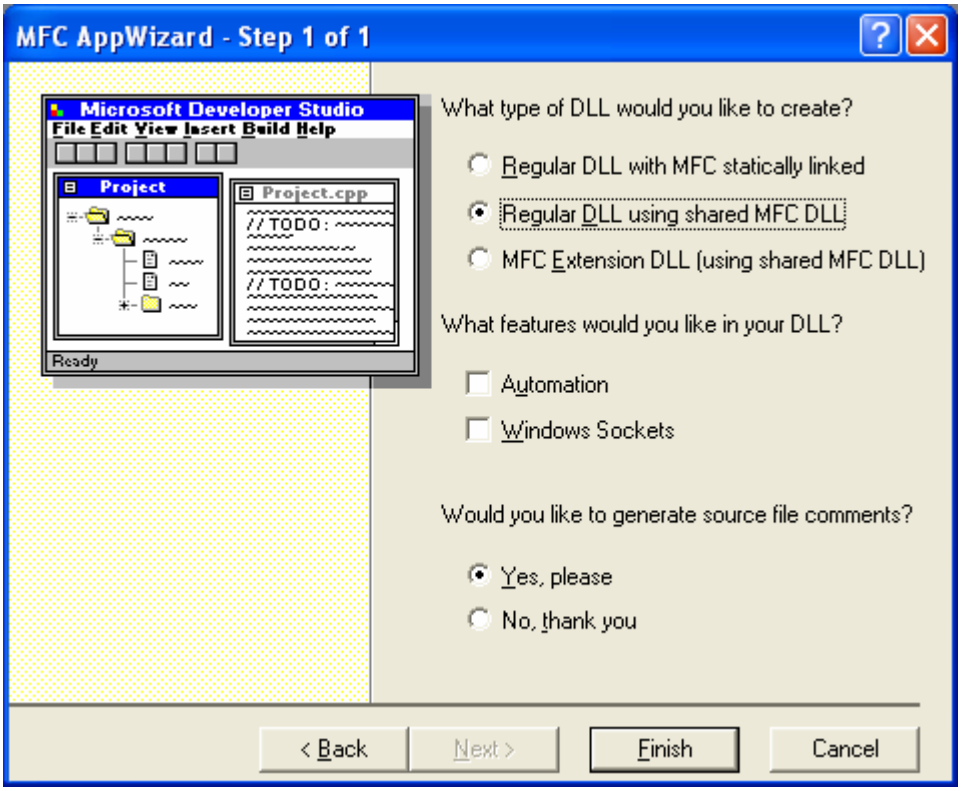

Figure 32: MYMFC22D DLL project, AppWizard step 1 of 1.

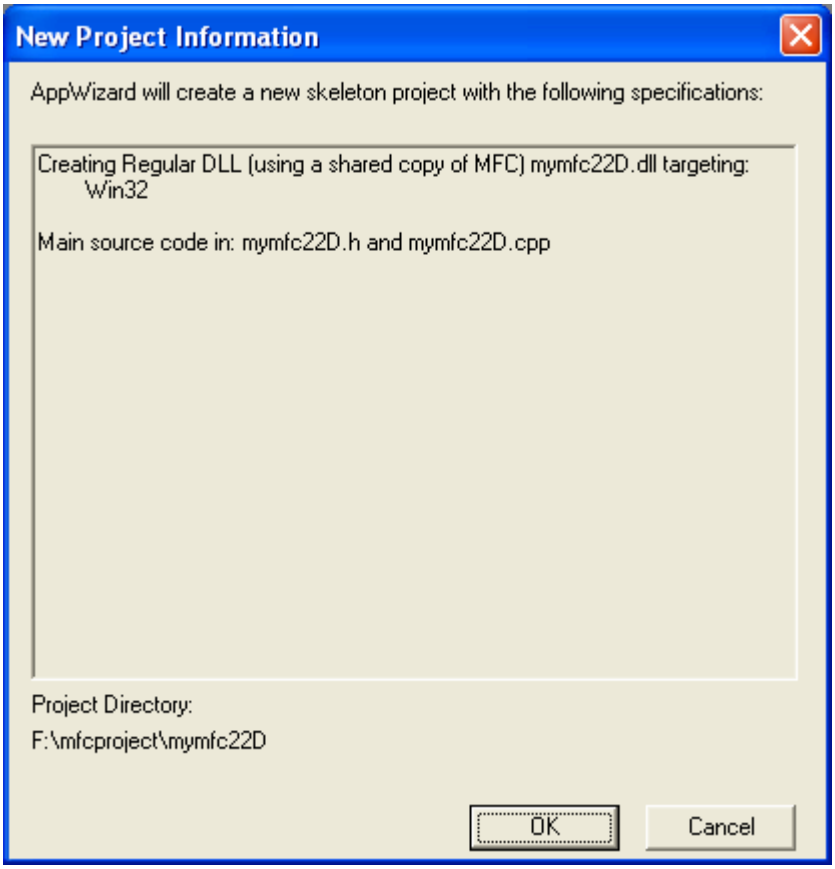

Figure 33: MYMFC22D project summary.

Listing 9 shows the code for the primary source file, with the added code in the `InitInstance()` function in orange. The dummy exported `Mymfc22DEntry()` function exists solely to allow the DLL to be implicitly linked. The client program must include a call to this function. That call must be in an executable path in the program or the compiler will eliminate the call. As an alternative, the client program could call the Win32 `LoadLibrary()` function in its `InitInstance()` function to explicitly link the DLL.

MYMFC22D.CPP

```
// mymfc22D.cpp : Defines the initialization routines for the DLL.
//

#include "stdafx.h"
#include "mymfc22D.h"
#include "RygWnd.h"

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

extern "C" __declspec(dllexport) void Mymfc22DEntry() {} // dummy
function

(generated comment lines omitted)

/////////////////////////////////////////////////////////////////////////
// CMymfc22DApp

BEGIN_MESSAGE_MAP(CMymfc22DApp, CWinApp)
    //{{AFX_MSG_MAP(CMymfc22DApp)
        // 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()

/////////////////////////////////////////////////////////////////////////
// CMymfc22DApp construction

CMymfc22DApp::CMymfc22DApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////
// The one and only CMymfc22DApp object

CMymfc22DApp theApp;

BOOL CMymfc22DApp::InitInstance()
{
    CRygWnd::RegisterWndClass(AfxGetInstanceHandle());
    return CWinApp::InitInstance();
}
```

Listing 9: The `mymfc22D.cpp`, primary source listing.

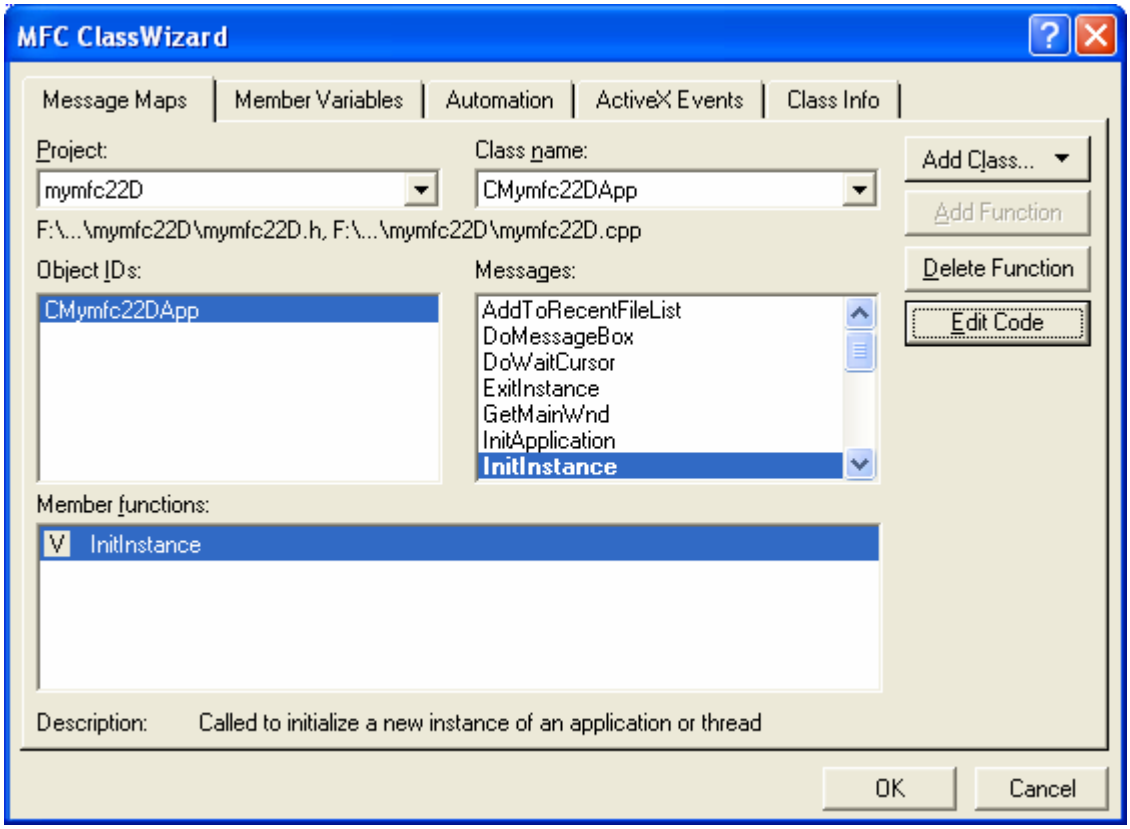

Figure 34: Message mapping for `InitInstance()` function.

The following is the `InitInstance()` code.

```
// The one and only CMymfc22DApp object

CMymfc22DApp theApp;

BOOL CMymfc22DApp::InitInstance()
{
    // TODO: Add your specialized code here and/or call the base class
    CRygWnd::RegisterWndClass(AfxGetInstanceHandle());
    return CWinApp::InitInstance();
}
```

Listing 10.

Listing 11 shows the code for the `CRygWnd` class, including the global `RygWndProc()` function. Click the **Add Class** button in ClassWizard to create this class.

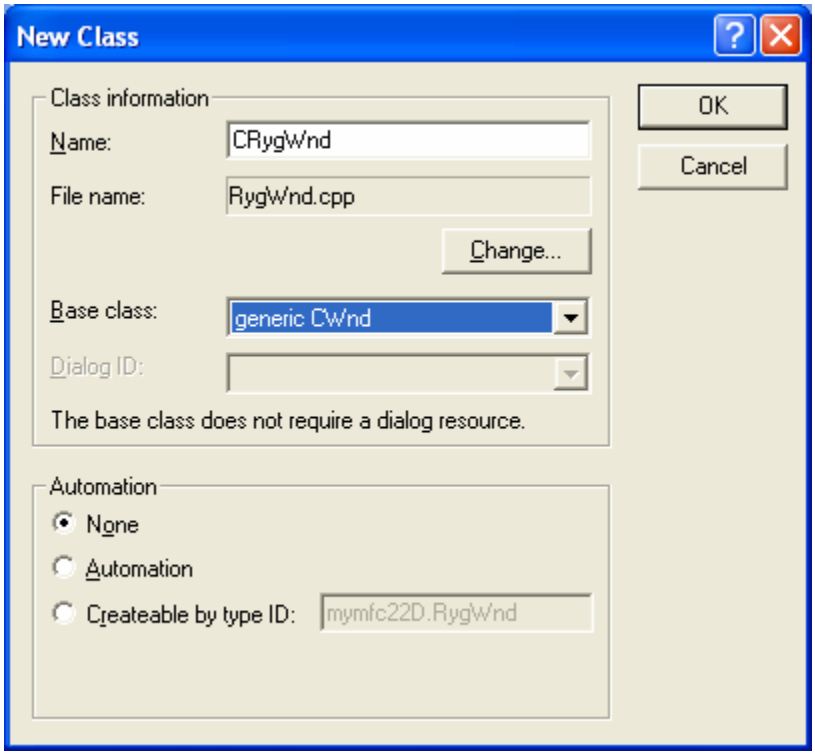

Figure 35: Creating and adding new class, `CRygWnd` to the project using **generic `CWnd`** as its base class.

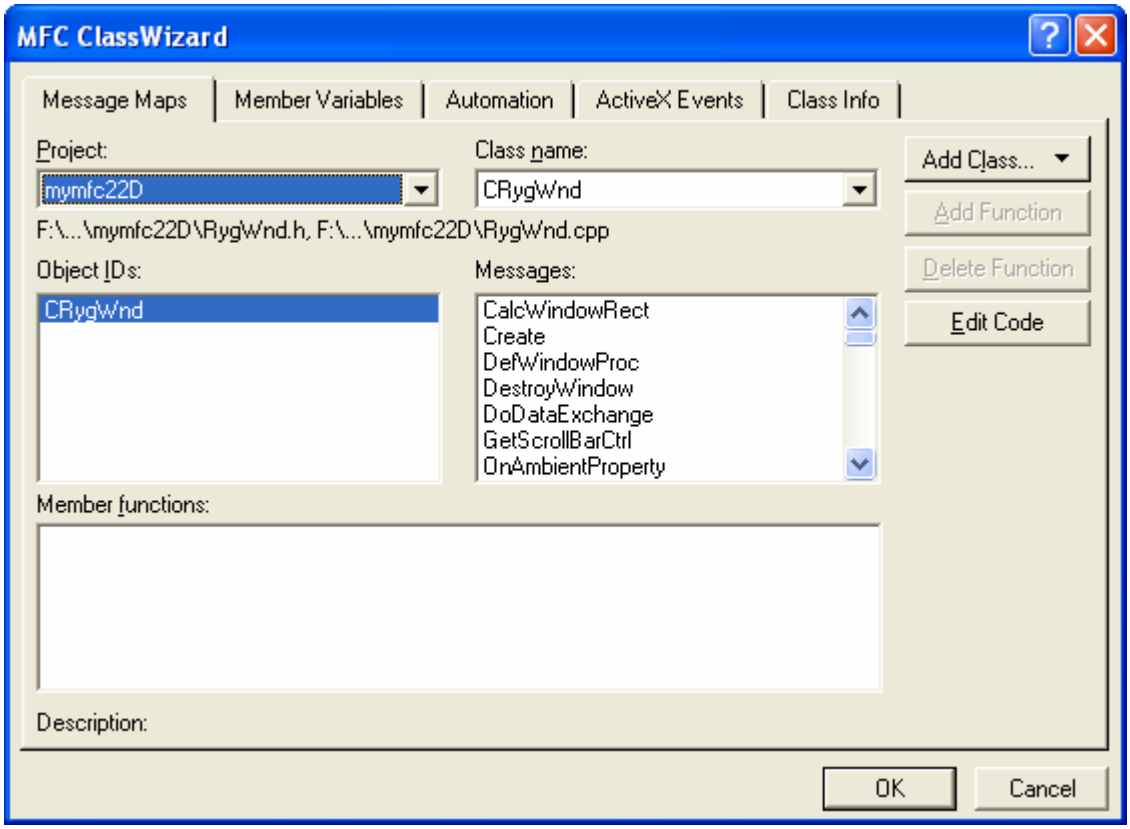

Figure 36: `CRygWnd` class integrated into the project.

The code that paints the traffic light isn't very interesting, so we'll concentrate on the functions that are common to most custom controls. The static `RegisterWndClass()` member function actually registers the `RYG` window class and must be called as soon as the DLL is loaded. The `OnLButtonDown()` handler is called when the user presses the left mouse button inside the control window. It sends the clicked notification message to the parent window. The overridden `PostNcDestroy()` function is important because it deletes the `CRygWnd` object when the client program destroys the control window. The `OnGetState()` and `OnSetState()` functions are called in response to user-defined messages sent by the client. **Remember to copy the DLL to your system directory**.
If you feel tedious to go step by step in completing the codes, just copy and paste the following codes into the **RygWnd.h** and **RygWnd.cpp** files.

RYGWND.H

```
#if
!defined(AFX_RYGWND_H__1AA889D5_9788_11D0_BED2_00C04FC2A0C2__INCLUDED_)
#define AFX_RYGWND_H__1AA889D5_9788_11D0_BED2_00C04FC2A0C2__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// RygWnd.h : header file
//

/////////////////////////////////////////////////////////////////////////
// CRygWnd window

#define RYG_SETSTATE WM_USER + 0
#define RYG_GETSTATE WM_USER + 1

LRESULT CALLBACK AFX_EXPORT
    RygWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

class CRygWnd : public CWnd
{
private:
    int m_nState; // 0=off, 1=red, 2=yellow, 3=green
    static CRect  s_rect;
    static CPoint s_point;
    static CRect  s_rColor[3];
    static CBrush s_bColor[4];

// Construction
public:
    CRygWnd();
public:
    static BOOL RegisterWndClass(HINSTANCE hInstance);

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CRygWnd)
    protected:
    virtual void PostNcDestroy();
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CRygWnd();

    // Generated message map functions
private:
```

```
    void SetMapping(CDC* pDC);
    void UpdateColor(CDC* pDC, int n);
protected:
    //{{AFX_MSG(CRygWnd)
    afx_msg void OnPaint();
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    //}}AFX_MSG
    afx_msg LRESULT OnSetState(WPARAM wParam, LPARAM lParam);
    afx_msg LRESULT OnGetState(WPARAM wParam, LPARAM lParam);
    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations
//  immediately before the previous line.
#endif //
!defined(AFX_RYGWND_H__1AA889D5_9788_11D0_BED2_00C04FC2A0C2__INCLUDED_)
```

RYGWND.CPP

```
// RygWnd.cpp : implementation file
//

#include "stdafx.h"
#include "mymfc22D.h"
#include "RygWnd.h"

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

LRESULT CALLBACK AFX_EXPORT
    RygWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    CWnd* pWnd;

    pWnd = CWnd::FromHandlePermanent(hWnd);
    if (pWnd == NULL) {
        // Assume that client created a CRygWnd window
        pWnd = new CRygWnd();
        pWnd->Attach(hWnd);
    }
    ASSERT(pWnd->m_hWnd == hWnd);
    ASSERT(pWnd == CWnd::FromHandlePermanent(hWnd));
    LRESULT lResult = AfxCallWndProc(pWnd, hWnd, message, wParam,
lParam);
    return lResult;
}

/////////////////////////////////////////////////////////////////////////////
// CRygWnd

// static data members
CRect  CRygWnd::s_rect(-500, 1000, 500, -1000); // outer rectangle
CPoint CRygWnd::s_point(300, 300); // rounded corners
CRect  CRygWnd::s_rColor[] = {CRect(-250, 800, 250, 300),
                              CRect(-250, 250, 250, -250),
                              CRect(-250, -300, 250, -800)};
CBrush CRygWnd::s_bColor[] = {RGB(192, 192, 192),
                              RGB(0xFF, 0x00, 0x00),
                              RGB(0xFF, 0xFF, 0x00),
```

```
                                RGB(0x00, 0xFF, 0x00)};

BOOL CRygWnd::RegisterWndClass(HINSTANCE hInstance) // static member
                                                    //  function
{
    WNDCLASS wc;
    wc.lpszClassName = "RYG"; // matches class name in client
    wc.hInstance = hInstance;
    wc.lpfnWndProc = RygWndProc;
    wc.hCursor = ::LoadCursor(NULL, IDC_ARROW);
    wc.hIcon = 0;
    wc.lpszMenuName = NULL;
    wc.hbrBackground = (HBRUSH) ::GetStockObject(LTGRAY_BRUSH);
    wc.style = CS_GLOBALCLASS;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    return (::RegisterClass(&wc) != 0);
}

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

CRygWnd::CRygWnd()
{
    m_nState = 0;
    TRACE("CRygWnd constructor\n");
}

CRygWnd::~CRygWnd()
{
    TRACE("CRygWnd destructor\n");
}


BEGIN_MESSAGE_MAP(CRygWnd, CWnd)
    //{{AFX_MSG_MAP(CRygWnd)
    ON_WM_PAINT()
    ON_WM_LBUTTONDOWN()
    //}}AFX_MSG_MAP
    ON_MESSAGE(RYG_SETSTATE, OnSetState)
    ON_MESSAGE(RYG_GETSTATE, OnGetState)
END_MESSAGE_MAP()
void CRygWnd::SetMapping(CDC* pDC)
{
    CRect clientRect;
    GetClientRect(clientRect);
    pDC->SetMapMode(MM_ISOTROPIC);
    pDC->SetWindowExt(1000, 2000);
    pDC->SetViewportExt(clientRect.right, -clientRect.bottom);
    pDC->SetViewportOrg(clientRect.right / 2, clientRect.bottom / 2);
}

void CRygWnd::UpdateColor(CDC* pDC, int n)
{
    if (m_nState == n + 1) {
        pDC->SelectObject(&s_bColor[n+1]);
    }
    else {
        pDC->SelectObject(&s_bColor[0]);
    }
    pDC->Ellipse(s_rColor[n]);
}

/////////////////////////////////////////////////////////////////////////
// CRygWnd message handlers

void CRygWnd::OnPaint()
```

```
{
    int i;
    CPaintDC dc(this); // device context for painting
    SetMapping(&dc);
    dc.SelectStockObject(DKGRAY_BRUSH);
    dc.RoundRect(s_rect, s_point);
    for (i = 0; i < 3; i++) {
        UpdateColor(&dc, i);
    }
}

void CRygWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
    // Notification code is HIWORD of wParam, 0 in this case
    GetParent()->SendMessage(WM_COMMAND, GetDlgCtrlID(),
        (LONG) GetSafeHwnd()); // 0
}
void CRygWnd::PostNcDestroy()
{
    TRACE("CRygWnd::PostNcDestroy\n");
    delete this; // CWnd::PostNcDestroy does nothing
}

LRESULT CRygWnd::OnSetState(WPARAM wParam, LPARAM lParam)
{
    TRACE("CRygWnd::SetState, wParam = %d\n", wParam);
    m_nState = (int) wParam;
    Invalidate(FALSE);
    return 0L;
}

LRESULT CRygWnd::OnGetState(WPARAM wParam, LPARAM lParam)
{
    TRACE("CRygWnd::GetState\n");
    return m_nState;
}
```

Listing 11: The `CRygWnd` class listing.

As usual, build the program and copy the **mymfc22D.dll** into the Windows system directory.

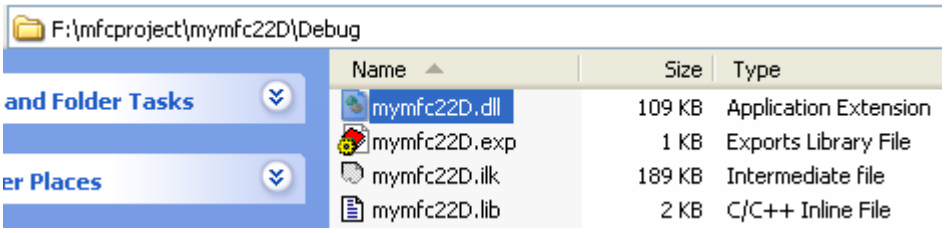

Figure 37: The generated DLL file of MYMFC22D program.

Revising the Updated MYMFC22B Example: Adding Code to Test `mymfc22D.dll`

The MYMFC22B program already links to the MYMFC22A and **MYMFC22C DLLs**. Now you'll revise the project to implicitly link to the MYMFC22D custom control.

Here are the steps for updating the MYMFC22B example:

Add a new dialog resource and class to \mfcproject**mymfc22B**.Use the dialog editor to create the `IDD_MYMFC22D` template with a custom control with child window ID `IDC_RYG`, as shown here.

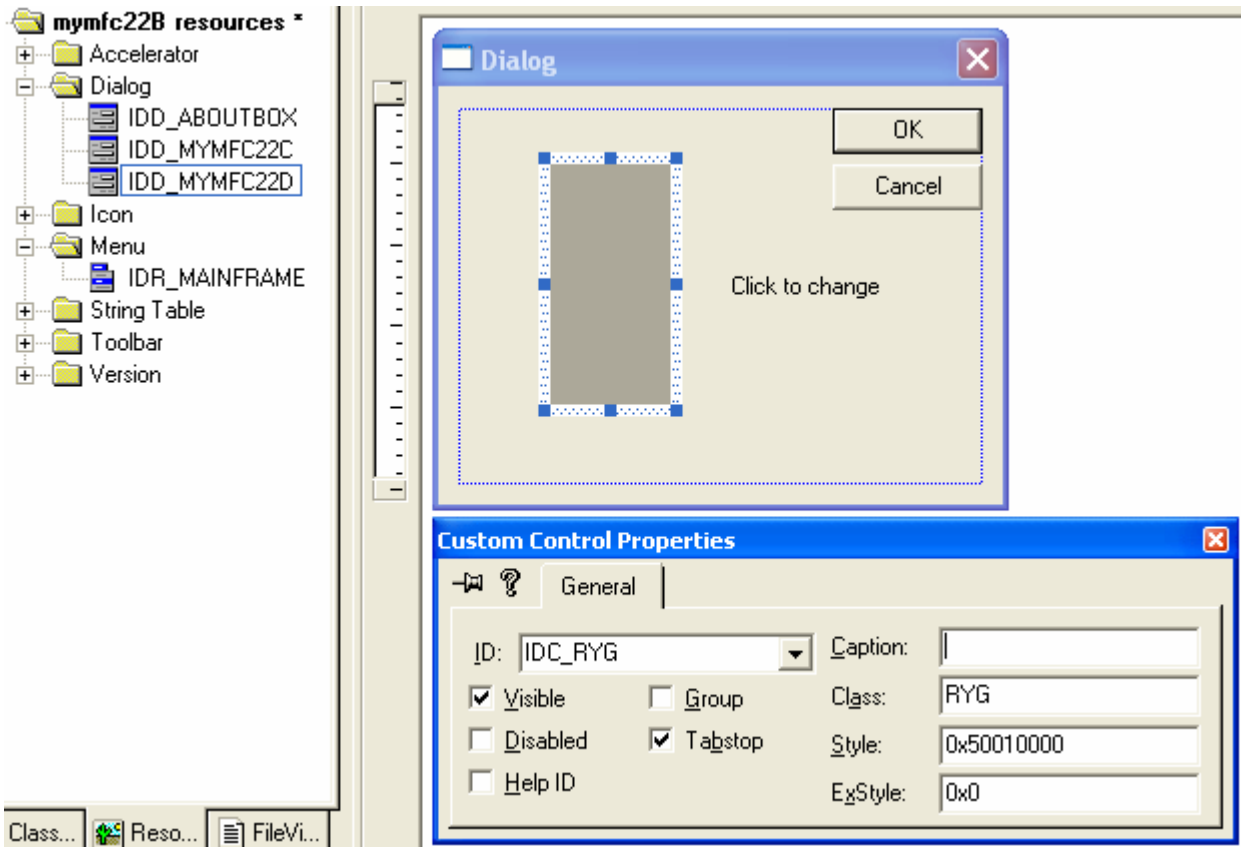

Figure 38: Adding a new dialog resource and modifying its properties.

Specify `RYG` as the window class name of the custom control, as shown.

Then use ClassWizard to generate a class `CTest22dDialog`, derived from `CDialog`.

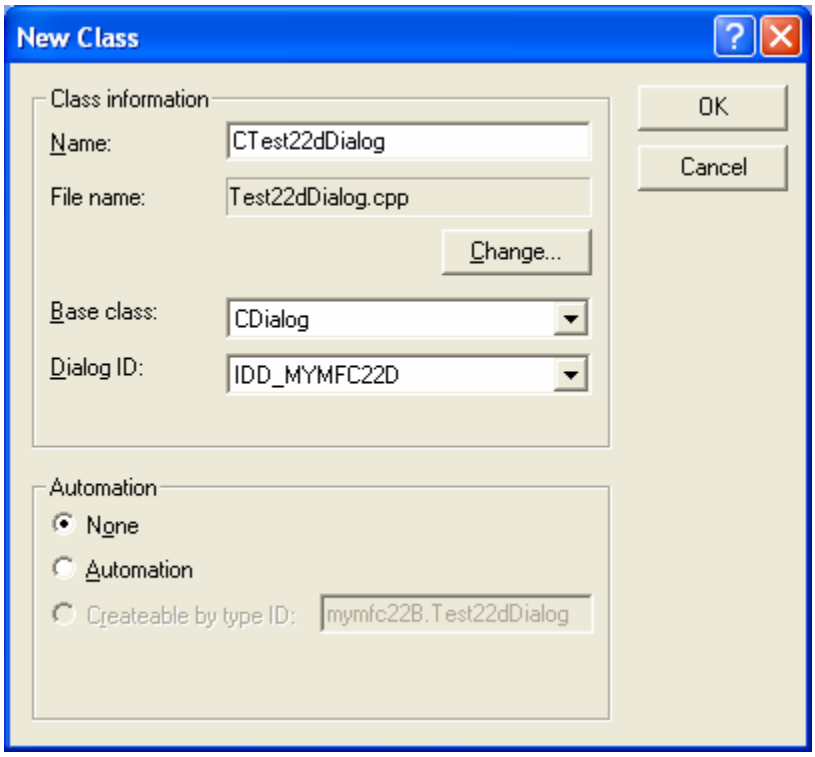

Figure 39: Creating and adding new class `CTest22dDialog`, derived from `CDialog`.

Edit the `Test22dDialog.h` file. Add the following private data member (added manually instead of using ClassView):

```
	enum { OFF, RED, YELLOW, GREEN };
```

```
enum { OFF, RED, YELLOW, GREEN } m_nState;

/////////////////////////////////////////////
// CTest22dDialog dialog

class CTest22dDialog : public CDialog
{
// Construction
```

Listing 12.

and `m_nState` variable in **Test22dDialog.h**,

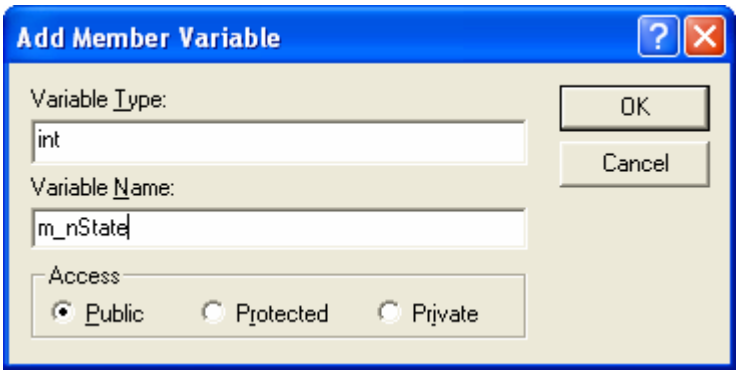

Figure 40: Adding member variable through the ClassView.

```
class CTest22dDialog : public CDialog
{
// Construction
public:
    int m_nState;
    CTest22dDialog(CWnd* pParent = NULL);
```

Listing 13.

Also add the following import and user-defined message IDs:

```
	extern "C" __declspec(dllimport) void Mymfc22DEntry(); // dummy function
	#define RYG_SETSTATE WM_USER + 0
	#define RYG_GETSTATE WM_USER + 1
```

```
enum { OFF, RED, YELLOW, GREEN } m_nState;

extern "C" __declspec(dllimport) void Mymfc22DEntry(); // dummy function
#define RYG_SETSTATE WM_USER + 0
#define RYG_GETSTATE WM_USER + 1

/////////////////////////////////////////////////////////////////////////////
// CTest22dDialog dialog
```

Listing 14.

Edit the constructor in **Test22dDialog.cpp** to initialize the state data member. Add the following code:

```
CTest22dDialog::CTest22dDialog(CWnd* pParent /*=NULL*/)
    : CDialog(CTest22dDialog::IDD, pParent)
{
    //{{AFX_DATA_INIT(CTest22dDialog)
      // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
    m_nState = OFF;
    Mymfc22DEntry(); // Make sure DLL gets loaded
}
```

```
// CTest22dDialog dialog

CTest22dDialog::CTest22dDialog(CWnd* pParent /*=NULL*/)
    : CDialog(CTest22dDialog::IDD, pParent)
{
    //{{AFX_DATA_INIT(CTest22dDialog)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
    m_nState = OFF;
    Mymfc22DEntry(); // Make sure DLL gets loaded
}
```

Listing 15.

Map the control's clicked notification message. You can't use ClassWizard here, so you must add the message map entry and handler function in the **Test22dDialog.cpp** file, as shown here:

```
ON_CONTROL(0, IDC_RYG, OnClickedRyg) // Notification code is 0
```

```
BEGIN_MESSAGE_MAP(CTest22dDialog, CDialog)
    //{{AFX_MSG_MAP(CTest22dDialog)
        // NOTE: the ClassWizard will add message map macros here
    //}}AFX_MSG_MAP
    ON_CONTROL(0, IDC_RYG, OnClickedRyg) // Notification code is 0
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CTest22dDialog message handlers
```

Listing 16.

```
void CTest22dDialog::OnClickedRyg()
{
    switch(m_nState) {
    case OFF:
        m_nState = RED;
        break;
    case RED:
        m_nState = YELLOW;
        break;
    case YELLOW:
        m_nState = GREEN;
        break;
    case GREEN:
        m_nState = OFF;
        break;
    }
    GetDlgItem(IDC_RYG)->SendMessage(RYG_SETSTATE, m_nState);
    return;
}
```

```
// CTest22dDialog message handlers

void CTest22dDialog::OnClickedRyg()
{
    switch(m_nState) {
    case OFF:
        m_nState = RED;
        break;
    case RED:
        m_nState = YELLOW;
        break;
    case YELLOW:
        m_nState = GREEN;
        break;
    case GREEN:
        m_nState = OFF;
        break;
    }
    GetDlgItem(IDC_RYG)->SendMessage(RYG_SETSTATE, m_nState);
    return;
}
```

Listing 17.

When the dialog gets the clicked notification message, it sends the `RYG_SETSTATE` message back to the control in order to change the color. Don't forget to add this prototype in the **Test22dDialog.h** file:

```
   afx_msg void OnClickedRyg();
```

```
// Implementation
protected:

    // Generated message map func
    //{{AFX_MSG(CTest22dDialog)
        // NOTE: the ClassWizard
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
    afx_msg void OnClickedRyg();
};
```

Listing 18.

Integrate the `CTest22dDialog` class into the MYMFC22B application. You'll need to add a second item on the **Test** menu, a Mymfc22D DLL option with ID `ID_TEST_MYMFC22DDLL`.

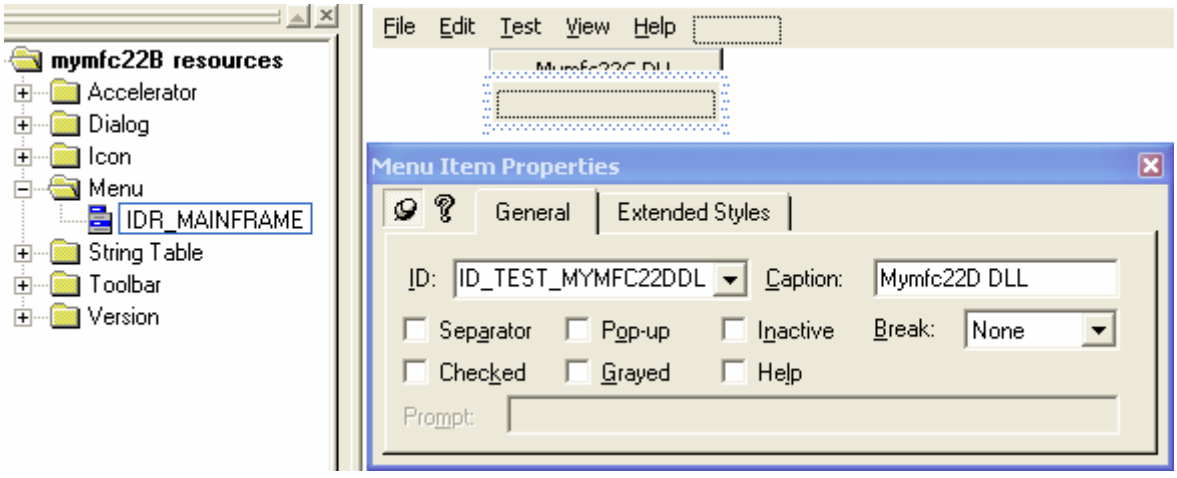

Figure 41: Adding a second item on the **Test** menu, a Mymfc22D DLL option with ID `ID_TEST_MYMFC22DDLL`.

Use ClassWizard to map this option to a member function in the `CMymfc22BView` class, and then code the handler in **Mymfc22BView.cpp** as follows:

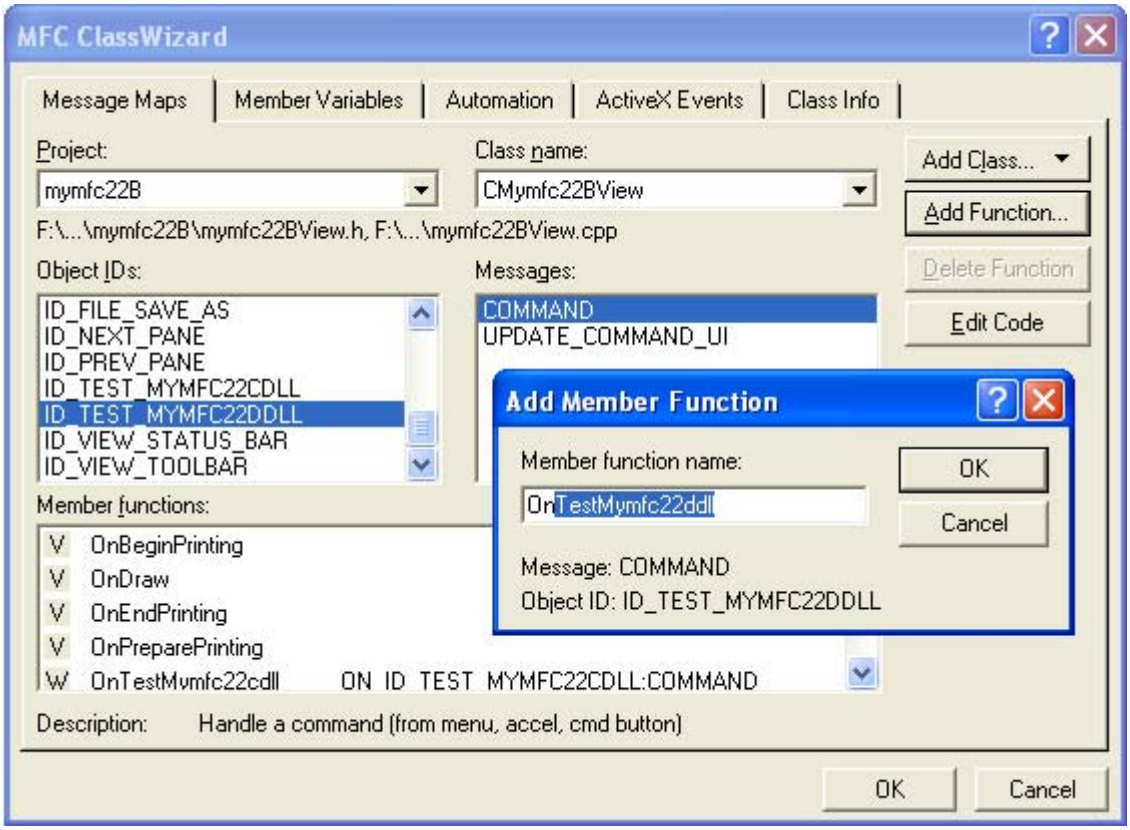

Figure 42: Mapping `ID_TEST_MYMFC22DDLL` to a member function in the `CMymfc22BView` class.

```
void CMymfc22BView::OnTestMymfc22Ddll()
{
    CTest22dDialog dlg;
    dlg.DoModal();
}
```

```
void CMymfc22BView::OnTestMymfc22ddll()
{
    // TODO: Add your command handler code here
    CTest22dDialog dlg;
    dlg.DoModal();
}
```

Listing 19.

Of course, you'll have to add the following `#include` line to **Mymfc22BView.cpp**:

```
#include "Test22dDialog.h"
```

```
//

#include "stdafx.h"
#include "mymfc22B.h"

#include "mymfc22BDoc.h"
#include "mymfc22BView.h"

#include "Test22cDialog.h"
#include "Test22dDialog.h"
```

Listing 20.

Add the MYMFC22D import library to the linker's input library list. Choose **Settings** from Visual C++'s **Project** menu, and then add **\mfcproject\mymfc22D\Debug\mymfc22D.lib** to the **Object/Library modules** control on the **Link** page. With this addition, the program should implicitly link to all three DLLs.

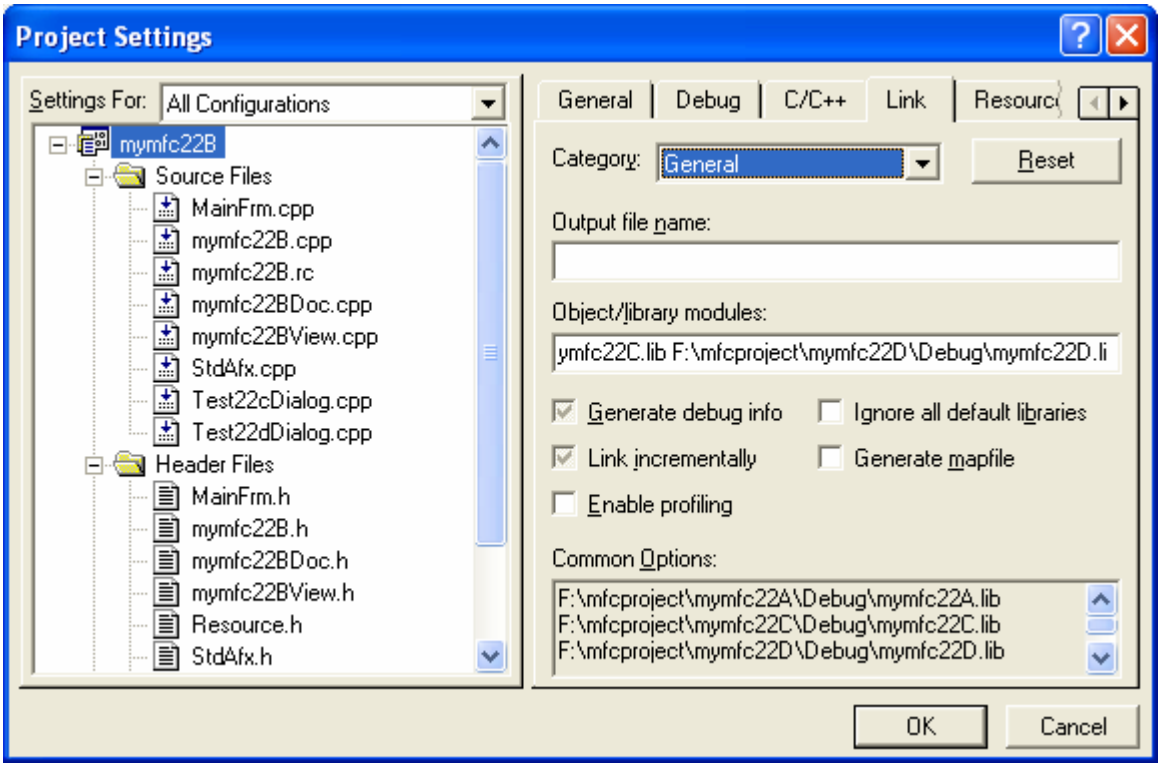

Figure 43: Adding the `mymfc22D.lib` (import) library to the linker's input library list, we have three DLL libraries here.

Build and test the updated MYMFC22B application. Choose **Mymfc22D DLL** from the **Test** menu. Try clicking the traffic light with the left mouse button. The traffic-light color should change. The result of clicking the traffic light several times is shown here.

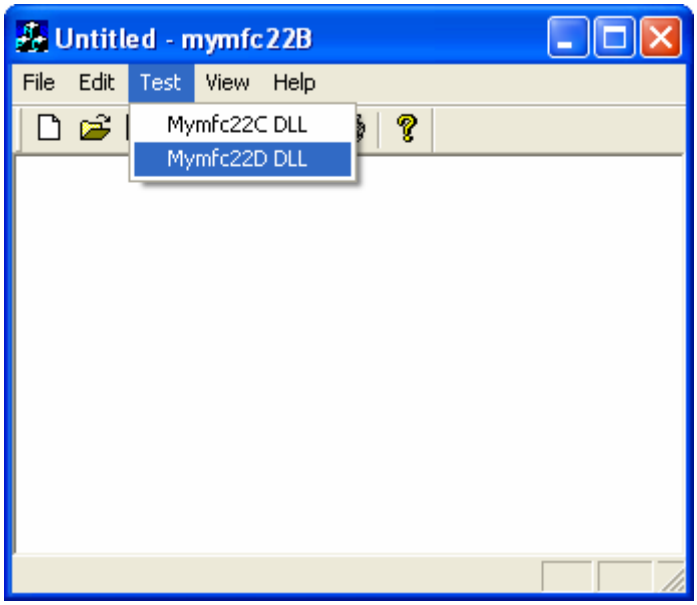

Figure 44: MYMFC22B program output with three DLLs linking.

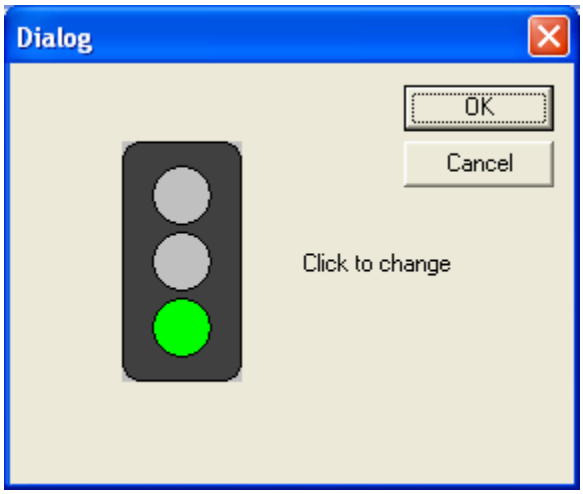

Figure 45: MYMFC22B program output using the third type of DLL.

Further reading and digging:

1. Win32 dynamic link library, DLL.
2. MSDN MFC 6.0 class library online documentation - used throughout this Tutorial.
3. MSDN MFC 7.0 class library online documentation - used in .Net framework and also backward compatible with 6.0 class library
4. MSDN Library
5. Windows data type.
6. Win32 programming Tutorial.
7. The best of C/C++, MFC, Windows and other related books.
8. Unicode and Multibyte character set: Story and program examples.