This article assumes you're familiar with C++
Download the code (41KB)

Internet Client SDK Part II: Dress Your Applications for Success with WinInet

Aaron Skonnard

Although WinInet has been around for a while, many programmers still shy away from it. Perhaps all they ever needed was a clear, concise overview of the API.

Web browsing objects give your Microsoft® Windows®-based application the ability to act as a custom Web browser. As my first article on the Internet Client SDK showed ("Boning up on the Internet Client SDK Part I: Web Browsing Objects," *MIND*, October 1997), this strategy can be used only for the object's exposed properties and methods. If your application needs more flexibility, such as direct access to Internet-based APIs, the WinInet API is the interface you'll need.

WinInet has been around for a while. In fact, there have already been a few articles in *MIND* devoted to the subject. Back in the Spring 1996 issue of *MIND*, Matthew Powell, Leon Braginski, and Jeffrey Richter all discussed some portion of the WinInet functionality. In the May 1997 issue of *MIND*, Steve Zimmerman discussed using WinInet in ActiveX™ controls (see Designing ActiveX Components Part II: Implementing Internet Communication with WinInet). Since the demand for Internet integration continues to increase, WinInet deserves more attention. In this article, I'll give a brief overview of the WinInet architecture and how to get started with it. Then I'll go into more detail on dial-up connections, general Internet functions, URLs, HTTP, FTP, and Gopher. Finally, I'll show you how to use the Visual C++® WinInet classes in a sample FTP client application.

WinInet Overview

WinInet is a high-level interface to the more complicated underlying Internet protocols (including HTTP, FTP, and Gopher). WinInet allows your application to act as an HTTP, FTP, or Gopher client without its having to understand or, more importantly, keep up with the ever-evolving protocol standards. If you use WinInet in your applications, when standards change you can let WinInet worry about the changes while your interface to the protocol remains the same. (To learn more about the protocol standards, see the RFC documents at http://www.rs.internic.net.) WinInet can be used to write product-ordering systems, stock tickers/analyzers, online banking systems, FTP clients, your own Internet browser, and so on. The possibilities are endless.

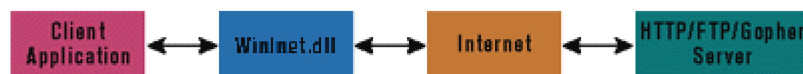

Figure 1: Sending and receiving information with WinInet

Figure 1 illustrates how information is sent and received using the WinInet API. The Windows-based application calls into WinInet.dll. WinInet is then responsible for contacting the server and receiving a response. Once WinInet receives the response, the

application can begin using the API—simple as that.
Before WinInet, adding Internet communications to Windows-based applications required expertise in sockets and protocol specifications. Even simple communications required considerable development time. WinInet lets you quickly and easily add Internet communications to your applications and even make them compatible with server-side scripts developed with CGI, ASP, and ISAPI. For example, let's assume you already have a server-side script set up for access via a browser and you want to use the same script from a wizard in your application. You can use WinInet to send the information gathered from the wizard dialogs back to your Web server with an HTTP POST request. Using this approach, a single server-side script can process requests made from both a browser and your Windows-based application.
To use the WinInet API, you need to include WinInet.h in your source files and link against WinInet.lib. Assuming you've installed the Internet Client SDK, you'll probably want to add INetSDK Path\Include to your compiler's include path and INetSDK Path\Lib to your compiler's lib path. If you have problems compiling or linking once you've added calls to WinInet functions, make sure these files are where you think they are.

Internet Sessions

WinInet gives your application the ability to host Internet sessions. An Internet session generally consists of the following three steps: connecting to the Internet, sending and receiving information using HTTP, FTP, or Gopher, and closing the connection.
How you'll connect to the Internet depends mostly on the target machine's Internet connection. If it's on a LAN with a direct connection to the net, verifying the connection is all that is necessary. Upon verification, the application is ready to start sending and receiving information. If the target machine uses dial-up networking to connect to the Internet, your application is responsible for initiating and verifying a successful dial-up connection.
In the second step (sending and receiving information using HTTP, FTP, or Gopher), you can use the same code for either connection scenario. Once you've established a valid Internet connection, WinInet does not distinguish between calls made across a LAN and calls made across a dial-up connection.
The last step, closing the connection, is a little easier if the target machine is connected to a LAN. You simply need to close all open Internet handles by calling InternetCloseHandle. If the machine uses dial-up networking, you may also need to disconnect the dial-up networking connection.
Generally, it's a good idea to plan for the worst-case scenario and not make any assumptions about the target machine's Internet connection. And with WinInet's new dial-up functions it's a snap to handle the process of establishing a dial-up networking connection.

Dial-Up and Autodial Functions

Figure 2 describes the new WinInet dial-up functions. Before the Microsoft Internet Explorer (IE) 4.0 Preview 2 release, WinInet offered only two dial-up functions: InternetAttemptConnect and InternetCheckConnection. InternetAttemptConnect checks to see if there is a connection to the Internet. If there isn't a connection, it displays the dial-up networking dialog box and allows the user to connect to an ISP. InternetCheckConnection determines whether a connection to the Internet already exists by trying to ping the server designated by a URL passed to the function.
While InternetAttemptConnect and InternetCheckConnection may be sufficient for many situations, both are somewhat limited. The problem with InternetAttemptConnect is that it requires (or at least used to before the new Connection Manager) user intervention to establish the connection. This is a problem if you want to give your program the ability to schedule a connection for the middle of the night to download a huge file. In my opinion, anything that requires user intervention is very limiting to both the programmer and the

user. InternetCheckConnection is also of limited use. Sure, it determines whether a connection to the Internet exists, but it cannot tell you anything about the type of connection.

IE 4.0 Preview 2 introduced seven more WinInet dial-up functions that improve upon the earlier two. InternetGetConnectedState provides the information that InternetCheckConnection lacked by determining the type of Internet connection being used. It can figure out whether the connection is over a modem, on a LAN, or even through a proxy. You simply pass in a combination of the following values, which represents the type of connection you're testing for:

```
INTERNET_CONNECTION_MODEM
INTERNET_CONNECTION_LAN
INTERNET_CONNECTION_PROXY
```

It returns a Boolean indicating whether that type of connection is in use.

Figure 3: Connection Manager

InternetAutodial is a simple solution to the user-intervention problem. It allows you to initiate an unattended connection by using the user's default dial-up networking entry. **Figure 3** shows the Connection Manager's nifty new dialog as displayed by InternetAutodial. Internet-Dial is even more flexible; you can specify which dial-up networking entry to use and how you want to make the connection. Like InternetAutodial, Internet-Dial is capable of making unattended connections. It even goes one step further and provides a flag (INTERNET_ DIAL_UNATTENDED) to turn off the dial-up UI completely.

For example, this code establishes an unattended connection, if the user isn't already connected, using any of the mentioned connection types:

```
DWORD dwConnectionTypes = INTERNET_CONNECTION_LAN |
                          INTERNET_CONNECTION_MODEM |
                          INTERNET_CONNECTION_PROXY;
if (!InternetGetConnectedState(&dwConnectionTypes, 0))
{
    InternetAutodial(INTERNET_AUTODIAL_FORCE_UNATTENDED,
                     0);
}
```

As you can see, these new functions have greatly improved the dial-up capabilities of WinInet and provide an easy interface to Microsoft's dial-up services. If you've ever written your own dial-up (RAS) utilities, you'll appreciate this new addition to the API.

General Internet Functions

Before I start exploring the general WinInet Internet functions, let me mention one common pitfall. Many developers begin using these functions, but can't get anything to work because they don't have a valid TCP/IP connection. This means more than just establishing a dial-up connection. You must also make sure that your network and dial-up settings are configured properly. If you're not sure, contact your ISP. A simple way to verify this is to open IE 4.0 and navigate to http://www.microsoft.com. If your browser can successfully navigate to Microsoft's site, you shouldn't have any problems using WinInet. If the browser fails to find the site, check your connection and solve the problem before venturing into the world of WinInet. It will save you a lot of headaches.

InternetOpen is the mother of all WinInet Internet functions (see **Figure 4**). Take a look at InternetOpen's function declaration:

```
HINTERNET InternetOpen(
    IN LPCSTR lpszAgent,
    IN DWORD dwAccessType,
    IN LPCSTR lpszProxyName,
    IN LPCSTR lpszProxyBypass,
    IN DWORD dwFlags
    );
```

If you're developing a Web site to communicate with your Windows-based application, you'll want to pay special attention to the first parameter, lpszAgent. This should contain the address of a string that identifies your application to the Web server. With this information, your Web server can recognize requests made by your application and, if desired, can customize the response to better fit your needs.

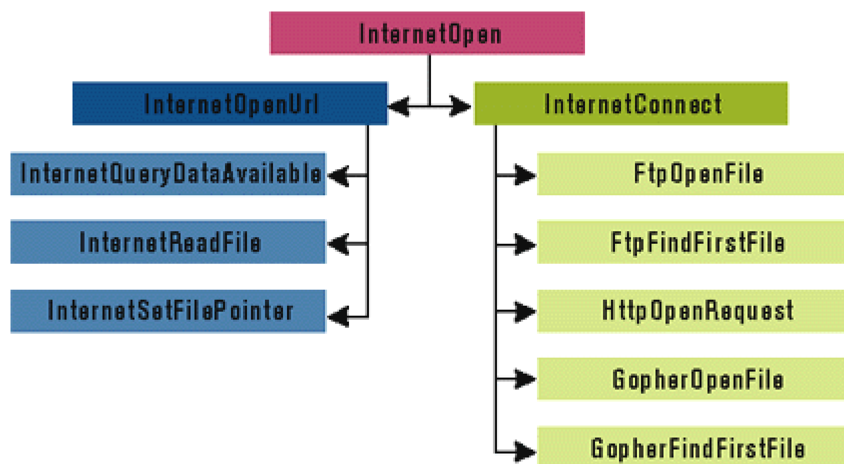

Figure 5: InternetOpen handle hierarchy

Your application must call InternetOpen before calling any other Internet function. This initializes the Internet DLL, preparing it to receive subsequent Internet calls from your application. The handle (HINTERNET) returned by InternetOpen is the root node in the Internet handle hierarchy. The root handle can then be used by InternetConnect and InternetOpenURL. Next, the handle returned by InternetConnect can be used by FtpOpenFile, Ftp- FindFirstFile, HttpOpenRequest, GopherOpenFile, and GopherFindFirstFile (see **Figure 5**). HTTP, FTP, and Gopher all have their own handle hierarchies as well. Get the idea? Appendix A in the WinInet section of the Internet Client SDK documentation provides detailed diagrams of the entire Internet handle hierarchy.

Now let's take a look at the second-highest function in the Internet handle hierarchy, InternetConnect:

```
HINTERNET InternetConnect(
    IN HINTERNET hInternetSession,
    IN LPCSTR lpszServerName,
    IN INTERNET_PORT nServerPort,
    IN LPCSTR lpszUsername,
```

```
    IN LPCSTR lpszPassword,
    IN DWORD dwService,
    IN DWORD dwFlags,
    IN DWORD dwContext
);
```

InternetConnect should be used to establish a connection with a service on the Web server. The service specified by dwService can be HTTP, FTP, or Gopher. The type of service that you connect to determines what types of functions you'll be able to call with the returned handle. You also need to specify the server name and port. The server name can either be the host name (such as www.microsoft.com) or the IP address of the server. As for the port value, you can either specify it yourself or use the default port for the requested service.

Once you've successfully established a connection with a service, you can start using the functions related to it.

```
HINTERNET hSession;
hSession = InternetOpen("MyApp", INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, 0);
if (hSession)
{
    HINTERNET hService;
    hService = InternetConnect(hSession, "ftp.microsoft.com",
                               INTERNET_DEFAULT_FTP_PORT, NULL, NULL,
                               INTERNET_SERVICE_FTP, 0, 0);
    if (hService)
    {
        // call ftp functions here using hService
    }
}
InternetCloseHandle(hSession);
```

This example establishes a connection with Microsoft's FTP site. Notice that I passed in NULL for both the lpszUsername and lpszPassword parameters. Depending on the type of service you're connecting to, these parameters default to different values. In this case, the FTP service defaults the lpszUsername parameter to anonymous and lpszPassword to the user's mail address. After InternetConnect returns a valid handle, I can use that handle to call the functions in the FTP handle hierarchy.

Whether it's due to a slow connection or a huge file download, Internet operations can take a considerable amount of time to complete. For that reason, it's very important to give the user some idea about the status of the operation at hand. InternetSetStatusCallback and InternetStatusCallback are especially useful for giving the user feedback during Internet operations. These two functions can display the status of any Internet request. Use InternetSetStatusCallback to register a callback function with a specific Internet handle. The registered function will be called each time the status of the given handle changes. The callback will also be used for any handles derived from the handle that registered the callback. InternetStatusCallback, on the other hand, is simply a placeholder for your callback function. It defines what values will be passed to the registered callback function. The most valuable of these is dwInternetStatus, a DWORD value containing the current status of the given Internet handle.

If you want your Internet requests to be made asynchronously, an InternetStatusCallback function is required. To make requests asynchronously, you must first call InternetOpen with INTERNET_FLAG_ASYNC set. Then, as long as you have an InternetStatusCallback function registered with the handle returned from InternetOpen, all requests made on handles derived from InternetOpen's handle will be made asynchronously. When the callback function receives the INTERNET_STATUS_REQUEST_COMPLETE status, the Internet operation is complete. If you don't have an InternetStatusCallback function registered properly, all requests will be made synchronously regardless of the INTERNET_FLAG_ASYNC flag.

This should give you a good overview of the most important general Internet functions. It would take a lot more space than I have to discuss all of the general Internet functions in detail. The WinInet section of the Internet Client SDK documentation does an excellent job of explaining how the rest of the functions behave.

URL Functions

As mentioned, you need to use InternetOpen and InternetConnect to establish a connection with one of the services on the server (like HTTP, FTP, or Gopher). Once you've established a connection, you can call the protocol-specific functions belonging to that service. Let's look at a general approach to using the HTTP, FTP, and Gopher protocols—the URL functions—before examining how to program them directly.

Figure 6 describes the behavior of each URL function. Of the five functions, only InternetOpenUrl actually sends and receives information. The other four exist only to make InternetOpenUrl easier to use. They simplify the process of creating, encoding, and breaking up Internet URLs.

If you keep your eye on the address box as you navigate around using Internet Explorer (IE) 4.0, you'll probably notice strange-looking character sequences in some of the URLs. For example, take a look at the following HTTP URL:

```
http://aarons.axtech.com/register.cgi?name=Aaron%20Skonnard
```

Notice the ?, &, and % characters found in the URL. If you don't understand what these characters are for, you're in for a lot of grief when trying to build and send HTTP URLs. The ? indicates the end of the server-side object location and the beginning of the name/value parameter pairs. Each name/value pair is separated by an & character. The % character represents an escape sequence for an unsafe character. Most non-alphanumeric characters, like blanks and punctuation, are unsafe to use in an URL. For example, the % 20 found in the example URL represents a blank space, which can't be used when passing an URL. The % symbol tells the Web server that the numbers to follow represent a character's hex value. When the Web server parses the URL, it knows to translate them accordingly.

InternetCanonicalizeUrl and In-ternetCombineUrl make URL encoding a piece of cake. In fact, you can build your URL without worrying about the unsafe characters. Then, before you send it (using Internet OpenUrl), call InternetCanonicalizeUrl or InternetCombineUrl and voilà! All unsafe characters will be encoded with hex values. Once you've mastered the URL creation step, you're ready to call InternetOpenUrl. The main advantage of InternetOpenUrl is sim-plicity. You can use InternetOpenUrl to send and receive data with HTTP, FTP, or Gopher, plus you don't have to worry about calling InternetConnect first. The only thing you really need to understand is how to build the URL for the protocol in use.

Figure 8: Fetching an HTML file

Take a look at **Figure 7**, which uses InternetOpenUrl to fetch and display the HTML file at http://www.microsoft.com/msj. To illustrate how you can use InternetOpenUrl with the different protocols, I created a simple dialog-based application that allows the user to type in a URL and fetch the specified file (see **Figure 8**). The command handler for the InternetOpenUrl button is almost identical to the example shown previously. I moved the call to InternetOpen to the dialog's InitInstance so it's only called once. The handle returned by InternetOpen is used by each call to InternetOpenUrl. I also moved the call to InternetCloseHandle to the dialog's OnClose method. Finally, I passed the URL edit box member variable (m_strURL) to Internet OpenUrl.

Figure 9: MSJ Web site

Figure 9 shows what IE 4.0 looks like when you type in http://www.microsoft.com/msj. **Figure 10** shows the file that InternetOpenUrl returns, the same HTML file that IE 4.0 is displaying. If you do the same thing for ftp://ftp.microsoft.com, **Figure 11** shows what IE 4.0 displays, and **Figure 12** shows the same HTML file returned from InternetOpenUrl. You could also do this for the Gopher protocol—try gopher://www.byu.edu.

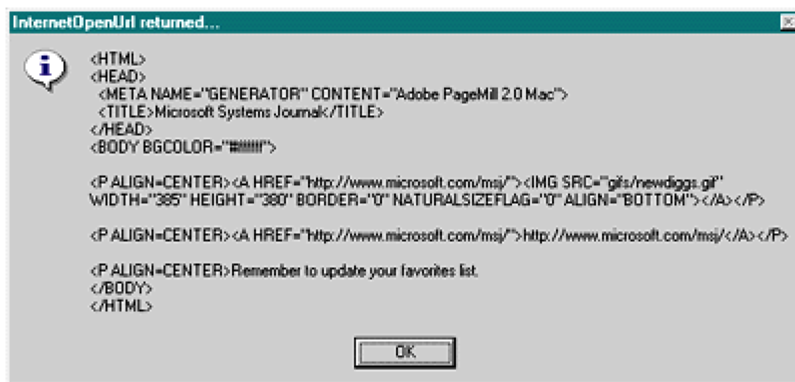

Figure 10: InternetOpenUrl output

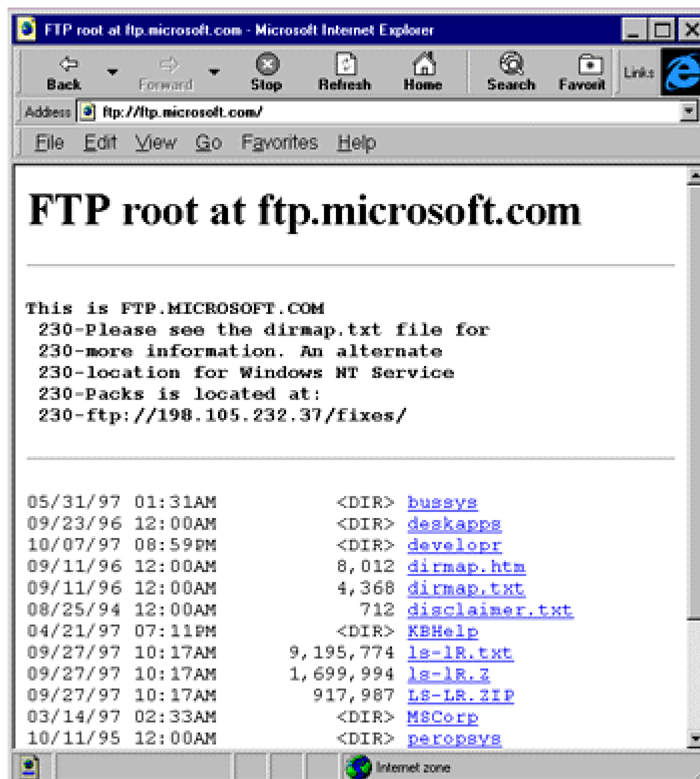

Figure 11: FTP root at ftp.microsoft.com

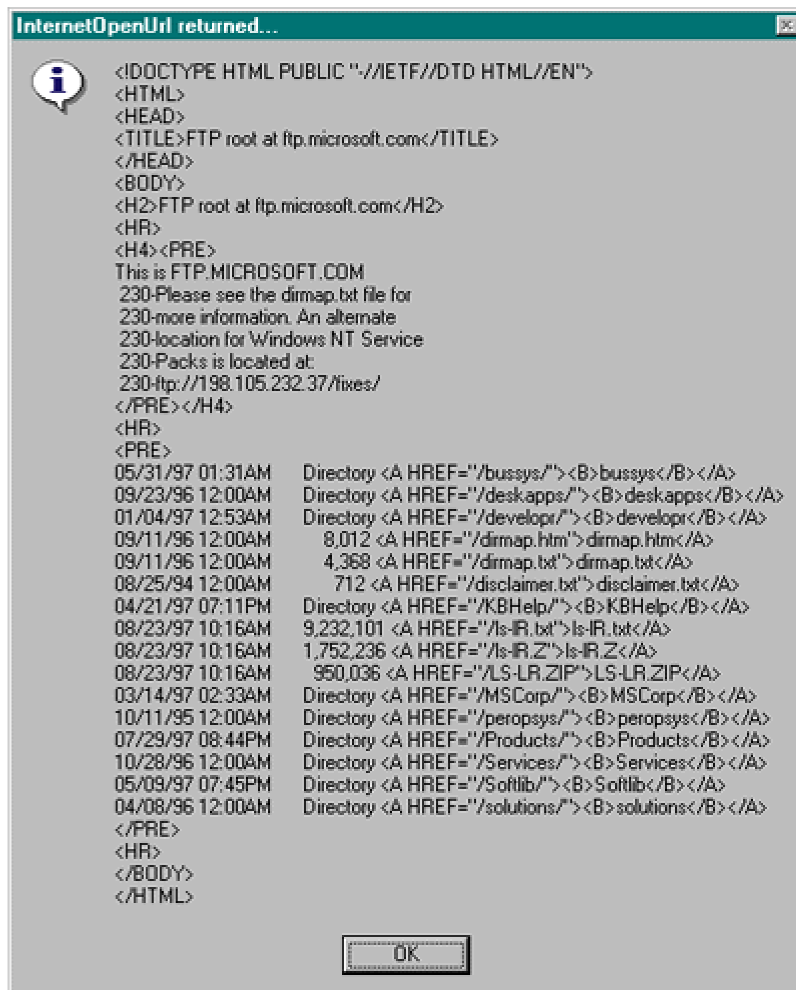

Figure 12: FTP root raw source

While InternetOpenUrl is simpler than the protocol-specific functions, it is somewhat limited in functionality. If you're still looking for a more customizable client/server communications system, you'll want to explore the protocol-specific functions.

FTP, HTTP, and Gopher

WinInet offers a set of functions for each of the three main Internet protocols. Because there is so much information, I'll only scratch the surface on the FTP and HTTP functions (see **Figures 13** and **14**). The WinInet section of the Internet Client SDK documentation contains complete details on the Gopher protocol (see **Figure 15**).

The WinInet FTP functions give your application the ability to manipulate files and directories on an FTP server. There are functions available for creating, removing, and changing directories as well as for finding, deleting, renaming, downloading, and sending files (see **Figure 13**). Remember, before using any of these functions you must already have established a connection with an FTP server by calling InternetOpen followed by InternetConnect (using the INTERNET_ SERVICE_FTP flag). Then you typically call FtpFindFirstFile or FtpSetCurrentDirectory.

Let's look at another example. The code in **Figure 16** demonstrates how to download the README.TXT file found in the developr/visual_c directory on Microsoft's FTP server. This is a simplified example. It assumes that you know the names and locations of the files and directories. But it does show you how simple it is to download a file from an FTP server. Improving this function to accept user input would essentially turn this example into your very own FTP client.

The HTTP functions described in **Figure 14** give you direct control over the behavior of an HTTP session in general and any HTTP request in particular. To begin using HTTP, first call InternetOpen and InternetConnect as you did with FTP (except don't forget to pass INTERNET_SERVICE_HTTP to InternetConnect). Then call HttpOpenRequest to create an HTTP request handle. This handle stores all of the request properties passed into HttpOpenRequest. The next step is to call HttpSendRequest. HttpSendRequest actually requests the specified object and sends any supplied data over the network. Once HttpSendRequest succeeds, you can read the response by using the InternetReadFile function.

Another cool feature of the WinInet HTTP functions is the ability to use the Secure Sockets Layer (SSL). This is especially helpful if you're developing a client/server system to send confidential information over the Internet. For example, if you're developing a purchasing system that needs to send credit card numbers over the Internet, using some type of secure communications is necessary. Otherwise, you'll probably start losing customers very quickly.

Currently, I'm developing a Windows-based Internet purchasing system for a client. Not too long ago the client asked me to upgrade all of the product communications to use SSL. Since I'm using WinInet for all Internet communications throughout the app, the upgrade required very little effort. In fact, it was as simple as adding INTERNET_FLAG_ SECURE (synonymous with https) on all calls to HttpOpenRequest and changing the port value used by InternetConnect to INTERNET_DEFAULT_HTTPS_ PORT. By making these simple changes, all information passed between the app and the server is now encrypted and secure. And best of all, it's transparent to the user of the WinInet-based program. In fact, if a developer didn't know better, she might not even realize that the encryption/decryption is taking place behind the scenes. However, since using SSL is obviously going to slow things down a bit, it's wise to only use SSL on requests that contain confidential information.

Figure 17 shows how to make a secure HTTP request. Since the request uses INTERNET_FLAG_ SECURE, the data string "CreditCard=0123456789" will be encrypted before it is sent. The response from the Web server will also be encrypted, but before you begin to read the file WinInet will have decrypted it for you already.

At this point, you can probably appreciate the simplicity of InternetOpenUrl. But after mastering the process of using the straight FTP and HTTP functions, you'll appreciate even more the flexibility and control you have over the communications functionality.

Visual C++ WinInet Classes

If you're an MFC developer, you'll be glad to know that beginning with version 4.2, MFC offers a set of WinInet classes to encapsulate the API functionality described in this article. But even if you're not an MFC guru, you may find that the MFC object-oriented approach fits better into your application design. The MFC WinInet classes offer plenty of advantages. Not only do the classes use familiar MFC file input and output, you get other object-oriented features like default parameters and exception handling. Plus, the classes automatically clean up open Internet handles and connections.

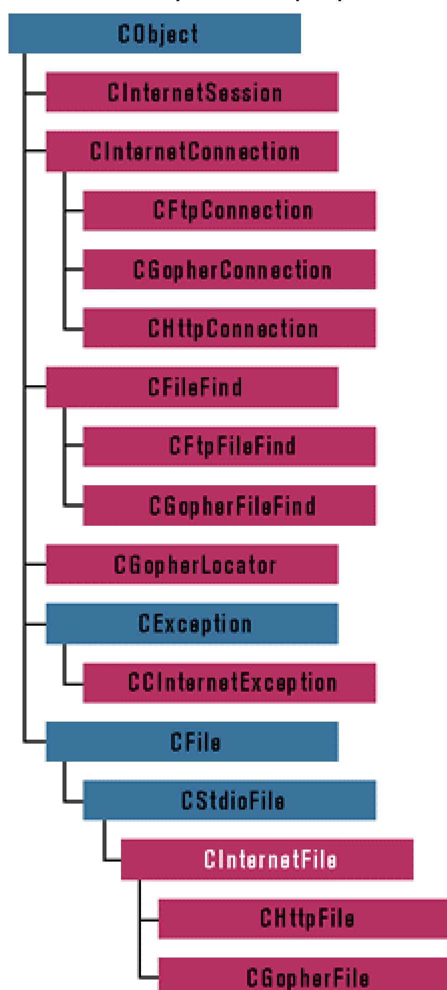

Figure 18: WinInet MFC classes

Figure 18 illustrates how the WinInet classes fit into the MFC hierarchy. Each of the WinInet MFC classes encapsulates functional groups of the WinInet API. For example, CInternetSession encapsulates the InternetOpen and InternetConnect functions. CFtpConnection encapsulates FTP-specific functions like FtpSetCurrentDirectory, FtpGetFile, and FtpPutFile. This object-oriented approach is easier for many developers to understand and implement.

Let's take a brief look at how you can use these classes in a simple MFC application. **Figure 19** contains the code from a sample I wrote called MyFTP. You can download it from the link at the top of this article. I minimized error checking and exception handling to keep things simple and easy to understand, so enhancing the code is up to you.

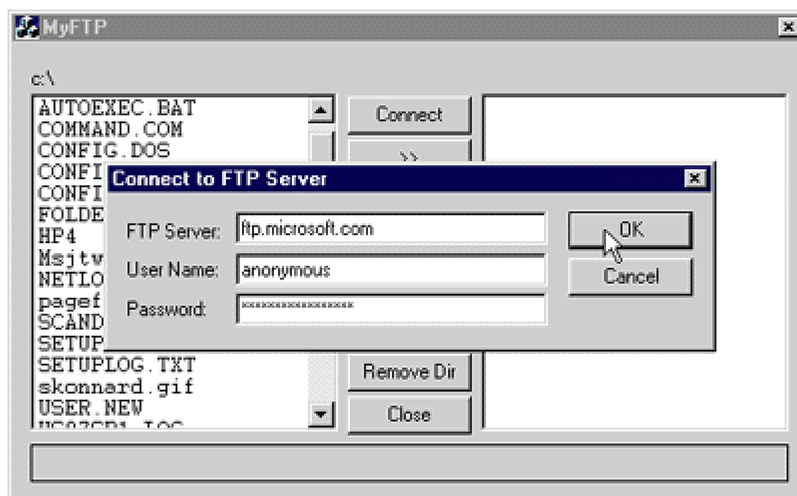

Figure 20: Connecting to an FTP server

MyFTP is a dialog-based application that performs the most common FTP functions. When the application starts, it checks for a connection to the Internet. If it cannot find a valid connection, it will invoke and establish an automatic dial-up connection (see the call to InternetAutodial in CMyFTPApp::InitInstance). Press Connect and type in the name of the FTP server, a user name, and a password for the FTP server you want a connection to (see **Figure 20**). Once you're successfully connected to an FTP server, the right-hand list box will be populated with the files and directories available on the FTP server. Click on a file in the FTP server list box and press the << (Get) button. After the file is successfully downloaded, it will show up in the local list box (see **Figure 21**).

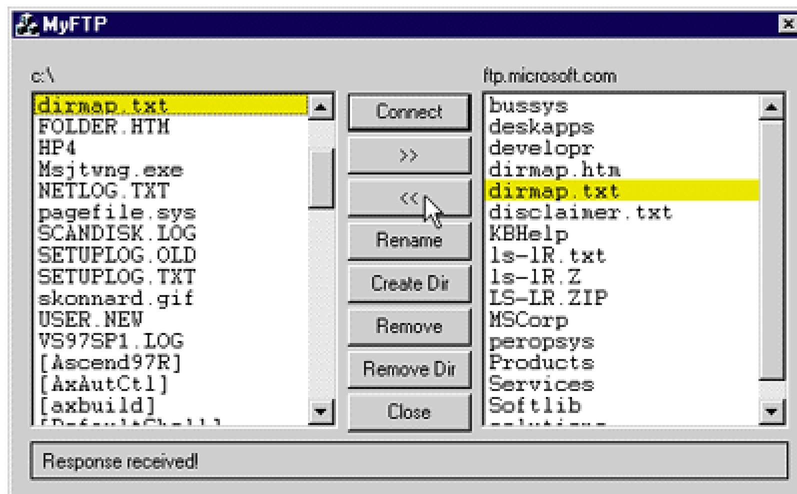

Figure 21: Local list box

Some of the functionality requires that you have write permissions on the FTP server (put file, create directory, delete file, and so on). If you have an FTP account with your ISP, you can connect to that FTP server and log in with your user name and password. Then you should be able to test this functionality as well.

Notice the status bar located on the bottom of the dialog box. Each time you send a request, the status will be updated with the current status text. I accomplished this by deriving my own class, CMyFTPSession, from CInternetSession and overriding the OnStatusCallback virtual method (see **Figure 19**). Inside my implementation of OnStatusCallback, I simply send a message to the dialog telling it to update the status bar text.

After you study the code, you'll notice that the API handle hierarchies and dependencies are evident in the MFC classes as well. There is really not much different about using the

MFC classes except for the object-oriented nature of the design. If you're like me, a piece of code is worth a thousand words. Hopefully this sample helps you understand the nitty-gritty details of using the WinInet MFC classes.

Conclusion

WinInet is another valuable tool found in the Internet Client SDK. It helps developers by providing a high-level API capable of making their applications Internet-aware. This encapsulation makes it possible for developers to forget about the underlying protocol specifics and concentrate on general communication functionality. Not only does WinInet provide dial-up capabilities, it also provides support for HTTP, FTP, and Gopher sessions. If you're looking for something even simpler, WinInet provides functions to support generic URL browsing through any of the protocols.

Even as protocol specifications change, the application's interface to WinInet will remain the same. But even more importantly, WinInet gives developers the flexibility they need to implement customized Internet application strategies.

The WinInet API was designed to be easy to learn and use for most developers who are familiar with Win32. To accommodate the legions of MFC developers out there, Microsoft also designed WinInet MFC classes that encapsulate the WinInet functionality.

See Internet Client SDK Part III: Common Controls

From the December 1997 issue of Microsoft Interactive Developer. Get it at your local newsstand, or better yet, subscribe.