

Programming the Microsoft Internet Information Server

Program examples compiled using Visual C++ 6.0 compiler on Windows XP Pro machine with Service Pack 2 and some Figure screen snapshots have been taken on Windows 2000 server. The Internet Information Services version is IIS 4.x/5.x/6.x on Windows 2000 Server SP 4 and Windows XP Pro SP 2. The Internet Explorer is 6.x. Topics and sub topics for this tutorial are listed below. A complete information about IIS installation, configuration and [testing a Web site is dumped HERE](#) and how to [setup FTP server also included HERE](#). Both Web and FTP servers were done on Windows 2000 Server SP4. Don't forget to read Tenouk's small [disclaimer](#).

Index:

- Introduction
- IIS Alternatives
- Microsoft IIS
- Installing and Controlling IIS
- Running Internet Service Manager
- IIS Security
- IIS Directories
- IIS Logging
- Testing IIS
- ISAPI Server Extensions
- Common Gateway Interface and ISAPI
- A Simple ISAPI Server Extension GET Request
- HTML Forms: GET vs. POST
- Writing an ISAPI Server Extension DLL
- The Simplest Idea
- The MFC ISAPI Server Extension Classes
- CHttpServer
- CHttpServerContext
- CHtmlStream
- A Practical ISAPI Server Extension: myex34a.dll
- MYEX34A From Scratch
- The Story
- The First Step: Getting the Order
- Optional Parameters
- The Second Step: Processing the Confirmation
- Building and Testing myex34a.dll
- Debugging the MYEX34A DLL
- ISAPI Database Access
- Using HTTP Cookies to Link Transactions
- How Cookies Work
- How an ISAPI Server Extension Processes Cookies
- Problems with Cookies
- WWW Authentication
- Basic Authentication
- Windows NT Challenge/Response Authentication
- The Secure Sockets Layer
- ISAPI Filters
- Writing an ISAPI Filter DLL
- The MFC ISAPI Filter Classes
- CHttpFilter
- CHttpFilterContext

A Sample ISAPI Filter: myex34b.dll, myex34c.exe
MYEX34B From Scratch: ISAPI Filter
Choosing the Notification
Sending Transaction Data to the Display Program
MYEX34C From Scratch
CMainFrame Class
CMyex34cApp Class
CMyex34cDoc Class
CMyex34cView Class
The Story of The Display Program
Building and Testing the MYEX34B ISAPI Filter

Introduction

In [Module 32](#), you used a "homemade" Web based on the Winsock APIs. In this module, you'll learn how to use and extend Microsoft Internet Information Server (IIS), which is bundled with Windows Server. IIS is actually three separate servers, one for HTTP (for the World Wide Web), one for FTP, and one for gopher. This module tells you how to write HTTP server extensions using the Microsoft **IIS application programming interface** (ISAPI) that is part of Microsoft **ActiveX technology**. You'll examine two kinds of extensions: an **ISAPI server extension** and an **ISAPI filter**, both of which are DLLs. An ISAPI server extension can perform Internet business transactions such as order entry. An ISAPI filter intercepts data traveling to and from the server and thus can perform specialized logging and other tasks.

IIS Alternatives

The exercises in this module assume that you have Windows Server and IIS. If you are running Windows Workstation, you can use Peer Web Services, which supports fewer connections and doesn't allow virtual servers. If you are running Microsoft Windows 95 or Windows 98, you can use Personal Web Server, which is packaged with Microsoft FrontPage. Internet Information Server, Peer Web Services, and Personal Web Server can all use ISAPI extension DLLs. See your server's documentation for operating details.

Microsoft IIS

Microsoft IIS is a high-performance Internet/intranet server that takes advantage of Windows NT features such as I/O completion ports, the Win32 function `TransmitFile()`, file-handle caching, and CPU scaling for threads.

Installing and Controlling IIS

When you install Windows Server, you are given the option of installing IIS. If you selected IIS at setup, the server will be running whenever Windows NT is running. IIS is a special kind of Win32 program called a **service** (actually three services, WWW, HTTP, and gopher, in one program called **inetinfo.exe**), which won't appear in the taskbar. You can control IIS from the **Services** icon in the **Control Panel** or through the **Administrative Tools** menu, but you'll probably want to use the Internet Service Manager program instead.

Running Internet Service Manager

You can run **Internet Service Manager** from the **Administrative Tools** menu that's accessible on the **Start Program** menu.

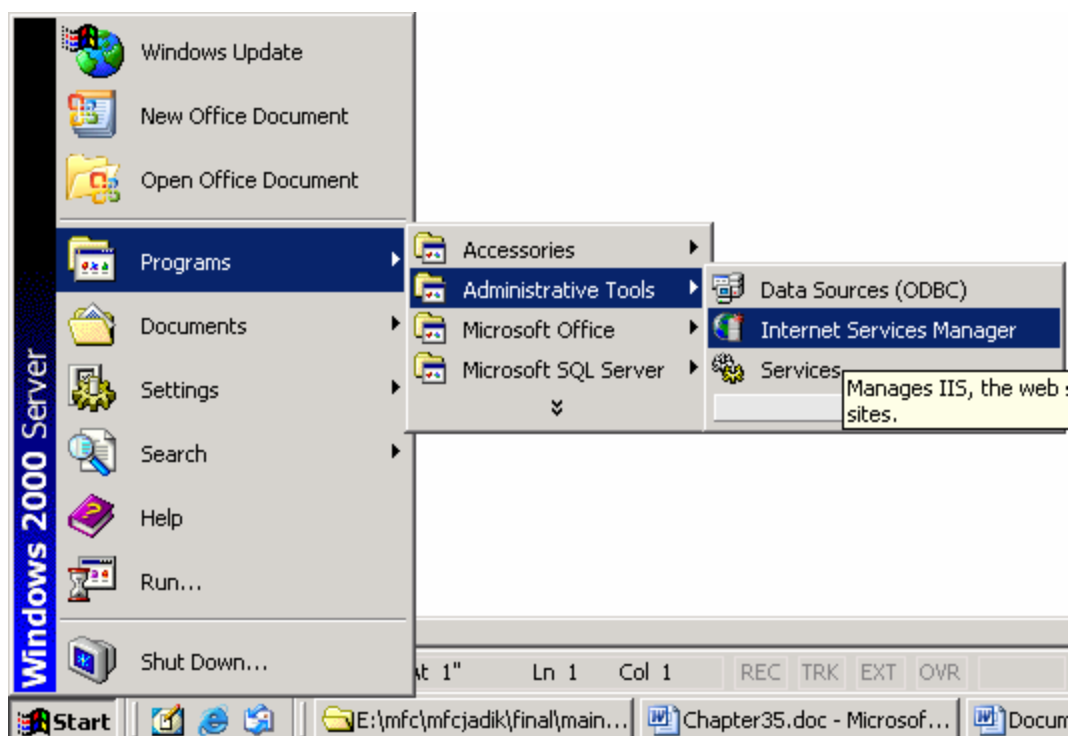


Figure 1: Internet Service Manager menu on Windows 2000 Server.

You can also run an HTML-based version of Internet Service Manager remotely from a browser. That version allows you to change service parameters, but it won't let you turn services on and off. Figure 2 shows the Microsoft Internet Service Manager screen with the World Wide Web (WWW) and FTP are running. You can select a service by clicking on its icon at the left. The triangle and square buttons on the toolbar of the screen allow you to turn the selected service on or off.

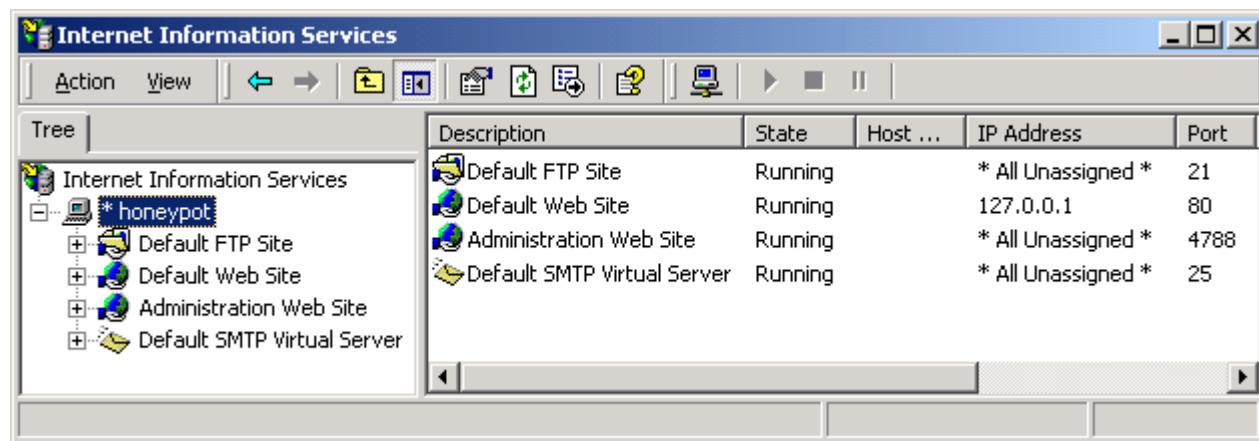


Figure 2: The Microsoft Internet Service Manager screen.

IIS Security

After you double-click on the WWW service icon of the Microsoft Internet Service Manager screen or using the context menu as shown below, you'll see a property sheet.

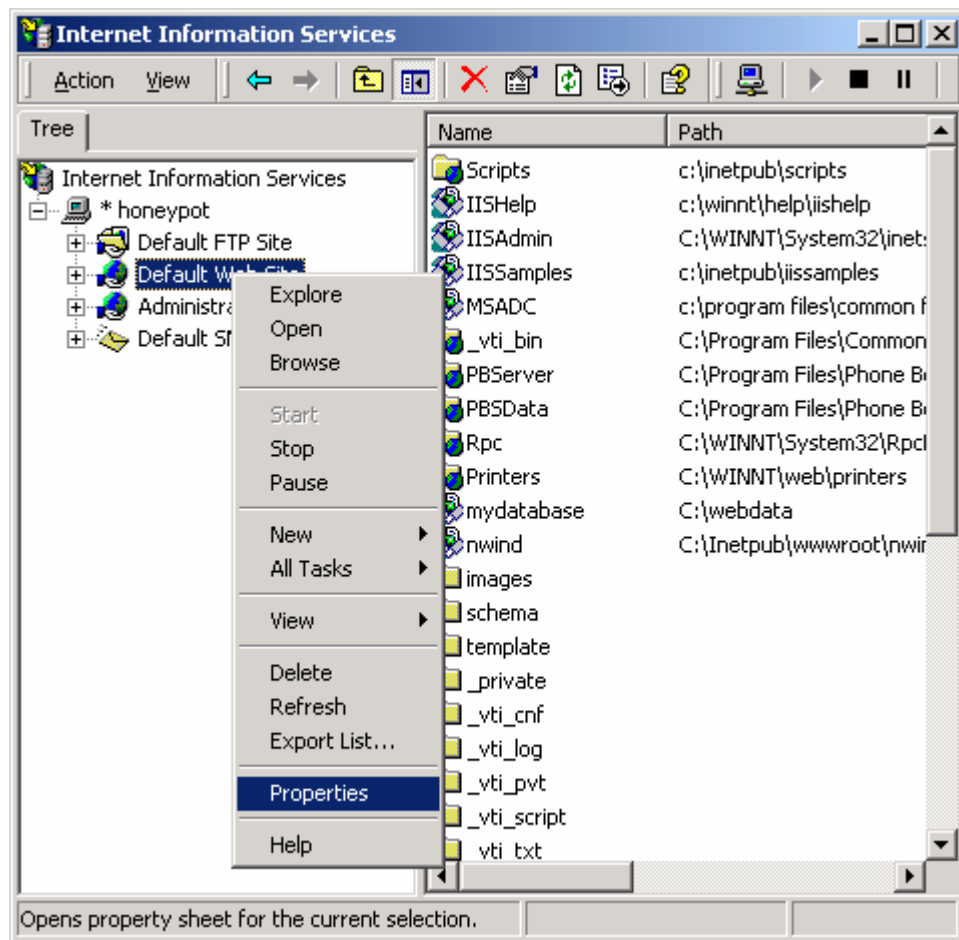


Figure 3: Invoking the web server properties pages through the context menu.

The **Properties** page lets you configure many aspects of IIS. When a client browser requests a file, the server impersonates a local user for the duration of the request and that user name determines which files the client can access. Which local user does the server impersonate? Most often, it's the one you see in the Username field, shown in Figure 4 and can be accessed by selecting the **Directory Security** tab and click the **Edit** button.

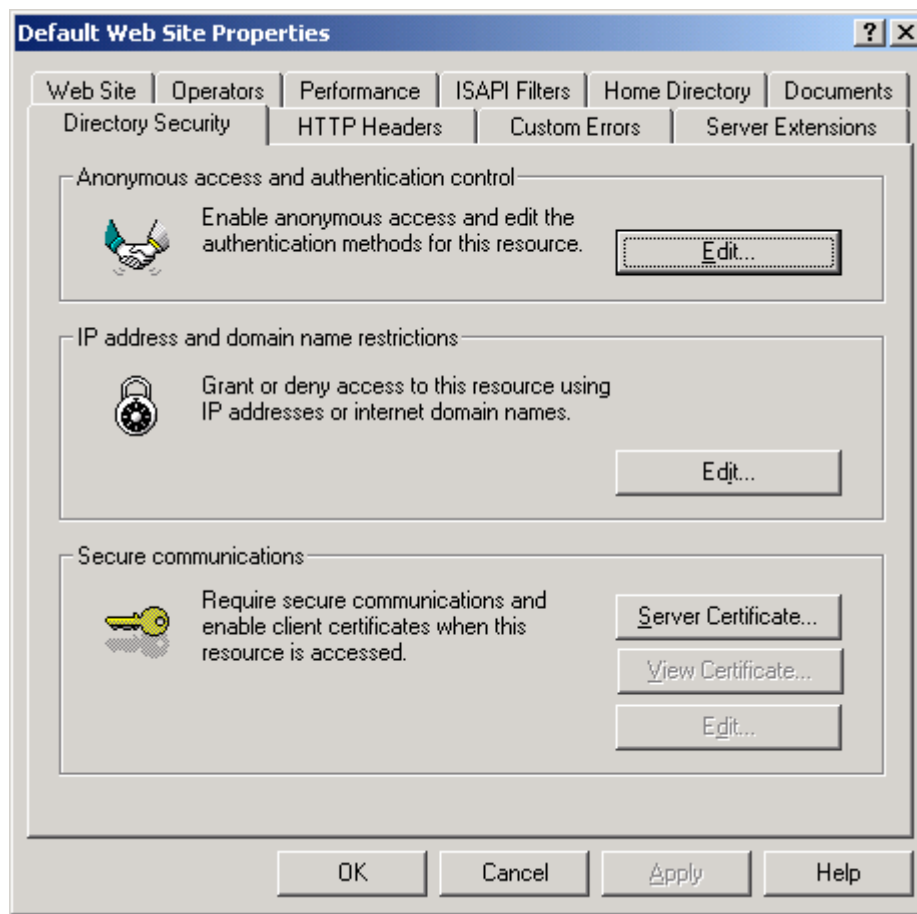


Figure 4: The WWW Service Properties screen.

Click the **Edit** button.

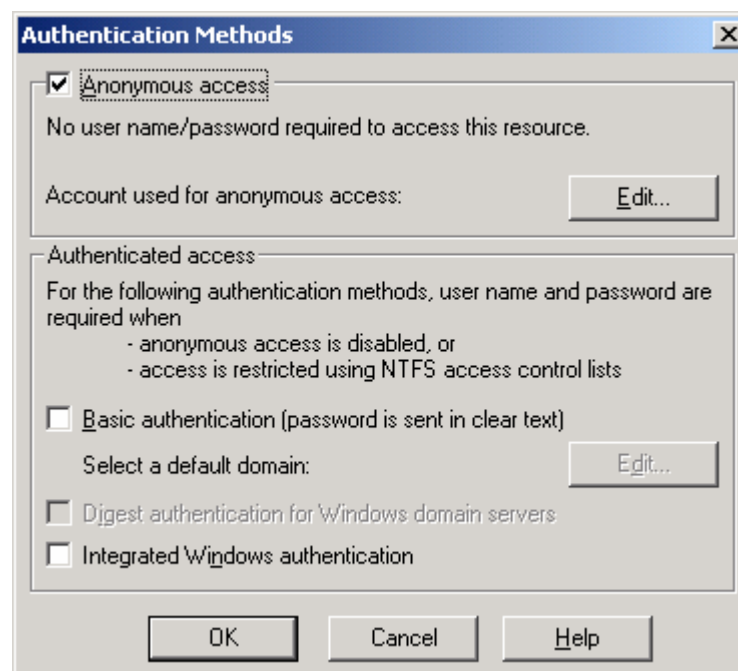


Figure 5: Web authentication method, an anonymous access.

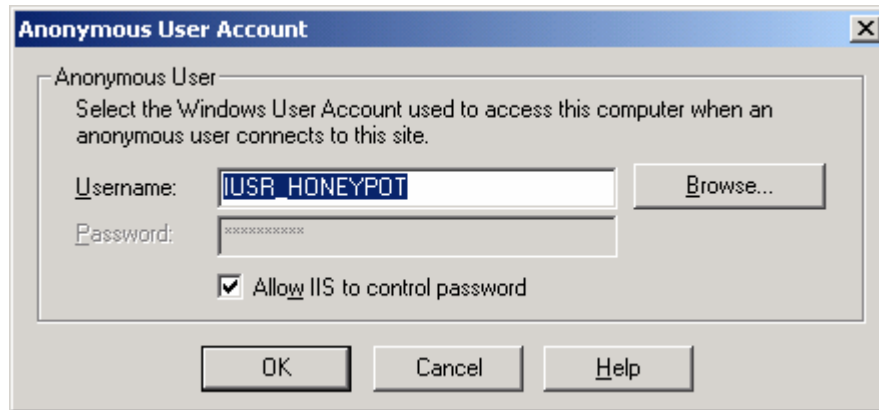


Figure 6: Anonymous account used by web user to access the web server (web site).

Most Web page visitors don't supply a user name and password, so they are considered anonymous users. Those users have the same rights they would have if they had logged on to your server locally as IUSR_MYMACHINENAME. That means that IUSR_MYMACHINENAME must appear in the list of users that is displayed when you run (from the Administrative Tools menu) **Active directory Users and Computers**, and the passwords must match. The IIS Setup program normally defines this anonymous user for you. You can define your own WWW anonymous user name, but you must be sure that the entry on the Properties page matches the entry in the computer's (or Windows NT domain's) user list. Note also the **Authenticated access** options. For the time being, stick to the **Anonymous Access** option only, which means that all Web users are logged on as IUSR_MYMACHINENAME. Later in this module, we'll explain Windows NT Challenge/Response.

IIS Directories

Remember SlowSoft's Web site from [Module 32](#)? If you requested the URL <http://slowsoft.com/newproducts.html>, the **newproducts.html** file would be displayed from the slowsoft.com home directory. Each server needs a **home directory**, even if that directory contains only subdirectories. The home directory does not need to be the server computer's root directory, however. As shown in Figure 7, the default WWW home directory is really c:\Inetpub\wwwroot, so clients read the disk file \wwwroot\newproducts.html. The content of the web site also can be on the shared directory/drive located on other server(s) or the web user can be redirected to other web addresses (URLs).

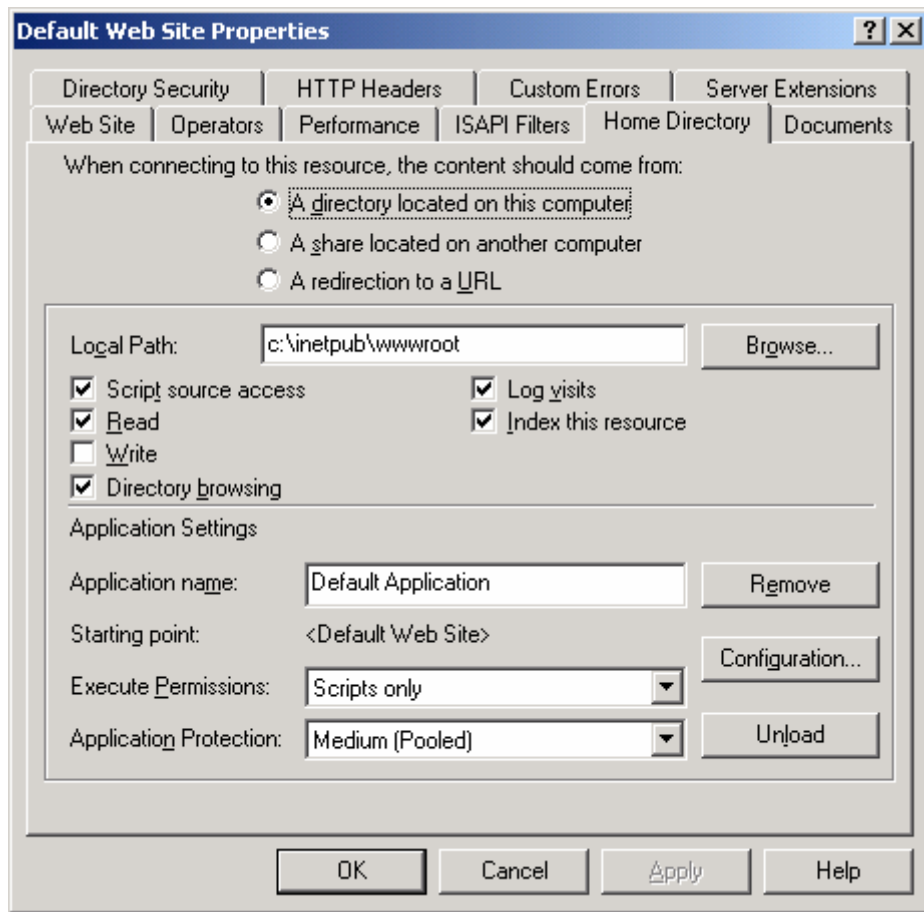


Figure 7: The **wwwroot**, default WWW home directory screen.

If you select the **Directory Browsing** option of the **Directories** page on the service property screen, browser clients can see a hypertext list of files in the server's directory instead.

Your server could get by with a home directory only, but the IIS virtual directory feature might be useful. The virtual directory can be created using **New Virtual Directory** context menu shown below. It is used to map the IIS directory to the real physical directory.

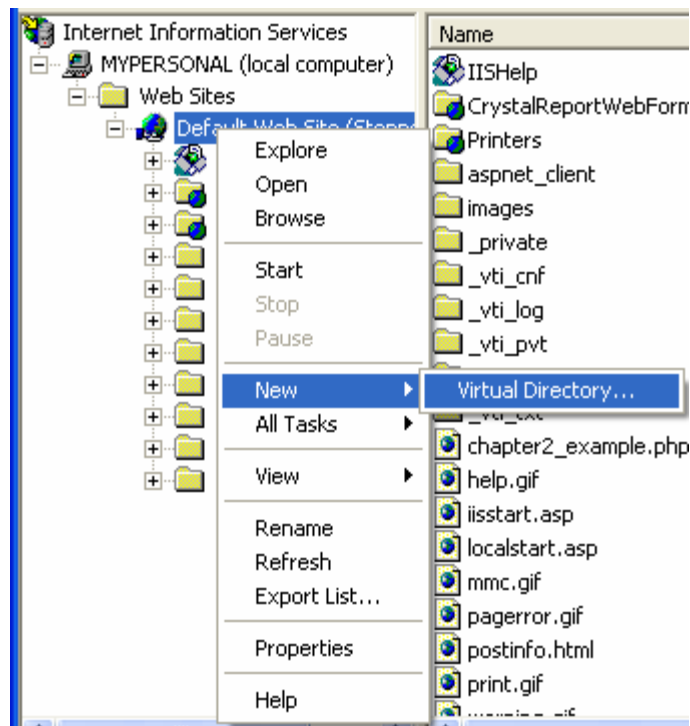


Figure 8: Creating virtual directory that can be used by web server to map to the real, physical directory.

If your computer was configured for multiple IP addresses, IIS would allow you to define virtual Web servers. Each virtual server would have its own home directory (and virtual directories) attached to a specified IP address, making it appear as though you had several server computers. Unfortunately, the IIS Web server listens on all the computer's IP addresses, so you can't run IIS simultaneously with the MYEX34A server with both listening on port 80. As described in [Module 32](#), browsers can issue a blind request. As Figure 9 shows, Internet Service Manager lets you specify the file that a blind request selects, usually **Default.htm**. From Figure 9, Default.htm will be used, if not available/found, the second document, Default.asp will be used and so on. If none of the default document available, browser returns a typical error page not found. However if the default document option is disabled, the start page conventionally named index.html, if available, will be used automatically.

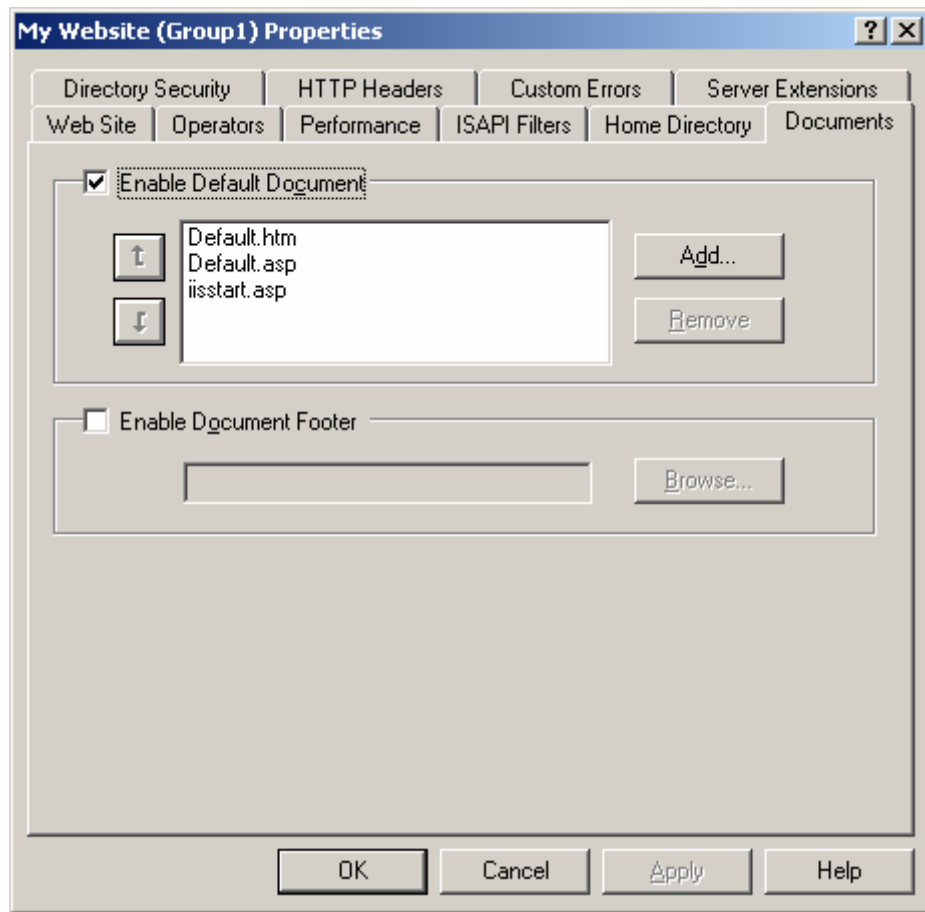


Figure 9: **Default.htm**, the default document of the web site. You can add or remove those default documents.

IIS Logging

IIS is capable of making log entries for all connections. You control logging from the Internet Service Manager's Logging property page. You can specify text log files, or you can specify logging to an SQL/ODBC database. Log entries consist of date, time, client IP address, file requested, query string, and so forth. You can access the logging settings by selecting the **Enable Logging** and clicking the **Properties** button as shown below.

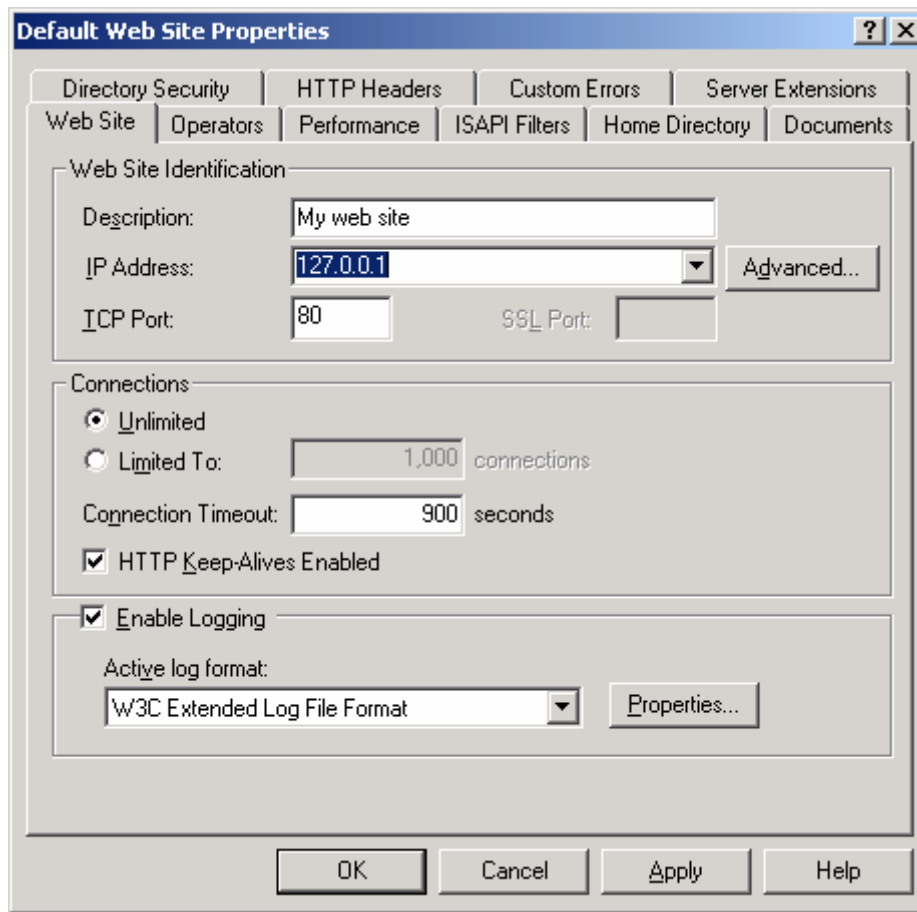


Figure 10: Web server with logging enabled.

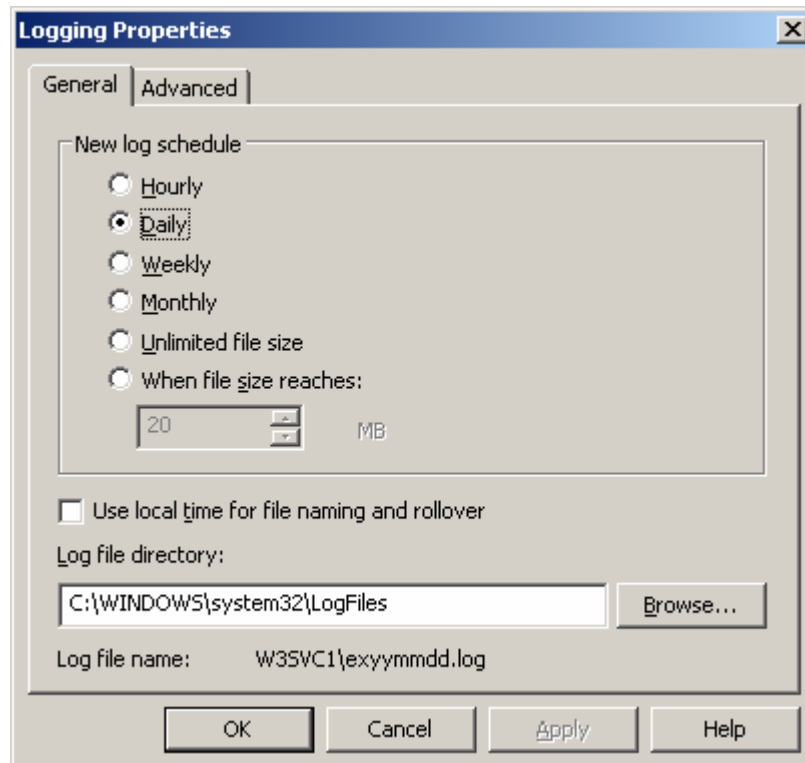


Figure 11: General settings for web server logging.

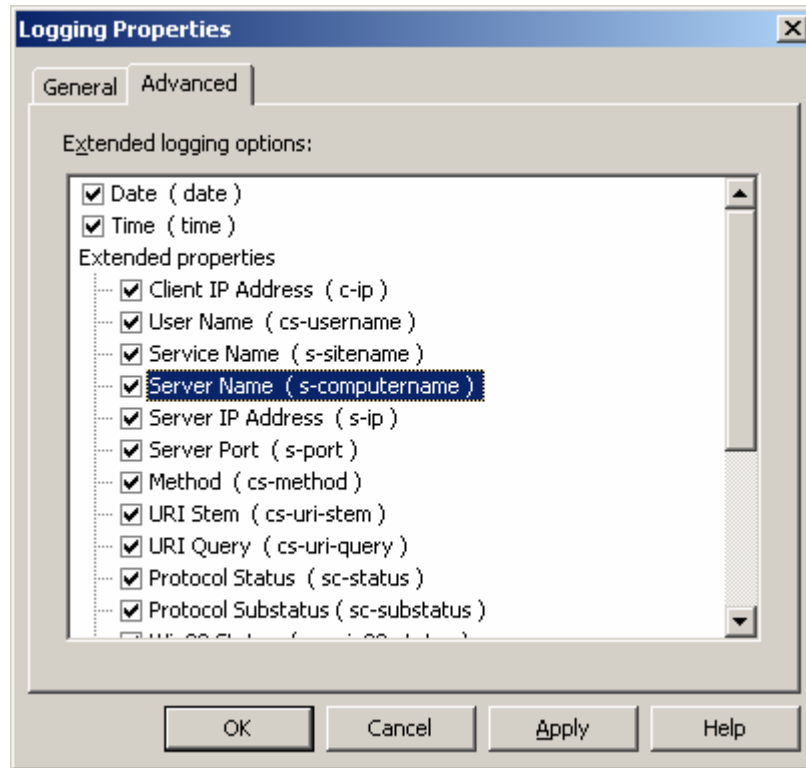


Figure 12: Advanced settings for web server logging.

Testing IIS

It's easy to test IIS with a browser or with any of the MYEX34A clients. Just make sure that IIS is running and that the MYEX34A server is not running. The default IIS home directory is **c:\Inetpub\wwwroot**, and some HTML files are installed there. If you're running a single machine, you can use the **localhost** (or 127.0.0.1) host name. For a network, use a name from the **Hosts** file. If you can't access the server from a remote machine, run ping to make sure the network is configured correctly. Don't try to build and run ISAPI DLLs until you have successfully tested IIS on your computer. The complete steps to configure IIS and test your web site using IIS 3.x/5.x on Windows 2000 Server are dumped [here](#).

ISAPI Server Extensions

An ISAPI server extension is a program (implemented as a DLL loaded by IIS) that runs in response to a GET or POST request from a client program (browser). The **browser can pass parameters to the program**, which are often values that the browser user types into edit controls, selects from list boxes, and so forth. The ISAPI server extension typically sends back HTML code based on those parameter values. You'll better understand this process when you see an example. Take note that nowadays better scripting languages with more functionalities and security features, such as Active Server Page (ASP), PHP, Java Server Page (JSP) etc. were used to replace and complement ISAPI server extension. What we are concern in this tutorial is more on how the whole system works.

Common Gateway Interface and ISAPI

Internet server programs were first developed for UNIX computers, so the standards were in place long before Microsoft introduced IIS. The [Common Gateway Interface](#) (CGI) standard, actually part of HTTP, evolved as a way for browser programs to interact with **scripts** or **separate executable programs** running on the server. Without altering the HTTP/CGI specifications, Microsoft designed IIS to allow any browser to load and run a **server DLL**. DLLs are part of the IIS process and thus are faster than scripts that might need to load separate executable programs. Because of your

experience, you'll probably find it easier to write an ISAPI DLL in C++ than to write a script in PERL, the standard Web scripting language for servers.

CGI shifts the programming burden to the server. Using CGI parameters, the browser sends small amounts of information to the server computer, and the server can do absolutely anything with this information, including access a database, generate images, and control peripheral devices. The server sends a file (HTML or otherwise) back to the browser. The file can be read from the server's disk, or it can be generated by the program. No ActiveX controls or Java applets are necessary, and the browser **can be running on any type of computer**.

A Simple ISAPI Server Extension GET Request

Suppose an HTML file contains the following tag (there is already **C:\Inetpub\Scripts** subdirectory when you install IIS and we will use this subdirectory to put all the DLL file of the program examples):

```
<a href="scripts/mymaps.dll?State=Idaho">Idaho Weather Map</a><p>
```

When the user clicks on **Idaho Weather Map**, the browser sends the server a CGI GET request like this:

```
GET scripts/mymaps.dll?State=Idaho HTTP/1.0
```

IIS then loads **mymaps.dll** from its scripts (virtual) directory, calls a default function (often named `Default()`), and passes it the `State` parameter `Idaho`. The DLL then goes to work generating a JPG file containing the up-to-the-minute satellite weather map for Idaho and sends it to the client. If `mymaps.dll` had more than one function, the tag could specify the function name like this:

```
<a href="scripts/mymaps.dll?GetMap?State=Idaho&Res=5">Idaho Weather  
Map</a><p>
```

In this case, the function `GetMap()` is called with two parameters, `State` and `Res`. You'll soon learn how to write an ISAPI server similar to `mymaps.dll`, but first you'll need to understand HTML forms, because you don't often see CGI GET requests by themselves.

HTML Forms: GET vs. POST

In the HTML code for the simple CGI GET request above, the state name was hard-coded in the tag. Why not let the user select the state from a drop-down list? For that, you need a form, and here's a simple one that can do the job. This file named **weathermap.html**

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">  
<HTML>  
  
<HEAD>  
<TITLE>Weathermap HTML Form</TITLE>  
</HEAD>  
  
<BODY>  
  
<H1></H1>  
<CENTER><B><FONT color="#FF0000" face="Verdana">Welcome to the Satellite  
Weathermap Service</FONT></B></CENTER>  
<H1></H1>  
<FORM action="scripts/mymaps.dll?GetMap" method="POST">  
  <P>Select your state: <SELECT name="State">  
    <OPTION>Alabama</OPTION>  
    <OPTION>Alaska</OPTION>  
    <OPTION>Idaho</OPTION>  
    <OPTION>Washington</OPTION>  
  </SELECT> </P>  
  <P><INPUT type="submit"><INPUT type="reset"> </P>
```

```
</FORM>
</BODY>
</HTML>
```

Listing 1.

If you looked at this HTML file with a browser, you would see the form shown in Figure 22.

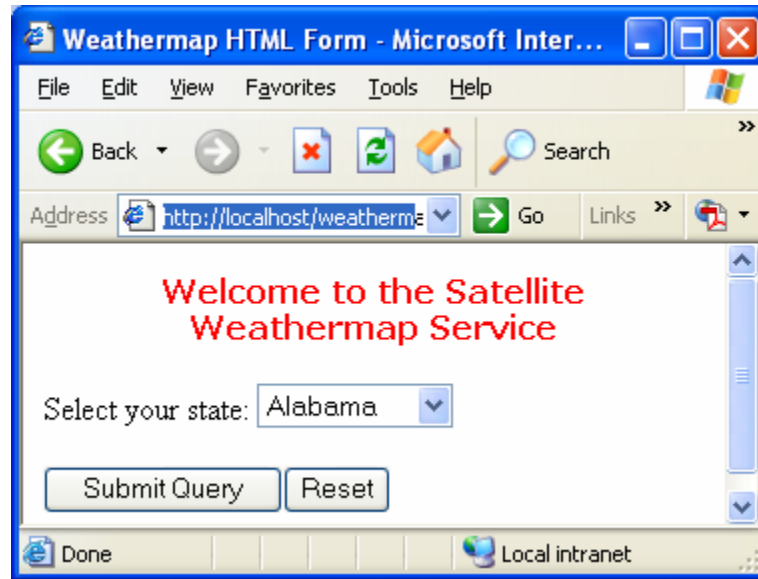


Figure 13: The Weathermap HTML form window.

The select tag provides the state name from a list of four states, and the all-important "submit" input tag displays the pushbutton that sends the form data to the server in the form of a CGI GET request that looks like this:

```
GET scripts/mymaps.dll?GetMap?State=Idaho HTTP/1.0
(various request headers)
(blank line)
```

Unfortunately, some early versions of the Netscape browser omit the function name in form-originated GET requests, giving you two choices: provide only a default function in your ISAPI DLL, and use the POST method inside a form instead of the GET method. If you want to use the POST option, change one HTML line in the form above to the following:

```
<form action="scripts/mymaps.dll?GetMap" method=POST>
```

Now here's what the browser sends to the server:

```
POST scripts/mymaps.dll?GetMap
(various request headers)
(blank line)

State=Idaho
```

Note that the parameter value is in the last line instead of in the request line. ISAPI DLLs are usually stored in a separate virtual directory on the server because these DLLs must have execute permission but do not need read permission. Clicking the **Edit Properties** button shown in Figure 7 will allow you to access these permissions from the **Internet Service Manager**, or you can double-click on a directory to change its properties.

Writing an ISAPI Server Extension DLL

Visual C++ gives you a quick start for writing ISAPI server extensions. Just select **ISAPI Extension Wizard** from the **Projects** list. After you click the **OK** button, your first screen looks like Figure 24.

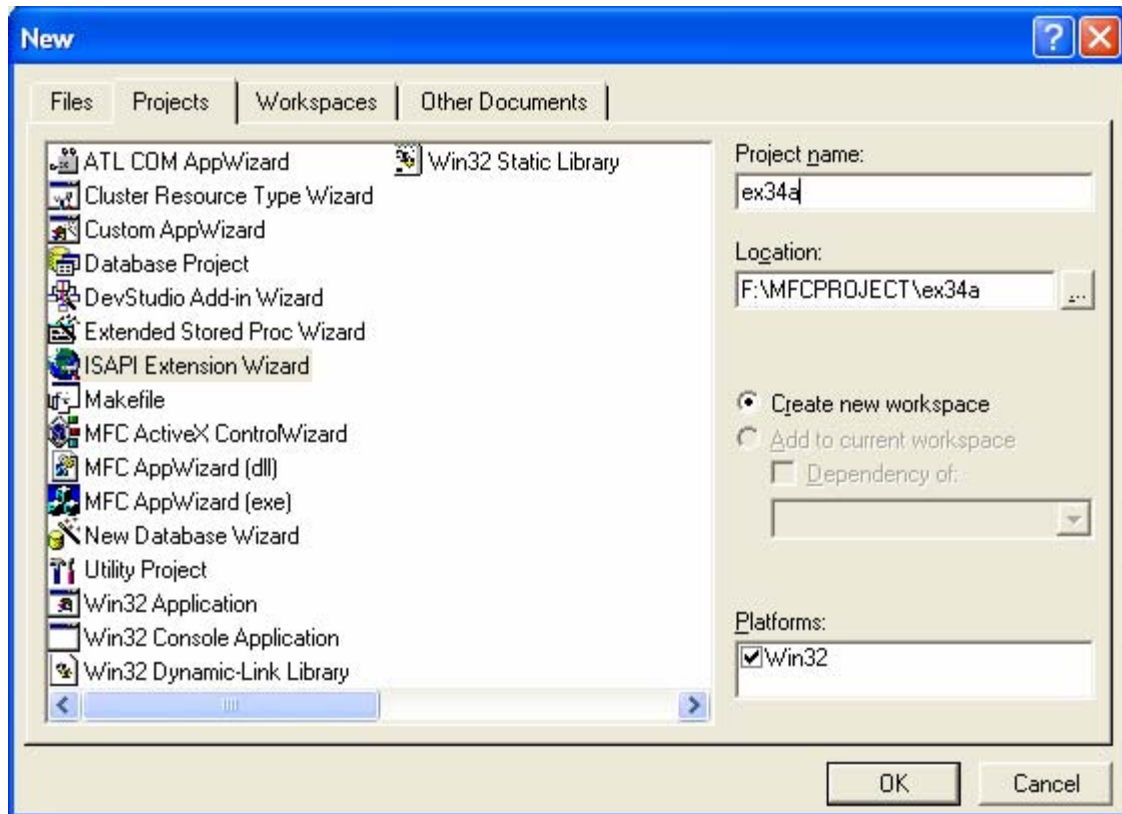


Figure 14: ISAP extension new project dialog.

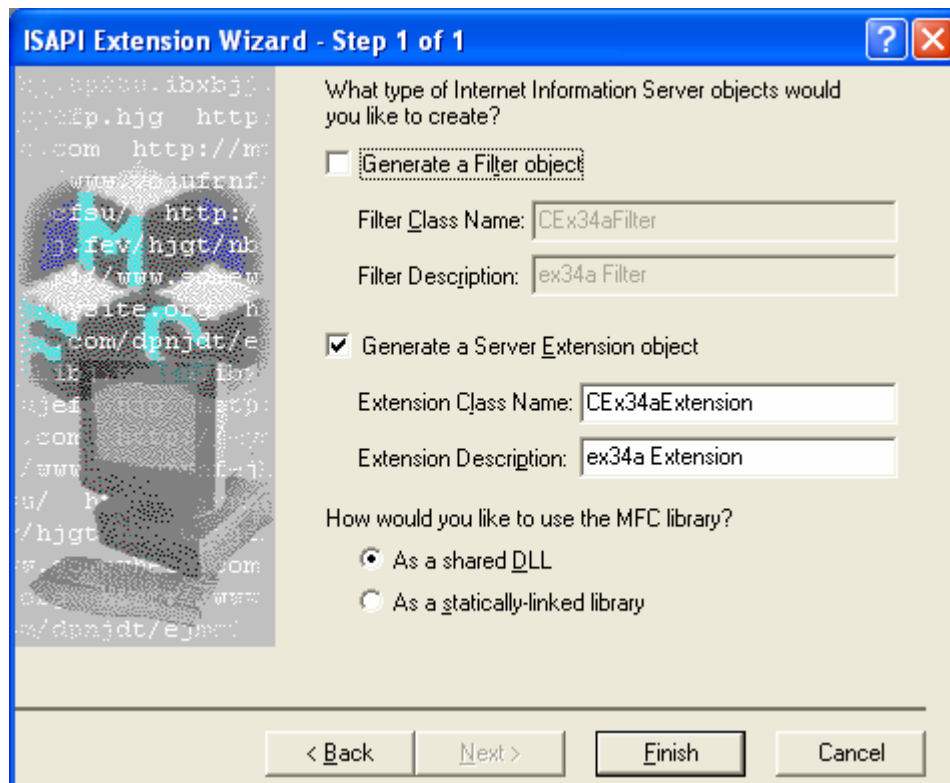


Figure 15: The Step 1 of 1 page of the ISAPI Extension Wizard.

Check the **Generate A Server Extension Object** box, and you've got a do-nothing DLL project with a class derived from the MFC `CHttpServer` class and a `Default()` member function. Now it's time for a little programming. You must write your ISAPI functions as members of the derived `CHttpServer` class, and you must write parse map macros to link them to IIS. There's no "parse map wizard," so you have to do some coding. It's okay to use the `Default()` function, but you need a `GetMap()` function too. First insert these lines into the wizard-generated parse map:

```
ON_PARSE_COMMAND(GetMap, CWeatherExtension, ITS_PSTR)
ON_PARSE_COMMAND_PARAMS("State")
```

Then write the `GetMap()` function:

```
void CWeatherExtension::GetMap(CHttpServerContext* pCtxt, LPCTSTR
pstrState)
{
    StartContent(pCtxt);
    WriteTitle(pCtxt);
    *pCtxt << "Visualize a weather map for the state of ";
    *pCtxt << pstrState;
    EndContent(pCtxt);
}
```

This function doesn't actually generate the weather map (what did you expect?), but it does display the selected state name in a custom HTML file. The `CHttpServer::StartContent` and `CHttpServer::EndContent` functions write the HTML boilerplate, and `CHttpServer::WriteTitle` calls the virtual `CHttpServer::GetTitle` function, which you need to override:

```
LPCTSTR CWeatherExtension::GetTitle() const
{
    return "Your custom weather map"; // for browser's title window
}
```

}

The MFC `CHttpServerContext` class has an overloaded `<<` operator, which you use to put text in the HTML file you're building. Behind the scenes, an attached object of the MFC class `CHTMLStream` represents the output to the server's socket.

The Simplest Idea

The following are the steps for the previous story. Here, we just to get some idea because **mymaps.dll** does nothing useful. Just follow the shown steps.

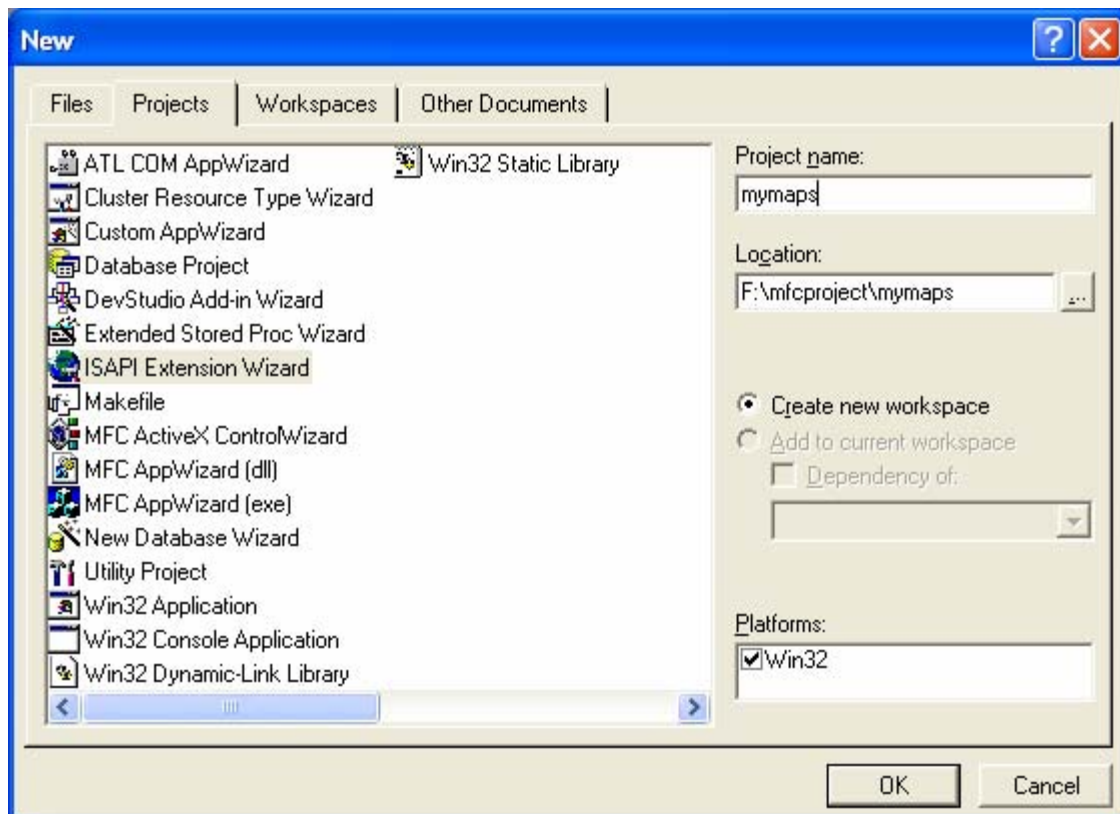


Figure 16: MYMAPS – ISAPI extension new project dialog.

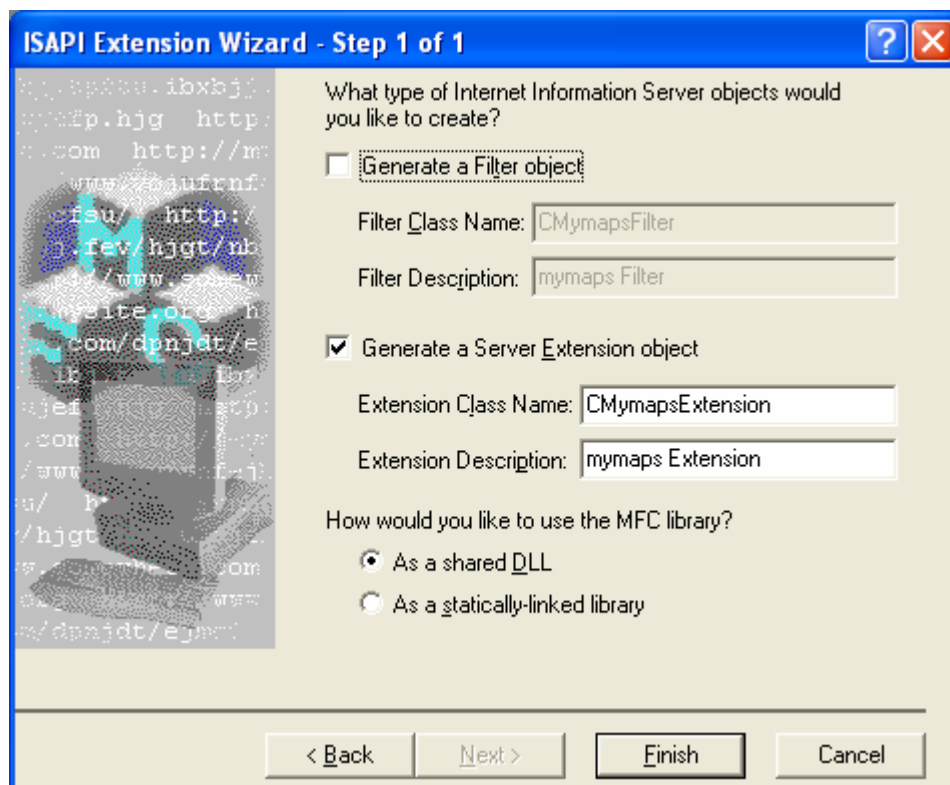


Figure 17: MYMAPS - Step 1 of 1 ISAPI Extension Wizard.

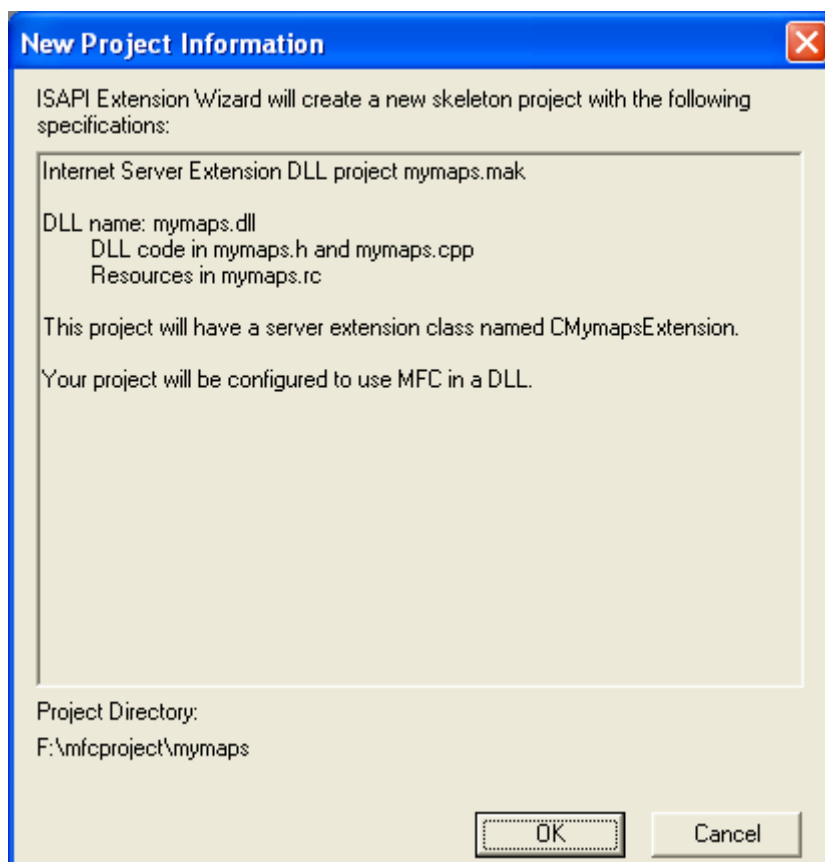


Figure 18: MYMAPS project summary.

Using ClassWizard, insert `GetTitle()` message handler.

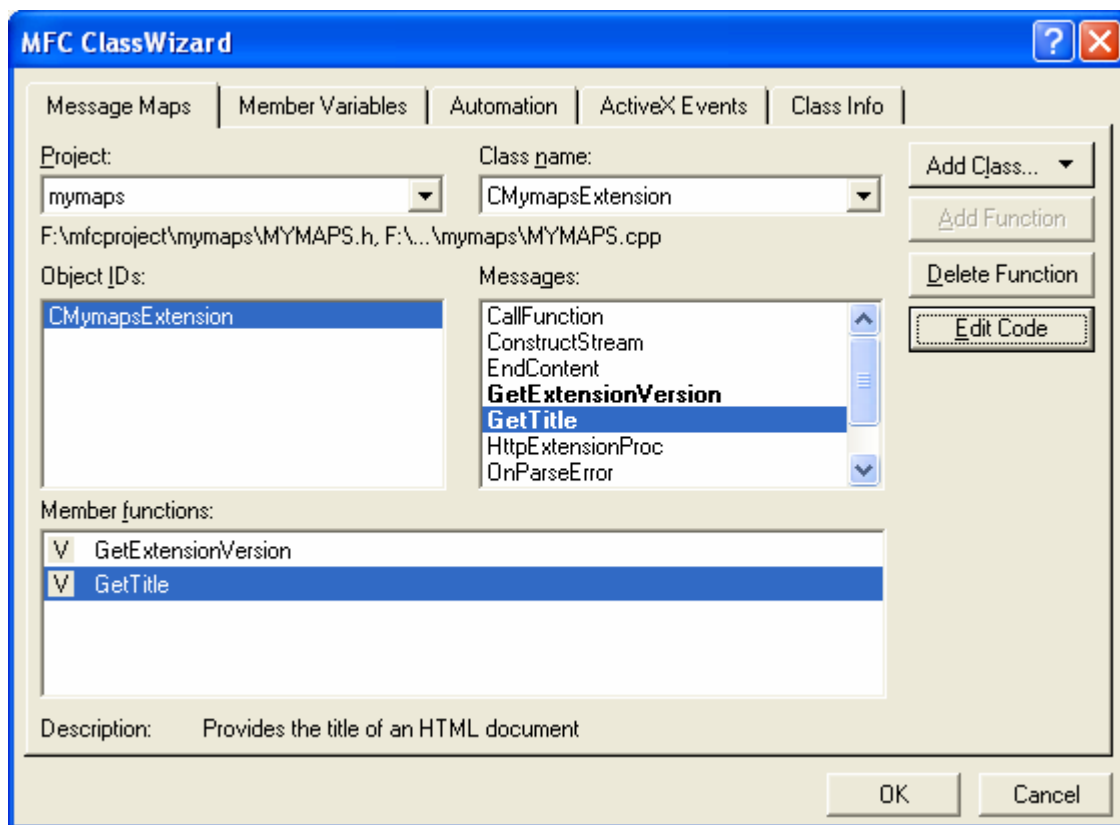


Figure 19: Adding `GetTitle()` virtual function using ClassWizard.

Edit the **mymaps.h** as shown below. The added/edited code shown in orange color.

```
#if !defined(AFX_MYMAPS_H__8290CF7E_3213_40CD_B52D_255EADB11E02__INCLUDED_)
#define AFX_MYMAPS_H__8290CF7E_3213_40CD_B52D_255EADB11E02__INCLUDED_

// MYMAPS.H - Header file for your Internet Server
// mymaps Extension

#include "resource.h"

class CMymapsExtension : public CHttpServer
{
public:
    CMymapsExtension();
    ~CMymapsExtension();

// Overrides
// ClassWizard generated virtual function overrides
// NOTE - the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code!
//{{AFX_VIRTUAL(CMymapsExtension)
public:
    virtual BOOL GetExtensionVersion(HSE_VERSION_INFO* pVer);
    virtual LPCTSTR GetTitle() const;
//}}AFX_VIRTUAL
// virtual BOOL TerminateExtension(DWORD dwFlags);

// TODO: Add handlers for your commands here.
```

```

        // For example:

        void Default(CHttpServerContext* pCtxt);
        void GetMap(CHttpServerContext* pCtxt, LPCTSTR pstrState);

        DECLARE_PARSE_MAP()

        //{AFX_MSG(CMymapsExtension)
        //}AFX_MSG
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif // !defined(AFX_MYMAPS_H__8290CF7E_3213_40CD_B52D_255EADB11E02__INCLUDED)

```

Listing 2.

And **mymaps.cpp** as shown below.

```

// MYMAPS.CPP - Implementation file for your Internet Server
//      mymaps Extension

#include "stdafx.h"
#include "mymaps.h"

////////////////////////////////////
// The one and only CWinApp object
// NOTE: You may remove this object if you alter your project to no
// longer use MFC in a DLL.

CWinApp theApp;

////////////////////////////////////
// command-parsing map

BEGIN_PARSE_MAP(CMymapsExtension, CHttpServer)
    // TODO: insert your ON_PARSE_COMMAND() and
    // ON_PARSE_COMMAND_PARAMS() here to hook up your commands.
    // For example:
    ON_PARSE_COMMAND(GetMap, CMymapsExtension, ITS_PSTR)
    ON_PARSE_COMMAND_PARAMS("State")
    ON_PARSE_COMMAND(Default, CMymapsExtension, ITS_EMPTY)
    DEFAULT_PARSE_COMMAND(Default, CMymapsExtension)
END_PARSE_MAP(CMymapsExtension)

////////////////////////////////////
// The one and only CMymapsExtension object

CMymapsExtension theExtension;

////////////////////////////////////
// CMymapsExtension implementation

CMymapsExtension::CMymapsExtension()
{
}

CMymapsExtension::~CMymapsExtension()
{
}

BOOL CMymapsExtension::GetExtensionVersion(HSE_VERSION_INFO* pVer)
{
}

```

```

        // Call default implementation for initialization
        CHttpServer::GetExtensionVersion(pVer);

        // Load description string
        TCHAR sz[HSE_MAX_EXT_DLL_NAME_LEN+1];
        ISAPIVERIFY(::LoadString(AfxGetResourceHandle(),
                                IDS_SERVER, sz, HSE_MAX_EXT_DLL_NAME_LEN));
        _tcscpy(pVer->lpszExtensionDesc, sz);
        return TRUE;
    }

// BOOL CMymapsExtension::TerminateExtension(DWORD dwFlags)
// {
//     // extension is being terminated
//     // TODO: Clean up any per-instance resources
//     return TRUE;
// }

////////////////////////////////////
// CMymapsExtension command handlers

void CMymapsExtension::Default(CHttpServerContext* pCtxt)
{
    StartContent(pCtxt);
    WriteTitle(pCtxt);

    *pCtxt << _T("This default message was produced by the Internet");
    *pCtxt << _T(" Server DLL Wizard. Edit your CMymapsExtension::Default()");
    *pCtxt << _T(" implementation to change it.\r\n");

    EndContent(pCtxt);
}

void CMymapsExtension::GetMap(CHttpServerContext* pCtxt, LPCTSTR pstrState)
{
    StartContent(pCtxt);
    WriteTitle(pCtxt);
    *pCtxt << "Visualize a weather map for the state of ";
    *pCtxt << pstrState;
    EndContent(pCtxt);
}

// Do not edit the following lines, which are needed by ClassWizard.
#ifdef 0
BEGIN_MESSAGE_MAP(CMymapsExtension, CHttpServer)
    //{{AFX_MSG_MAP(CMymapsExtension)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
#endif // 0

////////////////////////////////////
// If your extension will not use MFC, you'll need this code to make
// sure the extension objects can find the resource handle for the
// module. If you convert your extension to not be dependent on MFC,
// remove the comments around the following AfxGetResourceHandle()
// and DllMain() functions, as well as the g_hInstance global.

/****

static HINSTANCE g_hInstance;

HINSTANCE AFXISAPI AfxGetResourceHandle()
{
    return g_hInstance;
}

```

```

BOOL WINAPI DllMain(HINSTANCE hInst, ULONG ulReason,
                    LPVOID lpReserved)
{
    if (ulReason == DLL_PROCESS_ATTACH)
    {
        g_hInstance = hInst;
    }

    return TRUE;
}

****/

LPCTSTR CMymapsExtension::GetTitle() const
{
    // TODO: Add your specialized code here and/or call the base class
    return "Your custom weather map"; // for browser's title window
}

```

Listing 3.

Build the program, generating **mymaps.dll** file under the project's **Debug** directory. Copy **mymaps.dll** to **c:\inetpub\scripts** sub directory.

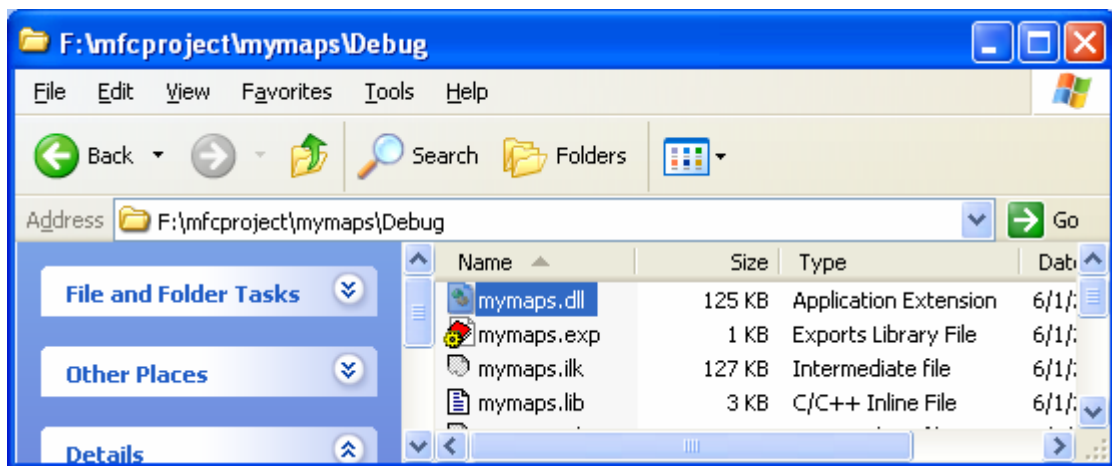


Figure 20: The generated DLL under the Debug directory.

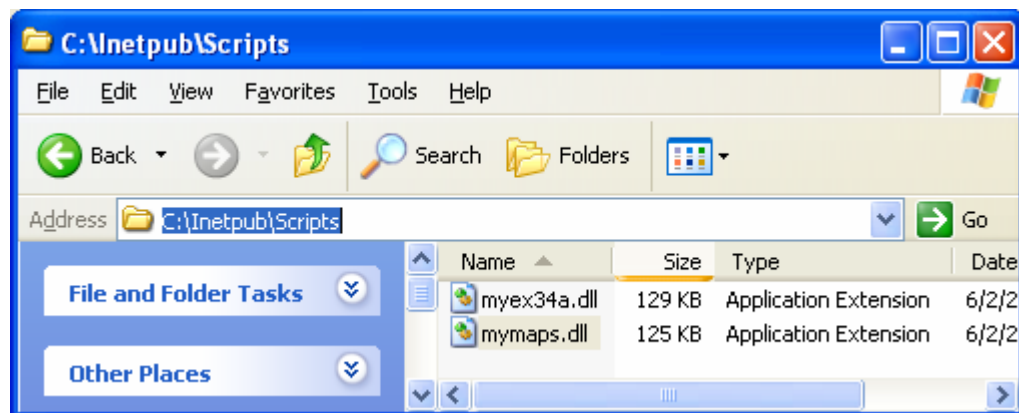


Figure 21: Copying the DLL to **scripts** directory of the default web site directory.

Verify the scripts sub directory through IIS.

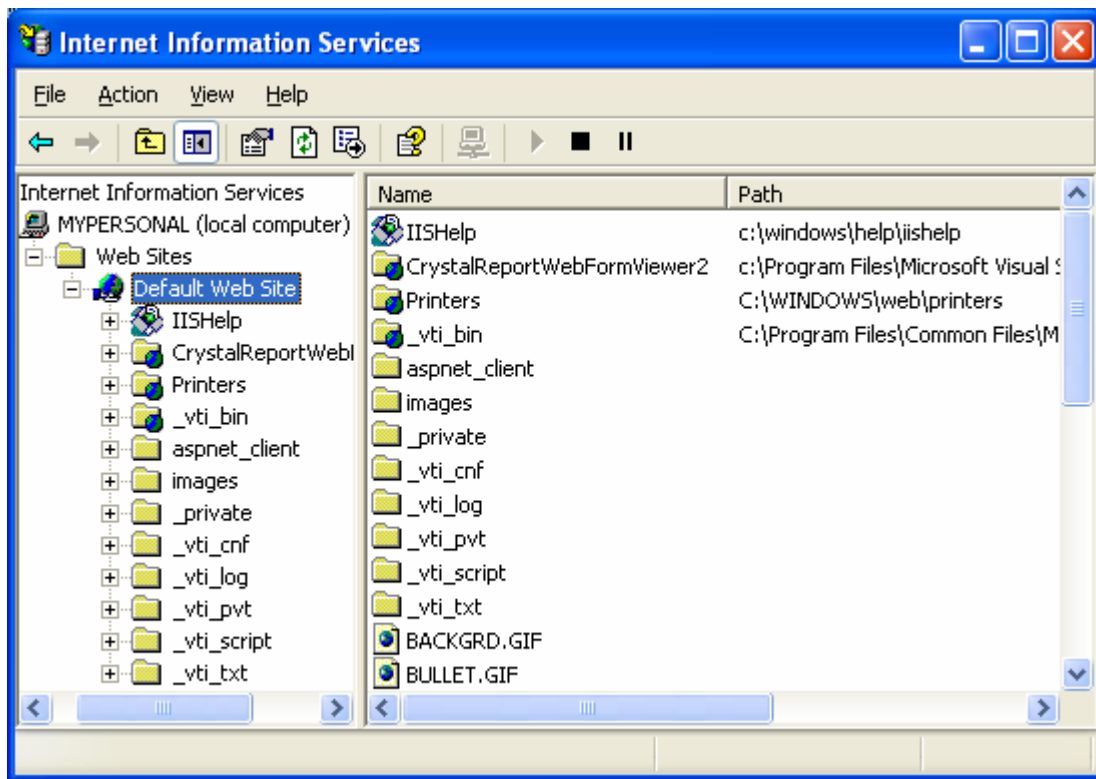


Figure 22: The **scripts** sub directory nowhere to be seen.

If the scripts sub directory cannot be seen, we need to create a virtual directory to map to the physical script subdirectory. Select the **Web Site** directory and right click. Select **New Virtual Directory** context menu.

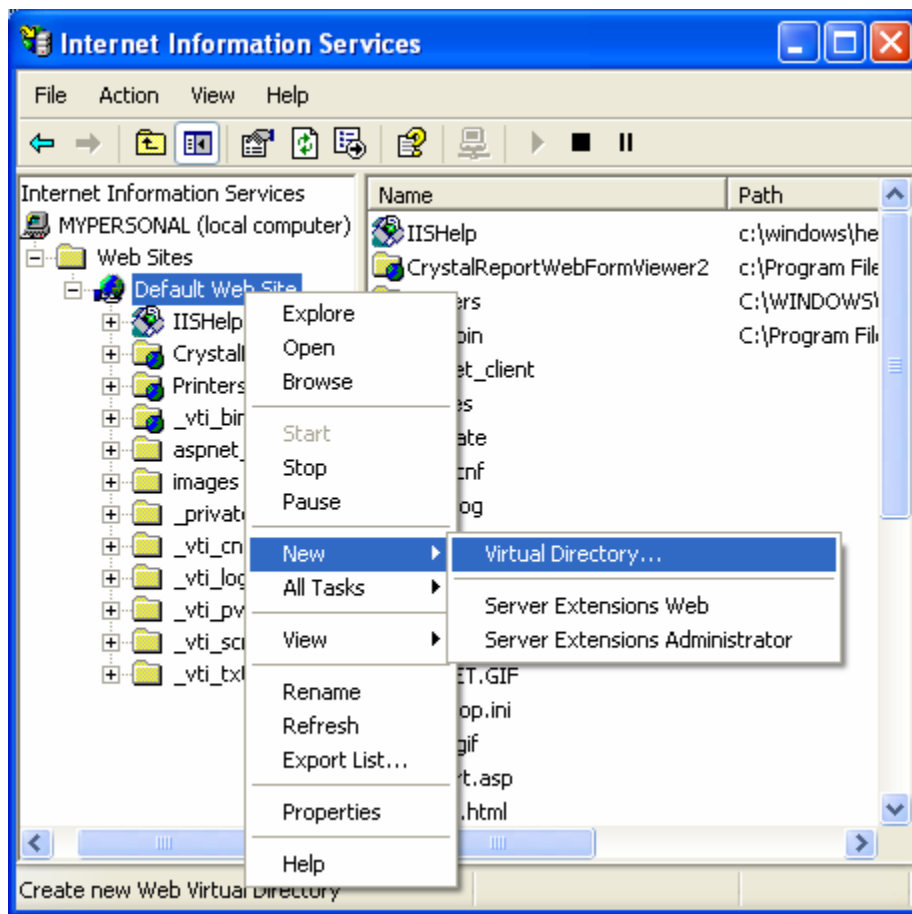


Figure 23: Invoking the **Virtual Directory** context menu.

Follow the **Virtual Directory Creation Wizard** instructions.



Figure 24: The Virtual Directory Creation Wizard welcome page.

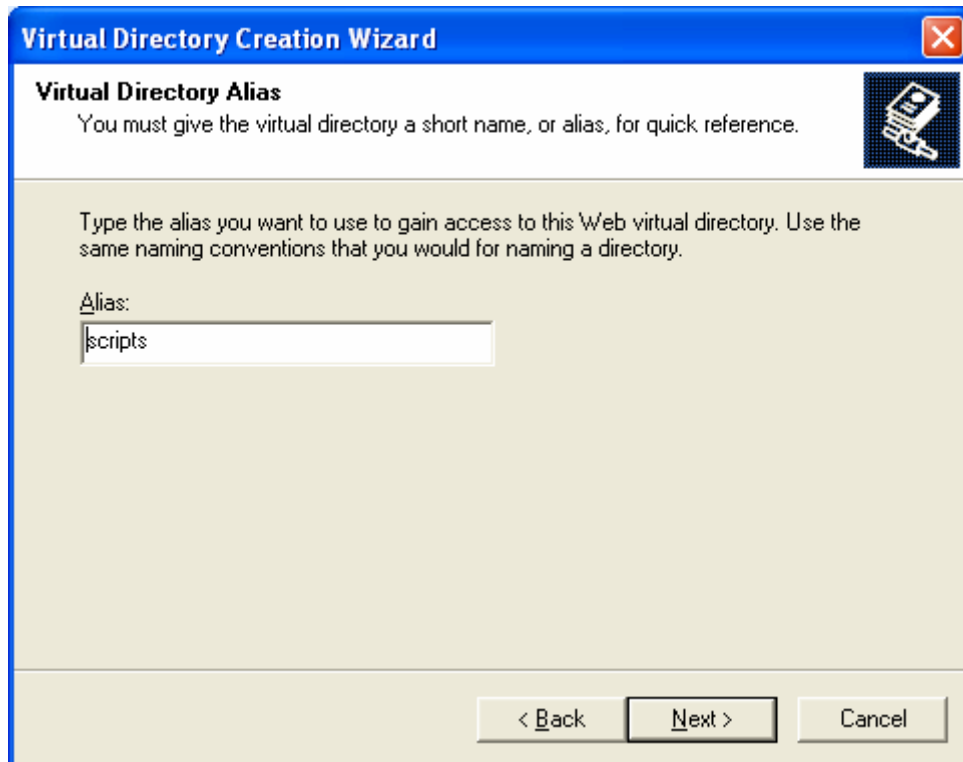


Figure 25: Entering the virtual directory alias.

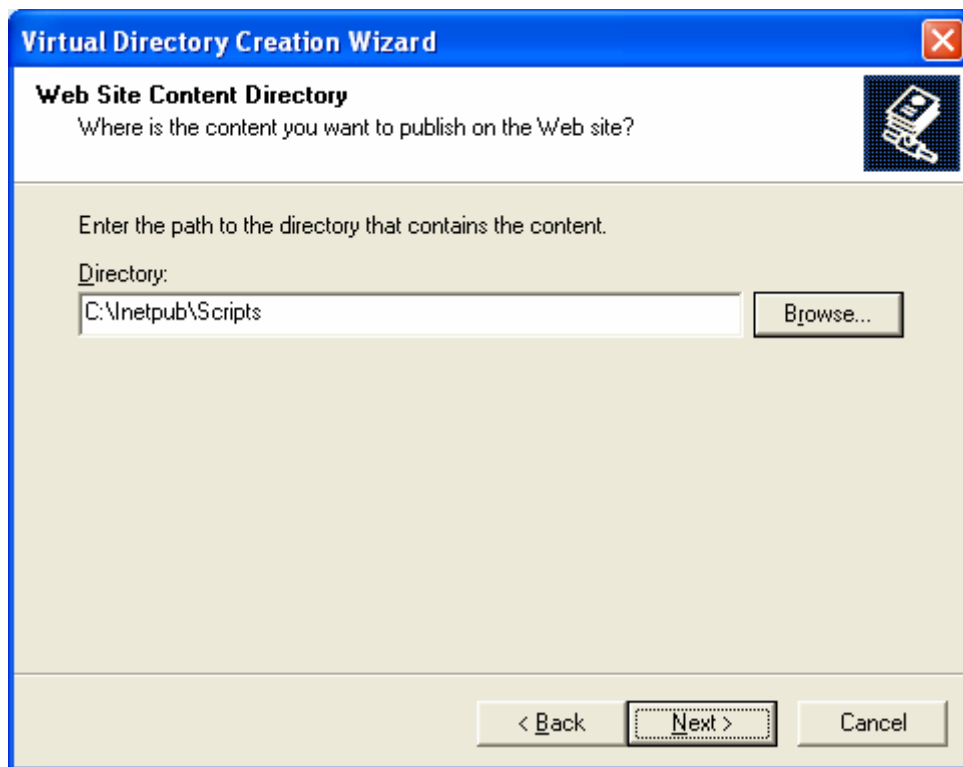


Figure 26: Entering the full path of the DLL.

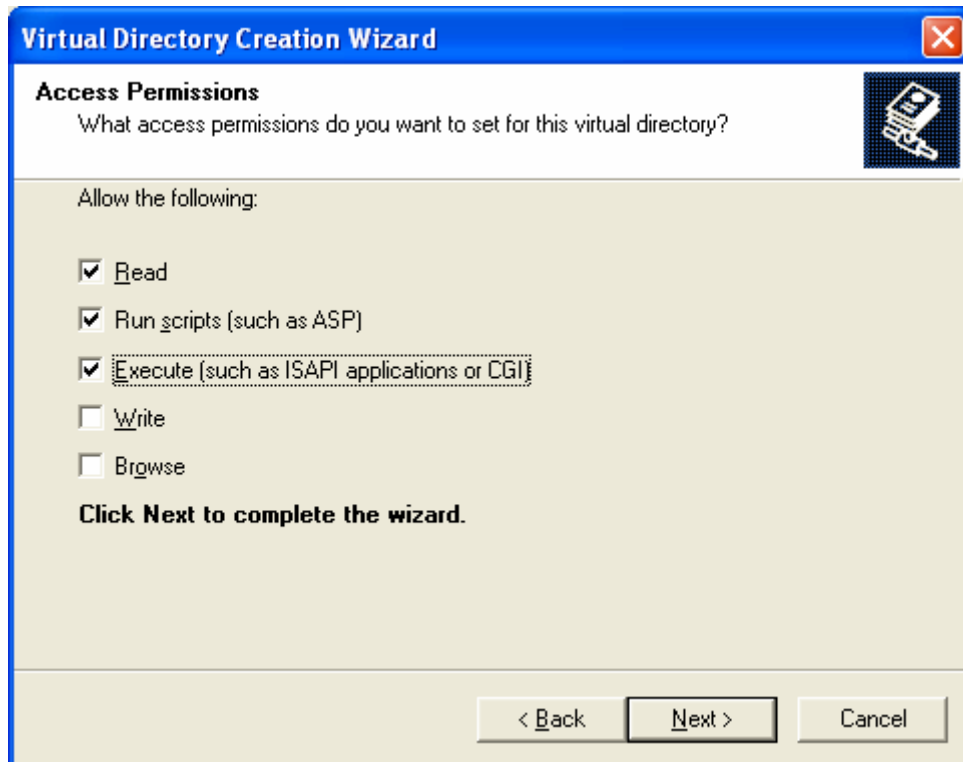


Figure 27: Setting the virtual directory access permission.

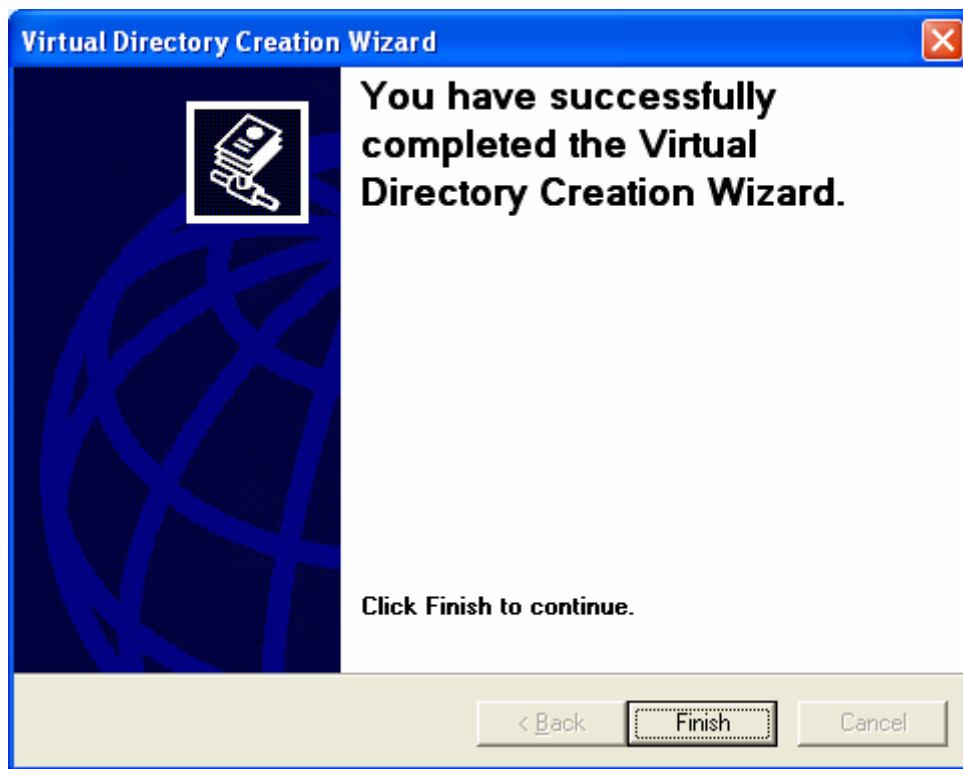


Figure 28: The final page of the Virtual Directory Creation Wizard.

Now we can see the scripts sub directory.

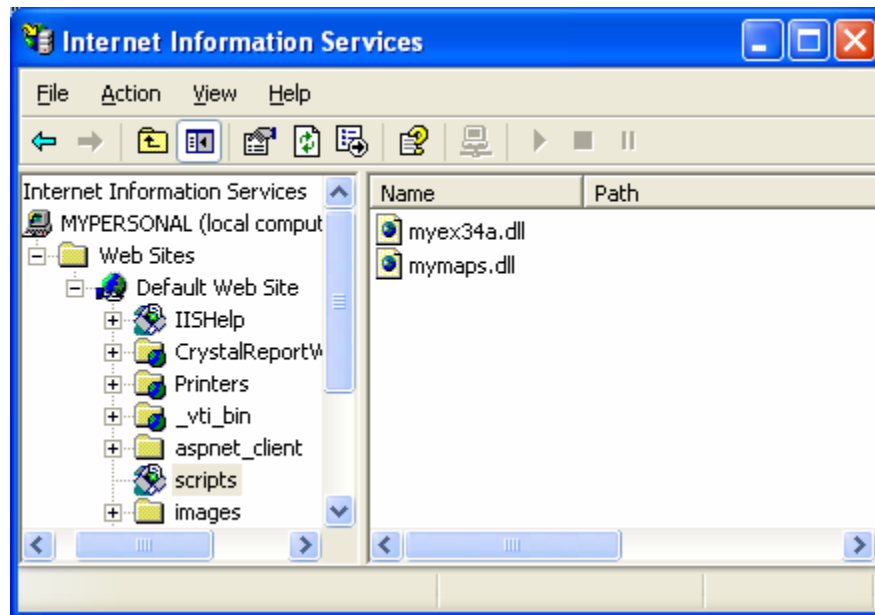


Figure 29: Then, the scripts sub directory can be seen.

Next, create HTML page named **weathermap.html** with the following simple content.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>

<HEAD>
```

```

<TITLE>Weathermap HTML Form</TITLE>
</HEAD>

<BODY>

<H1></H1>
<CENTER><B><FONT color="#FF0000" face="Verdana">Welcome to the Satellite
Weathermap Service</FONT></B></CENTER>
<H1></H1>
<FORM action="scripts/mymaps.dll?GetMap" method="POST">
  <P>Select your state: <SELECT name="State">
    <OPTION>Alabama</OPTION>
    <OPTION>Alaska</OPTION>
    <OPTION>Idaho</OPTION>
    <OPTION>Washington</OPTION>
  </SELECT> </P>
  <P><INPUT type="submit"><INPUT type="reset"> </P>
</FORM>

</BODY>

</HTML>

```

Listing 4.

Put this file under the **c:\inetpub\wwwroot**. Launch browser and type the local address as shown below.

<http://localhost/weathermap.html>

The following web page will be displayed. When you select a state and click the **Submit Query** button, **mymaps.dll** will be executed.

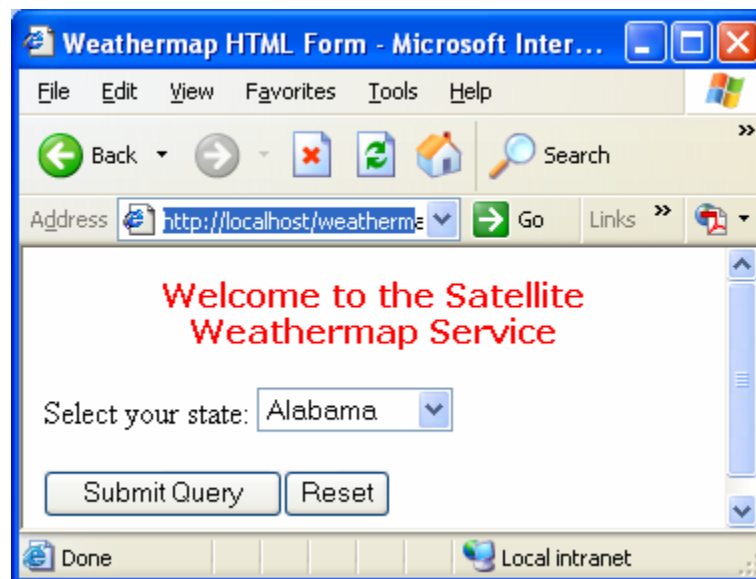


Figure 30: The Weathermap page in action.

Select a state and click the **Submit Query** button.

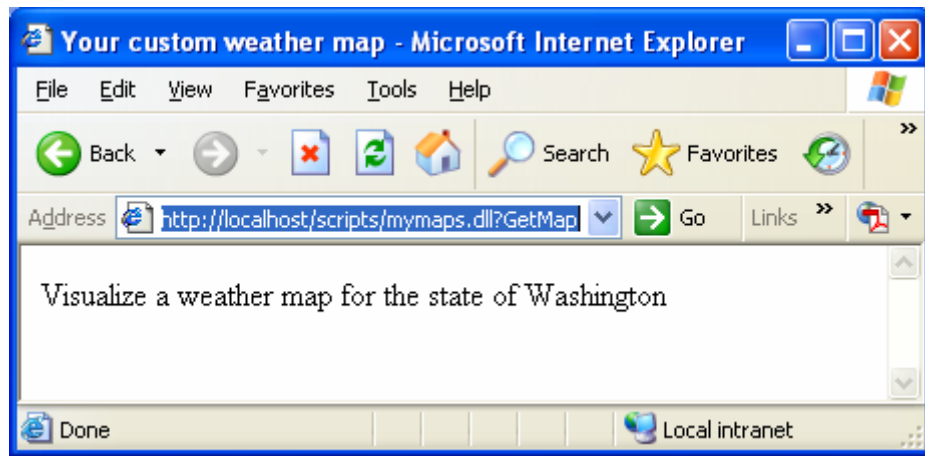


Figure 31: Invoking and executing `GetMap()` in **mymaps.dll**.

Hopefully you got the idea now.

The MFC ISAPI Server Extension Classes

Now is a good time to review the three MFC classes that are used to create an MFC ISAPI server extension. Remember that these classes are for ISAPI server extensions **only**. Don't even think of using them in ordinary Winsock or WinInet applications.

CHttpServer

With the help of the ISAPI Extension Wizard, you derive a class from `CHttpServer` for each ISAPI server extension DLL that you create. You need one member function for each extension function (including the default function), and you need an overridden `GetTitle()` function. The framework calls your extension functions in response to client requests, using the connections established in the parse map. The ISAPI Extension Wizard provides an overridden `GetExtensionVersion()` function, which you seldom edit unless you need initialization code to be executed when the DLL is loaded. One of the `CHttpServer` member functions that you're likely to call is `AddHeader()`, which adds special response headers, such as `Set-Cookie`, before the response is sent to the server.

CHttpServerContext

There's one `CHttpServer` object per DLL, but there is one `CHttpServerContext` object for each server transaction request. Your extension functions each provide a pointer to one of these objects. You don't derive from `CHttpServerContext`, so you can't easily have variables for individual transactions. Because different IIS threads can manage transactions, you have to be careful to perform synchronization for any data members of your `CHttpServer` class or global variables. You've already seen the use of the `StartContent()`, `EndContent()`, and `WriteTitle()` functions of the `CHttpServer` class plus the overloaded `>>` operator. You might also need to call the `CHttpServerContext::GetServerVariable` function to read information sent by the client in the request headers.

CHtmlStream

Most of the time, you don't use the `CHtmlStream` class directly. The `CHttpServerContext` class has a `CHtmlStream` data member, `m_pStream`, that's hooked up to the `>>` operator and serves as the output for HTML data. You could access the `CHtmlStream` object and call its `Write` member function if you needed to send binary data to the client. Because objects of the `CHtmlStream` class accumulate bytes in memory before sending them to the client, you need an alternative approach if your DLL relays large files directly from disk.

A Practical ISAPI Server Extension: **myex34a.dll**

The weather map server isn't interesting enough to make into a real project. You'll probably find the MYEX34A example more to your taste. It's a real Internet commerce application, a pizza-ordering program. Imagine a computer-controlled pizza oven and a robot arm that selects frozen pizzas.

MYEX34A From Scratch

Use the previous module's [DEFAULT.HTM](#) file, rename it to **index.html** and put it under the **c:\Inetpub\wwwroot**. You also can create it by using the following code. Also make sure the **backgrd.gif** and **bullet.gif** files are also in the same sub directory.

```
<!-- This is DEFAULT.HTM file (myex33a) renamed to index.html -->
<HTML>

<HEAD>
<TITLE>Inside Visual C++ Home Page</TITLE>
</HEAD>

<BODY background="backgrd.gif">

<FONT color="#0000ff">
<H1></H1>
<CENTER><B><FONT face="Verdana">Welcome to the Nothing test page</FONT></B></CENTER>
<H1></H1>
<P></FONT></P>
<UL>
  <UL>
    <BR>
    <IMG src="bullet.GIF" align="MIDDLE" border="0"><A href="pizzaform.html"> Pizza
      form </A>
    </UL>
  </UL>
</UL>

</BODY>

</HTML>
```

Listing 5.

You can test the page using **http://localhost/** address provided that your IIS/web server was [properly configured](#).

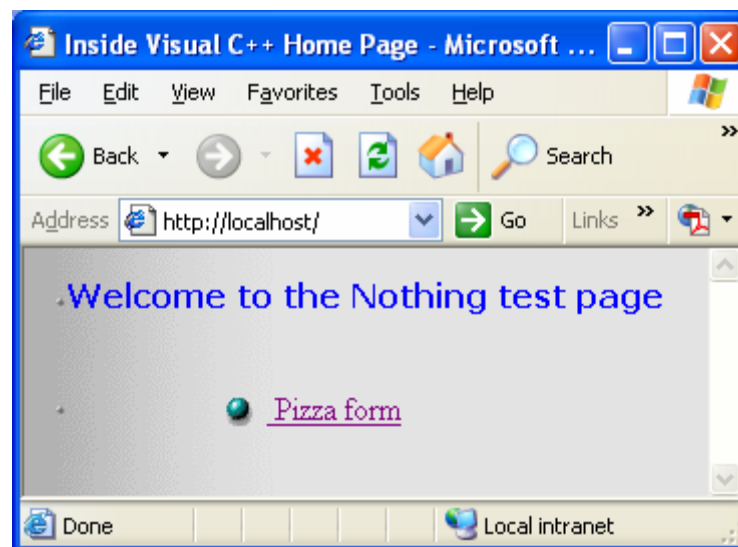


Figure 32: MYEX34A main web page in action.

Create another HTML page named **PizzaForm.html** and use the following code. Put this file under the **c:\Inetpub\wwwroot** directory.

```
<HTML>

<HEAD>
<TITLE>Inside Visual C++ HTML Form 1</TITLE>
</HEAD>

<BODY>

<H1></H1>
<CENTER><B><FONT color="#0000FF" face="Verdana" size="4">Welcome to
CyberPizza</FONT></B></CENTER>
<H1></H1>
<P><FONT color="#FF0000" face="Tahoma">Enter your order.</FONT> </P>
<FORM action="scripts/myex34a.dll?ProcessPizzaForm" method="POST">
  <P>Your Name: <INPUT type="text" name="name" value> </P>
  <P>Your Address: <INPUT type="text" name="address" value> </P>
  <P>Number of Pies: <INPUT type="text" name="quantity" value="1"> </P>
  <P>Pizza Size: </P>
  <MENU>
    <LI><INPUT type="radio" name="size" value="8">8-inch </LI>
    <LI><INPUT type="radio" name="size" value="10">10-inch </LI>
    <LI><INPUT type="radio" name="size" value="12" checked>12-inch </LI>
    <LI><INPUT type="radio" name="size" value="14">14-inch </LI>
  </MENU>
  <P>Toppings: </P>
  <P><INPUT type="checkbox" name="top1" value="Pepperoni" checked> Pepperoni
  <INPUT type="checkbox" name="top2" value="Onions"> Onions
  <INPUT type="checkbox" name="top3" value="Mushrooms"> Mushrooms
  <INPUT type="checkbox" name="top4" value="Sausage"> Sausage </P>
  <P><EM>(you can select multiple toppings)</EM> </P>
  <P><INPUT type="submit" value="Submit Order Now"><INPUT type="reset"> </P>
</FORM>

</BODY>

</HTML>
```

Listing 6.

Test the page by clicking the **Pizza form** link in the previous main page.

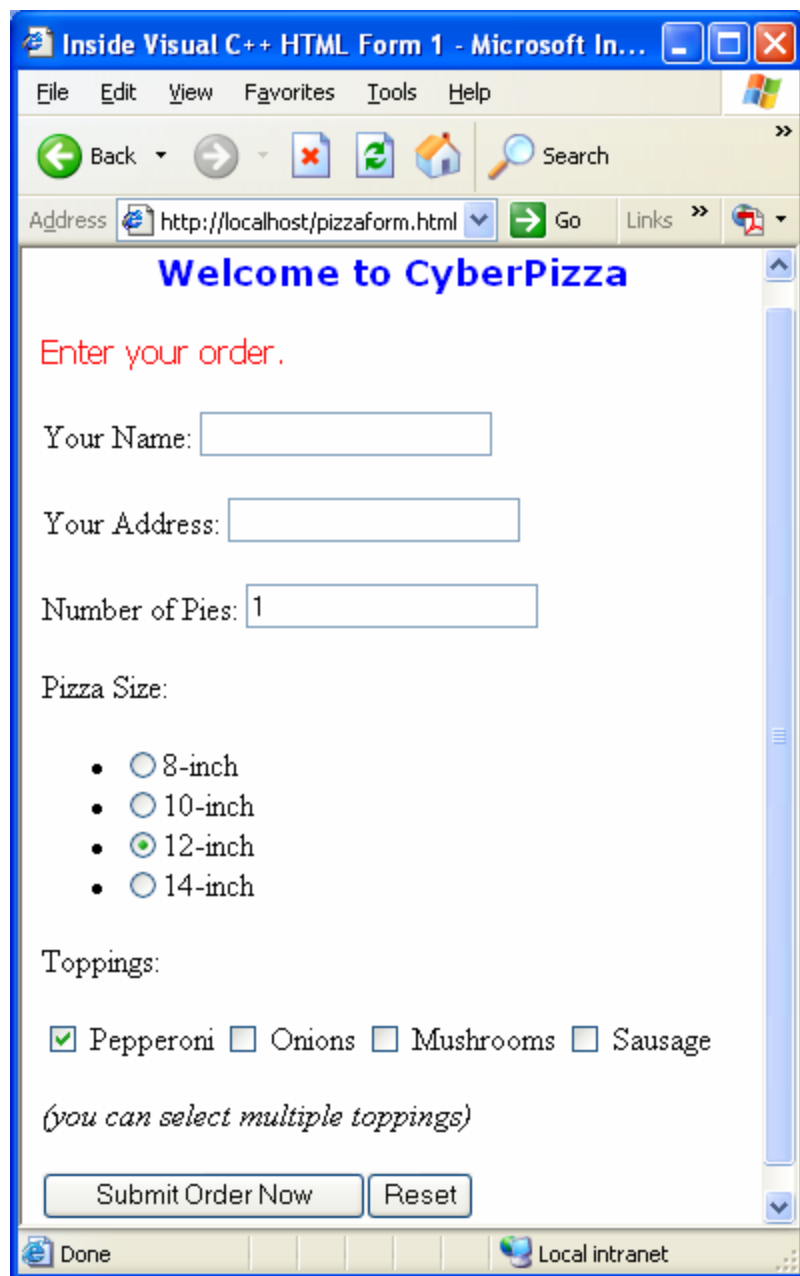


Figure 33: PizzaForm.html in action.

Next, let build the MYEX34A DLL program. Follow the shown steps.

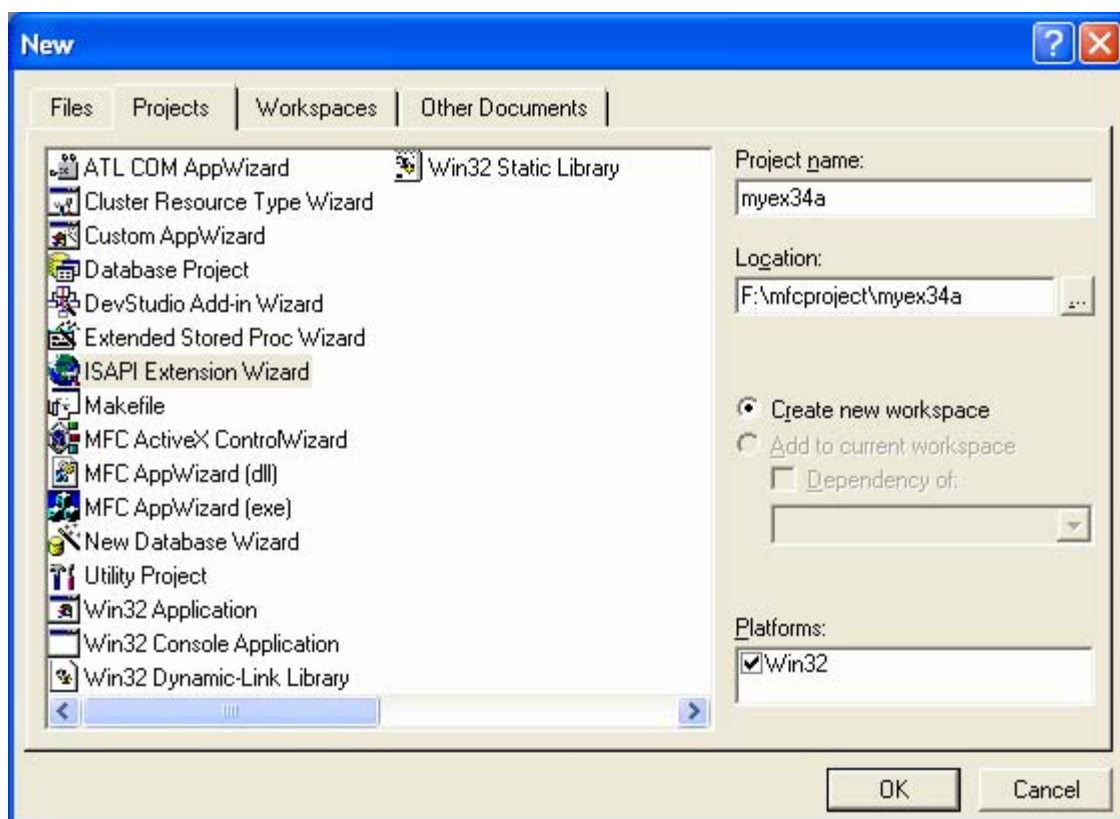


Figure 34: MYEX34A – ISAPI Extension new project dialog.

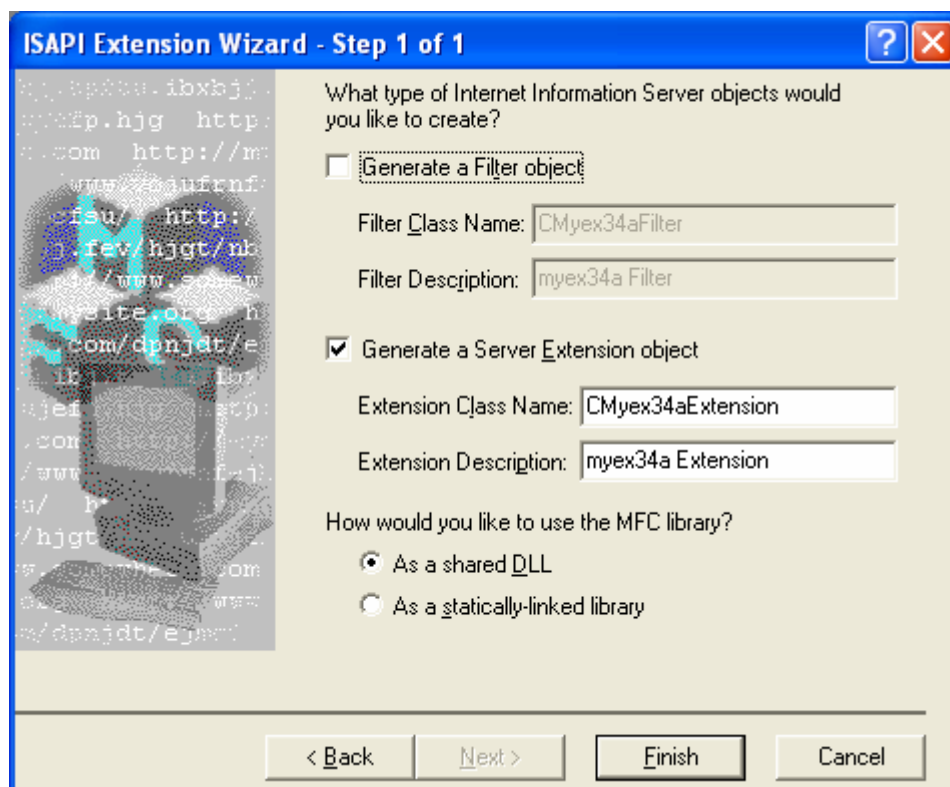


Figure 35: MYEX34A – ISAPI Extension Wizard Step 1 of 1.

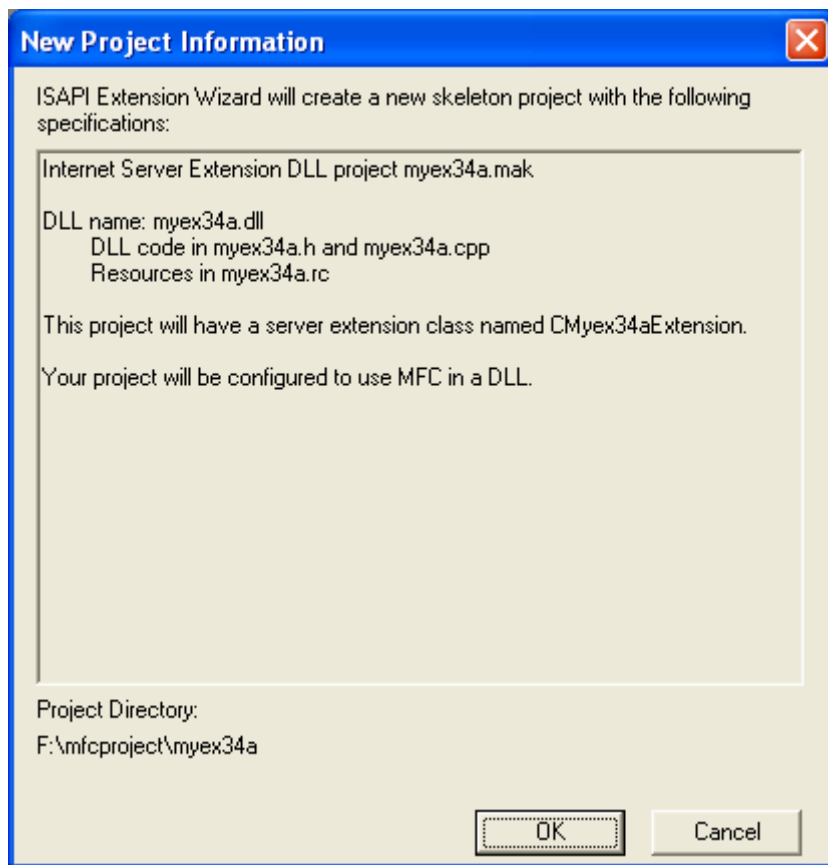


Figure 36: MYEX34A project summary.

Use ClassView or manually add the following member variables to CMyex34aExtension class (**myex34a.h**).

```
private:
    CCriticalSection m_cs;
    int m_nTotalPizzaOrders;
```

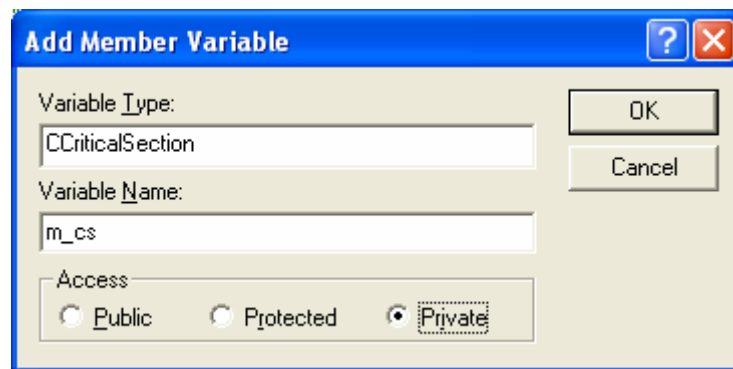


Figure 37: Adding private member variable to CMyex34aExtension class.

```
//}}AFX_MSG
private:
    int m_nTotalPizzaOrders;
    CCriticalSection m_cs;
};
```

Listing 7.

Then, add `GetTitle()` message handler function for the page title.

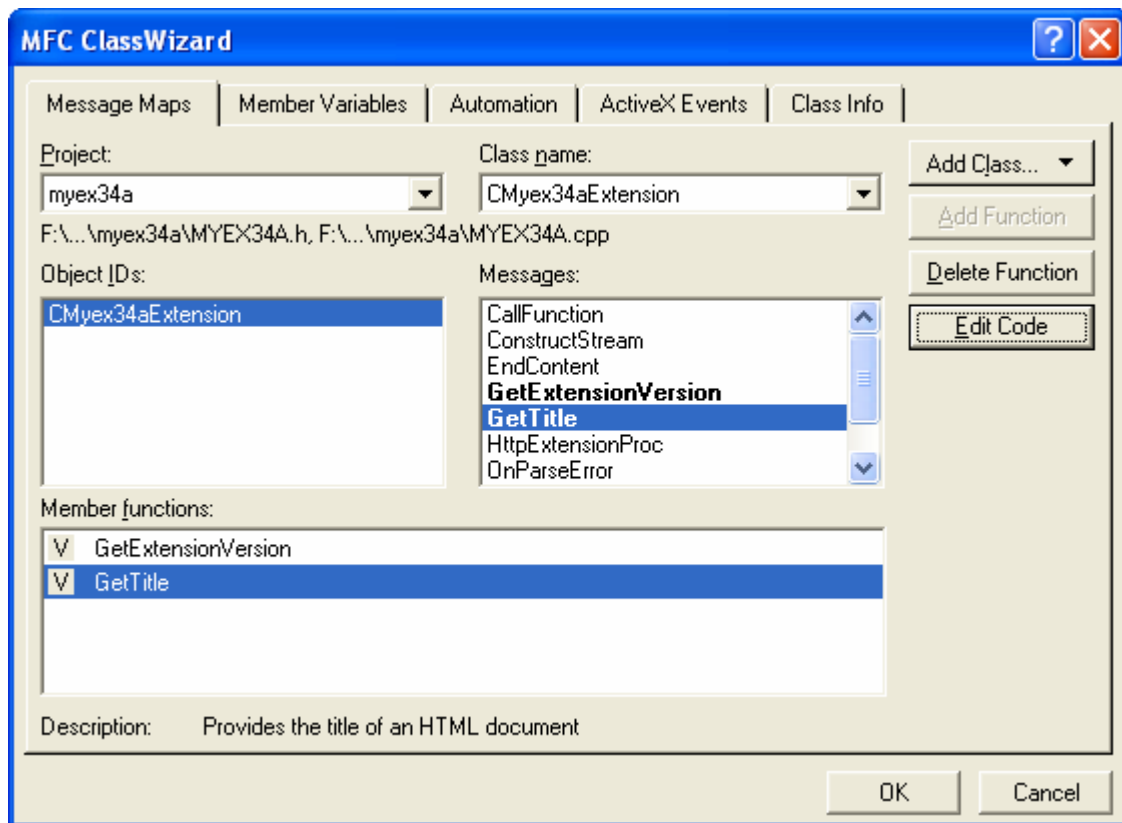


Figure 38: Adding `GetTitle()` message handler function to `CMyex34aExtension` class.

Add handlers for the commands.

```
void ConfirmOrder(CHttpServerContext* pCtxt, LPCTSTR pstrName);
void ProcessPizzaForm(CHttpServerContext* pCtxt, LPCTSTR pstrName, LPCTSTR
pstrAddr, int nQuantity, LPCTSTR pstrSize, LPCTSTR pstrTop1, LPCTSTR pstrTop2,
LPCTSTR pstrTop3, LPCTSTR pstrTop4);
void ProcessTimesheet(CHttpServerContext* pCtxt, int nPeriod, LPCTSTR
pstrEmployee, double dHours, int nJob);
```

```
// TODO: Add handlers for your commands here.
// For example:

void Default(CHttpServerContext* pCtxt);
void ConfirmOrder(CHttpServerContext* pCtxt, LPCTSTR pstrName);
void ProcessPizzaForm(CHttpServerContext* pCtxt, LPCTSTR pstrName,
    LPCTSTR pstrAddr, int nQuantity, LPCTSTR pstrSize, LPCTSTR pstrTop1,
    LPCTSTR pstrTop2, LPCTSTR pstrTop3, LPCTSTR pstrTop4);
void ProcessTimesheet(CHttpServerContext* pCtxt, int nPeriod,
    LPCTSTR pstrEmployee, double dHours, int nJob);
DECLARE_PARSE_MAP()

//{{AFX_MSG(CMyex34aExtension)
```

Listing 8.

Delete the `TerminateExtension()`, here, been commented out.

```

virtual LPCTSTR GetTitle() const;
//}}AFX_VIRTUAL
// virtual BOOL TerminateExtension(DWORD dwFlags);

// TODO: Add handlers for your commands here.
// For example:

```

Listing 9.

The CPP file. Manually, add command passing maps in **myex34a.cpp**.

```

ON_PARSE_COMMAND(ConfirmOrder, CMyex34aExtension, ITS_PSTR)
ON_PARSE_COMMAND_PARAMS("name")
ON_PARSE_COMMAND(ProcessPizzaForm, CMyex34aExtension, ITS_PSTR ITS_PSTR ITS_I4 ITS_PSTR
ITS_PSTR ITS_PSTR ITS_PSTR ITS_PSTR)
ON_PARSE_COMMAND_PARAMS("name address quantity size top1=~ top2=~ top3=~ top4=~")
ON_PARSE_COMMAND(ProcessTimesheet, CMyex34aExtension, ITS_I4 ITS_PSTR ITS_R8 ITS_I4)
ON_PARSE_COMMAND_PARAMS("Period Employee Hours Job")

// command-parsing map
BEGIN_PARSE_MAP(CMyex34aExtension, CHttpServer)
// TODO: insert your ON_PARSE_COMMAND() and
// ON_PARSE_COMMAND_PARAMS() here to hook up your commands.
// For example:
ON_PARSE_COMMAND(ConfirmOrder, CMyex34aExtension, ITS_PSTR)
ON_PARSE_COMMAND_PARAMS("name")
ON_PARSE_COMMAND(ProcessPizzaForm, CMyex34aExtension, ITS_PSTR
ON_PARSE_COMMAND_PARAMS("name address quantity size top1=~ top2=~ top3=~ top4=~")
ON_PARSE_COMMAND(ProcessTimesheet, CMyex34aExtension, ITS_I4 ITS_PSTR ITS_R8 ITS_I4)
ON_PARSE_COMMAND_PARAMS("Period Employee Hours Job")
ON_PARSE_COMMAND(Default, CMyex34aExtension, ITS_EMPTY)
DEFAULT_PARSE_COMMAND(Default, CMyex34aExtension)
END_PARSE_MAP(CMyex34aExtension)

////////////////////////////////////
// The one and only CMyex34aExtension object

```

Listing 10.

Initialize the `m_nTotalPizzaOrders` in the constructor as shown below.

```

CMyex34aExtension::CMyex34aExtension()
{
    m_nTotalPizzaOrders = 0;
}

// CMyex34aExtension implementation
CMyex34aExtension::CMyex34aExtension()
{
    m_nTotalPizzaOrders = 0;
}

```

Listing 11.

Finally, add the implementation part as shown below just after the `Default()` and don't forget to delete the `TerminateExtension()` implementation.

```

...
...
...

```

```

// BOOL CMyex34aExtension::TerminateExtension(DWORD dwFlags)
// {
//     // extension is being terminated
//     //TODO: Clean up any per-instance resources
//     return TRUE;
// }

////////////////////////////////////
// CMyex34aExtension command handlers

void CMyex34aExtension::Default(CHttpServerContext* pCtxt)
{
    StartContent(pCtxt);
    WriteTitle(pCtxt);

    *pCtxt << _T("This default message was produced by the Internet");
    *pCtxt << _T(" Server DLL Wizard. Edit your CMyex34aExtension::Default()");
    *pCtxt << _T(" implementation to change it.\r\n");

    EndContent(pCtxt);
}

// This must be defined first, coz we will use it in ProcessPizzaForm
void CMyex34aExtension::ConfirmOrder(CHttpServerContext* pCtxt, LPCTSTR pstrName)
{
    StartContent(pCtxt);
    WriteTitle(pCtxt);
    *pCtxt << "<p>Our courteous delivery person will arrive within 30 minutes.";
    *pCtxt << "<p>Thank you, " << pstrName << ", for using CyberPizza.";
    // now retrieve the order from disk by name, then make the pizza
    // be prepared to delete the order after a while if the customer doesn't confirm
    m_cs.Lock(); // gotta be threadsafe
    long int nTotal = ++m_nTotalPizzaOrders;
    m_cs.Unlock();
    *pCtxt << "<p>Total pizza orders = " << nTotal;
    EndContent(pCtxt);
}

void CMyex34aExtension::ProcessPizzaForm(CHttpServerContext* pCtxt, LPCTSTR pstrName,
    LPCTSTR pstrAddr, int nQuantity, LPCTSTR pstrSize,
    LPCTSTR pstrTop1, LPCTSTR pstrTop2, LPCTSTR pstrTop3, LPCTSTR
pstrTop4)
{
    StartContent(pCtxt);
    WriteTitle(pCtxt);
    if((strlen(pstrName) > 0) && (strlen(pstrAddr) > 0))
    {
        *pCtxt << " Your pizza order is as follows:";
        *pCtxt << "<p>Name: " << pstrName;
        *pCtxt << "<p>Address: " << pstrAddr;
        *pCtxt << "<p>Number of pies: " << (long int) nQuantity;
        *pCtxt << "<p>Size: " << pstrSize;
        *pCtxt << "<p>Toppings: " << pstrTop1 << " " << pstrTop2 << " " << pstrTop3
<< " " << pstrTop4;
        // Just fix the price...hahahaha...should has another routine lol!
        *pCtxt << "<p>The total cost is $23.49, including delivery.";
        *pCtxt << "<form action=\"myex34a.dll?ConfirmOrder\" method=POST>";
        *pCtxt << "<p><input type=\"hidden\" name=\"name\" value=\"\" << pstrName <<
\">"; // xref to original order
        *pCtxt << "<p><input type=\"submit\" value=\"Confirm and charge my credit
card\">";

        *pCtxt << "</form>";
        // store this order in a disk file or database, referenced by name
    }
    else {
        *pCtxt << " You forgot to enter your name or address. Back up and try

```

```

again." ;
    }
    EndContent(pCtxt);
}

void CMyx34aExtension::ProcessTimesheet(CHttpServerContext* pCtxt, int nPeriod,
                                         LPCTSTR pstrEmployee, double dHours, int nJob)
{
    StartContent(pCtxt);
    WriteTitle(pCtxt);
    *pCtxt << " Employee timesheet:";
    *pCtxt << "<p>Period: " << (long int) nPeriod;
    *pCtxt << "<p>Employee: " << pstrEmployee;
    *pCtxt << "<p>Hours: " << dHours;
    *pCtxt << "<p>Job: " << (long int) nJob;
    // store this transaction in a disk file or database
    EndContent(pCtxt);
}

// Do not edit the following lines, which are needed by ClassWizard.
#if 0
BEGIN_MESSAGE_MAP(CMyx34aExtension, CHttpServer)
    //{{AFX_MSG_MAP(CMyx34aExtension)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
#endif // 0

////////////////////////////////////
// If your extension will not use MFC, you'll need this code to make
// sure the extension objects can find the resource handle for the
// module. If you convert your extension to not be dependent on MFC,
// remove the comments around the following AfxGetResourceHandle()
// and DllMain() functions, as well as the g_hInstance global.

/****
static HINSTANCE g_hInstance;

HINSTANCE AFXISAPI AfxGetResourceHandle()
{
    return g_hInstance;
}

BOOL WINAPI DllMain(HINSTANCE hInst, ULONG ulReason,
                    LPVOID lpReserved)
{
    if (ulReason == DLL_PROCESS_ATTACH)
    {
        g_hInstance = hInst;
    }

    return TRUE;
}
****/

LPCTSTR CMyx34aExtension::GetTitle() const
{
    // TODO: Add your specialized code here and/or call the base class

    return "MyEx34a Pizza Extension";
}

```

Listing 12.

Build the program, generating **myex34a.dll**. Copy the DLL file to **c:\inetpub\scripts** sub directory. We put all the DLL in one place for easier management. You can create other directory/sub directory or put directly under the default wwwroot directory, make sure the path is matched with the code in your HTML and create virtual directory if needed.

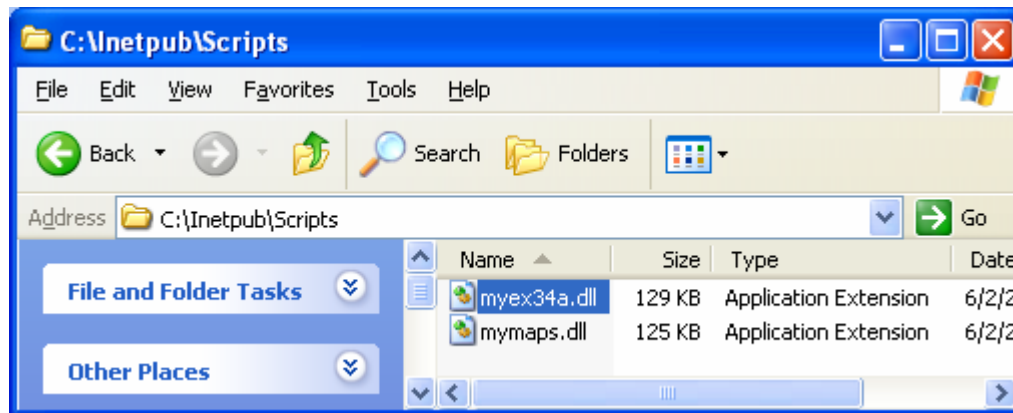


Figure 39: The DLL copied to scripts sub directory on Windows XP Pro.

Restart/start the World Wide Web publishing if required. Go to the browser address bar and type **http://localhost/** and press **Enter** or click the **Go** button. The following page should be launched.

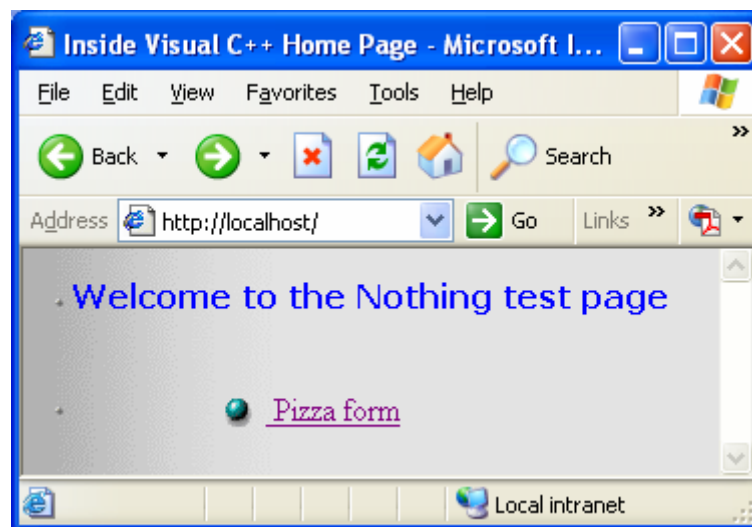


Figure 40: Pizza order web site in action.

When the **Pizza form** link is clicked, the following form will be displayed.

The image shows a web browser window titled "Inside Visual C++ HTML Form 1 - Microsoft In...". The address bar shows "http://localhost/pizzaform.html". The page content includes a welcome message, a form for entering order details, and a status bar at the bottom.

Welcome to CyberPizza

Enter your order.

Your Name:

Your Address:

Number of Pies:

Pizza Size:

- ☐ 8-inch
- ☐ 10-inch
- ☒ 12-inch
- ☐ 14-inch

Toppings:

☒ Pepperoni ☐ Onions ☐ Mushrooms ☐ Sausage

(you can select multiple toppings)

Done Local intranet

Figure 41: CyberPizza's Pizza form, ready for the order.

Fill up the data and click the **Submit Order Now** button.

The screenshot shows a web browser window titled "Inside Visual C++ HTML Form 1 - Microsoft In...". The address bar displays "http://localhost/pizzaform.html". The page content includes a welcome message, a form for entering order details, and two buttons at the bottom: "Submit Order Now" and "Reset".

Welcome to CyberPizza

Enter your order.

Your Name:

Your Address:

Number of Pies:

Pizza Size:

- ☐ 8-inch
- ☐ 10-inch
- ☐ 12-inch
- ☒ 14-inch

Toppings:

☒ Pepperoni ☒ Onions ☒ Mushrooms ☒ Sausage

(you can select multiple toppings)

Figure 42: Buying pizza by filling up the Pizza form.

Here, we have the confirmation page for the data filled, processed by `ProcessPizzaForm()`.

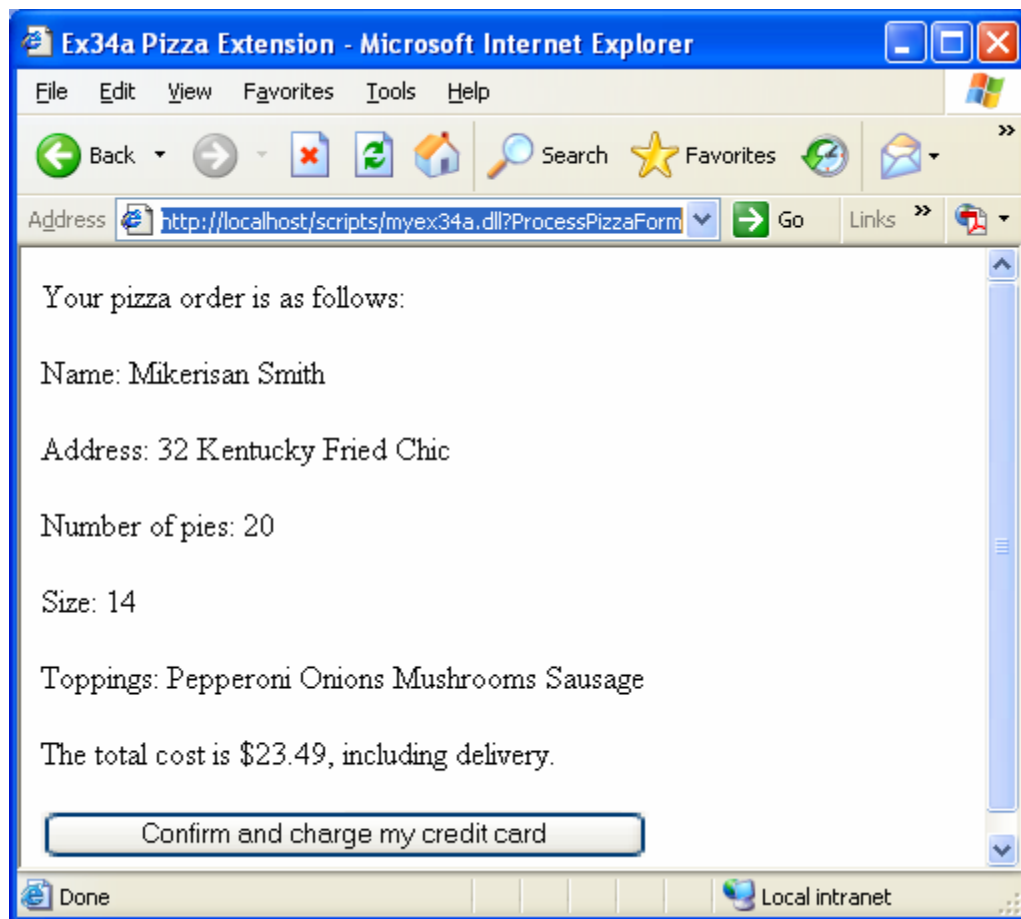


Figure 43: Confirmation page for the ordered pizza.

Click the **Confirm and charge my credit card** button. The following page displayed, processed by `ConfirmOrder()`.

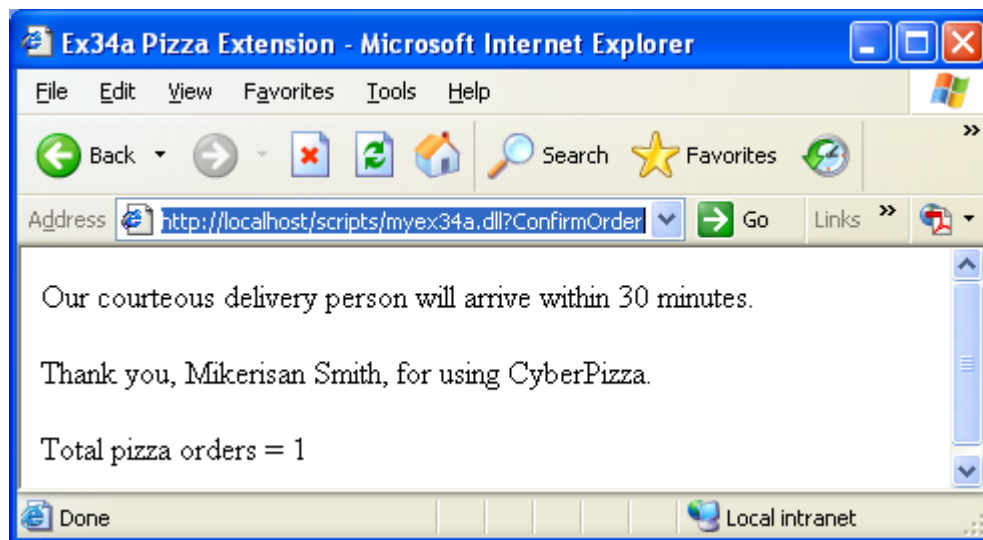


Figure 44: Confirmed page of the ordered pizza and the payment, ready to be delivered.

Try reorder the pizzas, the **Total pizza orders** will increase. If the required fields not filled and customer just clicks the **Submit Order Now**, the following message will be displayed.

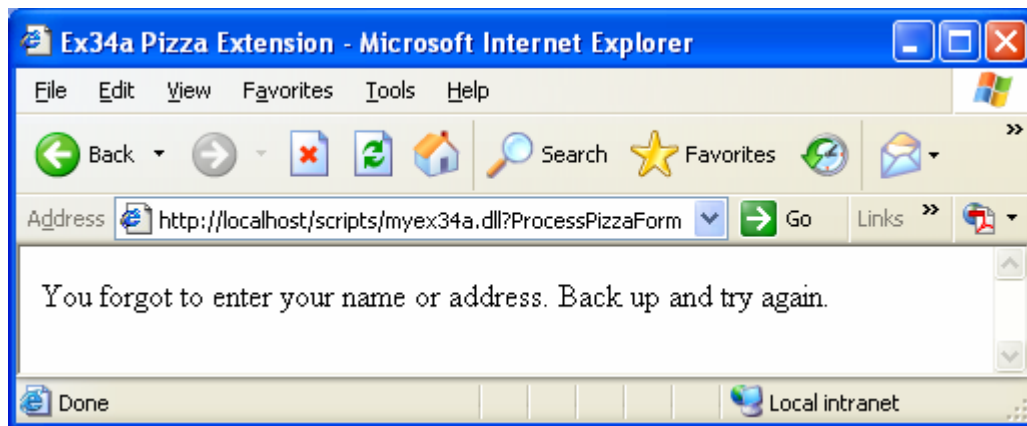


Figure 45: Form information validation message.

For all the web pages shown, take note that the DLL functions executed were displayed in the browser address bar.

The Story

The First Step: Getting the Order

Junior sales trainees are constantly admonished to "get the order." That's certainly necessary in any form of commerce, including the Internet. When the hungry customer hyperlinks to your site (by clicking on a picture of a pizza, of course), he or she simply downloads an HTML file that looks like this:

```
<HTML>

<HEAD>
<TITLE>Inside Visual C++ HTML Form 1</TITLE>
</HEAD>

<BODY>

<H1></H1>
<CENTER><B><FONT color="#0000FF" face="Verdana" size="4">Welcome to
CyberPizza</FONT></B></CENTER>
<H1></H1>
<P><FONT color="#FF0000" face="Tahoma">Enter your order.</FONT> </P>
<FORM action="scripts/myex34a.dll?ProcessPizzaForm" method="POST">
  <P>Your Name: <INPUT type="text" name="name" value> </P>
  <P>Your Address: <INPUT type="text" name="address" value> </P>
  <P>Number of Pies: <INPUT type="text" name="quantity" value="1"> </P>
  <P>Pizza Size: </P>
  <MENU>
    <LI><INPUT type="radio" name="size" value="8">8-inch </LI>
    <LI><INPUT type="radio" name="size" value="10">10-inch </LI>
    <LI><INPUT type="radio" name="size" value="12" checked>12-inch </LI>
    <LI><INPUT type="radio" name="size" value="14">14-inch </LI>
  </MENU>
  <P>Toppings: </P>
  <P><INPUT type="checkbox" name="top1" value="Pepperoni" checked> Pepperoni
  <INPUT type="checkbox" name="top2" value="Onions"> Onions
  <INPUT type="checkbox" name="top3" value="Mushrooms"> Mushrooms
  <INPUT type="checkbox" name="top4" value="Sausage"> Sausage </P>
  <P><EM>(you can select multiple toppings)</EM> </P>
  <P><INPUT type="submit" value="Submit Order Now"><INPUT type="reset"> </P>
</FORM>
```

```
</BODY>  
</HTML>
```

Listing 13.

Figure 47 shows how the order form appears in the browser.

The screenshot shows a Microsoft Internet Explorer window titled "Inside Visual C++ HTML Form 1 - Microsoft In...". The address bar shows "http://localhost/pizzaform.html". The page content includes a blue header "Welcome to CyberPizza", a red prompt "Enter your order.", and several form fields: "Your Name:" (text box), "Your Address:" (text box), "Number of Pies:" (text box with "1" entered), and "Pizza Size:" (radio buttons for 8-inch, 10-inch, 12-inch (selected), and 14-inch). Below these are checkboxes for toppings: "Pepperoni" (checked), "Onions", "Mushrooms", and "Sausage". A note "(you can select multiple toppings)" is present. At the bottom are "Submit Order Now" and "Reset" buttons. The status bar shows "Done" and "Local intranet".

Figure 46: The CyberPizza order form.

So far, no ISAPI DLL is involved. When the customer clicks the **Submit Order Now** button, the action begins. Here's an example what the server sees:

```
POST scripts/myex34a.dll?ProcessPizzaForm HTTP/1.0  
(request headers)  
(blank line)
```

```
name=Walter+Sullivan&address=Redmond%2C+WA&quantity=2&size=12&top1=Pepperoni&top3=Mushrooms
```

Looks like Walter Sullivan has ordered two 12-inch pepperoni and mushroom pizzas. The browser inserts a + sign in place of a space, the %2C is a comma, and the & is the parameter separator. Now let's look at the parse map entries in myex34a.cpp:

```
ON_PARSE_COMMAND(ProcessPizzaForm, CMyex34aExtension,
    ITS_PSTR ITS_PSTR ITS_I4 ITS_PSTR ITS_PSTR ITS_PSTR ITS_PSTR)
ON_PARSE_COMMAND_PARAMS("name address quantity size top1=~ top2=~ top3=~
top4=~")
```

Optional Parameters

When you write your parse map statements, you must understand the browser's rules for sending parameter values from a form. In the MYEX34A pizza form, the browser always sends parameters for text fields, even if the user enters no data. If the user left the **Name** field blank, for example, the browser would send name=&. For **check box** fields, however, it's a different story. The browser sends the check box parameter value only if the user checks the box. The parameters associated with check boxes are thus defined as optional parameters. If your parse macro for parameters looked like this:

```
ON_PARSE_COMMAND_PARAMS("name address quantity size top1 top2 top3 top4")
```

there would be trouble if the customer didn't check all the toppings. The HTTP request would simply fail, and the customer would have to search for another pizza site. The =~ symbols in the myex34a.cpp code designate the last four parameters as optional, with default values ~. If the **Toppings** option is checked, the form transmits the value; otherwise, it transmits a ~ character, which the DLL can test for. Optional parameters must be listed last. The DLL's ProcessPizzaForm() function reads the parameter values and produces an HTML confirmation form, which it sends to the customer. Here is part of the function's code:

```
*pCtxt << "<form action=\"myex34a.dll?ConfirmOrder\" method=POST>";
*pCtxt << "<p><input type=\"hidden\" name=\"name\" value=\"\"";
*pCtxt << pstrName << "\">"; // xref to original order
*pCtxt << "<p><input type=\"submit\" value=\"Confirm and charge my credit
card\">";
*pCtxt << "</form>";
// Store this order in a disk file or database, referenced by name
}
else {
    *pCtxt << "You forgot to enter name or address. Back up and try again. ";
}
EndContent(pCtxt);
```

The resulting browser screen is shown in Figure 47.

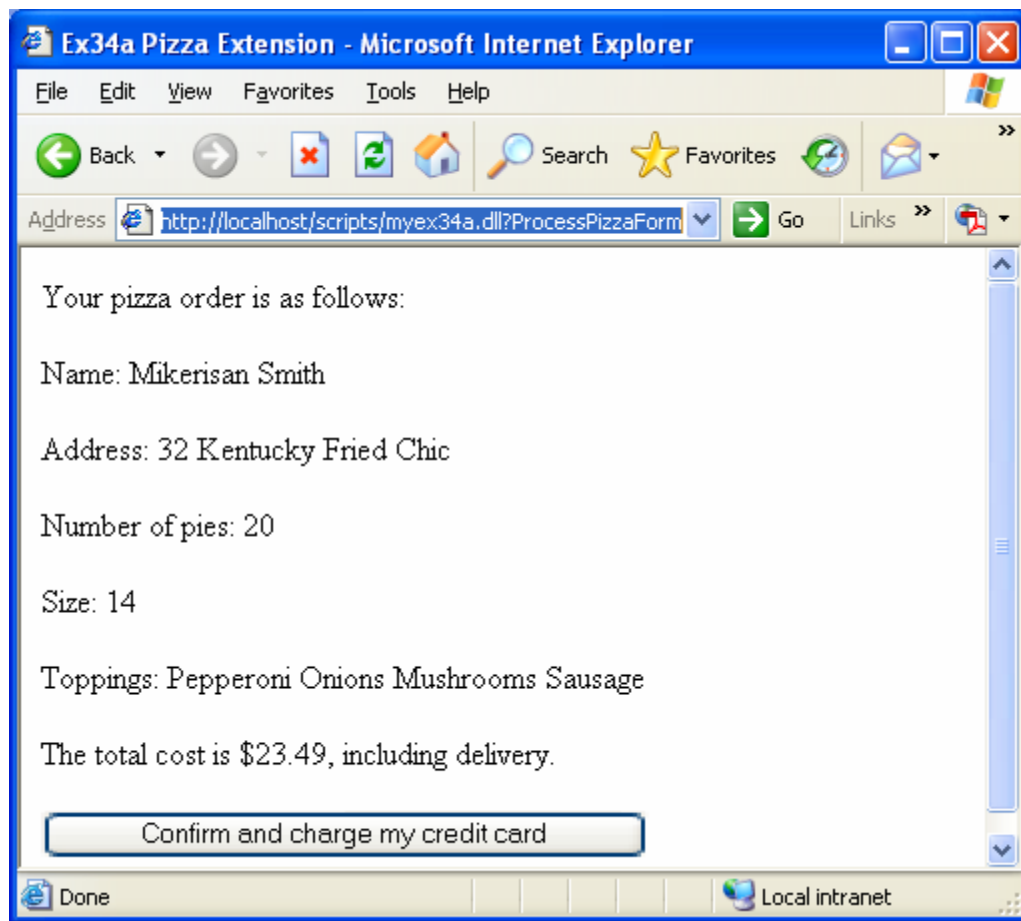


Figure 47: The pizza confirmation browser screen.

As you can see, we took a shortcut computing the price. To accept, the customer clicks the submit button named **Confirm And Charge My Credit Card**.

The Second Step: Processing the Confirmation

When the user clicks the **Confirm And Charge My Credit Card** button, the browser sends a second POST request to the server, specifying that the `CMyex34aExtension::ConfirmOrder` function be called. But now you have to solve a big problem. Each HTTP connection (request/response) is independent of all others. How are you going to link the confirmation request with the original order? Although there are different ways to do this, the most common approach is to send some text back with the confirmation in a hidden input tag. When the confirmation parameter values come back, the server uses the hidden text to match the confirmation to the original order, which it has stored somewhere on its hard disk. In the MYEX34A example, the customer's name is used in the hidden field, although it might be safer to use some combination of the name, date, and time. Here's the HTML code that `CMyex34aExtension::ProcessPizzaForm` sends to the customer as part of the confirmation form:

```
<input type="hidden" name="name" value="Walter Sullivan">
```

Here's the code for the `CMyex34aExtension::ConfirmOrder` function:

```
void CMyex34aExtension::ConfirmOrder(CHttpServerContext* pCtxt,
    LPCTSTR pstrName)
{
    StartContent(pCtxt);
    WriteTitle(pCtxt);
}
```

```

    *pCtxt << "<p>Our courteous delivery person will arrive within 30
minutes. ";
    *pCtxt << "<p>Thank you, " << pstrName << ", for using CyberPizza. ";
    // Now retrieve the order from disk by name, and then make the pizza.
    // Be prepared to delete the order after a while if the customer
    // doesn't confirm.
    m_cs.Lock(); // gotta be threadsafe
    long int nTotal = ++m_nTotalPizzaOrders;
    m_cs.Unlock();
    *pCtxt << "<p>Total pizza orders = " << nTotal;
    EndContent(pCtxt);
}

```

The customer's name comes back in the `pstrName` parameter, and that's what you use to retrieve the original order from disk. The function also keeps track of the total number of orders, using a critical section (`m_cs`) to ensure thread synchronization.

Building and Testing myex34a.dll

Building the project adds a DLL to the **Debug** subdirectory. You must copy this DLL to a directory that the server can find and copy **PizzaForm.html** also. You can use the **scripts** and **wwwroot** subdirectory that already under **\Inetpub**, or you can set up new virtual directories.

If you make changes to the MYEX34A DLL in the Visual C++ project, be sure to use Internet Service Manager to turn off the WWW service (because the old DLL stays loaded), copy the new DLL to the **scripts** directory, and then turn the WWW service on again. The revised DLL will be loaded as soon as the first client requests it. If everything has been installed correctly, you should be able to load **PizzaForm.html** from the browser and then order some pizza. Enjoy!

Debugging the MYEX34A DLL

The fact that IIS is a Windows NT service complicates debugging ISAPI DLLs. Services normally run as part of the operating system, controlled by the service manager database. They have their own window station, and they run on their own invisible desktop. This involves some of the murkier parts of Windows NT, and not much published information is available. However, you can use these steps to debug your MYEX34A DLL (or any ISAPI DLL):

1. Use the Internet Service Manager to stop all IIS services.

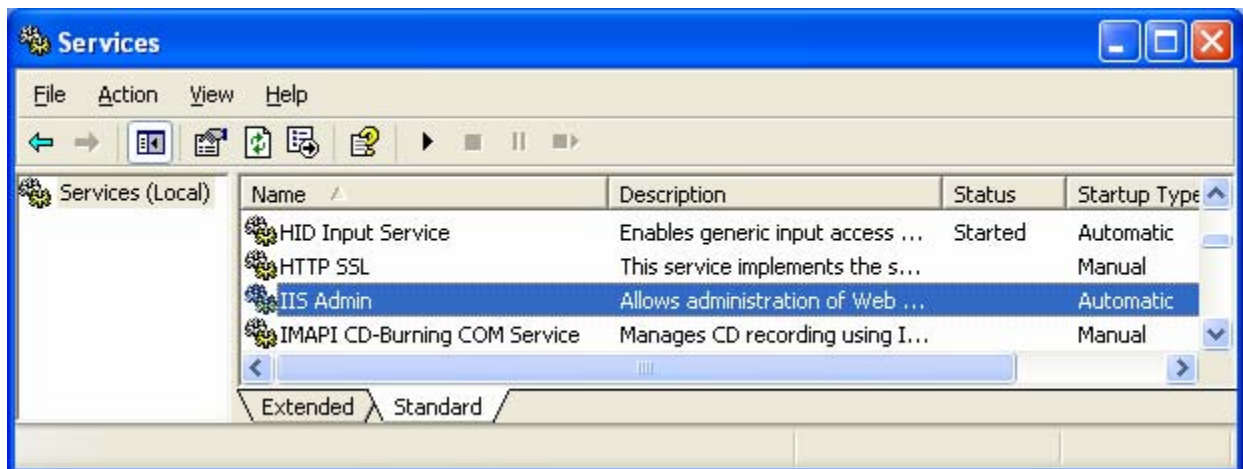


Figure 48: Stopping the IIS Admin and other dependent (World Wide Web publishing) services.

2. Choose **Settings** from the MYEX34A **Project** menu, and in the **Project Settings** dialog, type in the data as shown.

C:\WINDOWS\system32\inetrv\inetinfo.exe or C:\WINNT\system32\inetrv\inetinfo.exe depends on your Windows OS version.

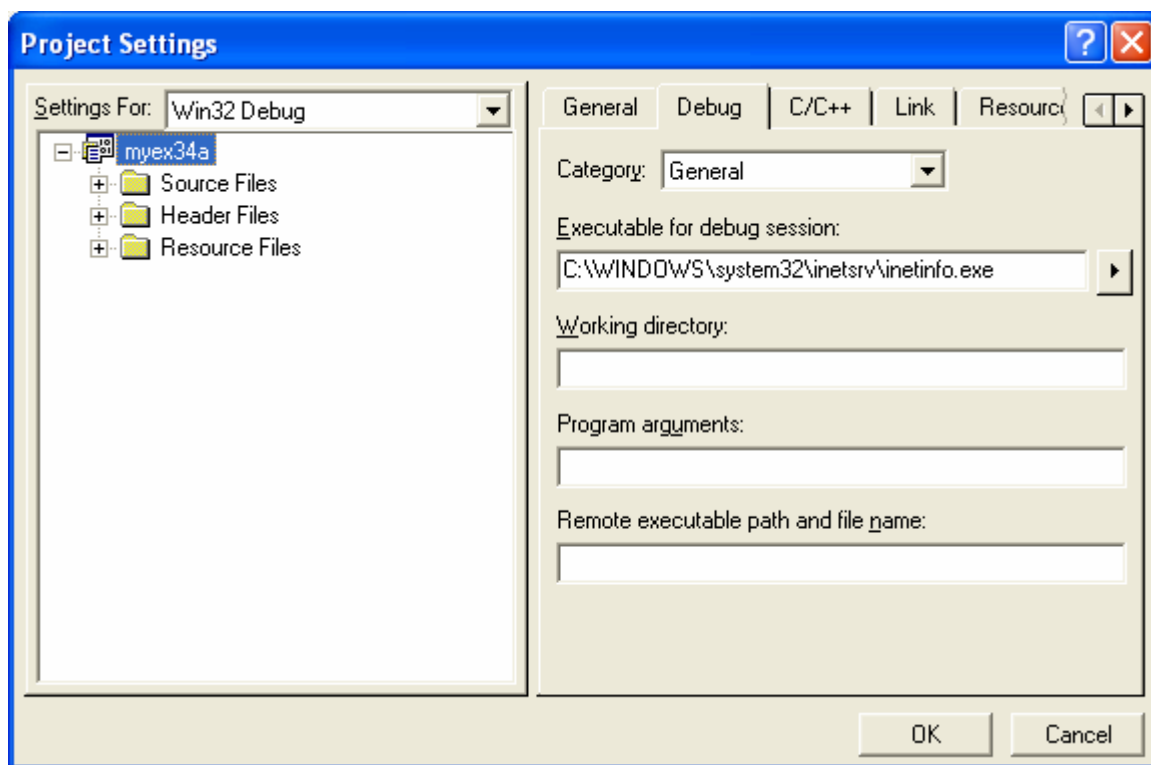


Figure 49: Entering the program that will be invoked during the debug process.

3. Start **Active Directory Users and Computers** (Administrative Tools menu). Choose User Rights from the Policies menu, check Show Advanced User Rights, select the right Act As Part Of The Operating System, and add your user group as shown on the facing page. For Windows 2000 and above you can use **Local Security Policy** (Administrative Tools), select **User Rights Assignment** under **Local Policies**.

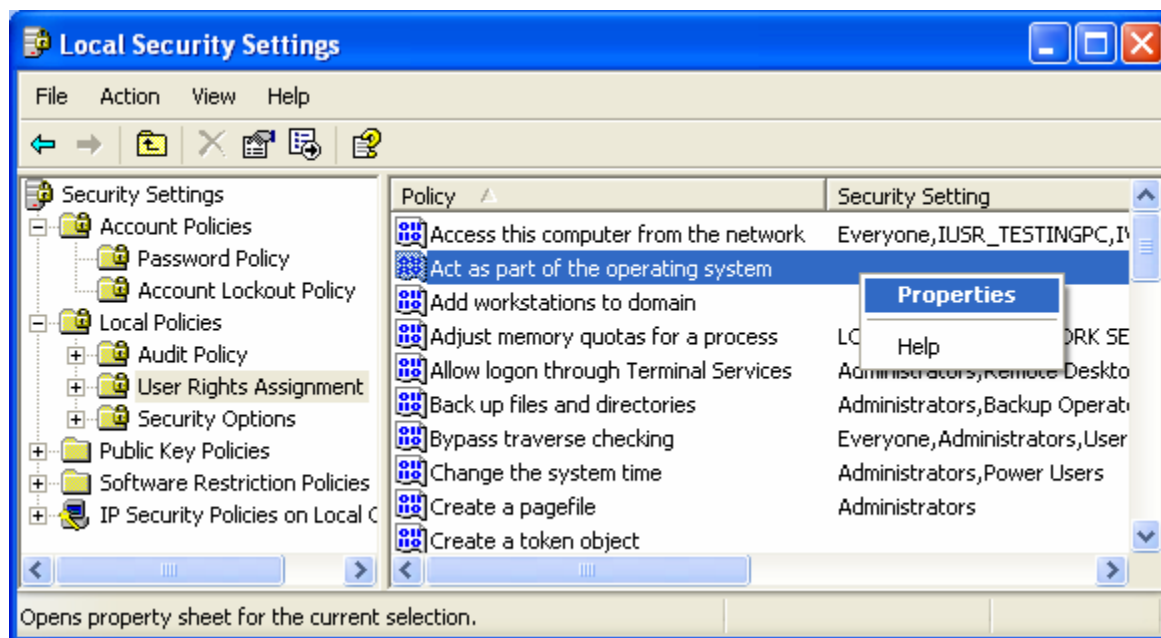


Figure 50: Changing the User Right Assignment setting through Local Security Settings.

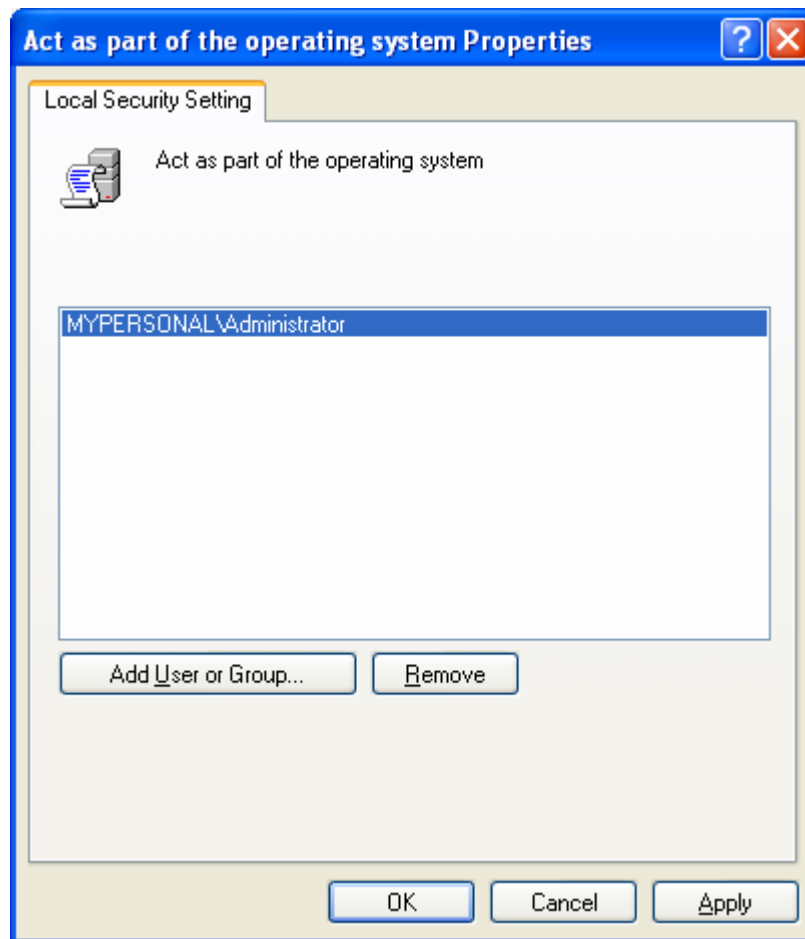


Figure 51: Adding a user to the Act as part of the operating system policy.

4. Repeat step 3 to set the right for **Generate Security Audits**.

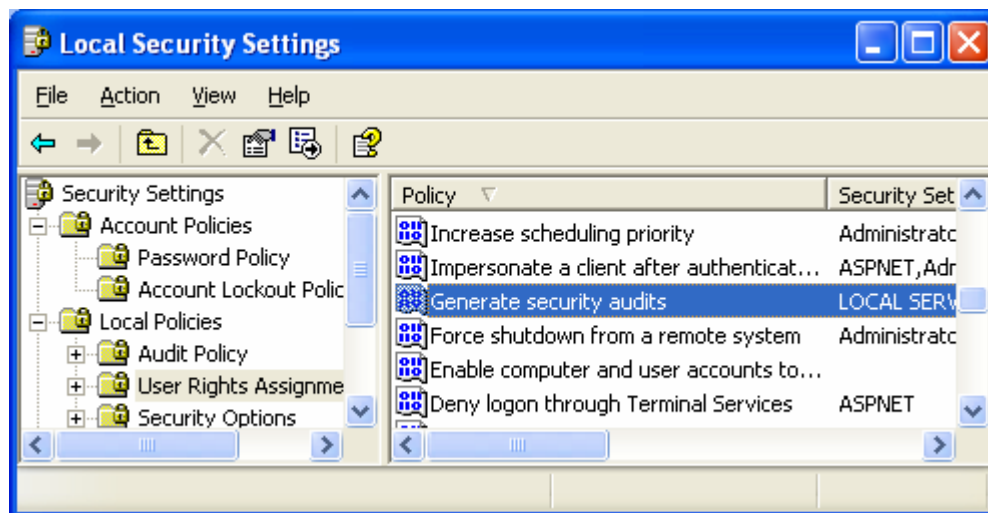


Figure 52: Generate Security Audits policy.

5. Log off and on to Windows NT to activate the new permission. (Don't forget this step.)
6. Make sure that the current MYEX34A DLL file has been copied into the scripts directory.

7. Start debugging. You can set breakpoints, step through code, and see the output of TRACE messages.

ISAPI Database Access

Your ISAPI server extension could use ODBC to access an SQL database. Before you write pages of ODBC code, however, check out the Internet Database Connector (IDC) described in the IIS documentation. The Internet Database Connector is a ready-to-run DLL, **Httpodbc.dll**, that collects SQL query parameters and formats the output. You control the process by writing an IDC file that describes the data source and an HTX file that is a template for the resulting HTML file. No C++ programming is necessary. The Internet Database Connector is for queries only. If you want to update a database, you must write your own ISAPI server extension with ODBC calls. Make sure your ODBC driver is multithreaded, as is the latest SQL server driver.

Using HTTP Cookies to Link Transactions

Now that you've wolfed down the pizza, it's time for some dessert. However, the cookies that we'll be digesting in this section are not made with chocolate chips. **Cookies are used to store information on our customers' hard disks.** In the MYEX34A example, the server stores the customer name in a hidden field of the confirmation form. That works fine for linking the confirmation to the order, but it doesn't help you track how many pizzas Walter ordered this year. If you notice that Walter consistently orders pepperoni pizzas, you might want to send him some e-mail when you have a surplus of pepperoni.

How Cookies Work

With cookies, you assign Walter a customer ID number with his first order and make him keep track of that number on his computer. The server assigns the number by sending a response header such as this one:

```
Set-Cookie: customer_id=12345; path=/; expires=Monday, 02-Sep-05 00:00:00 GMT
```

The string `customer_id` is the arbitrary cookie name you have assigned, the `/` value for path means that the browser sends the cookie value for any request to your site (named **CyberPizza.com**), and the expiration date is necessary for the browser to store the cookie value. When the browser sees the `Set-Cookie` response header, it creates (or replaces) an entry in its **cookies.txt** file as follows:

```
customer_id
12345
cyberpizza.com/
0
2096697344
0
2093550622
35
*
```

Every user profile will have **cookies.txt**. The following Figure shows the example.

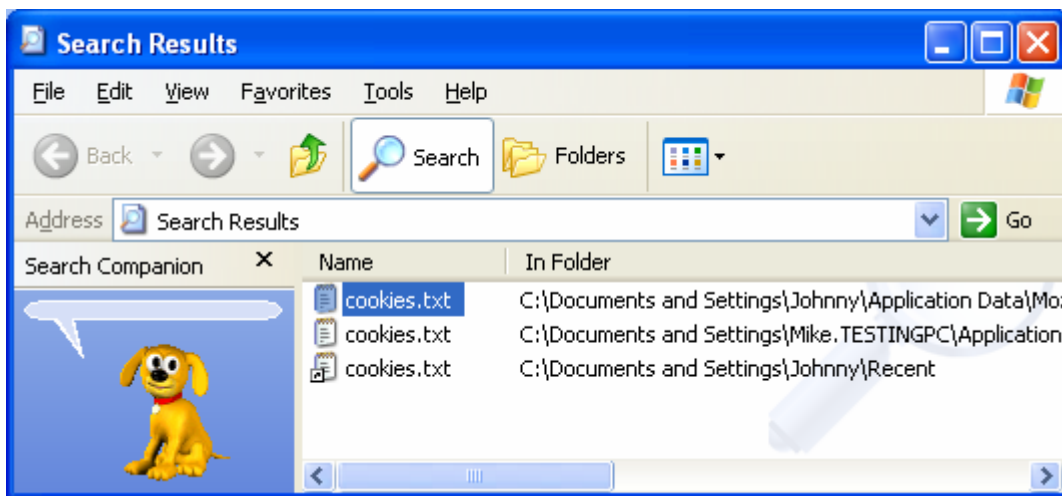


Figure 53: Cookies.txt for user profiles in Windows XP Pro.

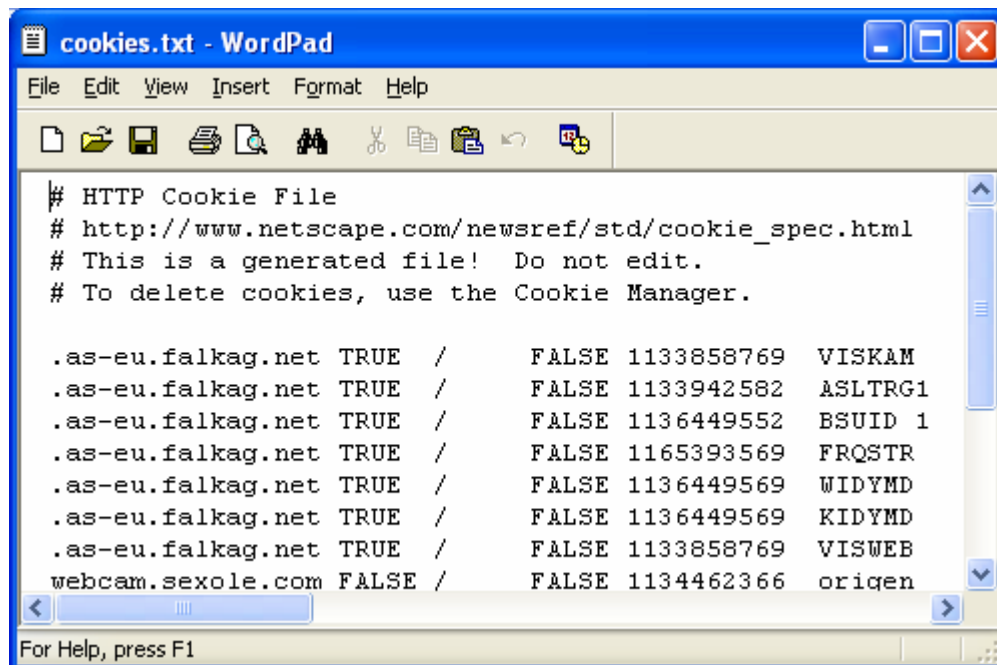


Figure 54: Cookies.txt's content example of the user profile for Firefox browser.

Thereafter, when the browser requests anything from CyberPizza.com, the browser sends a request header like this:

```
Cookie: customer_id=12345
```

How an ISAPI Server Extension Processes Cookies

Your ISAPI server extension function makes a call like this one to store the cookie at the browser:

```
AddHeader(pCtxt, "Set-Cookie: session_id=12345; path=/"
           " expires=Monday, 02-Sep-05 00:00:00 GMT\r\n");
```

To retrieve the cookie, another function uses code like this:

```
char strCookies[200];
DWORD dwLength = 200;
```

```
pCtxt->GetServerVariable("HTTP_COOKIE", strCookies, &dwLength);
```

The strCookies variable should now contain the text customer_id=12345.

Problems with Cookies

There was uproar some time ago when Internet users first discovered that companies were storing data on the users' PCs. New browser versions now ask permission before storing a cookie from a Web site. Customers could thus refuse to accept your cookie, for example, they could erase their **cookies.txt** file, or this file could become full. If you decide to use cookies at your Web site, you'll just have to deal with those possibilities.

WWW Authentication

Up to now, your IIS has been set to allow anonymous logons, which means that anyone in the world can access your server without supplying a user name or password. All users are logged on as IUSR_MYMACHINENAME and can access any files for which that user name has permissions. As stated in [Module 34](#), you should be using NTFS on your server for maximum security.

Basic Authentication

The simplest way to limit server access is to enable basic authentication. Then, if a client makes an anonymous request, the server sends back the response:

```
HTTP/1.0 401 Unauthorized
```

together with a response header like this:

```
WWW-Authenticate: Basic realm="xxx"
```

The client prompts the user for a user name and password, and then it resends the request with a request header something like this:

```
Authorization: Basic 2rc234ldfd8kdr
```

The string that follows **Basic** is a pseudoencrypted version of the user name and password, which the server decodes and uses to impersonate the client. The trouble with basic authentication is that intruders can pick up the user name and password and use it to gain access to your server. IIS and most browsers support basic authentication, but it's not very effective.

Windows NT Challenge/Response Authentication

Windows NT challenge/response authentication is often used for intranets running on Microsoft networks, but you can use it on the Internet as well. IIS supports it, but not all browsers do. If the server has challenge/response activated, a client making an ordinary request gets this response header:

```
WWW-Authenticate: NTLM
Authorization: NTLM T1RMTVNTUAABAAAAA5IAA ...
```

The string after NTLM is the well-encoded user name, the password is never sent over the network. The server issues a challenge, with a response header like this:

```
WWW-Authenticate: NTLM RPTUFJTgAAAAA ...
```

The client, which knows the password, does some math on the challenge code and the password and then sends back a response in a request header like this:

```
Authorization: NTLM AgACAAGAAAAAAAAA ...
```

The server, which has looked up the client's password from the user name, runs the same math on the password and challenge code. It then compares the client's response code against its own result. If the client's and the server's results match, the server honors the client's request by impersonating the client's user name and sending the requested data. When the client resends the request, the challenge/response dialog is performed over a single-socket connection with keep-alive capability as specified in the Connection request header.

WinInet fully supports Windows NT challenge/response authentication. Thus, Internet Explorer 4.0/above and the EX34A WinInet clients support it. If the client computer is logged on to a Windows NT domain, the user name and password are passed through. If the client is on the Internet, WinInet prompts for the user name and password. If you're writing WinInet code, you must use the `INTERNET_FLAG_KEEP_CONNECTION` flag in all `CHttpRequest::OpenRequest` and `CInternetSession::OpenURL` calls, as EX34A illustrates.

The Secure Sockets Layer

Windows NT challenge/response authentication controls only who logs on to a server. Anyone snooping on the Net can read the contents of the TCP/IP segments. The [secure sockets layer](#) (SSL) (the open source version – [OpenSSL](#)) goes one step further and encodes the actual requests and responses (with a performance hit, of course). Both IIS and WinInet support SSL. The secure sockets layer is described in the IIS documentation. The newest one is [Transport Layer Security](#) (TLS).

ISAPI Filters

An ISAPI server extension DLL is loaded the first time a client references it in a GET or POST request. An ISAPI filter DLL is loaded (based on a Registry entry) when the WWW service is started. The filter is then in the loop for all HTTP requests, so you can read and/or change any data that enters or leaves the server.

Writing an ISAPI Filter DLL

The ISAPI Extension Wizard makes writing filters as easy as writing server extensions. Choose **Generate A Filter Object**, and Step 2 looks like this.

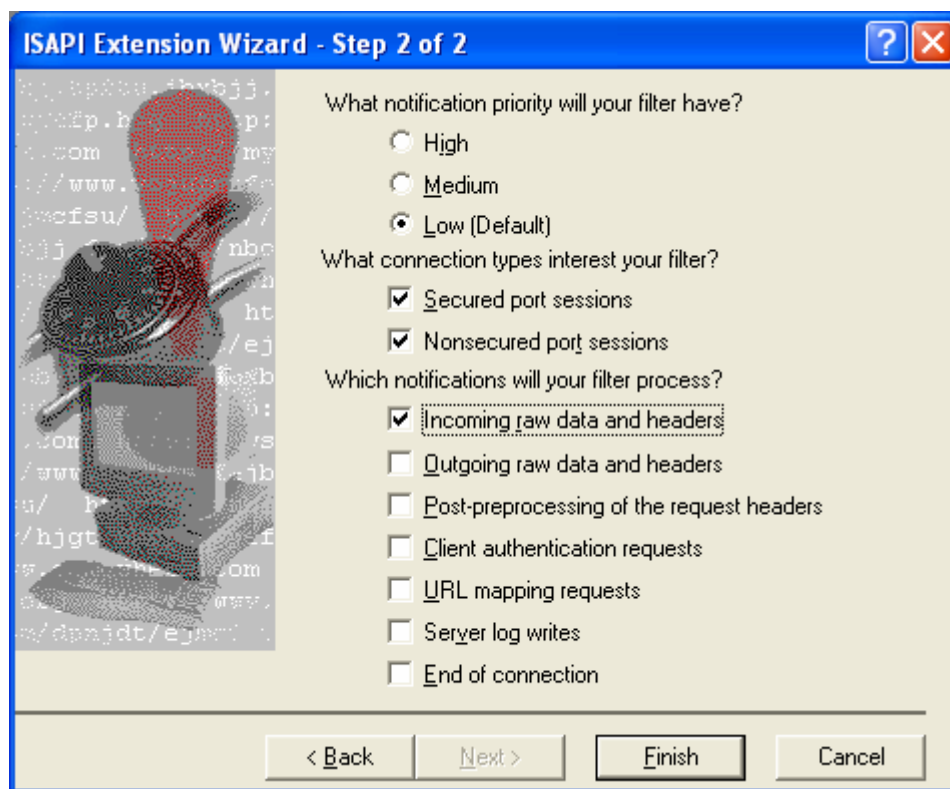


Figure 55: ISAPI Extension Wizard Step 2 of 2.

The list of options under **Which Notifications Will Your Filter Process?** refers to seven places where your filter can get control during the processing of an HTTP request. You check the boxes, and the wizard generates the code.

The MFC ISAPI Filter Classes

There are two MFC classes for ISAPI filters, `CHttpFilter` and `CHttpFilterContext`.

`CHttpFilter`

With the help of the ISAPI Extension Wizard, you derive a class from `CHttpFilter` for each ISAPI filter you create. There's just one object of this class. The class has virtual functions for each of seven notifications. The list of filters in the order in which IIS calls them is below.

```
virtual DWORD OnReadRawData(CHttpFilterContext* pCtxt,
                           PHTTP_FILTER_RAW_DATA pRawData);
virtual DWORD OnPreprocHeaders(CHttpFilterContext* pCtxt,
                              PHTTP_FILTER_PREPROC_HEADERS pHeaderInfo);
virtual DWORD OnUrlMap(CHttpFilterContext* pCtxt,
                      PHTTP_FILTER_URL_MAP pMapInfo);
virtual DWORD OnAuthentication(CHttpFilterContext* pCtxt,
                              PHTTP_FILTER_AUTHENT pAuthent);
virtual DWORD OnSendRawData(CHttpFilterContext* pCtxt,
                           PHTTP_FILTER_RAW_DATA pRawData);
virtual DWORD OnLog(CHttpFilterContext* pCtxt, PHTTP_FILTER_LOG pLog);
virtual DWORD OnEndOfNetSession(CHttpFilterContext* pCtxt);
```

If you override a function, you get control. It would be inefficient, however, if IIS made virtual function calls for every notification for each transaction. Another virtual function, `GetFilterVersion()`, is called once when the filter is loaded. The ISAPI Extension Wizard always overrides this function for you, and it sets flags in the function's `pVer` parameter, depending on which notifications you want. Here's a simplified sample with all the flags set:

```
BOOL CMyFilter::GetFilterVersion(PHTTP_FILTER_VERSION pVer)
{
    CHttpFilter::GetFilterVersion(pVer);
    pVer->dwFlags |= SF_NOTIFY_ORDER_LOW | SF_NOTIFY_NONSECURE_PORT |
        SF_NOTIFY_LOG | SF_NOTIFY_AUTHENTICATION |
        SF_NOTIFY_PREPROC_HEADERS | SF_NOTIFY_READ_RAW_DATA |
        SF_NOTIFY_SEND_RAW_DATA | SF_NOTIFY_URL_MAP |
        SF_NOTIFY_END_OF_NET_SESSION;
    return TRUE;
}
```

If you had specified URL mapping requests only, the wizard would have set only the `SF_NOTIFY_URL_MAP` flag and it would have overridden only `OnUrlMap()`. IIS would not call the other virtual functions, even if they were overridden in your derived class.

`CHttpFilterContext`

An object of this second MFC class exists for each server transaction, and each of the notification functions gives you a pointer to that object. The `CHttpFilterContext` member functions you might call are `GetServerVariable()`, `AddResponseHeaders()`, and `WriteClient()`.

A Sample ISAPI Filter: `myex34b.dll`, `myex34c.exe`

It was hard to come up with a cute application for ISAPI filters. The one we thought up, **`myex34b.dll`**, is a useful visual logging utility. IIS, of course, logs all transactions to a file (or database), but you must stop the server before you can see the log file entries. With this example, you have a real-time transaction viewer that you can customize.

MYEX34B From Scratch: ISAPI Filter

The following shows the steps to build MYEX34B, an ISAPI filter from scratch. Follow the shown steps.

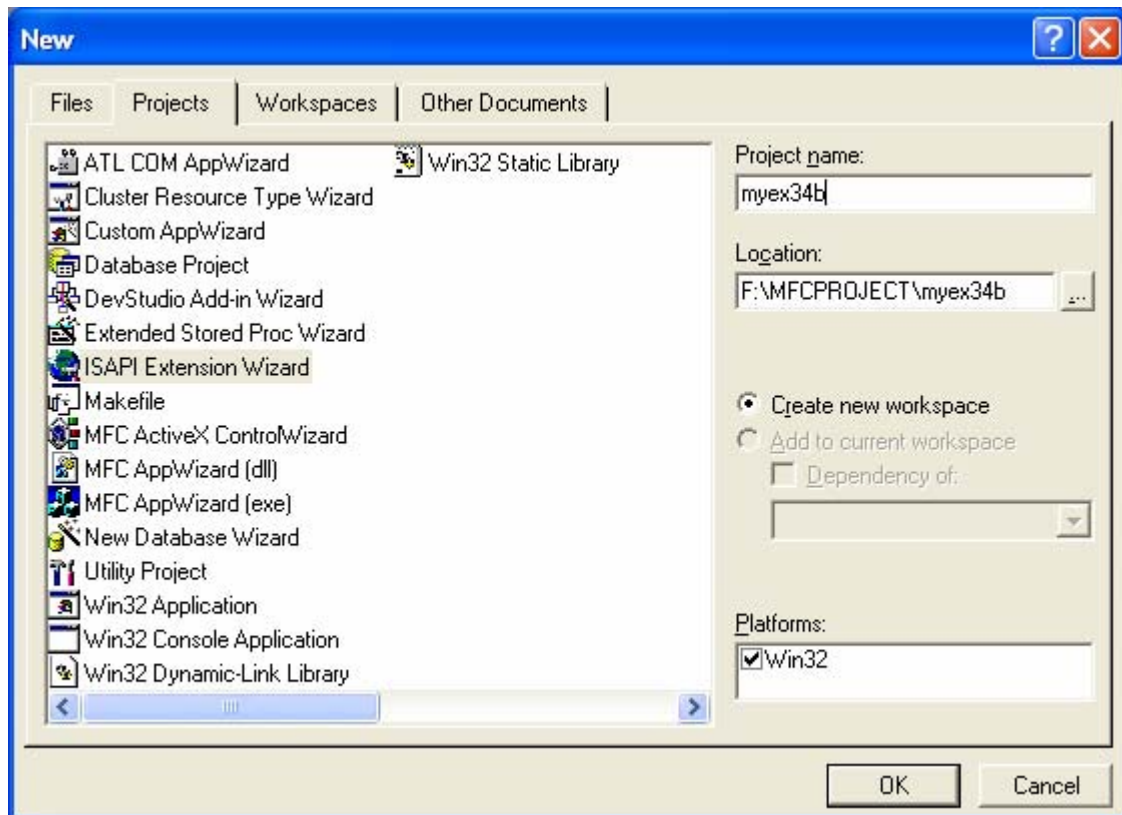


Figure 56: MYEX34B – ISAPI Extension Wizard new project dialog.

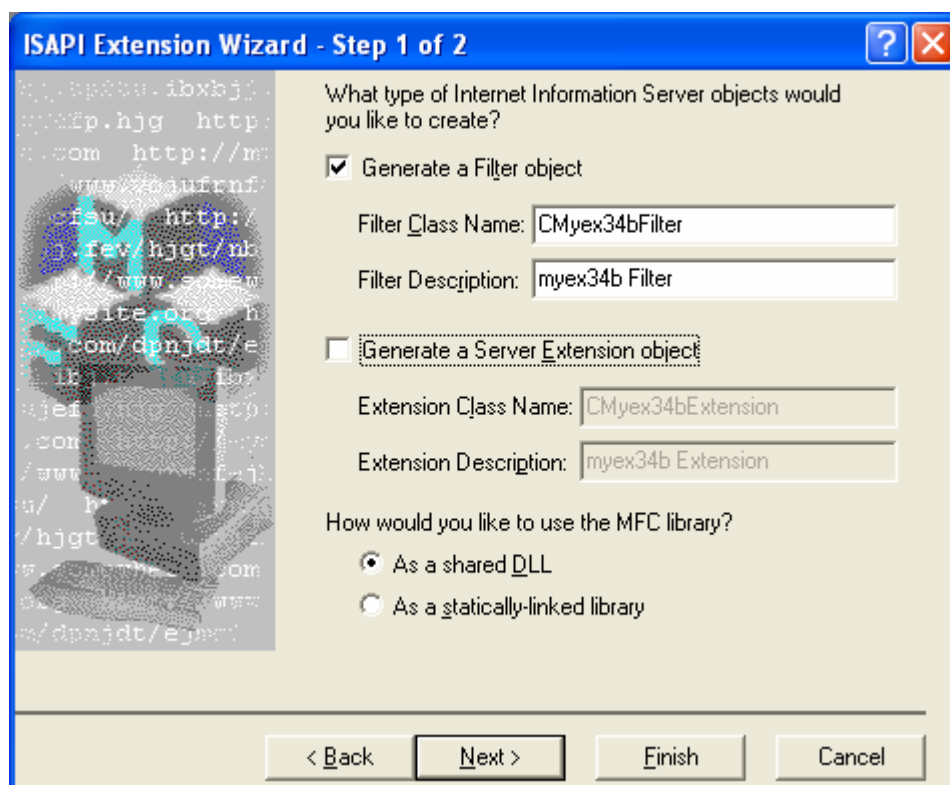


Figure 57: MYEX34B – ISAPI Extension Wizard step 1 of 2, selecting **Generate a Filter object** option.

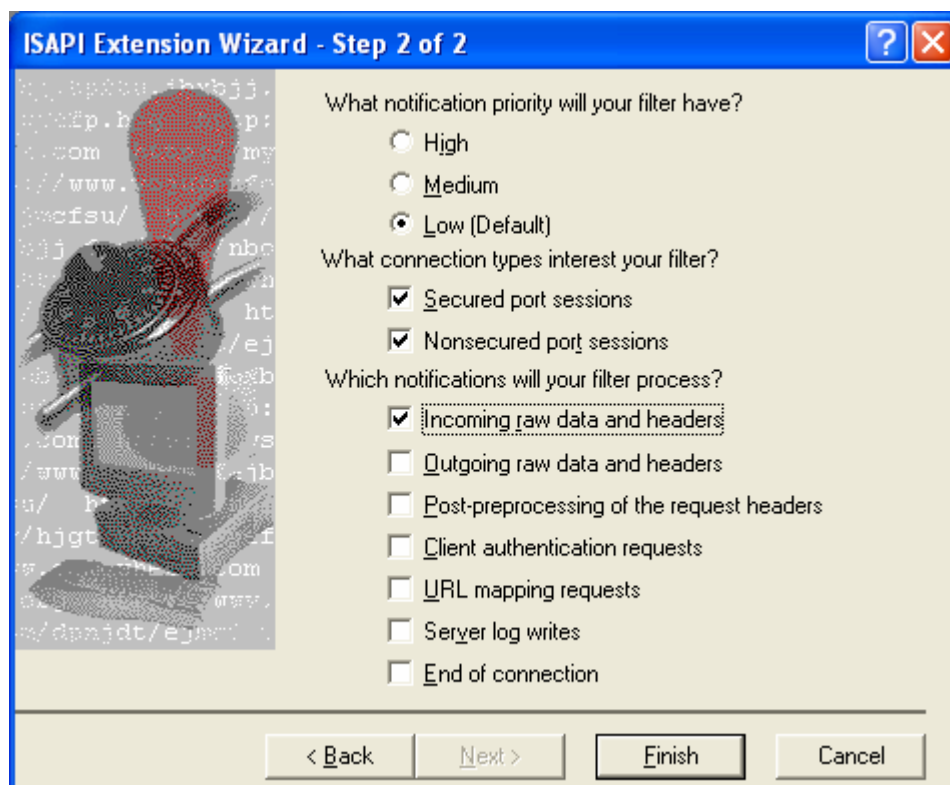


Figure 58: MYEX34B – ISAPI Extension Wizard step 2 of 2, selecting the notification priority, connection type and notification type options.

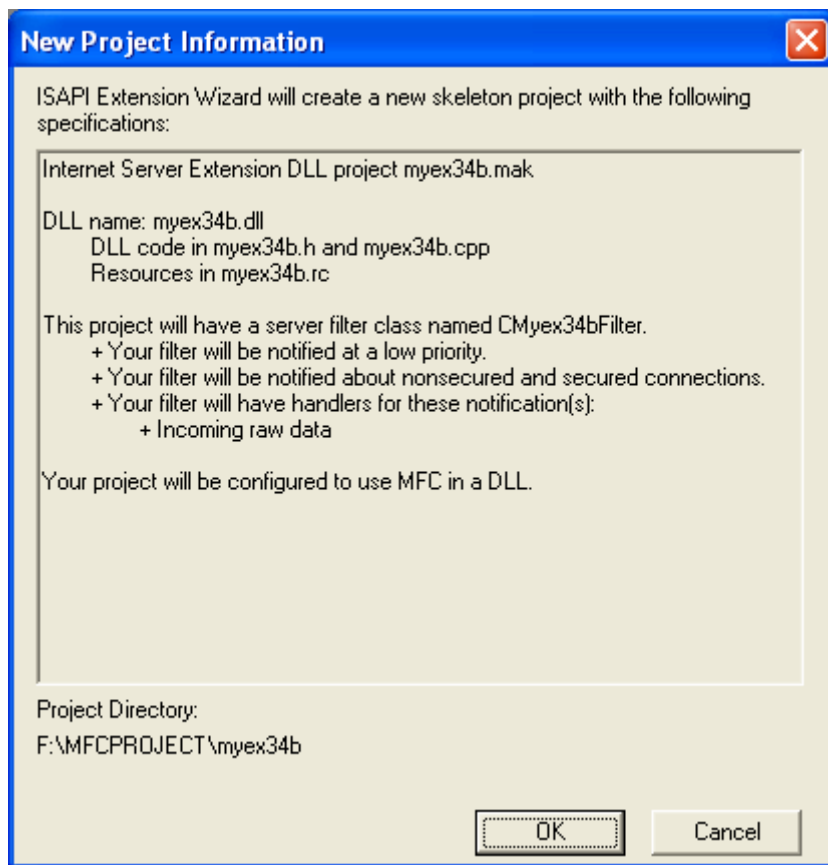


Figure 59: MYEX34B project summary.

Using ClassView, add the following two private member variables and a helper function.

```
HWND    m_hWndDest;
HANDLE  m_hProcessDest;
void SendTextToWindow(char* pchData);
```

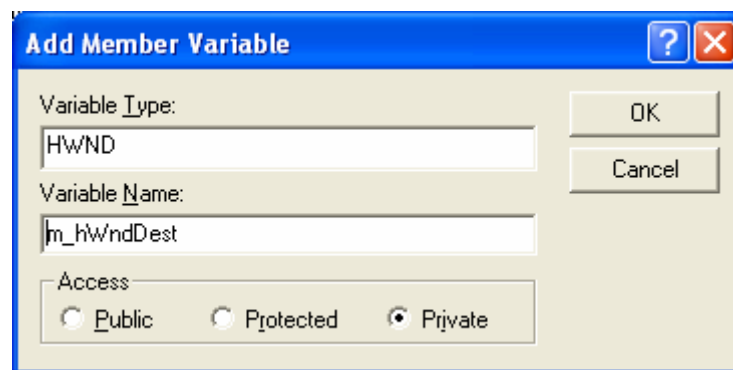


Figure 60: Adding m_hWndDest, a member variable through ClassView.

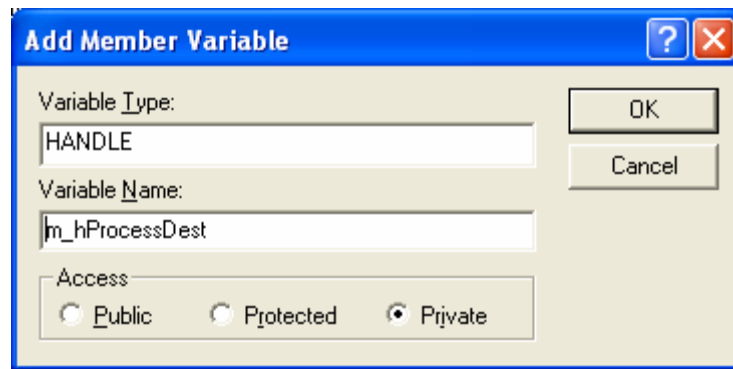


Figure 61: Adding m_hProcessDest member variable.

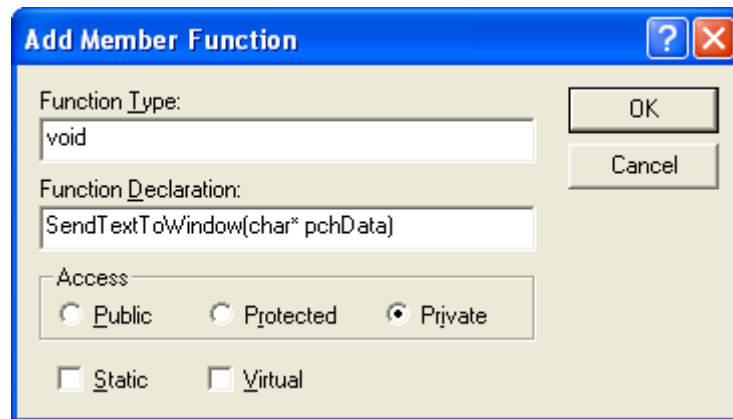


Figure 62: Adding SendToWindow() member function.

```
//}}AFX_MSG
private:
    void SendTextToWindow(char* pchData);
    HANDLE m_hProcessDest;
    HWND m_hWndDest;
};
//{{AFX_INSERT_LOCATION}} ...
```

Listing 14.

In myex34b.cpp, define the following constant.

```
#define WM_SENDTEXT WM_USER + 5

#include "stdafx.h"
#include "myex34b.h"

#define WM_SENDTEXT WM_USER + 5

////////////////////////////////////
// The one and only CWinApp object
```

Listing 15.

Add the following code at the end but before the return statement of the GetFilterVersion().

```

...
...
// m_hWndDest, m_hProcessDest used by SendTextToWindow
m_hProcessDest = NULL;
if((m_hWndDest = ::FindWindow(NULL, "myex34c")) != NULL)
{
    DWORD dwProcessId;
    GetWindowThreadProcessId(m_hWndDest, &dwProcessId);
    m_hProcessDest = OpenProcess(PROCESS_DUP_HANDLE, FALSE,
        dwProcessId);
    SendTextToWindow("MYEX34B filter started.\r\n");
}

    IDS_FILTER, sz, SF_MAX_FILTER_DESC_LEN));
_tcscpy(pVer->lpszFilterDesc, sz);

// m_hWndDest, m_hProcessDest used by SendTextToWindow
m_hProcessDest = NULL;
if((m_hWndDest = ::FindWindow(NULL, "myex34c")) != NULL)
{
    TRACE("The myex34c window found!\n");
    DWORD dwProcessId;
    GetWindowThreadProcessId(m_hWndDest, &dwProcessId);
    m_hProcessDest = OpenProcess(PROCESS_DUP_HANDLE, FALSE,
        dwProcessId);
    SendTextToWindow("MYEX34B filter started.\r\n");
}
return TRUE;
}

```

Listing 16.

Edit the OnReadRawData() as shown below.

```

DWORD CMyex34bFilter::OnReadRawData(CHttpFilterContext* pCtxt,
    PHTTP_FILTER_RAW_DATA pRawData)
{
    // TODO: React to this notification accordingly and
    // return the appropriate status code
    TRACE("CMyex34bFilter::OnReadRawData\n");
    // sends time/date, from IP, to IP, request data to a window
    char pchVar[50] = "";
    char pchOut[2000];
    DWORD dwSize = 50;
    BOOL bRet;
    CString strGmt = CTime::GetCurrentTime().FormatGmt("%m/%d/%y %H:%M:%S \
        GMT");
    strcpy(pchOut, strGmt);
    bRet = pCtxt->GetServerVariable("REMOTE_ADDR", pchVar, &dwSize);
    if(bRet && dwSize > 1) {
        strcat(pchOut, ", From ");
        strcat(pchOut, pchVar);
    }
    bRet = pCtxt->GetServerVariable("SERVER_NAME", pchVar, &dwSize);
    if(bRet && dwSize > 1) {
        strcat(pchOut, ", To ");
        strcat(pchOut, pchVar);
    }
    strcat(pchOut, "\r\n");
    int nLength = strlen(pchOut);
    // Raw data is not zero-terminated

```

```

        strncat(pchOut, (const char*) pRawData->pvInData, pRawData->cbInData);
        nLength += pRawData->cbInData;
        pchOut[nLength] = '\\0';
        SendTextToWindow(pchOut);
        return SF_STATUS_REQ_NEXT_NOTIFICATION;
    }

```

Edit the helper function, `SendTextToWindow()` as shown below.

```

void CMyex34bFilter::SendTextToWindow(char *pchData)
{
    if(m_hProcessDest != NULL) {
        int nSize = strlen(pchData) + 1;
        HANDLE hMMFReceiver;
        HANDLE hMMF = ::CreateFileMapping((HANDLE) 0xFFFFFFFF, NULL,
            PAGE_READWRITE, 0, nSize, NULL);
        ASSERT(hMMF != NULL);
        LPVOID lpvFile = ::MapViewOfFile(hMMF, FILE_MAP_WRITE, 0, 0, nSize);
        ASSERT(lpvFile != NULL);
        memcpy((char*) lpvFile, pchData, nSize);
        ::DuplicateHandle(::GetCurrentProcess(), hMMF, m_hProcessDest,
            &hMMFReceiver, 0, FALSE, DUPLICATE_SAME_ACCESS |
DUPLICATE_CLOSE_SOURCE);
        ::PostMessage(m_hWndDest, WM_SENDTEXT, (WPARAM) 0, (LPARAM) hMMFReceiver);
        ::UnmapViewOfFile(lpvFile);
    }
}

```

Build MYEX34B, generating the **myex34b.dll**. We need another program, MYEX34C, to display the transaction data/messages. Before that let have some story about MYEX34B.

Choosing the Notification

Start by looking at the list of `CHttpFilter` virtual member functions. Observe the calling sequence and the parameters. For the MYEX34B logging application, we chose `OnReadRawData()` because it allowed full access to the incoming request and header text (from `pRawData`) and to the source and destination IP addresses (from `pCtx->GetServerVariable`).

Sending Transaction Data to the Display Program

The ISAPI filter DLL can't display the transactions directly because it runs (as part of the IIS service process) on an invisible desktop. You need a separate program that displays text in a window, and you need a way to send data from the DLL to the display program. There are various ways to send the data across the process boundary. A conversation with Jeff Richter, the Windows guru who wrote *Advanced Windows* (Microsoft Press, 1997), led to the data being put in shared memory. Then a user-defined message, `WM_SENDTEXT`, is posted to the display program. These messages can queue up, so IIS can keep going independently of the display program.

We declared two handle data members in `CMyex34bFilter::m_hProcessDest` and `CMyex34bFilter::m_hWndDest`. We added code at the end of the `GetFilterVersion()` function to set these data members to the display program's process ID and main window handle. The code finds the display program's main window by its **title** (window's title), `myex34c`, and then it gets the display program's process ID.

```

m_hProcessDest = NULL;
if((m_hWndDest = ::FindWindow(NULL, "myex34c")) != NULL)
{
    DWORD dwProcessId;
    GetWindowThreadProcessId(m_hWndDest, &dwProcessId);
    m_hProcessDest = OpenProcess(PROCESS_DUP_HANDLE, FALSE, dwProcessId);
    SendTextToWindow("MYEX34B filter started\r\n");
}

```

```
}
```

Below is a helper function, `SendTextToWindow()`, which sends the `WM_SENDTEXT` message to the display program.

```
void CMyex34bFilter::SendTextToWindow(char* pchData)
{
    if(m_hProcessDest != NULL) {
        int nSize = strlen(pchData) + 1;

        HANDLE hMMFReceiver;
        HANDLE hMMF = ::CreateFileMapping((HANDLE) 0xFFFFFFFF, NULL,
            PAGE_READWRITE, 0, nSize, NULL);
        ASSERT(hMMF != NULL);
        LPVOID lpvFile = ::MapViewOfFile(hMMF, FILE_MAP_WRITE, 0, 0,
nSize);
        ASSERT(lpvFile != NULL);
        memcpy((char*) lpvFile, pchData, nSize);
        ::DuplicateHandle(::GetCurrentProcess(), hMMF, m_hProcessDest,
            &hMMFReceiver, 0, FALSE, DUPLICATE_SAME_ACCESS |
            DUPLICATE_CLOSE_SOURCE);
        ::PostMessage(m_hWndDest, WM_SENDTEXT, (WPARAM) 0,
            (LPARAM) hMMFReceiver);
        ::UnmapViewOfFile(lpvFile);
    }
}
```

The `DuplicateHandle()` function makes a copy of MYEX34B's map handle, which it sends to the MYEX34C program in a message parameter. The MYEX34C process ID, determined in `GetFilterVersion()`, is necessary for the `DuplicateHandle()` call. Here is the filter's `OnReadRawData()` function, which calls `SendTextToWindow()`:

```
DWORD CMyex34bFilter::OnReadRawData(CHttpFilterContext* pCtxt,
    PHTTP_FILTER_RAW_DATA pRawData)
{
    TRACE ("CMyex34bFilter::OnReadRawData\n");
    // sends time/date, from IP, to IP, request data to a window
    char pchVar[50] = "";
    char pchOut[2000];
    DWORD dwSize = 50;
    BOOL bRet;
    CString strGmt = CTime::GetCurrentTime().FormatGmt("%m/%d/%y
%H:%M:%S GMT");
    strcpy(pchOut, strGmt);
    bRet = pCtxt->GetServerVariable("REMOTE_ADDR", pchVar, &dwSize);
    if(bRet && dwSize > 1) {
        strcat(pchOut, ", From ");
        strcat(pchOut, pchVar);
    }
    bRet = pCtxt->GetServerVariable("SERVER_NAME", pchVar, &dwSize);
    if(bRet && dwSize > 1) {
        strcat(pchOut, ", To ");
        strcat(pchOut, pchVar);
    }
    strcat(pchOut, "\r\n");
    int nLength = strlen(pchOut);
    // Raw data is not zero-terminated
    strncat(pchOut, (const char*) pRawData->pvInData, pRawData->cbInData);
    nLength += pRawData->cbInData;
```

```

    pchOut[nLength] = '\\0';
    SendTextToWindow(pchOut);
    return SF_STATUS_REQ_NEXT_NOTIFICATION;
}

```

MYEX34C From Scratch

The following are the steps to build MYEX34C. Follow the shown steps.

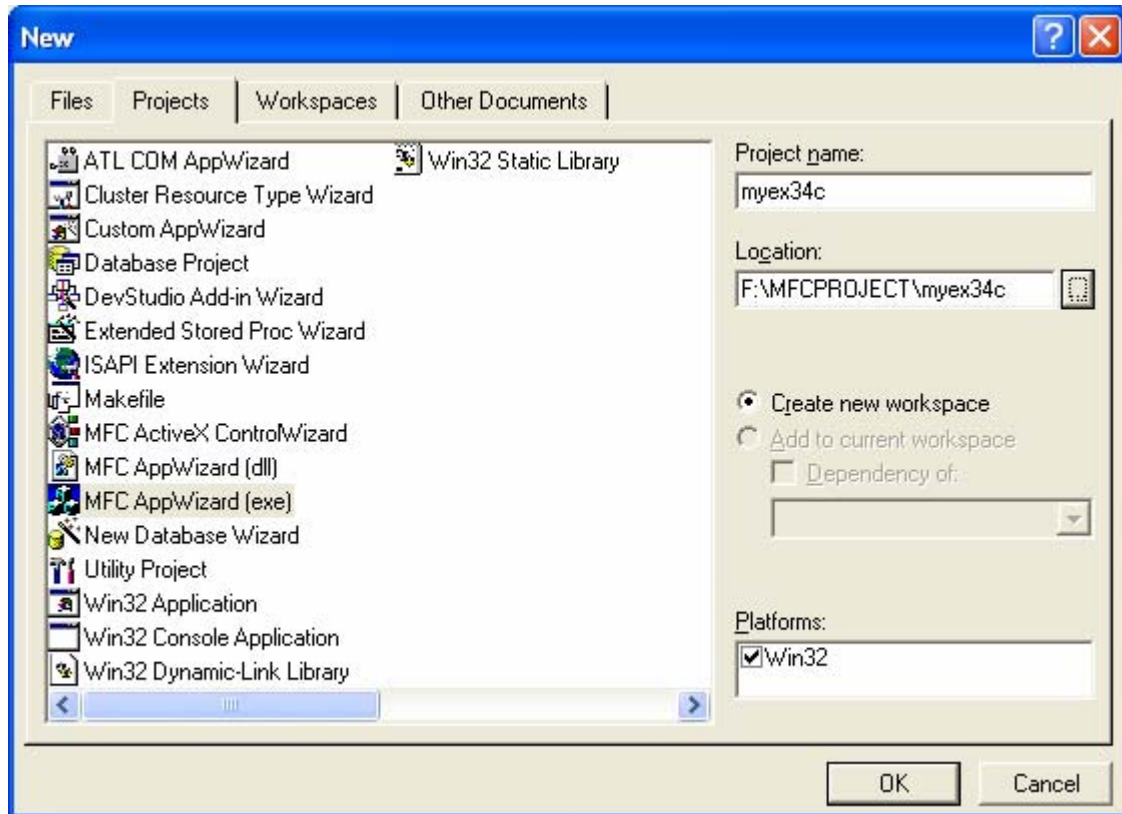


Figure 63: MYEX34C – new MFC AppWizard project dialog.

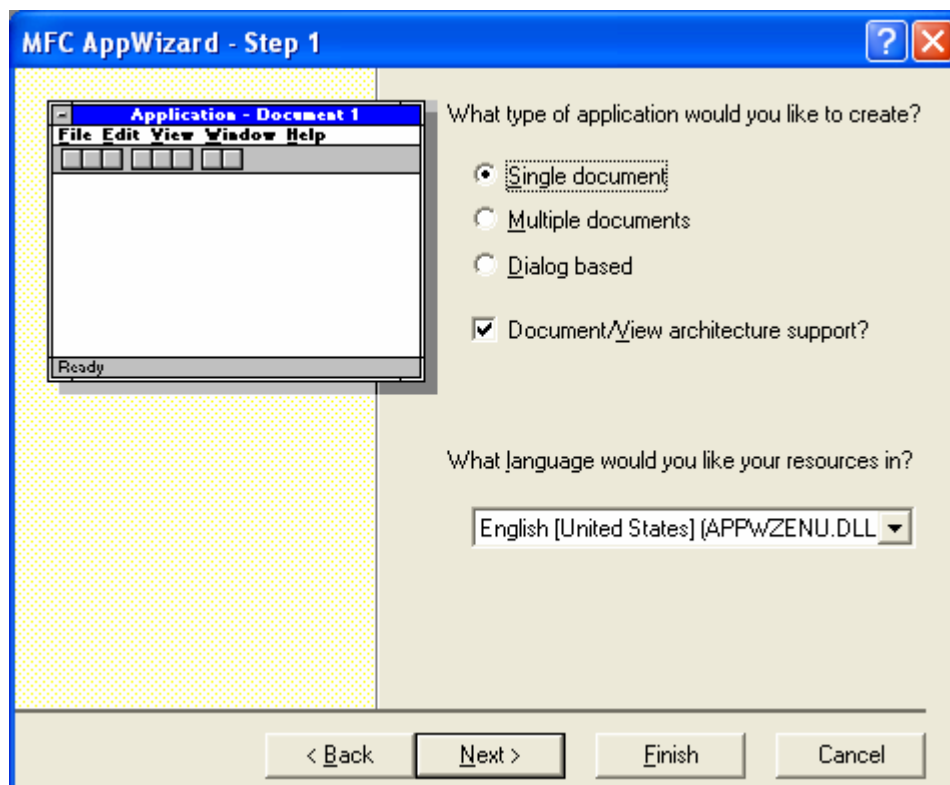


Figure 64: MYEX34C – Step 1 of 6, selecting SDI application.

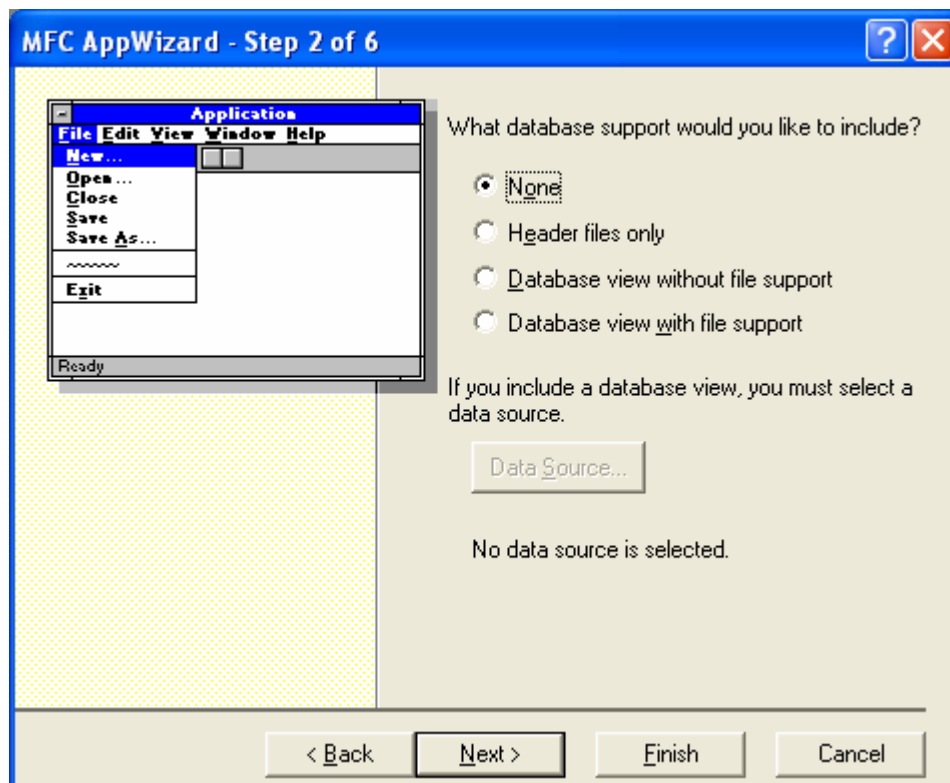


Figure 65: MYEX34C – Step 2 of 6.

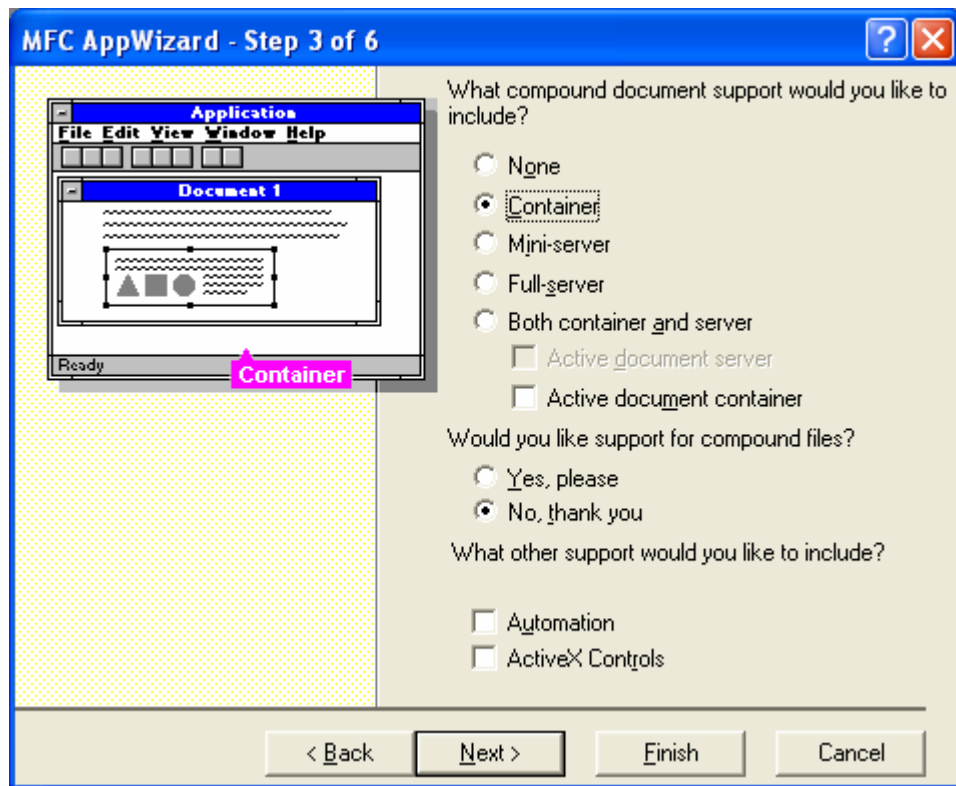


Figure 66: MYEX34C – Step 3 of 6, selecting Container in order to use CRichEditView class, deselect the Automation and ActiveX Controls.

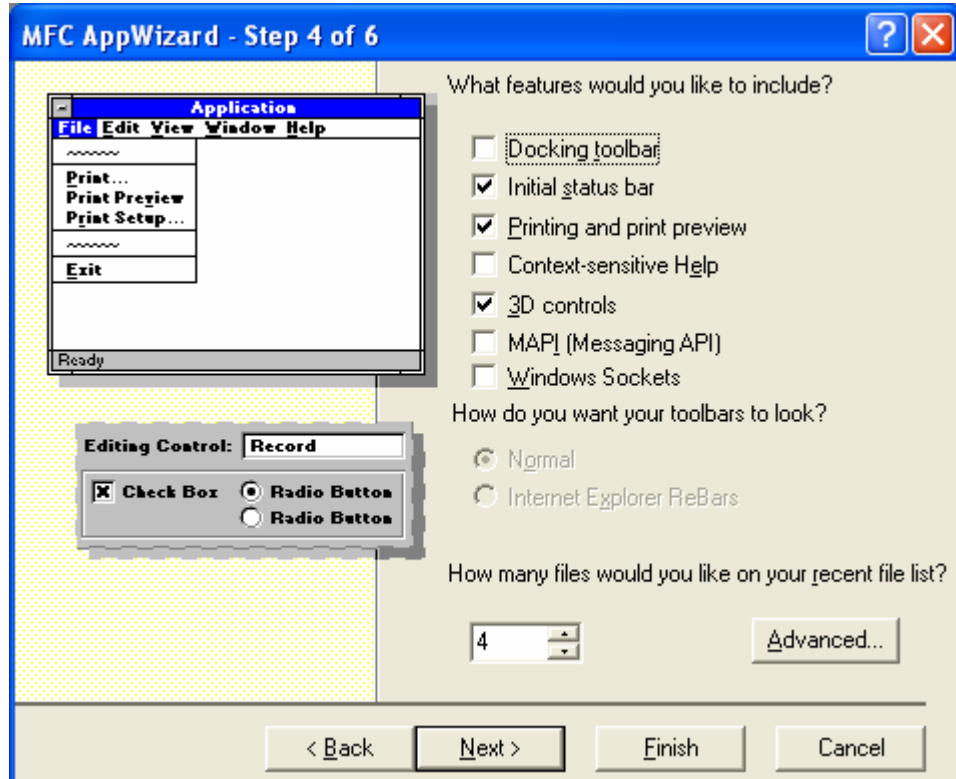


Figure 67: MYEX34C – Step 4 of 6, selecting SDI application, no Docking toolbar feature.

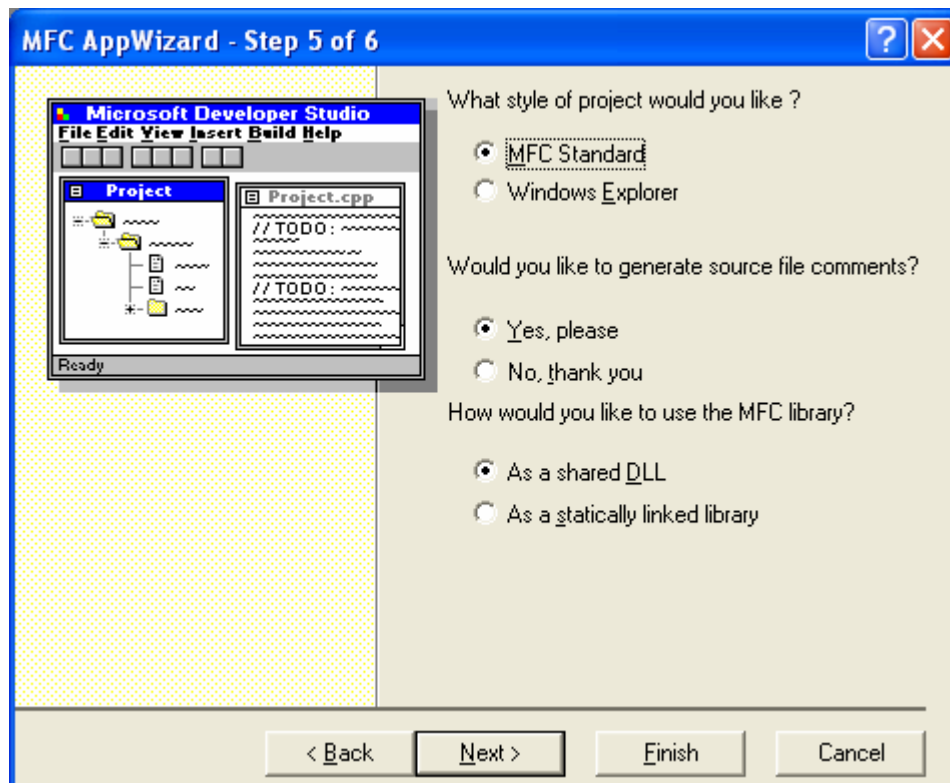


Figure 68: MYEX34C – Step 5 of 6.

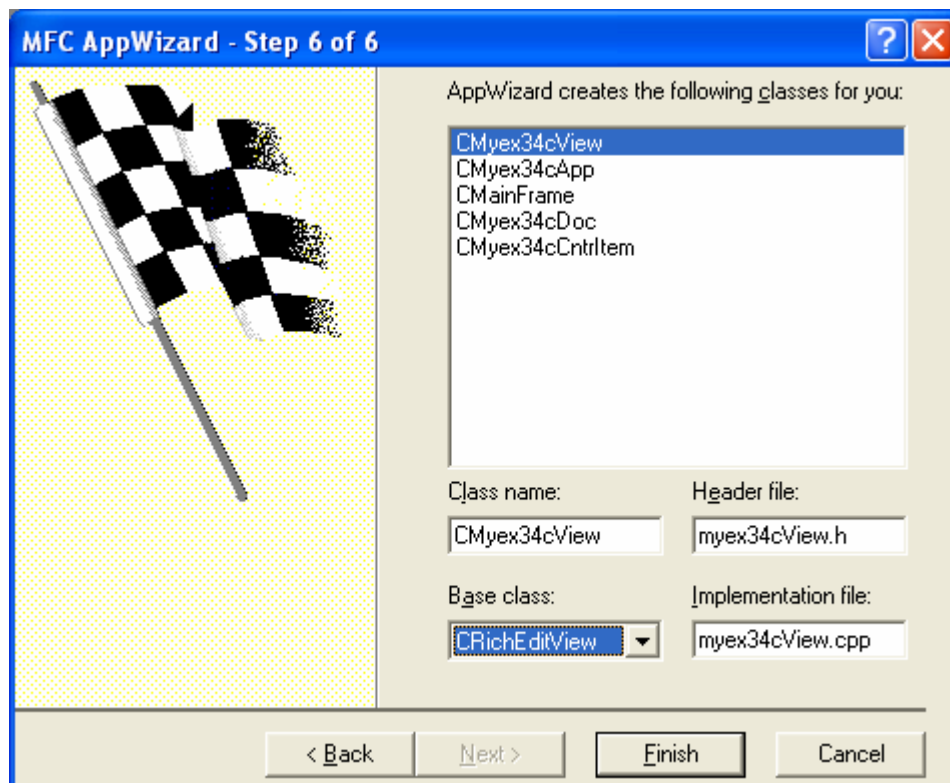


Figure 69: MYEX34C – Step 6 of 6, selecting CRichEditView for Doc and View as the base class.

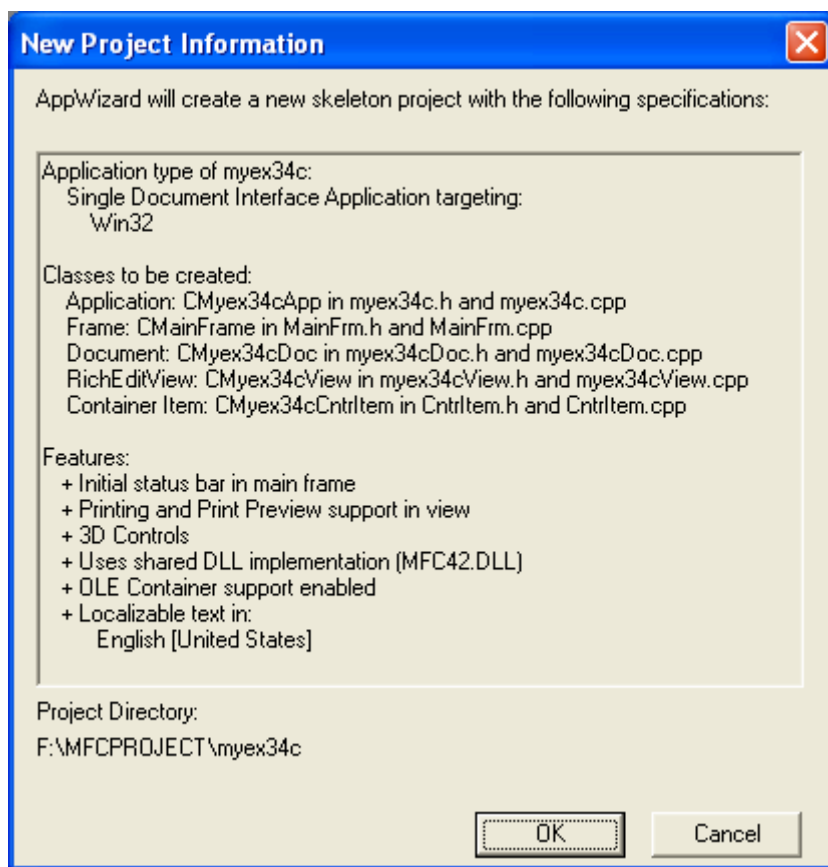


Figure 70: MYEX34C – project summary.

Add the following context menu.

ID	Caption
IDR_MENUCONTEXT	X
ID_EDIT_CLEAR_ALL	&Clear All

Table 1.

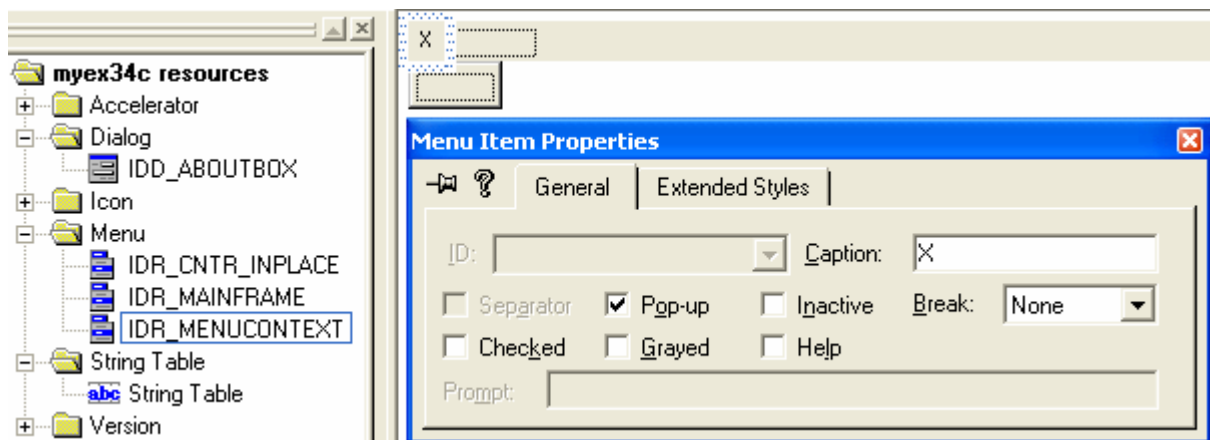


Figure 71: X menu property page.

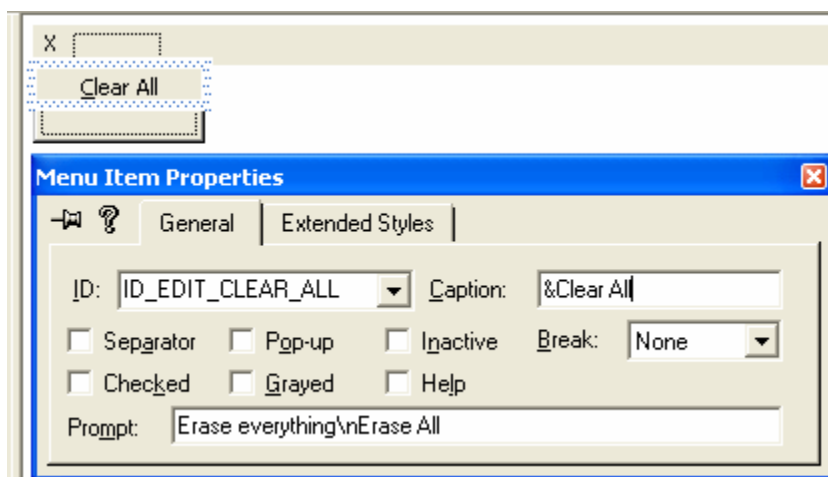


Figure 72: **Clear All** menu item property page.

Just delete the **View** menu for the `IDR_MAINFRAME`, because we don't need the **Status bar**.

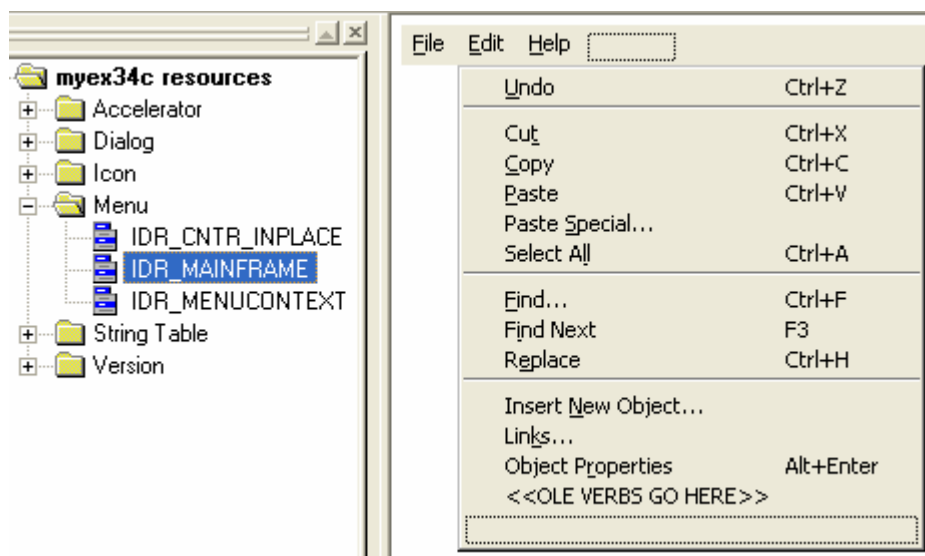


Figure 73: Deleting the **View** menu.

CMainFrame Class

Define WM_SENDTEXT, a user define Windows message as shown below in **MainFrm.h**.

```
#define WM_SENDTEXT WM_USER + 5

// MainFrm.h : interface of the CMainFrm
//
////////////////////////////////////

#define WM_SENDTEXT WM_USER + 5

#ifdef AFX_MAINFRM_H_1527736F_90B
#define AFX_MAINFRM_H_1527736F_90B
```

Listing 17.

Delete the `OnCreate()`. Add `OnSendText()` message map function and `OnUpdateFrameTitle()` virtual function. Unfortunately the `OnUpdateFrameTitle()` function not available in ClassWizard of my Visual C++ 6.0, may be available in Visual Studio or VC++ with Service Pack. This function added manually.

```
virtual void OnUpdateFrameTitle(BOOL bAddToTitle);

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CMainFrame)
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
virtual void OnUpdateFrameTitle(BOOL bAddToTitle);
//}}AFX_VIRTUAL

// Implementation
public:
```

Listing 18.

```
afx_msg LONG OnSendText(UINT wParam, LONG lParam);
```

```
// Generated message map functions
protected:
//{{AFX_MSG(CMainFrame)
//}}AFX_MSG
afx_msg LONG OnSendText(UINT wParam, LONG lParam);
DECLARE_MESSAGE_MAP()
};
```

Listing 19.

Add the following `#include` directive to **MainFrm.cpp**.

```
#include <afxpriv.h> // for AfxSetWindowText

#include "stdafx.h"
#include <afxpriv.h> // for AfxSetWindowText
#include "myex34c.h"

#include "MainFrm.h"
```

Listing 20.

Add the message map manually.

```
ON_MESSAGE(WM_SENDEXT, OnSendText)

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
//{{AFX_MSG_MAP(CMainFrame)
//}}AFX_MSG_MAP
ON_MESSAGE(WM_SENDEXT, OnSendText)
END_MESSAGE_MAP()
```

Listing 21.

Just delete the `OnCreate()` and edit `PreCreateWindow()` as shown below.

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
```

```

        return CFrameWnd::PreCreateWindow(cs);
    }

```

Add the following message handlers' implementation.

```

void CMainFrame::OnUpdateFrameTitle(BOOL bAddToTitle)
{
    // the ISAPI filter needs to find this window by name
    // just display the app name without the file name
    TRACE("CFrameWnd::OnUpdateFrameTitle\n");
    AfxSetWindowText(m_hWnd, AfxGetApp()->m_pszAppName);
}

LONG CMainFrame::OnSendText(UINT wParam, LONG lParam)
{
    TRACE("CMainFrame::OnSendText\n");
    LPVOID lpvFile = ::MapViewOfFile((HANDLE) lParam, FILE_MAP_READ, 0, 0,
    0);
    GetActiveView()->SendMessage(EM_SETSEL, (WPARAM) 999999, 1000000);
    GetActiveView()->SendMessage(EM_REPLACESEL, (WPARAM) 0,
    (LPARAM) lpvFile);
    ::UnmapViewOfFile(lpvFile);
    ::CloseHandle((HANDLE) lParam);

    return 0;
}

// CMainFrame message handlers
void CMainFrame::OnUpdateFrameTitle(BOOL bAddToTitle)
{
    // the ISAPI filter needs to find this window by name
    // just display the app name without the file name
    TRACE("CFrameWnd::OnUpdateFrameTitle\n");
    AfxSetWindowText(m_hWnd, AfxGetApp()->m_pszAppName);
}

LONG CMainFrame::OnSendText(UINT wParam, LONG lParam)
{
    TRACE("CMainFrame::OnSendText\n");
    LPVOID lpvFile = ::MapViewOfFile((HANDLE) lParam, FILE_MAP_READ, 0, 0,
    0);
    GetActiveView()->SendMessage(EM_SETSEL, (WPARAM) 999999, 1000000);
    GetActiveView()->SendMessage(EM_REPLACESEL, (WPARAM) 0,
    (LPARAM) lpvFile);
    ::UnmapViewOfFile(lpvFile);
    ::CloseHandle((HANDLE) lParam);

    return 0;
}

```

Listing 22.

CMyex34cApp Class

Add the following code after the InitInstance().

```
AfxEnableControlContainer();
```

```

BOOL CMyex34cApp::InitInstance()
{
    // Initialize OLE libraries
    if (!AfxOleInit())
    {
        AfxMessageBox(IDP_OLE_INIT_FAILED);
        return FALSE;
    }

    AfxEnableControlContainer();
    |
    // Standard initialization

```

Listing 23.

Add the following code just after the `AddDocTemplate()`.

```

    // Enable DDE Execute open
    EnableShellOpen();
    RegisterShellFileTypes(TRUE);

```

```

AddDocTemplate(pDocTemplate);

// Enable DDE Execute open
EnableShellOpen();
RegisterShellFileTypes(TRUE);
|
// Parse command line for stan
CCommandLineInfo cmdInfo;

```

Listing 24.

CMyex34cDoc Class

Using ClassWizard or ClassView, add `SaveModified()` virtual function.

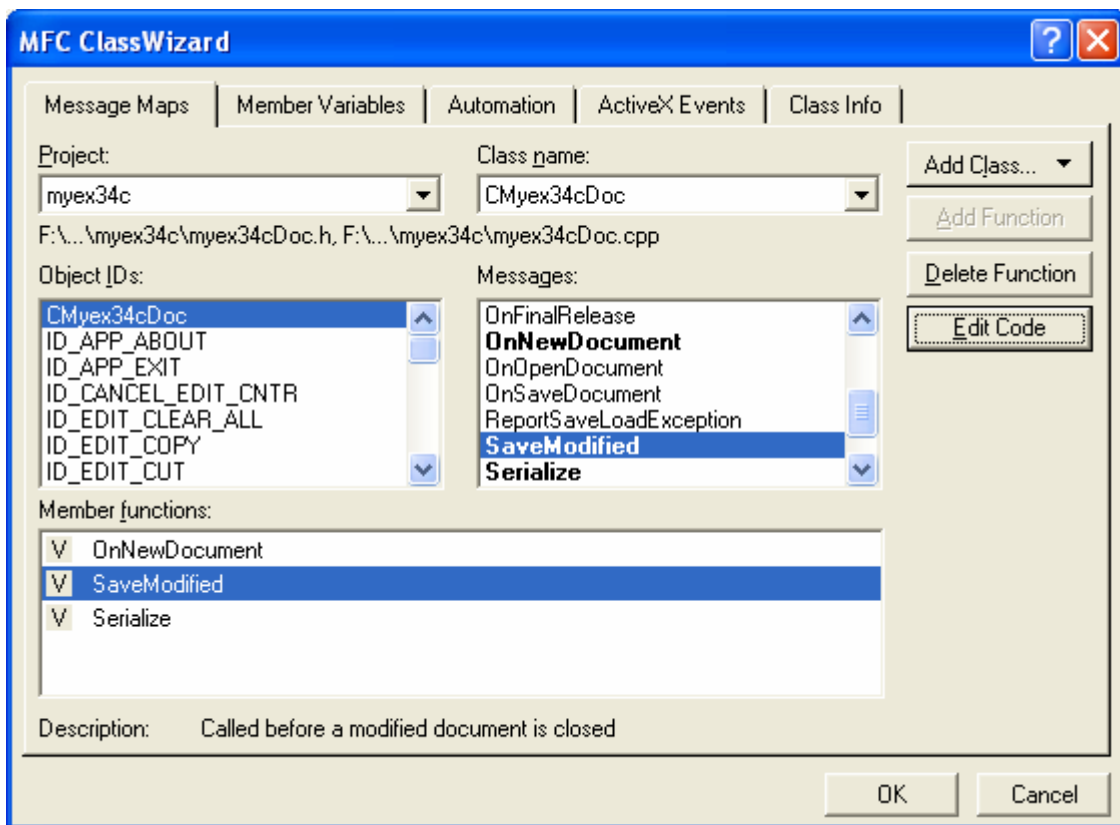


Figure 74: Adding SaveModified() virtual function.

Then, edit the code as shown below.

```

BOOL CMyex34cDoc::SaveModified()
{
    return TRUE;
}

// CMyex34cDoc commands
BOOL CMyex34cDoc::SaveModified()
{
    // TODO: Add your specialized code here and/or call the base class
    return TRUE;
}

```

Listing 25.

Change the following message map:

```

ON_UPDATE_COMMAND_UI_RANGE(ID_OLE_VERB_FIRST, ID_OLE_VERB_LAST,
    CRichEditDoc::OnUpdateObjectVerbMenu)

```

to

```

ON_UPDATE_COMMAND_UI(ID_OLE_VERB_FIRST,
    CRichEditDoc::OnUpdateObjectVerbMenu)

```

```

//}}AFX_MSG_MAP
// Enable default OLE container implementation
ON_UPDATE_COMMAND_UI(ID_OLE_EDIT_LINKS, CRichEditDoc::OnUpdateEditLinksMenu)
ON_COMMAND(ID_OLE_EDIT_LINKS, CRichEditDoc::OnEditLinks)
ON_UPDATE_COMMAND_UI(ID_OLE_VERB_FIRST, CRichEditDoc::OnUpdateObjectVerbMenu)
MESSAGE_MAP()

```

Listing 26.

CMyex34cView Class

Using ClassWizard or ClassView, add WM_RBUTTONDOWN windows message handler, also, command and command update for ID_EDIT_CLEAR_ALL.

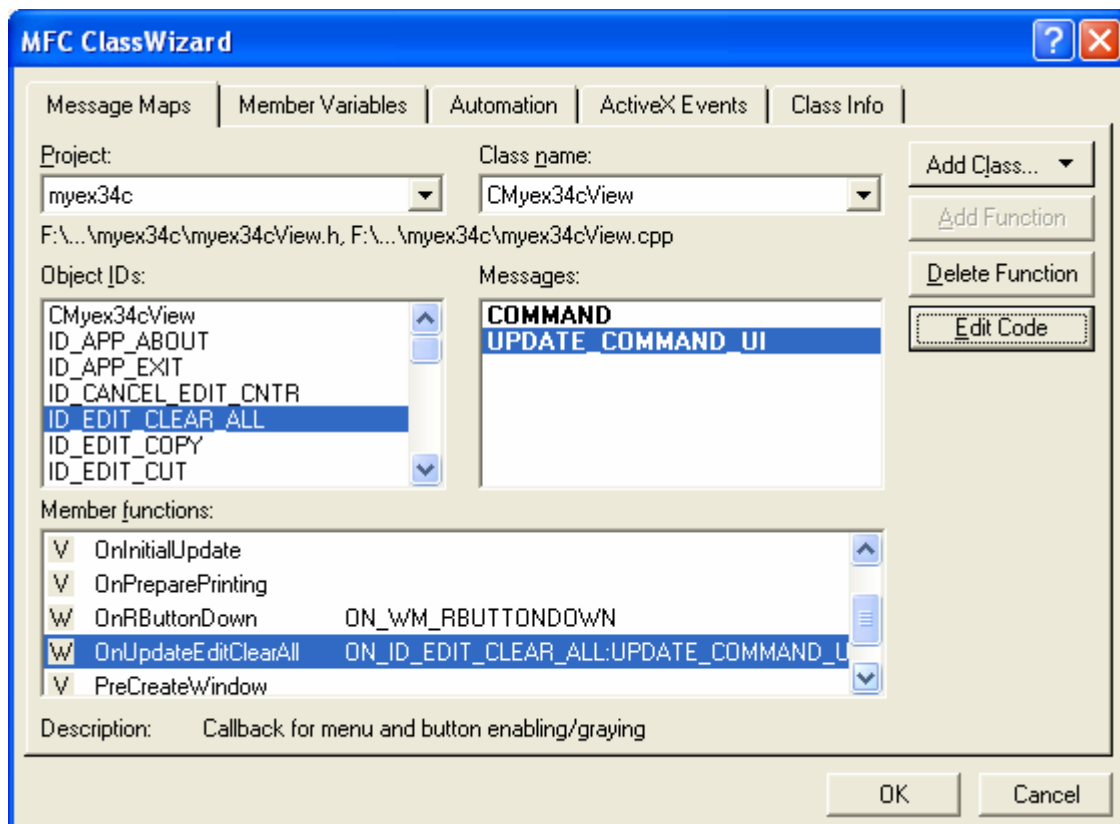


Figure 75: Adding Windows message, command and update commands.

```

// Generated message map functions
protected:
   //{{AFX_MSG(CMyex34cView)
    afx_msg void OnDestroy();
    afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnEditClearAll();
    afx_msg void OnUpdateEditClearAll(CCmdUI* pCmdUI);
   //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

```

Listing 27.

Edit the code for the implementation part as shown below.

```

BOOL CMyex34cView::PreCreateWindow(CREATESTRUCT& cs)

```

```

{
    BOOL bPreCreated = CRichEditView::PreCreateWindow(cs);
    cs.style |= ES_READONLY;
    return bPreCreated;
}

BOOL CMyx34cView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs
    BOOL bPreCreated = CRichEditView::PreCreateWindow(cs);
    cs.style |= ES_READONLY;
    return bPreCreated;
}

```

Listing 28.

```

void CMyx34cView::OnRButtonDown(UINT nFlags, CPoint point)
{
    TRACE("CMyx34cView::OnRButtonDown\n");
    CMenu menu;
    menu.LoadMenu(IDR_MENUCONTEXT);
    ClientToScreen(&point);
    menu.GetSubMenu(0)->TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON,
        point.x, point.y, this);
}

void CMyx34cView::OnEditClearAll()
{
    CRichEditCtrl& re = GetRichEditCtrl();
    re.SetSel(0, -1);
    // won't let us clear unless we reset readonly flag
    re.SetOptions(ECOOP_XOR, ECO_READONLY);
    re.Clear();
    re.SetOptions(ECOOP_SET, ECO_READONLY);
}

void CMyx34cView::OnUpdateEditClearAll(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(GetRichEditCtrl().GetTextLength() > 0);
}

```

```

// CMyex34cView message handlers

void CMyex34cView::OnRButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    TRACE("CMyex34cView::OnRButtonDown\n");
    CMenu menu;
    menu.LoadMenu(IDR_MENUCONTEXT);
    ClientToScreen(&point);
    menu.GetSubMenu(0)->TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON,
        point.x, point.y, this);
}

void CMyex34cView::OnEditClearAll()
{
    // TODO: Add your command handler code here
    CRichEditCtrl& re = GetRichEditCtrl();
    re.SetSel(0, -1);
    // won't let us clear unless we reset readonly flag
    re.SetOptions(ECOOP_XOR, ECO_READONLY);
    re.Clear();
    re.SetOptions(ECOOP_SET, ECO_READONLY);
}

void CMyex34cView::OnUpdateEditClearAll(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->Enable(GetRichEditCtrl().GetTextLength() > 0);
}

```

Listing 29.

Build and run MYEX34C. The following is the output.

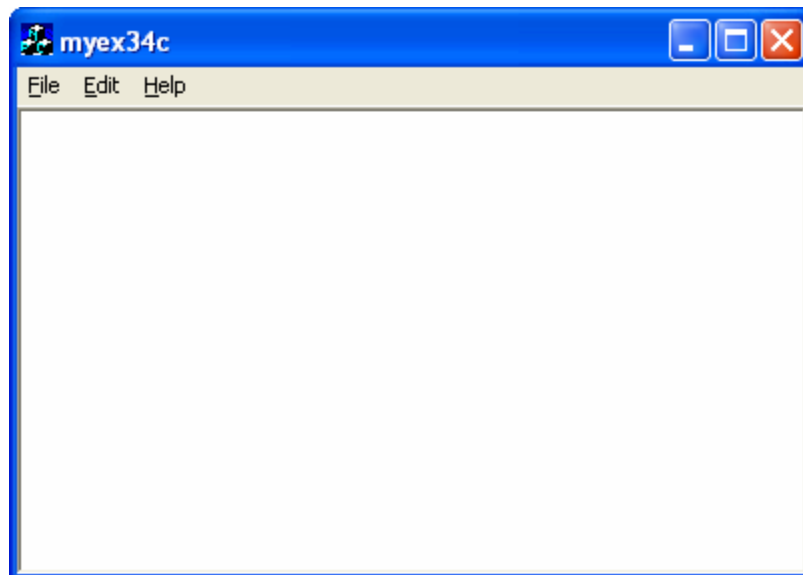


Figure 76: MYEX34C in action.

The Story of The Display Program

The display program, myex34c.exe, isn't very interesting. It's a standard AppWizard CRichEditView program with a WM_SENDETEXT handler in the main frame class:

```

LONG CMainFrame::OnSendText(UINT wParam, LONG lParam)
{

```

```

TRACE("CMainFrame::OnSendText\n");
LPVOID lpvFile = ::MapViewOfFile((HANDLE) lParam, FILE_MAP_READ, 0, 0,
0);
GetActiveView()->SendMessage(EM_SETSEL, (WPARAM) 999999, 1000000);
GetActiveView()->SendMessage(EM_REPLACESEL, (WPARAM) 0,
(LPARAM) lpvFile);
::UnmapViewOfFile(lpvFile);
::CloseHandle((HANDLE) lParam);

return 0;
}

```

This function just relays the text to the view. The MYEX34C CMainFrame class overrides OnUpdateFrameTitle() to eliminate the document name from the main window's title. This ensures that the DLL can find the MYEX34C window by name. The view class maps the WM_RBUTTONDOWN message to implement a context menu for erasing the view text. Apparently rich edit view windows don't support the WM_CONTEXTMENU message.

Building and Testing the MYEX34B ISAPI Filter

Build both the MYEX34B and MYEX34C projects, and then start the MYEX34C program. For **IIS 1.0 – 3.0**, to specify loading of your new filter DLL, you must manually update the Registry.

Run the **Regedit/regedt32** application, and then double-click on **Filter DLLs** in

\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC\Parameters. Add the full pathname of the DLL separated from other DLL names if any, with a comma.

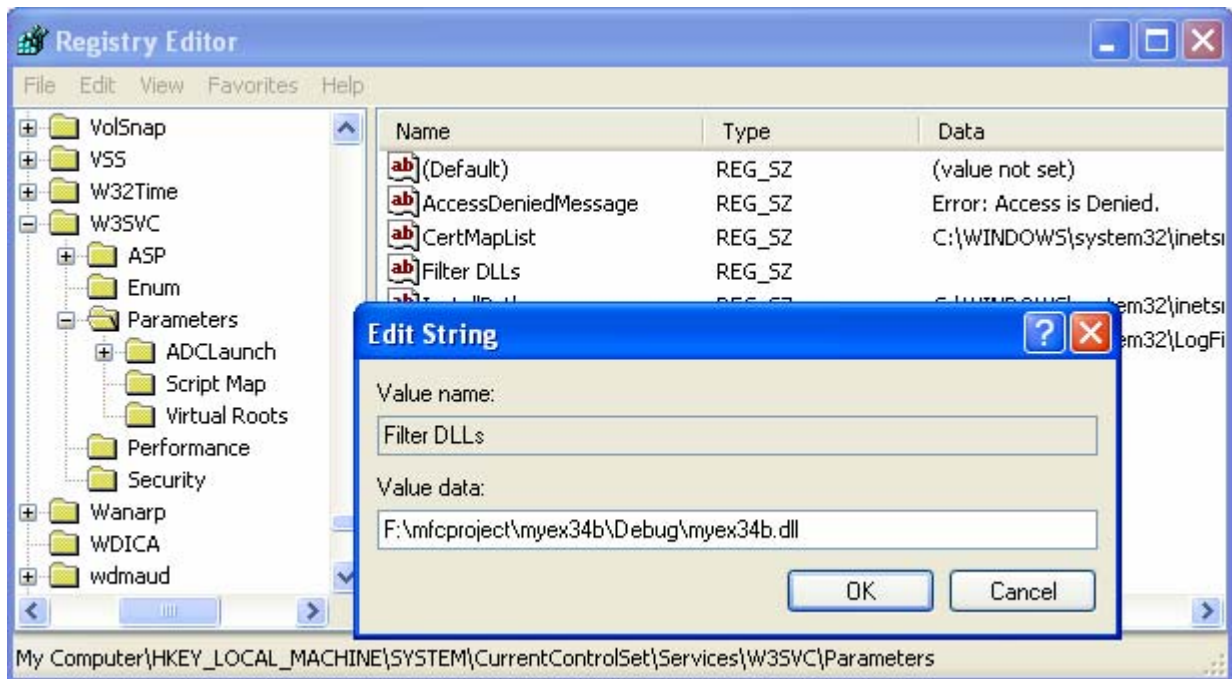


Figure 77: Adding the full path of the DLL in registry – Windows XP Pro

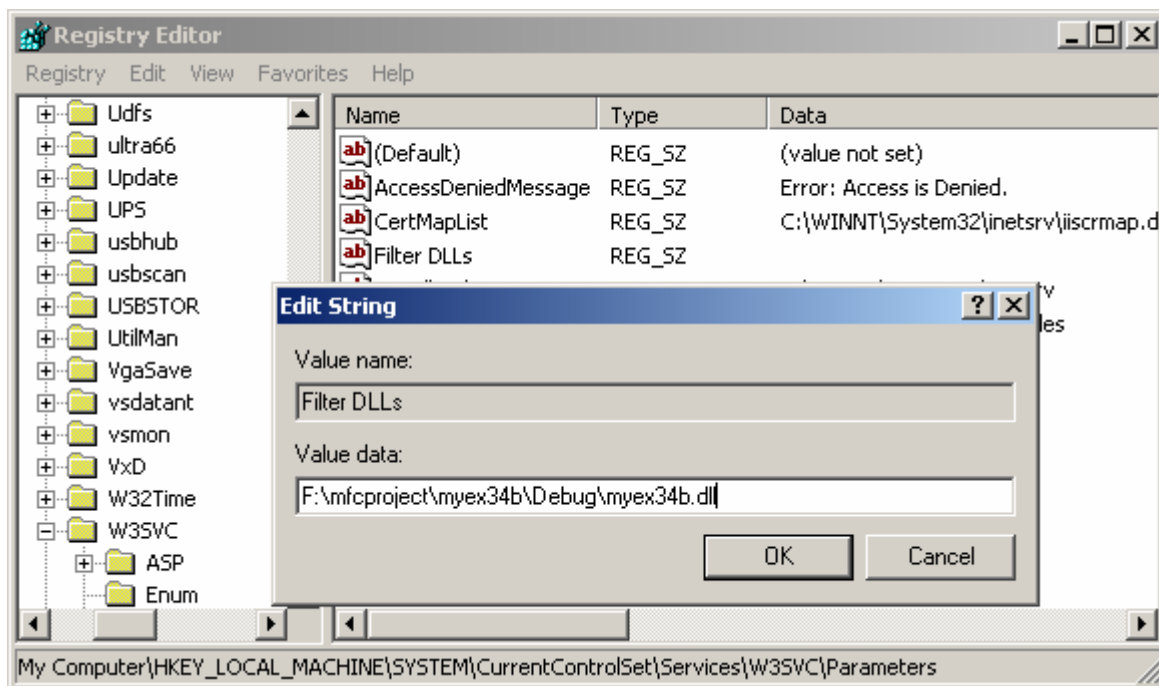


Figure 78: Adding the full path of the DLL in registry – Windows 2000 Server.

To install an ISAPI filter for use with a Microsoft Internet Information Server version **4.0 - 6.0**, follow the following steps:

1. Copy the filter DLL to an appropriate folder, such as the **Scripts** or **Cgi-bin** subdirectory.
2. Open the Internet Service Manager (MMC).
3. Select the appropriate level for the ISAPI filter:
 - a. To use the ISAPI filter with all Web sites (global), select the **ServerName** icon. A global filter is installed for the entire IIS service and sees requests for all Web sites.
 - b. To use the ISAPI filter with a specific Web site, select the icon for that Web site (for example, the default Web site). A site filter is installed at the site level and sees requests only for the site where it is installed.
4. Right-click the level (icon) that you selected (for this example, we have to choose the global level).

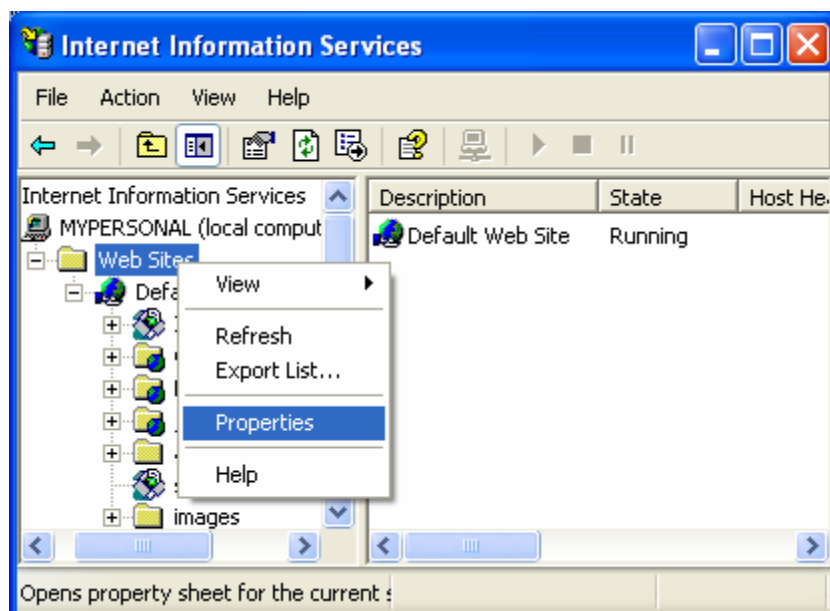


Figure 79: Invoking the Properties page of the Web site.

5. Click the **ISAPI Filters** tab.
For Windows 2000 Server, to configure an ISAPI filter for all Web sites, first click the **Edit** button that is next to the **Master Properties** of the WWW Service.
6. Click Add.
7. Type a name for the ISAPI filter.
8. Click Browse and select the ISAPI filter that you copied in step 1.

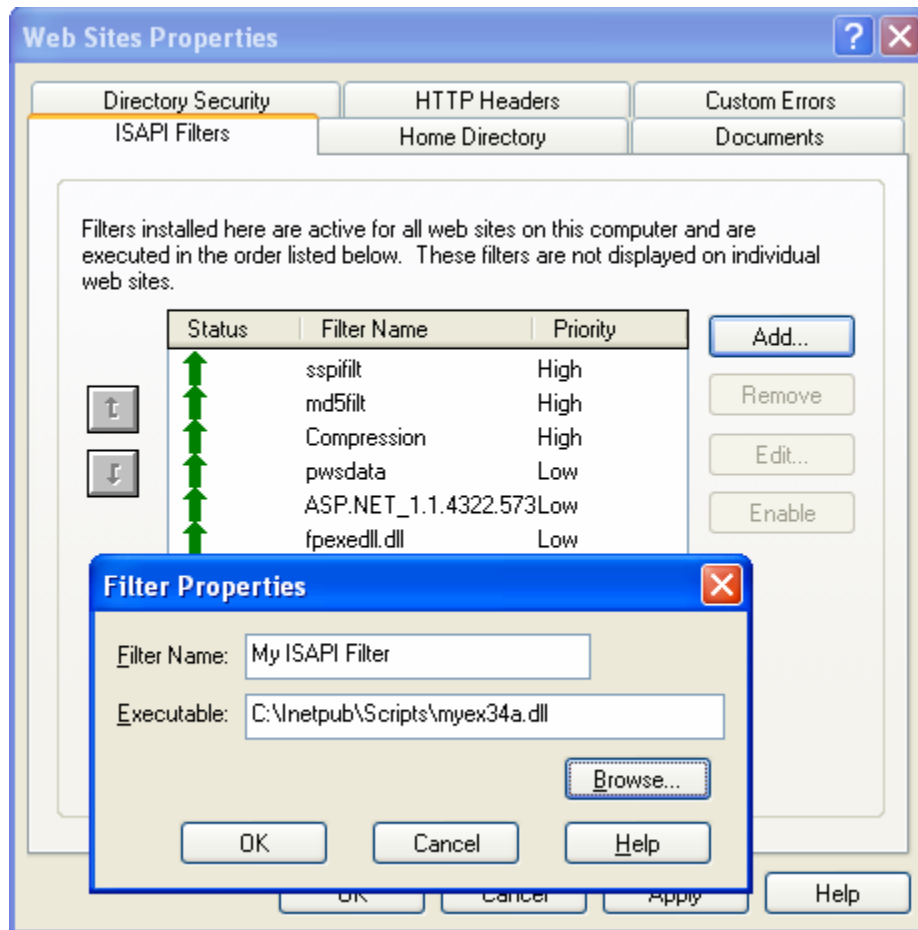


Figure 80: Entering the filter properties.

9. Click OK.
10. Stop the IISADMIN service. To do this, either type **net stop iisadmin /y** at a command prompt, or use the **Services** applet that is located in Control Panel (in Windows NT 4.0) or Administrative Tools (in Windows 2000) or .
11. Start the World Wide Web Publishing Service by typing **net start w3svc** at a command prompt, or by using the Services applet that is located in Control Panel (in Windows NT 4.0) or Administrative Tools (in Windows 2000) or for steps 11 and 12, you can use the IIS context menu when the Server directory is selected and right clicked as shown below. All other dependencies (World Wide Web Publishing Service in this case) will also be restarted when you restart the IIS.

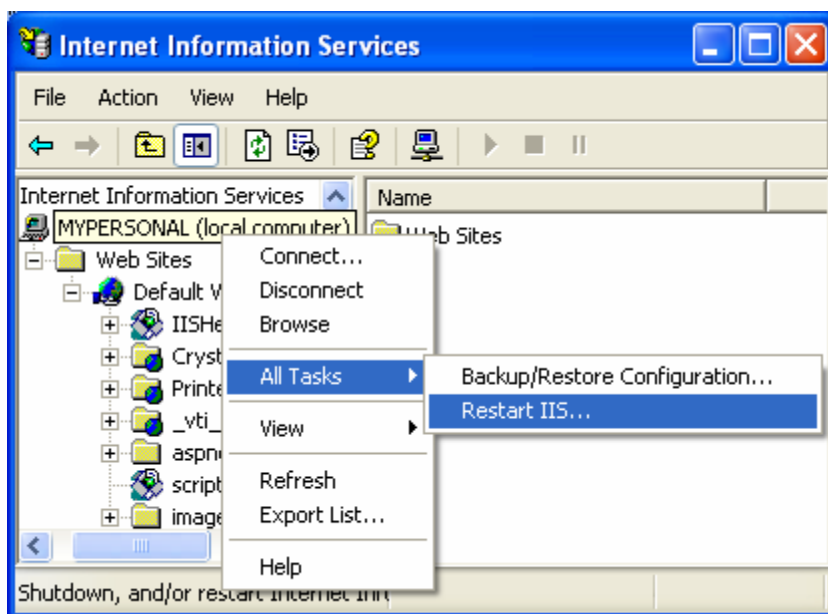


Figure 81: Restarting the IIS.

12. Repeat the previous step for any other services that were stopped in step 11 if needed.
13. Browse back to the ISAPI Filters tab (by following steps 1-5) and verify that the filter is loaded properly. You should see a green arrow that is pointing up under the **Status** column.

NOTE: The ISAPI Filters tab specifies a load order, with the filter at the top of the list loading first. Normally **Sspifilt.dll**, the ISAPI filter for SSL, is at the top of the list to allow any other filters to access data before IIS encrypts and transmits or decrypts and receives TTPS traffic.

If your ISAPI filter failed to load, you may try removing the added filter and re-add the filter in the Debug directory of the project directly as shown below. This happens mostly because of the version of function (library) used in the source code not compatible with Windows operating system version.

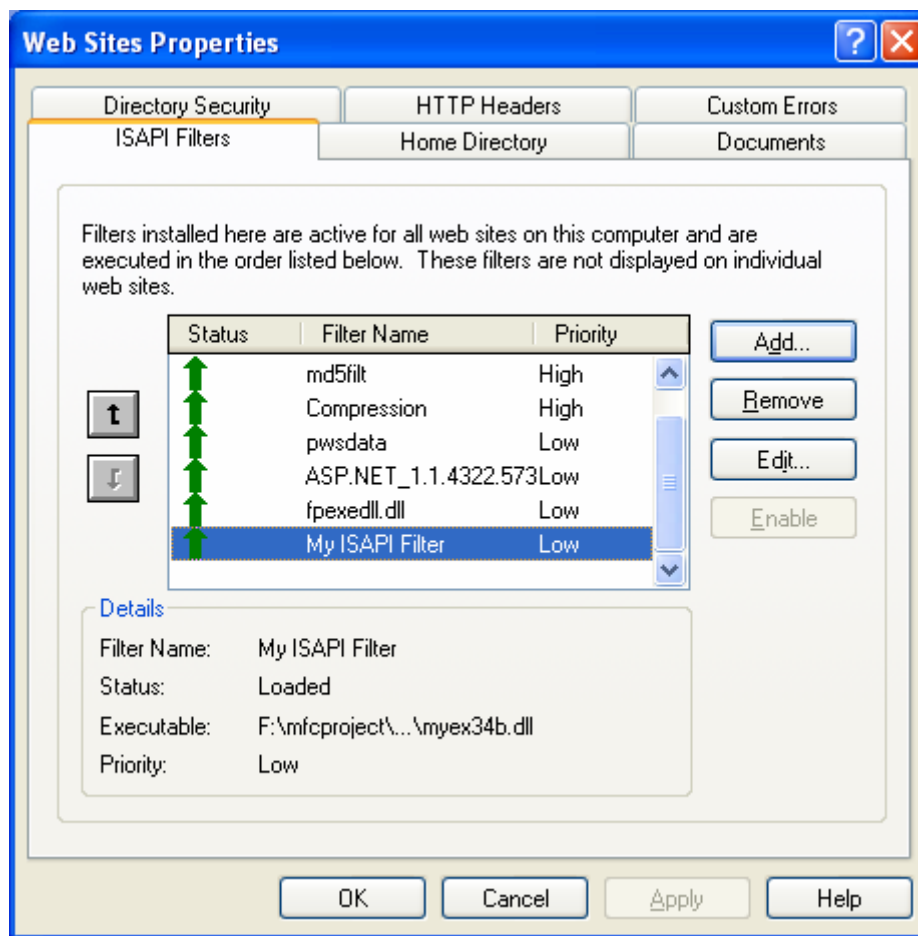


Figure 82: Using the debug version of the filter DLL.

The steps for adding an ISAPI filter to IIS 6.0 are the same as the steps for IIS 4.0 and IIS 5.0, but if you use an ISAPI filter that implements the SF_NOTIFY_READ_RAW_DATA event, you must change IIS 6.0 from its default configuration. Site filters cannot register for SF_NOTIFY_READ_RAW_DATA notifications. Global filters, however, can register for SF_NOTIFY_READ_RAW_DATA notifications. In this case, you must install the ISAPI filter at global level.

There's one more thing to do. You must change the IIS mode to allow the service to interact with the MYEX34C display program. To do this, click on the **Services** icon in the **Control Panel/Administrative Tools**, double-click on **World Wide Web Publishing Service**, and then check **Allow Service To Interact With Desktop**.

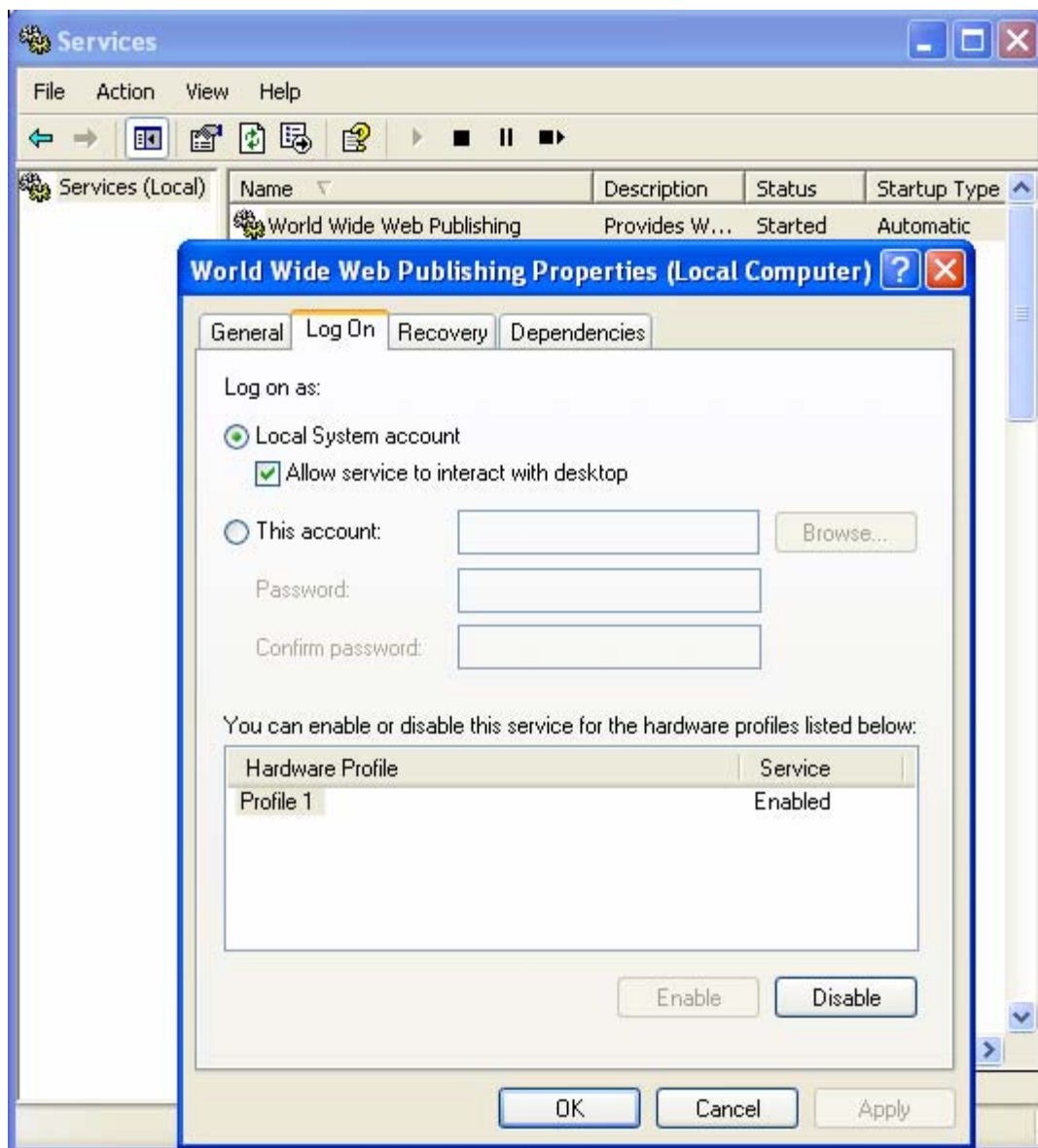


Figure 83: Enabling the **Allow service to interact with desktop** for World Wide Web Publishing service.

Finally, use **Internet Service Manager** to stop and restart the WWW service to load the filter DLL.

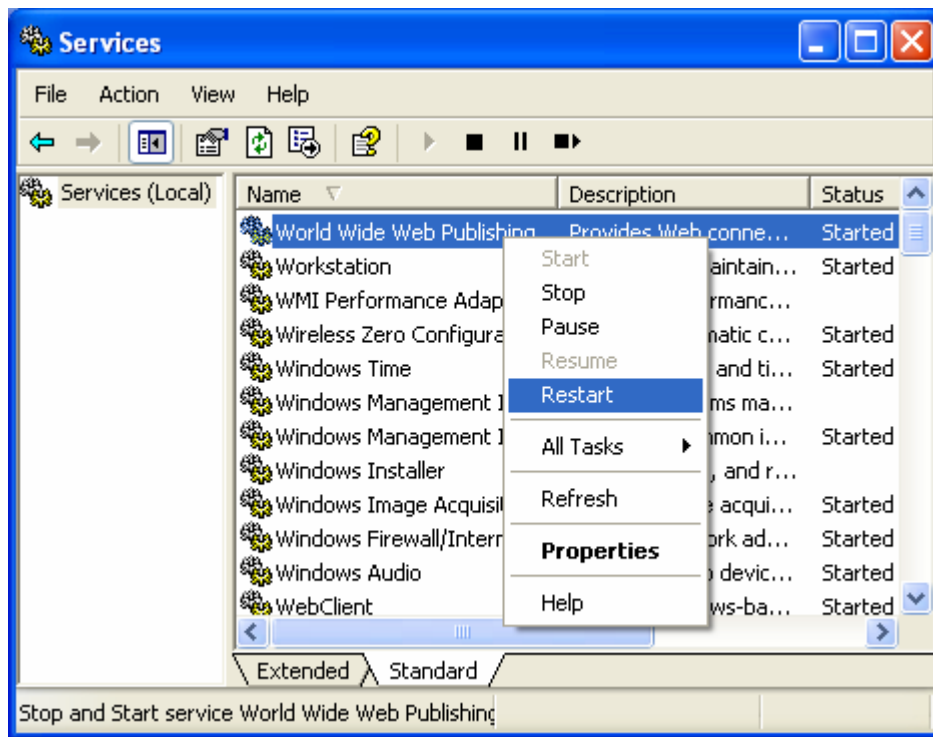


Figure 84: Restarting (start/stop) the World Wide Web service through the **Services** snap-in.

When you use the browser to retrieve pages from the server, you should see output something like this.

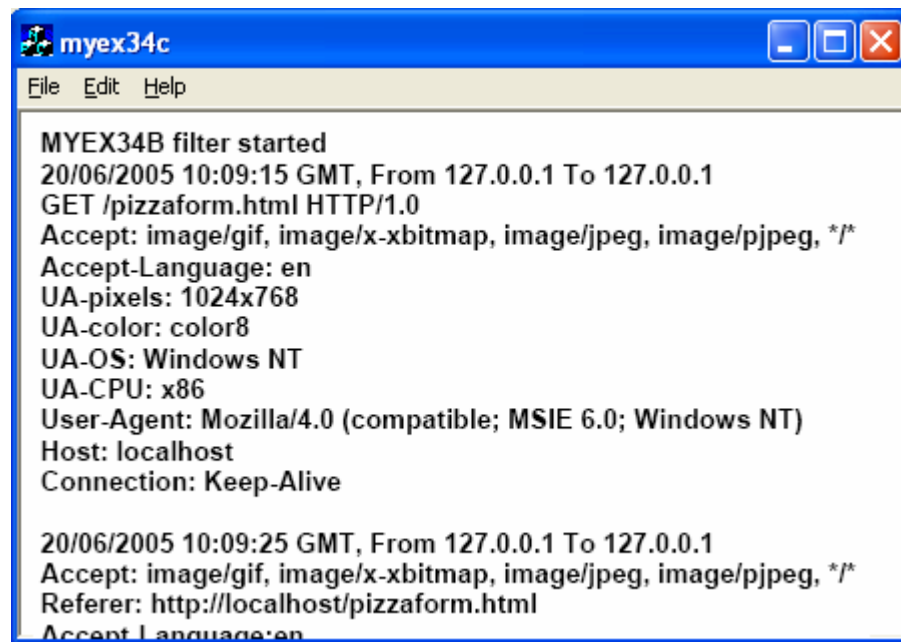


Figure 85: MYEX34C with user transaction data.

You can use the same steps for debugging an ISAPI filter that you used for an ISAPI server extension.

-----End-----

Further reading and digging:

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