Beating The System: Easy Internet, Part 1

by Dave Jewell

A huge assortment of client-side Delphi components are available for internet programming. They range from the Net Masters controls which are bundled with Delphi, through the excellent (and free!) offerings such as those from Frank Piette and the WinShoes people, to serious commercial products like the recently released Internet Professional toolkit from TurboPower (www.turbopower.com). This last, incidentally, is reviewed in the August issue of *Developers Review*.

The majority of these components sit on top of the WinSock library, which implements the standard TCP/IP protocol stack. Version 2.0 of WinSock is primarily implemented inside a DLL called WS2_32.DLL. From this, you might be forgiven for thinking that Microsoft's web applications such as OutLook Express, Internet Explorer, etc, simply sit on top of WinSock, but in fact, this isn't the case. Enter WININET.DLL, which provides a higher level of functionality and is extensively used by Microsoft's internet applications. If you're using Internet Explorer 4.0 or later, then you can guarantee that the WinINet DLL is available. Because WinINet is the workhorse of Internet Explorer, most of the functionality contained within the library relates to HTTP and FTP access (with a little Gopher functionality thrown in), as we'll see.

Despite WININET.DLL having been around for some time now, many programmers seem to be unaware of it and the extent to which it can simplify web programming. In this month's article, I'll take you on a tour of WinINet, with particular emphasis on the FTP functions, and I'll show you a simple program which illustrates how to browse a server directory using WinINet functions.

Getting Connected

Borland provide a wrapper for WinINet with recent versions of Delphi and C++Builder, meaning that you only need to add `WinINet` to the `uses` clause of your application to get started. Once you've done that, getting an application online is no more complex than executing the code in Listing 1.

Wasn't that easy?! No faffing about with `RASDial`, no mucking about with phone book entries, just one simple call and you've got a connection to the internet. `InternetAutodial` attempts to dial your default internet connection and, as you'll gather from the code, returns `True` on success, `False` if it fails. If you execute `InternetAutodial` while your PC is already online, it detects the connection and returns `True` immediately. The first parameter to the call is a flag which determines how the connection is made; passing `Internet_AutoDial_Force_Online` forces the computer to go online, displaying the standard RAS dial-up dialog (see Figure 1) and waiting for the user to hit the `Enter` key. If the user cancels the dialog, then the operation is aborted and the `InternetAutodial` returns `False`. The second parameter is a window handle which is presumably (the Microsoft documentation is nothing if not vague) used as a parent handle for any dialog boxes which need to be displayed; you can pass zero here if you want.

If you want to frighten the horses, `Internet_AutoDial_Force_Unattended` can be passed as the flag parameter. In this case, the RAS dial-up dialog is still displayed while your modem is dialling and the connection is being authenticated, but it no longer waits for a keypress before starting. In other words, this flag allows you to write programs which ride roughshod over the current setting of the `Connect automatically` checkbox in the RAS dial-up dialog, connecting whenever they want without the user's permission or intervention. Thanks to British Telecom, most of us are still using dial-up modems and paying for connect time, which means that users of such a badly-behaved program will hastily remove it from their hard disk. That said, there are obviously applications (such as email checkers) which have a bona fide reason for unattended internet access.

Hanging up is just as easy, as you can see from the sample code in Listing 2. If successful, the `InternetAutodialHangup` routine returns `True`. The only parameter to this function is reserved, and should always be zero.

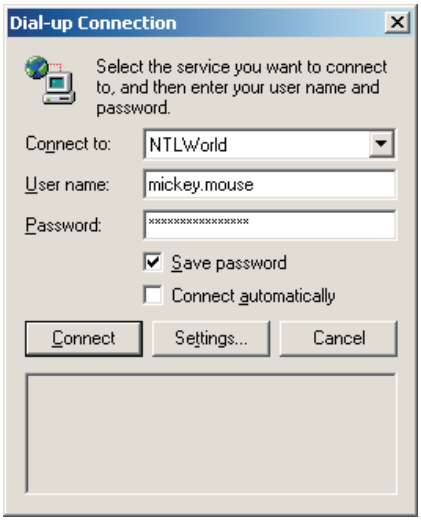
Dead easy, so far, right? Well, yes and no. There is one big caveat you've got to watch for and this concerns the business of sharing a dial-up connection with other applications. In previous versions of Internet Explorer it was possible to dial out using the web browser, start surfing a site and then `Alt-Tab` over to Outlook Express to read your mail. The problem here was that if Outlook Express was configured to drop the line after doing an email send/receive, then drop the line is exactly what it would do, regardless of whether or not it was the application that had dialled

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if InternetAutodial(Internet_AutoDial_Force_Online, Handle) then
    ShowMessage ('We''re connected!');
end;
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
procedure TForm1.DisconnectClick(Sender: TObject);
begin
  if InternetAutodialHangup (0) then
    ShowMessage ('Disconnected');
end;
```



➤ *Figure 1: The RAS dial-up dialog can be bypassed by WinINet clients performing unattended operations such as reading email.*

out in the first place! This problem has been fixed in Internet Explorer 5.0, but unfortunately the fixed functionality doesn't seem to work for non-Microsoft WinINet clients.

The bottom line? Create a simple do-nothing Delphi application using Listings 1 and 2, and then fire up two instances of the program, A and B. The first instance, A, will initiate a dial-up connection to the Internet while B will simply use this existing connection. Even though B didn't create the connection, closing B will drop the line which really *isn't* what we want. Why didn't Microsoft implement some sort of reference-counting facility inside the WinINet code so that the last application to call `InternetAutodialHangup` is the one that physically drops the line? Oh well.

Be that as it may, there are a handful of other WinINet routines that relate to dialling and establishing a connection. One of these is the `InternetCheckConnection` call which looks like this:

```
function
  InternetCheckConnection(
  lpszUrl: PChar;
  dwFlags: DWORD;
  dwReserved: DWORD): BOOL;
  stdcall;
```

This routine checks to see if a specific connection to the internet can be established. The first parameter can either specify a URL or be `Nil`. If it's not `Nil`, the WinINet code extracts the host server name from the URL and attempts to 'ping' the server to check for connectivity. There are a number of other variations on this theme including `InternetGoOnline`, `InternetGetConnectState` and others, but let's get on to some of the more interesting, higher level functionality. Incidentally, many of the higher level routines that we're going to be looking at will automatically initiate a connection for you when required to do so, so in many cases no explicit call to `InternetAutodial` (or whatever) is actually required.

Starting A Session

The most important single routine in the WinINet library is `InternetOpen`. This routine has to be called before most of the other HTTP or FTP related routines are called. The Delphi prototype of `InternetOpen` looks like this:

```
function InternetOpen(
  lpszAgent: PChar;
  dwAccessType: DWORD;
  lpszProxy, lpszProxyBypass:
  PChar; dwFlags: DWORD):
  HINTERNET; stdcall;
```

The first parameter corresponds to the 'agent' name that's used by the HTTP protocol. You'll typically want to set it to some unique name for your application. The access type parameter relates to the way in which host names are resolved and whether or not a proxy is involved. The simplest option here is simply to set it to the value zero (corresponding to `Internet_Open_Type_PreConfig`) which simply tells WinINet to retrieve all necessary information from the registry.

The `lpszProxy` parameter is a pointer to the name of a proxy server. Again, if this isn't relevant to your application, you can set it to `Nil`. The `lpszProxyBypass` parameter points to an optional list of host names which you don't want to route through the proxy and, as before, can be set to `Nil` in most cases. Finally, the `dwFlags` parameter allows you to set a number of specific options. For example, by passing `Internet_Flag_Async` as a flag, you ensure that this session (ie all the internet operations that 'descend' from this call to `InternetOpen`) are non-blocking asynchronous operations. Obviously, if you go down this route more work is required in the structure of your code so as to know when a particular pending operation has completed. Another possible flag setting is `Internet_Flag_From_Cache`. This directs that all operations are to be performed relative to the local cache, which is maintained by Internet Explorer, without actually going online. If a requested item isn't present in the cache, then an error is returned.

The result of the function is what one might call a 'session handle'. For some odd reason, (this is probably because Borland simply copied the original Microsoft type declarations) the `HINTERNET` type is defined in terms of a pointer rather than a `THandle`. This means that when testing a session handle for failure, we have to compare it with `Nil` instead of zero. The same observation applies to the various other 'handles' in the WinINet API.

OK, we've got a session handle, so what can we do with it? Let's suppose we're trying to start an FTP session. Most of the Microsoft articles describing WinINet services have used the example of a connection to ftp.microsoft.com, so I'll do the same. The code in Listing 3 shows how to open a connection to this server.

As you can see, the code checks that it's got a valid session handle and, if so, calls another routine called `InternetConnect`. If you have not called one of the routines I mentioned earlier to establish a network connection, then the computer will attempt to dial out at the time `InternetConnect` is called. This routine is specifically designed to set up an FTP or HTTP connection with a designated server (remember that HTTP and FTP functionality is what WinINet is all about). Briefly, the first parameter is the session handle that we got earlier from `InternetOpen`. The second parameter is the

name of the server that we want to talk to. This can optionally be a straight IP address such as 11.22.33.44. Next comes the port number: this is the port on which the server is listening for connection requests. For FTP servers, this is usually 21 and for HTTP servers it's 80. Secure HTTP servers use a port number of 443. In this case, we specify a value of `Internet_Default_FTP_Port` which equates to 21.

The port number is then followed by username and password strings. For anonymous FTP access, you can set both of these strings to `Nil`. If we're requesting access to an FTP server, then a `Nil` username string will be converted by the WinINet code to 'anonymous' and a `Nil` password will be converted to your email address. The next parameter specifies the type of service we want. Obviously, `Internet_Service_FTP` indicates we want to use the FTP service. The penultimate parameter specifies the flags to use. As far as I can tell, only one optional flag is defined, `Internet_Flag_Passive`, which specifies a passive type of FTP connection. The final parameter is basically an application-defined context number which can typically be used to recover the application 'context' inside one of the call-back routines defined by the WinINet API. In an OOP based programming language such as Delphi, you'd typically pass `Self` for this parameter. As with `InternetOpen`, `InternetConnect` returns a handle of type `HINTERNET`.

Fun With FTP

We're now all set to explore the FTP-related goodies made available by WinINet. Once we've connected to our FTP server of choice, it would be nice to find out what directory we are located in. This is accomplished through a call to `FtpGetCurrentDirectory`, which looks like this:

```
function FtpGetCurrentDirectory
  (hConnect: HINTERNET;
  lpszCurrentDirectory: PChar;
  var lpdwCurrentDirectory:
  DWORD): BOOL;
  stdcall;
```

As with the other WinINet routines, True is returned on success and False on failure. The first parameter is the connection handle we got from the call to `InternetConnect`. The second parameter points to a buffer where the current directory name will be placed whereas `lpdwCurrentDirectory` specifies the size of this buffer on input, and the number of characters copied to the buffer on exit, easy. In the case of ftp.microsoft.com you'll find that the initial directory is set to `/`, which of course is Unix-speak for the root directory. Remember that FTP servers are typically UNIX or Linux beasts and backslashes will be conspicuous by their absence!

If you want to get to a different directory, you can use the `FtpSetCurrentDirectory` routine which works exactly as advertised. Once again, it returns `True` on success and takes the connection handle mentioned above. The second parameter is a pointer to the required directory. This can be a fully qualified name starting from the root directory, or a path relative to the current directory:

```
function
  FtpSetCurrentDirectory(
  hConnect: HINTERNET;
  lpszDirectory: PChar): BOOL;
  stdcall;
```

Of course, navigating directories isn't much use if we don't know what files and directories are out there. Enter the `FtpFindFirstFile` and `InternetFindNextFile` routines. These, naturally, are analogous to the standard `FindFirst` and `FindNext` routines we all know and love, the big difference being that they're enumerating files on a server that may be half way around the world. Needless to say, they're also a lot slower! Here's what the `FtpFindFirstFile` routine looks like:

```
function FtpFindFirstFile
  (hConnect: HINTERNET;
  lpszSearchFile: PChar;
  var lpFindFileData:
  TWin32FindData;
  dwFlags: DWORD; dwContext:
  DWORD): HINTERNET; stdcall;
```

As you'd expect by now, `hConnect` is the familiar connection handle whereas `lpszSearchFile` is the file that we're looking for. If you want to include every file in your search (ie simply enumerate the contents of the current directory) then you can set this parameter to `*.*`. The `lpFindFileData` parameter points to a standard `TWin32FindData` record as used by the Windows API routines `FindFirstFile` and `FindNextFile`.

There are some significant caveats here: bet you thought it was getting too easy, right? To begin with, you shouldn't rely upon fields such as file creation date/time in the `TWin32FindData` record. We're probably talking to a UNIX box and Microsoft's WinINet documentation specifically states that this sort of information might not be available, in which case the WinINet code simply fills in its 'best guess' of what these fields should be. Thus, creation date and last access date will very often be the same as the modification date.

A more severe restriction applies to the use of `FtpFindFirstFile` itself. Amazingly, you're only allowed to use this routine *once* within the context of a single FTP connection handle. To put this another way, once you've called `InternetConnect` you're allowed one use of `FtpFindFirstFile`. If you

➤ *Listing 3*

```
procedure TForm1.FormShow(Sender:TObject);
begin
  hSession := InternetOpen('DelphiMagWinINetDemo',
    Internet_Open_Type_PreConfig, Nil, Nil, 0);
  if hSession <> Nil then begin
    hService := InternetConnect (hSession, 'ftp.microsoft.com',
    Internet_Default_FTP_Port, Nil, Nil, Internet_Service_FTP, 0, 0);
    if hService <> Nil then
      ShowMessage ('Connected to ftp.microsoft.com')
    else
      ShowMessage ('Can''t connect to ftp.microsoft.com');
    // ..... more code here.....
  end;
end;
```

want to, for example, write a tree-walking directory enumerator (possibly not a good idea with a slow FTP server!) then every time you move to a new directory you'll need to close the current connection handle and then open another one with `InternetConnect`. And yes, I think this is pretty daft too.

Continuing with the description of `FtpFindFirstFile`, the next parameter is a flag which, amongst other things, controls whether WinINet should look in the cache or ignore the cache and go straight to the server. The last parameter is another application-specified context number, as we saw in the final parameter to `InternetConnect`. The return value from `FtpFindFirstFile` is either another non-zero `HINTERNET` handle or zero. If zero is returned, then you can use the standard API `GetLastError` routine to determine what specific error occurred. When enumerating an empty directory, `FtpFindFirstFile` will return zero and `GetLastError` will return `Error_No_More_Files`.

If successful, the handle returned from `FtpFindFirstFile` can be used for subsequent calls to `InternetFindNextFile`. This routine is much simpler, taking only the handle and a pointer to the `TWin32FindData` record:

```
function InternetFindNextFile(
  hFind: HINTERNET;
  lpvFindData: Pointer): BOOL;
  stdcall;
```

I know what you're thinking; why did some dummy define the second parameter as an anonymous pointer instead of another `var TWin32FindData` parameter? Well, there is a little method to the madness, but not much. As you may have inferred from the name, `InternetFindNextFile` isn't used merely by FTP sessions, but by Gopher sessions too. Hence, depending on whether the first parameter was returned from `FtpFindFirstFile` or `GopherFindFirstFile`, the second parameter will point to either a `TWin32FindData` record or a `GOPHER_FIND_DATA` record. A crazy way to define an API? Regular readers of this column will know that further comment on Microsoft's API-writing skills is entirely superfluous.

Trouble?

As I've already pointed out, talking to any sort of server over a dial-up connection, HTTP, FTP or whatever, can be a tediously slow process. Many other factors come in here, such as how busy the server is, network loading and so forth. If you're seriously interested in writing robust web applications then you'll want to 'stress-test' your programs under conditions which simulate heavy network loading. One way I've found to do this is to fire up Outlook Express, subscribe to one of the many 'binaries' newsgroups and download multi-megabytes of assorted binary files in the background while testing your application in the foreground. If your program survives such cruelty without timing out, throwing an un-trapped exception or whatever, it'll probably survive in the real world.

In a similar vein, end-users aren't going to be too impressed with an FTP explorer-style program which spends five minutes enumerating the files on the remote server before displaying any sign of activity to the user (yes, yes, I know that Internet Explorer does exactly that, but surely we can aspire to something better?). The keyword here is feedback, it's important to let the end-user know that your program isn't out to lunch but that something is still happening. To support this, the WinINet library has the concept of the callback routine: an application-supplied function called whenever something has happened. When using WinINet synchronously, callback routines are optional, but they are obviously mandatory when using WinINet asynchronously, because there's no other mechanism for checking on the progress of a particular request.

Because Delphi makes it easy to create threaded applications, my own inclination would be to stick with synchronous (blocking) WinINet calls, offloading the bulk of the WinINet processing onto a background thread which uses callback routines to gather progress information and communicate with the primary thread via the `TThread.Synchronize` method.

So how does the callback mechanism work? WinINet provides a routine called `InternetSetStatusCallback`:

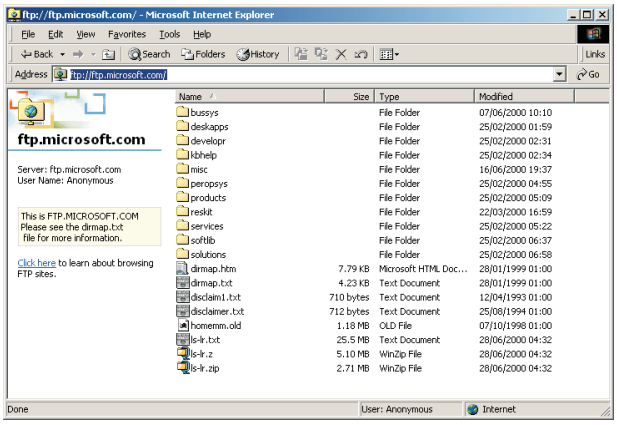
```
function
  InternetSetStatusCallback(
  hInet: HINTERNET;
  lpfnInternetCallback:
  PFNInternetStatusCallback):
  PFNInternetStatusCallback;
  stdcall;
```

The first parameter is our old friend the `HINTERNET` handle. A nice feature of the WinINet API is that any callback function, once applied to a specific handle, will also automatically apply to any derived handles. Thus, you can theoretically (see the later caveat) create a session handle with `InternetOpen`, set up a callback function on that handle, and any derived FTP, HTTP or Gopher handles will automatically 'inherit' the callback that you've set up.

The second parameter is a pointer to our callback routine, which looks like this:

```
procedure MyWinINetCallBack(
  hHandle: HINTERNET;
  dwContext: DWord;
  dwStatus: DWord;
  pStatus: Pointer;
  dwLen: DWord); stdcall;
```

As ever, API-level callbacks must be declared using the `stdcall` attribute otherwise access violations will inevitably result. The first parameter is the handle which 'raised' the callback event. It might be the handle that was originally passed to `InternetSetStatusCallback`, or it might be a derived handle. The `dwStatus` parameter provides a large number of possible status codes which it would be too tedious to list here. The meaning of the `pStatus` parameter depends on the status code in question and `dwLen` specifies how much data is addressed via `pStatus`. I'm sorry if that seems a



➤ *Figure 2: Internet Explorer contains built-in FTP browsing facilities courtesy of the WinINet library it sits on top of.*

bit vague, but if you've got access to the Platform SDK, then take a look at the documentation for yourself and you'll see what I mean. Fortunately, there is a simpler way to recover human readable status information at the time a callback is triggered, as we'll see later. Finally, the function result returned from `InternetSetStatusCallback` is the address of any previously existing callback function.

Here's that caveat I mentioned. I said 'theoretically' earlier because there's one interesting subtlety which requires some caution. As I've already mentioned, it's tempting to use the `dwContext` parameter (used with many WinINet functions) to pass `Self` so that you can recover the context of a Delphi object within the callback routine. However, you'll notice that the `InternetOpen` routine itself doesn't take a context number, and it's therefore impossible to associate a `TObject` instance with a callback routine at the session level. This raises the question of exactly what context number gets passed when a session-level notification takes place? In my own experiments, I was unable to persuade a session-level event to trigger a callback, so I decided that the best option was to play safe and set up the callback at the connection handle level, ie FTP, HTTP or whatever.

Putting It All Together

In Figure 2 you can see what the ftp.microsoft.com site looks like as seen from Internet Explorer. Figure 3 shows the same information viewed from my little FTP browser demo, the source code of which can be seen in Listing 4. This is a 'belt and braces' implementation, which does not use threading, but it does illustrate how to put together the various WinINet calls that I've discussed so far.

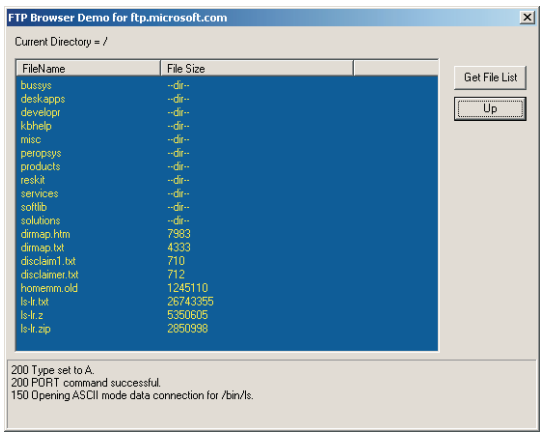
If you double click a directory name, the program will go into that directory and redisplay the files contained therein. To 'back out' to the previous directory level, just click the succinctly named `Up` button. The current directory name is displayed in the top left corner of the window and any status messages received via callback routines are displayed in the bottom pane.

As you'll see from the code, the action starts in the `GetFileListButtonClick` routine which calls `InternetOpen` to create a new session handle. The parameters used here represent just about the simplest way of calling the routine. If a valid handle is returned, then the `GetFileList` method is called to retrieve the current directory listing from the server. A `finally` clause ensures that the session handle is closed after the directory listing has been retrieved.

The `GetFileList` routine is considerably more complicated. Firstly, it sets the cursor to an hourglass before opening an FTP connection to Microsoft's server. It is at this point that the computer dials out, if an internet connection does not already exist. Notice how the `Self` parameter is passed as the final parameter to `InternetConnect` after casting it to a `Cardinal`. If this call succeeds, then a callback function, `MyCallback`, is set up and the status pane caption is altered to reflect the fact that a connection has been established.

The program keeps track of the current directory location on the FTP server and this is set using the `FtpSetCurrentDirectory` call. The current path is also retrieved at this point before the code goes into the main enumeration loop. This is preceded by a call to `FtpFindFirstFile` which retrieves the initial 'find handle' and initialises the `TWin32FindData` with information pertaining to the first file entry. You'll notice that each time round the loop, the `GetLastError` routine is called to see if we've reached the end of the enumeration. Each retrieved file or directory name is added to the listview control and the `SubItems` property of the associated item is initialised to reflect whether we're dealing with a directory or a file. If the latter, then the file size is retrieved. (Yes, yes, I know I've only bothered with the low 32-bits of the file size, but how many 4Gb files have you downloaded recently? Then again, this is the Microsoft FTP site, so you may be right. ☺)

There's one *very important* point which I should make about the callback routine, `MyCallBack`. As you can see, the second parameter corresponds to the Delphi object instance which we passed in earlier. This *completely contradicts* Microsoft's documentation which states that you get a pointer to the application supplied value, you



➤ *Figure 3: Source to this FTP browser is in Listing 4: it shows the rudiments of working with the WinINet library.*

don't! My first crack at a callback routine kept keeling over with an access violation because I was using `var` to access the `Self` parameter. Once I realised that the Microsoft documentation was in error, I removed the `var` keyword and everything started working. And yes, Cathy, this is yet another reason why this month's copy was late. Eleventh Hour Productions strikes again!

Once a callback notification has been received, the routine calls `InternetGetLastResponseInfo` to retrieve a response string from the WinINet library. For sure, you may well get more detailed information by examining each and every possible status word as per the Microsoft documentation, but I needed a quick way of seeing what sort of responses were being received. As you can see from Figure 3, when working with an FTP connection, the callback mechanism can be used to get standard FTP response strings from the server.

The rest of the code is pretty trivial. The `FileListDblClick` routine checks to see if a directory has been double clicked and, if so, forms a new directory path, before calling the `GetFileListButtonClick` routine to display the new directory contents. Similarly, the `UpButtonClick` routine is used to return to the parent directory level. The `FileListCompare` code is

➤ *Listing 4*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons, ComCtrls, WinINet,
  ExtCtrls;
type
  TForm1 = class(TForm)
    FileList: TListView;
    GetFileListButton: TButton;
    CurrentDir: TLabel;
    Panel1: TPanel;
    Status: TLabel;
    UpButton: TButton;
    procedure GetFileListButtonClick(Sender: TObject);
    procedure FileListDblClick(Sender: TObject);
    procedure UpButtonClick(Sender: TObject);
    procedure FileListCompare(Sender: TObject; Item1, Item2:
      TListItem; Data: Integer; var Compare: Integer);
  private
    hSession: HInternet;
    hFTP: HInternet;
    hFind: HInternet;
    ThisDir: String;
    procedure GetFileList (const Dir: String);
  public
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure MyCallBack (hHandle: HINTERNET; Self: TForm1;
  dwStatus: DWord; pStatus: Pointer; dwLen: DWord); stdcall;
var
  ErrNum, BuffSize: DWord;
  szBuff: array [0..1024] of Char;
begin
  BuffSize := sizeof (szBuff);
  if InternetGetLastResponseInfo(ErrNum, szBuff,
    BuffSize) then
    if (BuffSize > 0) and (szBuff [0] <> #0) then
      Self.Status.Caption := szBuff;
end;
procedure TForm1.GetFileList (const Dir: String);
var
  Item: TListItem;
  szDirSize: Cardinal;
  FileData: TWin32FindData;
  szDirectory: array [0..512] of Char;
begin
  Screen.Cursor := crHourGlass;
  try
    hFTP := InternetConnect (hSession, 'ftp.microsoft.com',
      Internet_Default_FTP_Port, Nil, Nil,
      Internet_Service_FTP, 0, Cardinal (Self));
    if hFTP <> Nil then try
      InternetSetStatusCallback (hFTP, @MyCallback);
      Status.Caption := 'Connected to ftp.microsoft.com';
      if Dir <> '' then
        FtpSetCurrentDirectory(hFTP, PChar(Dir));
      // Get current directory
      szDirSize := sizeof (szDirectory);
      if FtpGetCurrentDirectory(hFTP, szDirectory,
        szDirSize) then begin
        ThisDir := szDirectory;
        CurrentDir.Caption :=
          'Current Directory = ' + szDirectory;
      end;
      // The main enumeration loop
      FileList.Items.Clear;
      hFind := FtpFindFirstFile(hFTP, '*.*', FileData,
        0, Cardinal(Self));
      if hFind <> Nil then try
        while True do begin
          if GetLastError = Error_No_More_Files then break;
          // We've got a file
          Item := FileList.Items.Add;
          Item.Caption := FileData.cFileName;
          // Is this a directory or a file?
          if FileData.dwFileAttributes =
            File_Attribute_Directory then begin
            Item.Data := Pointer (1);
            Item.SubItems.Add ('--dir--');
          end else begin
            Item.Data := Nil;
            Item.SubItems.Add(IntToStr(
              FileData.nFileSizeLow));
          end;
          if not InternetFindNextFile(hFind, @FileData)
            then break;
        end;
      finally
        InternetCloseHandle (hFind);
      end;
    finally
      InternetCloseHandle (hFTP);
    end;
  finally
    Screen.Cursor := crDefault;
  end;
end;
procedure TForm1.GetFileListButtonClick (Sender: TObject);
begin
  hSession := InternetOpen('DelphiMagWinINetDemo',
    Internet_Open_Type_PreConfig, Nil, Nil, 0);
  if hSession <> Nil then try
    GetFileList (ThisDir);
  finally
    InternetCloseHandle (hSession);
  end;
end;
procedure TForm1.FileListDblClick(Sender: TObject);
var
  Item: TListItem;
  NewDir: String;
begin
  // Is this a directory double-click?
  Item := FileList.Selected;
  if (Item <> Nil) and (Item.Data <> Nil) then begin
    NewDir := ThisDir;
    if NewDir = '' then NewDir := '/';
    if NewDir [Length (NewDir)] <> '/' then
      NewDir := NewDir + '/';
    ThisDir := NewDir + Item.Caption;
    GetFileListButtonClick(Sender);
  end;
end;
procedure TForm1.UpButtonClick(Sender: TObject);
var
  p: PChar;
  NewDir: array [0..512] of Char;
begin
  if (ThisDir <> '') and (ThisDir <> '/') then begin
    StrPCopy (NewDir, ThisDir);
    p := StrRScan (NewDir, '/');
    if p <> Nil then begin
      p^ := #0;
      ThisDir := NewDir;
      if ThisDir = '' then ThisDir := '/';
      GetFileListButtonClick (Sender);
    end;
  end;
end;
procedure TForm1.FileListCompare(Sender: TObject; Item1,
  Item2: TListItem; Data: Integer; var Compare: Integer);
begin
  Compare :=
    Ord(Ord(Item1.Data <> Nil) < Ord(Item2.Data <> Nil));
end;
end.
```

used by the listview control to ensure that all directories are listed before files. If you compare Figures 2 and 3, you'll see that the display order is identical.

Conclusions

And that's about it for this month. Next time round, we'll look at what other FTP functionality is available and wrap things up into a reusable component that's capable of downloading (and maybe even uploading) files from an FTP server using a separate background thread. If space and time allow, we'll also take a look at the HTTP-related calls that are available.

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is the Technical Editor of *Developers Review* which is also published by iTec. You can contact Dave as TechEditor@itecuk.com