Beating The System: Easy Internet, Part 2

by Dave Jewell

In last month's article on WinINet programming, I introduced you to the often overlooked WinINet DLL, looked at basic issues such as establishing an internet connection and then described a simple WinINet-based program which could be used to browse the contents of an FTP site. This month, we'll take a look at some of the other routines in WinINet, and rewrite the code so as to end up with a reusable component.

The Gentle Art Of FTP File Fetching

But first, we need to tackle the thorny issue of downloading a specified file from our FTP server. You'll notice that I cunningly avoided this topic last month, concentrating instead on directory browsing. The fact is that, even with WinINet, downloading a file can be somewhat more complicated than it ought to be. To begin with, let's look at the obvious (and wrong) way to do it.

The simplest way of retrieving a file from an FTP server with WinINet is to use the `FtpGetFile` routine, the function prototype for which is shown in Listing 1. As discussed last month, the first parameter, `hConnect`, is a handle to a valid FTP session and `lpszRemoteFile` is the name of the required file on the remote server (this can be an absolute file location or relative to the currently set directory). `lpszNewFile` specifies where we want to place the file on the local machine and the `fFailIfExists` parameter is used to indicate that the function should fail if the specified local file already exists.

➤ *Listing 1*

```
function FtpGetFile (hConnect: HINTERNET; lpszRemoteFile: PChar;
  lpszNewFile: PChar; fFailIfExists: BOOL; dwFlagsAndAttributes: DWORD;
  dwFlags: DWORD; dwContext: DWORD): BOOL stdcall;
```

My advice here would be to explicitly check if the local file is present yourself! After all, it's not exactly a lot of work to call `FileExists`, and it's better to be safe than sorry. I wouldn't want you to think I don't trust Microsoft's code, but...☺

Next up, `dwFlagsAndAttributes` specifies the required attributes of the newly created local file, whereas `dwFlags` determines whether it's a binary or ASCII transfer and allows other flags pertaining to the cache (not relevant to us) to be specified. Finally, `dwContext` allows an application-supplied value to be passed, for use by callback routines. Again, this was discussed last month.

Once you've called `FtpGetFile` and received a result of `True`, that's the end of the story: you've got the file. But because this is a very high level routine, everything happens inside the call and there's very little opportunity to get in on the act. For sure, you could pass `Internet_Flag_Async` to the initial `InternetOpen` call, thereby ensuring that `FtpGetFile` is called asynchronously, but how do you then keep track of the progress of the file transfer? As I discussed last time, WinINet provides a generic mechanism for implementing callbacks via the `InternetSetStatusCallback` routine and it would be nice if we could use this mechanism to receive progress on the state of a lengthy file transfer. Unfortunately, such niceties didn't occur to the designer of WinINet because file transfer progress info isn't on the menu as far as callbacks are concerned.

As is usually the case with Microsoft's APIs, it's really a case of 'if you want the job done properly, then do it yourself'. The secret is to use the `FtpOpenFile` call rather than `FtpGetFile`:

```
function FtpOpenFile(
  hConnect: HINTERNET;
  lpszFileName: PChar;
  dwAccess: DWORD;
  dwFlags: DWORD;
  dwContext: DWORD): HINTERNET;
  stdcall;
```

Once again, the first and second parameters specify the FTP session handle and the name of the file that we're after. The `dwAccess` parameter can be set to either `GENERIC_READ` or `GENERIC_WRITE`. Write, do I hear you cry? Well, bear in mind that `FtpOpenFile` does exactly what it says, it simply opens a file and gives us a handle to it. It can be used prior to file upload operations as well as downloads. Once we've got a handle to the required file, we can then read the file, a bit at a time, by using the `InternetReadFile` call:

```
function InternetReadFile(
  hFile: HINTERNET;
  lpBuffer: Pointer;
  dwNumberOfBytesToRead: DWORD;
  var lpdwNumberOfBytesRead:
  DWORD): BOOL; stdcall;
```

The first parameter to this routine is `hFile`, the file handle we got from the call to `FtpOpenFile`. This is followed by a pointer to a buffer into which the data is read. Finally, the next parameter specifies the size of the supplied buffer and the final, `var`, parameter is used to indicate the number of bytes actually read. There's a lot to say about the `InternetReadFile` routine. To begin with, astute readers will have inferred from the name that it's not FTP-specific. The same routine is also used for HTTP file transfers and in conjunction with the Gopher functionality contained within WinINet. Essentially, it's a general purpose 'chunk of data' reader.

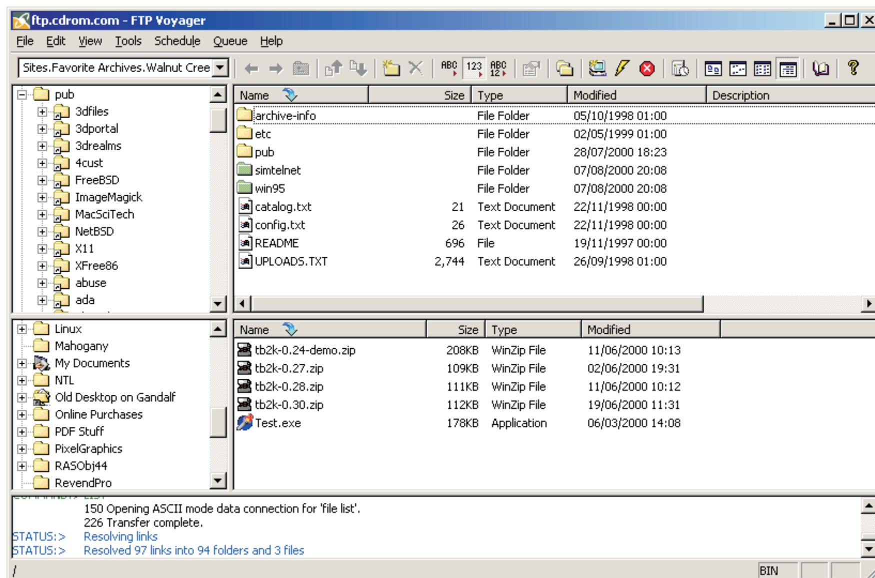
As I mentioned last month, I reckon that the easiest approach when working with WinINet is to use synchronous calls which have

been offloaded onto a background thread. In other words, my overall strategy when downloading a file via FTP with WinINet is to establish a network connection, open an FTP session, call `FtpOpenFile` to get a handle to the required file and then (and only then) create a background thread which receives the data from the remote server. The thread basically has to sit in a loop reading data from the server until the end of the file is reached. Once the file has been downloaded, then the thread is destroyed.

To use `InternetReadFile` you need a buffer which is big enough to hold a reasonable amount of data, say 4Kb. When used synchronously, the routine won't return until that amount of data has been read from the server. If no error occurred, and `lpdwNumberOfBytesRead` is less than the supplied buffer size, then you know that you've got to the end of the file. Clearly, allocating a very large buffer means that you'll only infrequently be able to update status information, progress bars, etc, so you may as well have used `FtpGetFile`. On the other hand, a very small buffer size (say 256 bytes) will radically increase the overhead involved in the file transfer and slow down the overall operation. I felt that 4Kb was a reasonable compromise, and this seems to agree with many commercial applications, such as *FTP Voyager*, which you can see updating the 'bytes received' count in increments of 4,096.

Making Progress

This raises another issue: how do you know how big the file is? If you've used Internet Explorer to download many files from the internet, you'll have seen that most of the time IE knows how big the remote file is and can therefore indicate that you are (for example)



➤ *Figure 1: You can use the WinINet FTP functionality as the basis of a powerful FTP browser program such as FTP Voyager from Deerfield (visit www.ftpvoyager.com for more details).*

57% of the way through the download. However, you'll also have noticed that there are occasions when IE isn't able to determine the size of the remote file, and it simply reports *'Unknown file size'* and contents itself with displaying the number of bytes that have been downloaded so far. This is a pretty common scenario when performing FTP transfers: you can't guarantee that the other end of the connection knows exactly how many bytes are going to be transferred and you need to cater for this in your progress dialog. If you don't know the overall file size, then don't display a progress bar, because you will profoundly irritate your users when the bar gets to the end of the track and then starts over again from scratch! (I've seen applications that do this!).

The simplest way of determining the size of the remote file is through a call to the `FtpFindFirstFile` routine which we looked at last time. As you'll remember, this fills in a `TWin32FindData` structure that gives us, amongst other things, the size of the file. If the call to `TfpFindFirstFile` fails, then we just make do as best we can.

Putting all this together, I came up with the reusable class, `TFTPFileReader`, shown in Listing 2. This isn't a component in the normal sense; as you can see, it derives directly from `Tobject`. Think of it more as a drop-in 'helper' class that's intended for incorporation into a larger project and reused from there. For maximum flexibility (and, admittedly, to make life easier for myself) I wrote it in such a way that the onus is on the host application to establish a connection to the internet. A handle to this connection, of type `HINTERNET`, must be established as discussed last month and passed to the file reader class via the `NetConnection` property. At the same time, the host application needs to fill in the `SourceFileName`, `DestFileName` and `ServerName` fields so that the file reader knows what file's being downloaded, what it should be called on the local machine and where to download it from, respectively.

➤ *Listing 2*

```
unit FTPReader;
interface
uses Windows, WinINet, SysUtils, Classes, Dialogs;
const
  BufSize = $1000;
type
  TFTPFileReader = class;
  TFTPFileReaderThread = class (TThread)
  private
    Owner: TFTPFileReader;
    procedure DoProgress;
  public
    procedure Execute; override;
  end;
  TFTPFileReader = class (TObject)
  private
{ CONTINUED ON FACING PAGE... }
```

```
{ ...CONTINUED FROM FACING PAGE }
    fNetConnection: HInternet;
    fFTPSession: HInternet;
    fFileHandle: HInternet;
    fSourceFileName: String;
    fServerName: String;
    fDestStream: TFileStream;
    fDestFileName: String;
    fUserName: String;
    fPassword: String;
    fFileSize: Integer;
    fOwnFTPSession: Boolean;
    fServerPort: Integer;
    fTotalBytesRead: Integer;
    fOnProgress: TNotifyEvent;
    fCompletionString: String;
    fOnCompletetion: TNotifyEvent;
    fThread: TFTPFileReaderThread;
    fBuffer: array [0..BufSize - 1] of Char;
    procedure Cleanup (Fail: Boolean);
    procedure Panic (const Message: String);
    function StartSession: Boolean;
    procedure ThreadTerminated (Sender: TObject);
  public
    destructor Destroy; override;
    procedure Execute;
    procedure CancelTransfer;
    property FileSize: Integer read fFileSize;
    property TotalBytesRead: Integer read fTotalBytesRead;
    property CompletionString: String
      read fCompletionString;
    property NetConnection: HInternet read fNetConnection
      write fNetConnection;
    property FTPSession: HInternet read fFTPSession
      write fFTPSession;
    property SourceFileName: String read fSourceFileName
      write fSourceFileName;
    property DestFileName: String read fDestFileName
      write fDestFileName;
    property ServerName: String read fServerName
      write fServerName;
    property ServerPort: Integer read fServerPort
      write fServerPort;
    property OnProgress: TNotifyEvent read fOnProgress
      write fOnProgress;
    property OnCompletion: TNotifyEvent read fOnCompletetion
      write fOnCompletetion;
  end;
implementation
destructor TFTPFileReader.Destroy;
begin
  Cleanup (False);
  Inherited Destroy;
end;
procedure TFTPFileReader.Cleanup (Fail: Boolean);
begin
  // Close the destination file stream if open;
  fDestStream.Free;
  fDestStream := Nil;
  if Fail then
    DeleteFile (fDestFileName);
  // Close the source file if its open
  if fFileHandle <> Nil then begin
    InternetCloseHandle (fFileHandle);
    fFileHandle := Nil;
  end;
  // Tear down the FTP session, if any
  if fOwnFTPSession and (fFTPSession <> Nil) then begin
    InternetCloseHandle (fFTPSession);
    fFTPSession := Nil;
    fOwnFTPSession := False;
  end;
end;
procedure TFTPFileReader.Panic (const Message: String);
begin
  Cleanup (True);
  raise Exception.Create (ClassName + ': ' + Message);
end;
procedure TFTPFileReader.CancelTransfer;
begin
  Cleanup (True);
end;
function TFTPFileReader.StartSession: Boolean;
var
  FindHandle: HInternet;
  FindData: TWin32FindData;
  szUserName, szPassword: PChar;
begin
  // Do we need to create an FTP session ?
  if fFTPSession = Nil then begin
    if fNetConnection = Nil then
      Panic ('No Network connection specified');
    if (fUserName = '') or (fPassword = '') then begin
      szUserName := Nil;
      szPassword := Nil;
    end else begin
      szUserName := @fUserName [1];
      szPassword := @fPassword [1];
    end;
    fFTPSession := InternetConnect(fNetConnection,
      PChar(fServerName), fServerPort, szUserName,
      szPassword, Internet_Service_FTP, 0, 0);
    if fFTPSession = Nil then
      Panic ('Can''t create an FTP session');
    fOwnFTPSession := True;
  end;
  // We've got an FTP session, how big is the file?
  FindHandle := FtpFindFirstFile(fFTPSession,
    PChar(fSourceFileName), FindData, 0, 0);
  if FindHandle <> Nil then begin
    fFileSize := FindData.nFileSizeLow;
    InternetCloseHandle (FindHandle);
  end;
  // Now, try and open the file for transfer
  fFileHandle := FtpOpenFile (fFTPSession,
    PChar(fSourceFileName), Generic_Read,
    Ftp_Transfer_Type_Binary, 0);
  Result := fFileHandle <> Nil;
end;
procedure TFTPFileReader.Execute;
begin
  // Perform all needed sanity checks....
  if fServerName = '' then
    Panic ('No server name specified');
  if fSourceFileName = '' then
    Panic ('No source filename specified');
  if fDestFileName = '' then
    Panic ('No destination filename specified');
  if fServerPort = 0 then
    fServerPort := Internet_Default_FTP_Port;
  // So far so good, now create an FTP session
  if not StartSession then
    Panic ('Requested file not found') else begin
    // Time to create the background thread, create the
    // source file and start rolling
    try
      fDestStream :=
        TFileStream.Create (fDestFileName, fmCreate);
    except
      Panic ('Can''t create destination file');
    end;
    fThread := TFTPFileReaderThread.Create (True);
    fThread.FreeOnTerminate := True;
    fThread.OnTerminate := ThreadTerminated;
    fThread.Owner := Self;
    fThread.Resume;
  end;
end;
procedure TFTPFileReader.ThreadTerminated (Sender: TObject);
begin
  if Assigned(OnCompletion) then
    OnCompletion(Self);
end;
// TFTPFileReaderThread
procedure TFTPFileReaderThread.DoProgress;
begin
  if Assigned(Owner.OnProgress) then
    Owner.OnProgress(Owner);
end;
procedure TFTPFileReaderThread.Execute;
var
  BytesRead: DWord;
  ErrNum, BuffSize: DWord;
  szBuff: array [0..1024] of Char;
begin
  while not Terminated do begin
    BytesRead := 0;
    if InternetReadFile (Owner.fFileHandle, @Owner.fBuffer,
      sizeof (Owner.fBuffer), BytesRead) then begin
      // If got more data, write it to destination file
      if (BytesRead > 0) and
        (BytesRead <= sizeof(Owner.fBuffer)) then begin
        Owner.fDestStream.Write (Owner.fBuffer, BytesRead);
        // Update progress info
        Owner.fTotalBytesRead :=
          Owner.fTotalBytesRead + Integer(BytesRead);
        Synchronize(DoProgress);
      end;
      // If we didn't get any data, but InternetReadFile
      // returned True, then it's EOF. Just terminate the
      // thread by leaving Execute.
      if BytesRead = 0 then begin
        Owner.fCompletionString := 'OK';
        Exit;
      end;
    end else begin
      // It looks bad. InternetReadFile has returned False
      // which basically means its screwed up. Get the last
      // response string from the server and pass it back.
      BuffSize := sizeof (szBuff);
      Owner.fCompletionString := 'Unknown Error';
      if InternetGetLastResponseInfo(ErrNum, szBuff,
        BuffSize) then
        if (BuffSize > 0) and (szBuff [0] <> #0) then
          Owner.fCompletionString := szBuff;
      // We're out'a here
      Exit;
    end;
  end;
end;
end.
```

You don't need to set up the ServerPort property because if you leave it set to zero then it'll automatically be initialised to 21, the default port number for FTP connections. Thus, having set up the other properties previously mentioned, you can just call the Execute method of the file reader class and let it get on with it. Note that the Execute method is essentially asynchronous: it establishes an FTP connection with the server and then returns more or less immediately while the FTP transfer continues in the background. This is done by creating a separate thread to handle the download, as we'll see shortly.

In addition to the above comments, you will see that the TFTPFileReader class also has an extra property, FTPSession. If this is left as zero (the default), then the file reader will create an FTP session of its own. However, by placing the handle of an existing FTP session into this property, the file reader can be made to 'inherit' an FTP session from the host application. Why did I do things like this? Stay with me and I'll explain all.

At any point during the download, if the user gets bored (or whatever), the host application is free to call the CancelTransfer method, which does exactly what you'd expect it to do. Of course, this raises the question of how the host app is supposed to know how the file transfer is progressing and when it has completed. In order to do this, I've added a few other goodies to the class. To begin with, you'll see that there are FileSize and TotalBytesRead properties which indicate the total size of the file and the number of bytes that have been downloaded, respectively. As I've already pointed out, it might not be possible to retrieve the file size from the server, in which case FileSize will report zero. I also resisted the temptation to start calculating 'percentage done', passing references to TProgressBar controls and similar stuff. Good software design always decouples the 'grunt code' from the user interface. Although I don't *always* remember to put this principle into practice, I have done so here! In theory, the host application could create multiple instances of TFTPFileReader, presenting a fancy user interface which shows how each file download is progressing. None of this is relevant to the file reader itself.

Of course, this doesn't mean that the host app should have to continually poll the file reader object in order to determine the current state of play. Instead, I've added a couple of event handlers to take care of things. Firstly, there's an OnProgress event which is triggered whenever fresh data is received from the remote machine. The frequency with which this event gets triggered depends entirely on the value you assign to BufSize since it only fires when another 'buffer's-worth' of data has arrived. With a 56K modem, this happens just under twice per second. If you've got an ADSL connection then firstly I hate you with a passion *[Steady, Dave, I warned you about those death threats to readers before... Ed]* and secondly you might want to implement a somewhat larger buffer size.

You'll also notice that the file reader class has an OnCompletion event. Unsurprisingly, this gets triggered when the file transfer has completed. Inside your OnCompletion event handler, you should check the value of the read-only CompletionString property. This will be set to OK if the file transfer completed successfully, and some other error message if things went pear-shaped for whatever reason.

➤ *Table 1: Usernames and passwords.*

Username	Password	Name Sent	Password Sent
Nil	Nil	'anonymous'	User's email address
Not Nil	Nil	Username	Empty string
Nil	Not Nil	ERROR	ERROR
Not Nil	Not Nil	Username	Password

How It Works

With that as an introduction, let's roll up our sleeves and see how TFTPFileReader works its magic.

Having set up the various properties as mentioned earlier, nothing happens until we call the Execute method: there isn't even a custom constructor for this class. Once Execute is invoked, we perform a few simple sanity checks to ensure the essential properties have been set up, calling the Panic helper routine if things don't look good. As mentioned earlier, the FTP server port number gets initialised to Internet_Default_FTP_Port if no custom port number has been specified.

Next, the StartSession method gets called. If no FTP session handle was supplied, then we create a new FTP session inside this method. As you'll see if you read the MSDN documentation, the InternetConnect routine expects the FTP username and password to be passed as two PChar arguments. Although an empty String in 32-bit Delphi is internally represented as a Nil pointer, it unfortunately gets transmogrified into a non-Nil pointer if you cast it to a PChar! This is why the code here is more verbose than it might otherwise be.

Table 1 has been taken from Microsoft's documentation and 'Pascalised'. As you can see, it shows the various permutations of username and password that can be passed to the InternetConnect routine, and how they translate into what's actually sent to the server. I felt that this was needlessly complicated, so I just wrote the code in such a way as to eliminate the two middle conditions, one of which isn't allowed anyhow! Thus, leave both properties as empty strings for anonymous FTP,

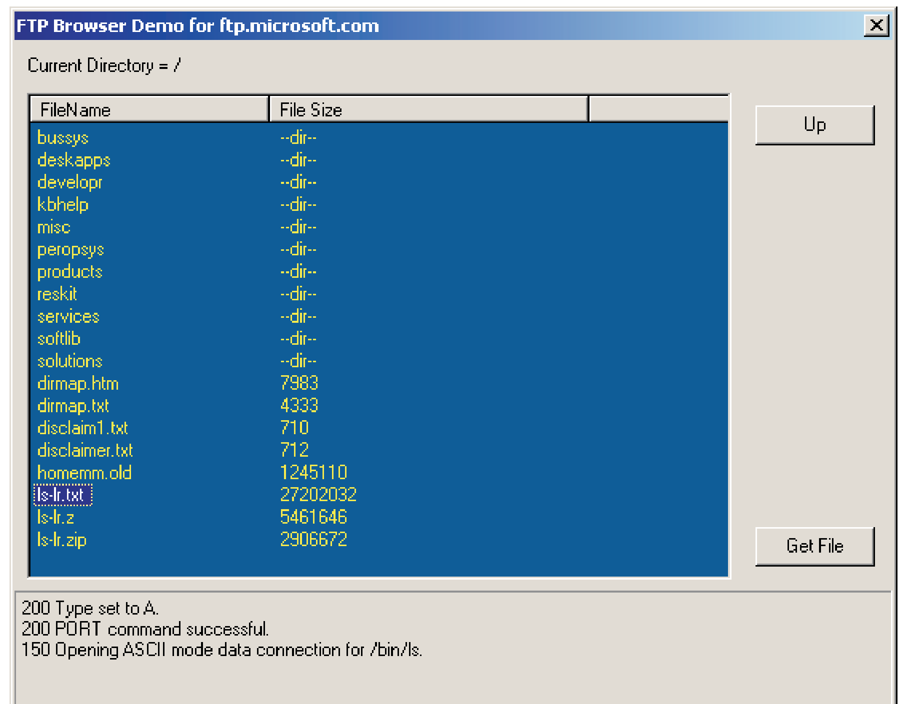
and set them both for servers that require a specific username and password.

Once we've established an FTP session with the server, the next job is to figure out, if possible, the size of the file we're about to retrieve. This is done by passing our FTP session handle to `FtpFindFirstFile`. If this returns a non-nil value, then we retrieve the file size information from the `TWin32FindData` structure (well, the first 32 bits of it, anyway!) and use this to set the `FileSize` property.

Time for an important note. In last month's article, I complained that `FtpFindFirstFile` could only be used once during a specific FTP session. In fact, on closer reading, this turned out to be a misunderstanding on my part. What it says is that you can't have more than one 'find file' enumeration in force at any one time. To put this another way, once you've called `FtpFindFirstFile` on a particular FTP session, you won't be able to do so again until you've closed the existing handle through a corresponding call to `InternetCloseHandle`. In this context, you can think of closing a 'find file' handle as being analogous to calling `FindClose` after using `FindFirst/FindNext`. However, whereas you can nest `FindFirst/FindNext` calls, you can't do the same with the FTP routines. This is the reason I wrote the file reader class in such a way as to optionally share FTP sessions, since it now makes a lot more sense for it to be able to do so.

The final job of the `StartSession` routine is to call `FtpOpenFile`, retrieving a handle to the remote file. Notice that I always open the file in binary mode, ie a byte-for-byte copy. This will obviously work for binary and text files.

Back in the `Execute` method, a `TfileStream` object is created for writing the destination file, and then comes the fun part: the `Execute` method finishes by creating the background thread which does the work of driving the file transfer. Rather than copying umpteen variables from the file reader into the thread object, I just pass a pointer to the thread's 'owner' and then



➤ *Figure 2: My browser doesn't look vastly different from last month, except that it's lost the Get File List button, which was essentially redundant, and sprouted a new Get File button for initiating an FTP transfer.*

use this field to get at required information from inside the thread. The thread is created in a suspended state, but as soon as the `Owner` field and `OnTerminate` handler have been set up, `Resume` is called, which, as you'll know if you've done any thread programming before, causes the thread's `Execute` method to start executing.

Inside the thread's `Execute` method (not to be confused with the `Execute` method of the `TFTPFileReader` class!) we do what every well-behaved thread does and that's to sit in a loop checking the state of the `Terminated` property. If this gets set to `True`, then it means that the thread should shut down. Inside the loop, we call `InternetReadFile`, trying to read the remote file data. Each time that this routine returns success, we write the newly retrieved data to the destination file stream and update the count of the total number of bytes that have been retrieved. Finally, the thread's `DoProgress` method is called, invoking the file reader's `OnProgress` event handler if one has been assigned. Notice that this has to be done via a `Synchronize` call so that the event handler is executed within the context of the application's primary thread. As seasoned Delphi developers know to their cost, the VCL is not re-entrant!

If the `InternetReadFile` routine returned success but didn't actually give us any data, this indicates that the file transfer has completed. In this case, we set the completion string to `OK` as mentioned earlier and exit the routine. This terminates the thread, thereby triggering the thread's `OnTerminate` event which calls the file reader's `ThreadTerminated` method. Inside this method, we call the housekeeping `Cleanup` routine which closes the destination file stream and tears down the FTP session.

Phew! If the unthinkable happens and `InternetReadFile` reports an error, then we use `InternetGetLastResponseInfo` to retrieve a (hopefully!) human readable error string, place it in the `CompletionString` property and terminate the `Execute` loop as before.

Of course, the code I've presented here is specific to FTP file downloads, but not very. While writing this stuff I realised that it'd be quite easy to create a generic

file download program which uses virtual methods to open a session, stop a session and 'drive' the actual file transfer. In this way, you could write an abstract file transfer class, inheriting a different descendant from this class according to whether you're doing an FTP download or using the HTTP functionality, for example. This is left as an exercise for you, dear reader, but hopefully you get the idea.

The FTP Browser Revisited

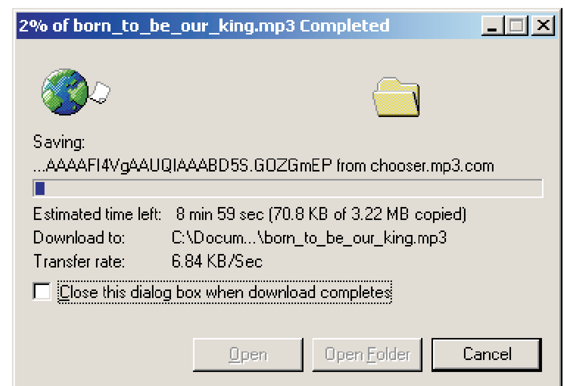
As mentioned elsewhere, I eventually sussed the fact that you *can* perform multiple file enumerations within one FTP session! This being the case, I've modified last month's testbed code to create a single FTP session when it starts running, and to destroy the FTP session only when the application terminates. This simplifies the code and makes things run more quickly because we're not forever having to create a new session.

I've also added the ability to download individual files from the server by double-clicking a file, or by selecting a file and then pressing the `Get File` button (see Figure 2). During a download, the status panel at the bottom of the window reflects the number of bytes that have been transferred and, if available, the total size of the file. I had originally intended to implement a fancy download dialog, complete with cancel button, progress bar and animated AVI, but alas, time was against me as usual. If you do want to implement such a progress dialog, bear in mind that the standard AVI animations implemented via the `CommonAVI` property of the `TAnimate` control don't include what I'll refer to as the 'web copy' animation shown in Figure 3. Fortunately, there's an easy way around this, because I've used a resource editor to extract the necessary .AVI file for you. It's included with this month's files as WEBCOPY.AVI.

If you want to use this animation in a project of your own, just embed the animation into a .RES file as a custom resource of type `AVI`. You can then load the animation into the `TAnimate` control at runtime by setting the controls `ResHandle` property to `HInstance`, set its `ResID` property to whatever numeric ID you've assigned to the resource, and you can then set the `Active` property in the usual way to start the animation.

Listing 3 shows the main additions which I made to last month's FTP browser code in order to integrate it with the reusable FTP file reader class. Briefly, you'll see that I've added a `OnListSelectItem` handler which takes care of ensuring that the file download button is only enabled if a file is currently selected. The `FileTransferComplete` method is invoked when the FTP file reader triggers an `OnCompletion` event. It takes care of updating the status text label to show the success (or failure) of the file transfer. It also destroys the file reader itself and emits a beep to notify the user that the file transfer is complete. Similarly, the `FileTransferProgress` method is invoked whenever more data is received from the remote end. As mentioned already, it simply updates the status label to indicate how much data has been received thus far.

The real meat of Listing 3 is the `DownloadClick` routine which initiates a file transfer. After checking that we're dealing with a file, and confirming that the user does want to perform a download, the code then obtains a destination filename in the time-honoured



➤ *Figure 3: If you want to implement a fancy file download progress dialog such as the one built into Internet Explorer, take a look at the WEBCOPY.AVI file included with this month's project files.*

➤ *Listing 3*

```
procedure TForm1.FileListSelectItem (Sender: TObject; Item:
  TListItem; Selected: Boolean);
begin
  if Selected then
    Download.Enabled := Item.Data = Nil
  else
    Download.Enabled := False;
end;
procedure TForm1.FileTransferComplete (Sender: TObject);
begin
  Status.Caption := 'File Transfer Complete: Result = ' +
    FileReader.CompletionString;
  FileReader.Free;  FileReader := Nil;
  Screen.Cursor := crDefault;
  Enabled := True;
  MessageBeep (0);
end;
procedure TForm1.FileTransferProgress (Sender: TObject);
begin
  Status.Caption := 'Bytes transferred: ' +
    IntToStr(FileReader.TotalBytesRead);
  if FileReader.FileSize <> 0 then
    Status.Caption := Status.Caption + ' out of ' +
      IntToStr(FileReader.FileSize) + ' bytes';
end;
procedure TForm1.DownloadClick(Sender: TObject);
var
  Item: TListItem;
  DirName: String;
begin
  Item := FileList.Selected;
  if (Item <> Nil) and (Item.Data = Nil) then begin
    DirName := Copy(CurrentDir.Caption, 21, MaxInt);
    if MessageDlg('Download ' + Item.Caption +
      ' from directory "' + DirName + '" ?', mtConfirmation,
      [mbYes, mbNo], 0) = mrYes then begin
        SaveDialog.FileName := Item.Caption;
        if SaveDialog.Execute then begin
          Screen.Cursor := crHourGlass;
          Enabled := False;
          FileReader := TFTPFileReader.Create;
          FileReader.SourceFileName := Item.Caption;
          FileReader.DestFileName := SaveDialog.FileName;
          FileReader.ServerName := 'ftp.microsoft.com';
          FileReader.NetConnection := hSession;
          FileReader.FTPSession := hFTP;
          FileReader.OnCompletion := FileTransferComplete;
          FileReader.OnProgress := FileTransferProgress;
          FileReader.Execute;
        end;
    end;
  end;
end;
```

manner and then creates a `TFTPFileReader` object to manage the transfer. The various properties are set up as previously described, and the `OnProgress` and `OnCompletion` event hooks are initialised. It's then just a case of calling the `Execute` method and Bob's your uncle.

Conclusions

I'll admit that this FTP browser/download program would win few prizes for user friendliness. I've included no facility for cancelling the transfer, though that would be easy to incorporate, but more importantly I've neatly side-stepped the issue of multiple file transfers by disabling the main form during a transfer, thus preventing you from downloading anything else! Obviously, you wouldn't do that in a real program, would you?

As it stands, the `TFTPFileReader` class could certainly be instantiated multiple times, but if you go down this route then you'll need to bear in mind Microsoft's dire warnings about only allowing one active file transfer within a single FTP session. What this really means is that if you want to download more than one file at once, then you'll need to open multiple FTP sessions, one for each file. I haven't experimented with this but, in theory, if you leave the `FTPSession` property of the file reader object set to zero, then it'll open a new FTP session just for that file transfer. Now you can see why I allowed for the class to optionally 'inherit' an FTP session handle from the host program.

I hope that this month's and last month's articles have given you a feel for the FTP functionality contained within WinINet. I'll revisit WinINet soon and look at what's available on the HTTP front.

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