# The Delphi CLINIC

**Edited by Brian Long**

***Problems with your Delphi project?***

***Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com***

## QuickReport And Printers

**Q** How can I programmatically tell QuickReports which printer to use when printing, without forcing the user to open up the standard printer setup dialog?

**A** TQuickRep component has a PrinterSettings property, which itself has a PrinterIndex property. This corresponds exactly to the PrinterIndex property of the normal Delphi Printer object (from the Printers unit).

As an example, the project QRPrinters.Dpr on this month's disk shows all available printers in a listbox. This is done by copying the Printers property of the Printer object, which is a TStrings object, to the Items property of the listbox and setting the listbox's ItemIndex to match Printer.PrinterIndex.

The user can select a printer from the listbox and the listbox's ItemIndex property is assigned to the PrinterIndex property of the aforementioned QuickReports PrinterSettings property and also the normal Printer object (ensuring non-QuickReports printing also goes to the specified printer). They can then press a button to print a simple report (a trivial report based on a DBDEMOS table) and it will go to the selected printer. Listing 1 shows the key parts of the code.

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  //Get list of printers displayed in the listbox on the form
  lstPrinters.Items := Printer.Printers;
  //Highlight default printer
  lstPrinters.ItemIndex := Printer.PrinterIndex
end;
procedure TMainForm.lstPrintersClick(Sender: TObject);
begin
  //Record chosen printer as new current printer and as QuickReports target
  Printer.PrinterIndex := lstPrinters.ItemIndex
  ReportForm.QuickRep1.PrinterSettings.PrinterIndex := lstPrinters.ItemIndex;
end;
procedure TMainForm.btnPrintReportClick(Sender: TObject);
begin
  //Print report on newly selected printer
  ReportForm.QuickRep1.Print
end;
```

➤ *Listing 1: Telling QuickReports to print to a specified printer.*

```
BOOL CopyFile(
  LPCTSTR lpExistingFileName,// pointer to name of an existing file
  LPCTSTR lpNewFileName,     // pointer to filename to copy to
  BOOL bFailIfExists  // flag for operation if file exists
  );
```

➤ *Listing 2: The C declaration for the Win32 CopyFile API.*

## File Copying

**Q** I want to incorporate a backup facility by copying my application's database files to a Zip Drive, but I can't seem to find a way of doing this from within the application itself. In the Delphi help files there is mention of a Win32 API CopyFile function, but without any indication of the syntax required to use it. Once again, can you help?

**A** A good source of file manipulation routines (including a CopyFile routine) can be found in the FMXUtils unit of the FilManEx demo (a file manager example project). This demo can be found in Delphi's Demos\Doc\FilManEx directory. Nick Hodges mentioned this unit in the *Tips & Tricks* column all the way back in Issue 2 of *The Delphi Magazine*. Alternatively, you could use either of the file copying routines listed in my *File Handling 5: Streaming Components* article from Issue 10.

Neither of these solutions use the Win32 CopyFile API. Instead they use Delphi file handling routines or file streams to do the job. However, using this API could simplify the implementation of a file copying routine significantly, since it is fully self-contained. To get to the Windows API help for this routine, simply type it into a blank area in the Delphi editor and press F1. If you are given an option of going to the MAPI help file or the Win32 help file (as happens with Delphi 2), choose the latter.

The API help, as usual, is written for C programmers, and shows the CopyFile API declared as shown in Listing 2.

This is saying it takes two C strings for the source and destination filenames, and a Boolean that indicates what to do if the target file exists. A value of True means the function will fail, returning False, whereas a value of False means the file will be overwritten. For many Delphi programmers, looking at C syntax just causes headaches, so it is always sensible to look at the Delphi definition of it.

Pressing the Quick Info button in the API help file (see Figure 1) tells us that CopyFile is supported on Windows 95 (and therefore Windows 98) and Windows NT, which is good news. It also says that (for C programmers) the API information is declared in the WinBase.h header file.

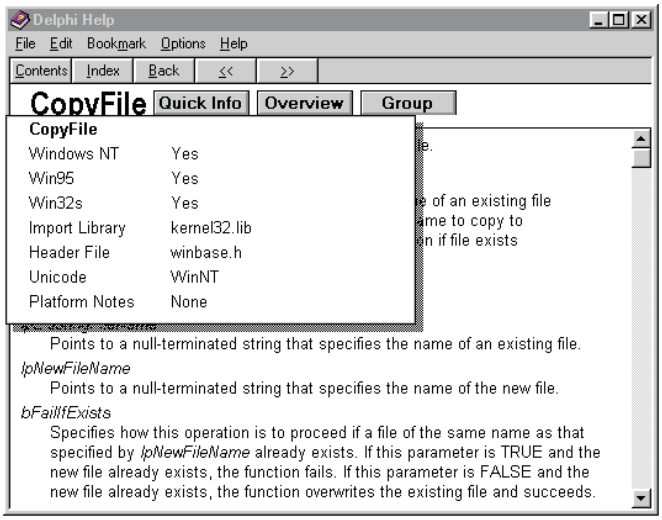
What we as Delphi programmers need to know is that generally you

can work out which unit contains the import declaration for an API by seeing which C header file declares it. Normally, if the API appears in a unit at all, the unit has the same name as the header file, with a few exceptions. So, if the header file was listed as shellapi.h, the import unit would be `ShellAPI`. If the header file was listed as mmsystem.h, the import unit would be `MMSystem`. The header files winreg.h, winver.h, winnetwk.h, wingdi.h, winuser.h and winbase.h (basically those in the form winXXXX.h) are dealt with by the main `Windows` import unit, automatically added to the `uses` clause of each form unit.

To see the Delphi declaration of `CopyFile`, open the Windows.pas unit source file. In Delphi 4 and later, you can click on the `Windows` unit in a `uses` clause, press `Ctrl+Enter` and the file will open straight away, as the relevant directory is on the browsing path. For Delphi 2 and 3, pressing `Ctrl+Enter` will show an open dialog. You will need to navigate to Delphi's Source\RTL\Win directory to find the unit.

Once the unit is open, do a search to find `CopyFile`. The declaration will look like Listing 3.

The two filename parameters are declared as type `PChar` and the `Boolean` is declared as type `BOOL`,




➤ *Figure 1: Finding information from the Win32 API help file.*

which equates to the Delphi type `LongBool`. Having clarified the situation, we can test out the API.

This month's disk has a test project called CopyAFile.Dpr on it. The simple user interface has one button that launches an open dialog so you can choose a file to copy. It then launches a save dialog so you can choose where you want it copied to. The file is then copied using a small procedure called `FileCopy`. This uses `CopyFile` to attempt to copy the file.

If the file already exists, the first call to `CopyFile` fails (thanks to the third parameter being `True`), and the Windows `GetLastError` API will return `ERROR_FILE_EXISTS`. Given this, the user is asked if it is okay to overwrite the file. If the user says it is, `CopyFile` is called again, but with the last parameter being `False`. The `FileCopy` procedure can be seen in Listing 4, along with the code that calls it after showing the two dialogs.

`CopyFile` has an associated routine called `CopyFileEx`. This extended version supports callbacks, which would allow you to display file copy progress information. However, it is only supported in Windows NT 4 and higher, not in Windows 95 or 98. So instead of looking at that API, we will try the Windows shell generic file operation API, `SHFileOperation`. This is what Windows Explorer uses to copy, delete, rename and move files, and so can give progress feedback automatically (the dialog with the document flying between folders).

To copy a file, you pass a suitably set up `SHFILEOPSTRUCT` record (called a `TSHFileOpStruct` by all 32-bit versions of Delphi) to `SHFileOperation`. The fields of this record are as follows. The `Wnd` field needs the window handle of a form, so any dialogs (such as confirmation or progress) can be located sensibly on the screen and be properly modal. `wFunc` indicates the type of operation, which can be `FO_COPY`, `FO_MOVE`, `FO_DELETE` or `FO_RENAME`.

The `pFrom` field holds the filename(s) that need to be copied. This is a `PChar` field, and so the filenames are null-terminated. If there is more than one file, the filenames are just appended one after another. To indicate there are no more files, the last filename should be followed by another null terminator character. The `pTo` field holds the folder that the file should

➤ *Listing 3: The Delphi declaration for the Win32 CopyFile API.*

```
function CopyFile(
  lpExistingFileName, lpNewFileName: PChar;
  bFailIfExists: BOOL): BOOL; stdcall;
```

➤ *Listing 4: A file copying routine, using the CopyFile API.*

```
procedure FileCopy(const Source, Target: String);
var
  LastError: DWord;
begin
  if not CopyFile(PChar(Source), PChar(Target), True) then begin
    LastError := GetLastError;
    //If file exists, check it's okay to overwrite
    if LastError = ERROR_FILE_EXISTS then begin
      if MessageDlg('Destination file exists. Overwrite?',
        mtConfirmation, [mbYes, mbNo], 0) = mrYes then
        //If it's okay, then overwrite
        Win32Check(CopyFile(PChar(Source), PChar(Target), False))
    end else
      //If there was some other problem, report it
      raise EWin32Error.Create(SysErrorMessage(LastError));
  end
end;
procedure TForm1.btnCopyClick(Sender: TObject);
begin
  if dlgSource.Execute and dlgTarget.Execute then
    FileCopy(dlgSource.FileName, dlgTarget.FileName)
end;
```

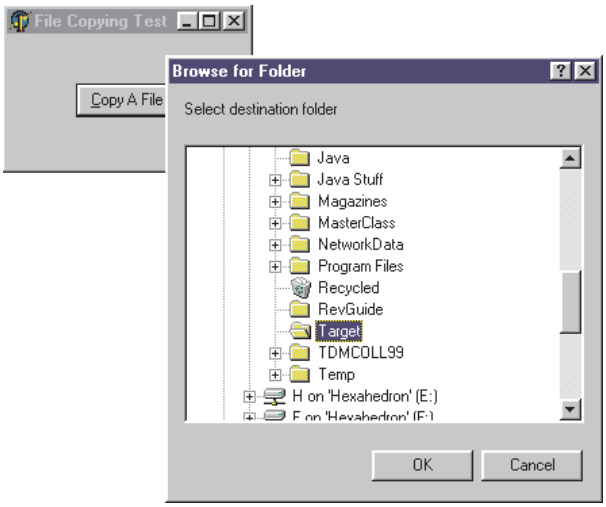
be copied to (as opposed to the target filename).

The `fFlags` field holds whatever special flags you would like to pass. These allow you to control whether confirmations will be made for file overwrites (`FOF_NOCONFIRMATION`), whether a progress dialog will be displayed, assuming the file is large enough (`FOF_SILENT`), whether filenames will be shown on the progress dialog (`FOF_SIMPLEPROGRESS`), etc.

If the user cancels any operation during the file copy, the `fAnyOperationAborted` field gets set to `True`. Finally, the `lpszProgressTitle` field points to a string that is written instead of file names on the progress dialog, if you specified the `FOF_SIMPLEPROGRESS` flag.

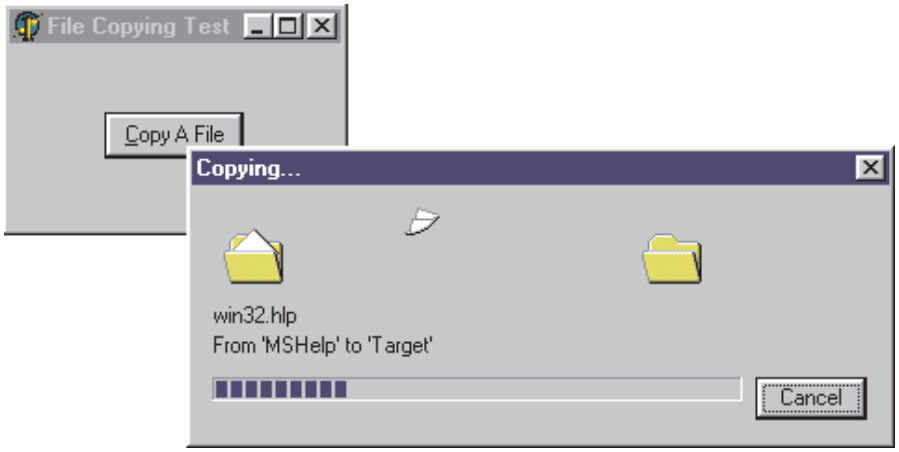
So, given a filename and a target folder, we can copy a file with `SHFileOperation`. An open dialog allows the simple test application to get a file selected for copying. In order to select a target folder, a save dialog does not really work (it wants a file name as well). So we'll use the `SHBrowseForFolder` shell API, as used in last issue's *Delphi Clinic* to browse for a computer on the network. This time it will be used, from within a function called `GetFolder`, to select a folder. Figure 2 shows the folder browsing dialog.

➤ *Figure 2: Browsing for a folder.*




CopyAFile2.Dpr is the test project for this file copying experiment. The useful parts of the code are shown in Listing 5, the progress dialog is in Figure 3.

➤ *Figure 3: A progress dialog showing whilst copying a file.*




## ActiveX Warning

**Q** I have been testing Delphi's Web Deployment. I have created a simple ActiveForm and placed one button on it, displaying *Hello World* when pressed. The problem is that when Internet Explorer loads the ActiveX I get a Security Warning with the message: '*Do you want to install and run Simple.ocx. The publisher cannot be determined due to the problems below. Authenticode signature not found*'. How do I stop this message from appearing?

**A** When someone using a Web browser loads an HTML page containing a reference to an ActiveX, the Web browser will probably try to download the ActiveX so its code can be executed. The problem with this is that the ActiveX could potentially contain malicious code that might do harm to the user's machine contents. Internet Explorer employs security measures to try

➤ *Listing 5: A file copying routine, using the SHFileOperation API.*

```
uses
  ShellAPI, ShlObj;
procedure FileCopy(const Source, Target: String);
var
  FOS: TSHFileOpStruct;
  SourceLst: String;
begin
  FillChar(FOS, SizeOf(FOS), 0);
  FOS.Wnd := Application.MainForm.Handle;
  FOS.wFunc := FO_COPY;
  //File name list must have 2 null terminators. It
  //gets 1 anyway. The other we pass in explicitly
  SourceLst := Source + #0;
  FOS.pFrom := PChar(SourceLst);
  FOS.pTo := PChar(Target);
  //Uncomment these 2 lines if you do not want the file
  //name details shown on the copy progress dialog
  //FOS.fFlags := FOF_SIMPLEPROGRESS;
  //FOS.lpszProgressTitle := 'Please wait...'
  SHFileOperation(FOS);
end;
function GetFolder: String;
var
  BrowseInfo: TBrowseInfo;
  Folder: array[0..MAX_PATH] of Char;
begin
  FillChar(BrowseInfo, SizeOf(BrowseInfo), 0);
  BrowseInfo.hwndOwner := Application.MainForm.Handle;
  BrowseInfo.lpszTitle := 'Select destination folder';
  BrowseInfo.ulFlags := BIF_RETURNONLYFSDIRS or
    BIF_DONTGOBELOWDOMAIN;
  SHGetPathFromIDList(SHBrowseForFolder(BrowseInfo),
    Folder);
  Result := Folder;
end;
procedure TForm1.btnCopyClick(Sender: TObject);
var
  TargetFolder: String;
begin
  if dlgSource.Execute then begin
    TargetFolder := GetFolder;
    if TargetFolder <> '' then
      FileCopy(dlgSource.FileName, TargetFolder)
  end
end;
```

and reduce the risk of such problems.

There are several levels of security that you can enable with Internet Explorer in its Options dialog. The simplest (but least secure) way of avoiding the warning dialog you see is to set the security to a lower setting, so no check is made against binary files. This is fine if you are confident that all binary files that will be downloaded are harmless, and on an intranet system, this may be a valid thing to assume.

However, Microsoft recommend that you code sign your ActiveX controls to verify their integrity. You use Microsoft Authenticode utilities to generate code signatures, which are sent to a Local Registration Agency or Certification Authority (such as VeriSign or GTE) who can then issue a Software Publisher Certificate. Once the certificate has been received, your binary files can be code signed with more Authenticode utilities.

Authenticode itself cannot guarantee that signed code will do no harm. However, an Authenticode signed ActiveX tells the person downloading it that its manufacturer is participating in the infrastructure of trusted entities. It also tells the person who the software publisher is and verifies whether the file has been tampered with since it was released by them.

For more information about Authenticode technology, visit www.microsoft.com/security.

More information on code signing can be found in the VeriSign code signing FAQ at http://digitalid.verisign.com/id_faqs.htm.

## Custom Error Strings From DLL Routines

**Q** According to the MSDN it is possible to extend the Windows error handling capabilities (GetLastError and SetLastError) for third-party DLLs. It should be possible then for any application to use SysErrorMessage (if you stick to Delphi-speak) or the Windows API to get descriptive error strings for the error numbers.

```
function SysErrorMessage(ErrorCode: Integer): string;
var
  Len: Integer;
  Buffer: array[0..255] of Char;
begin
  Len := FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM or
    FORMAT_MESSAGE_ARGUMENT_ARRAY, nil, ErrorCode, 0, Buffer,
    SizeOf(Buffer), nil);
  while (Len > 0) and (Buffer[Len - 1] in [#0..#32, '.']) do
    Dec(Len);
  SetString(Result, Buffer, Len);
end;
```

➤ *Listing 6: Delphi's SysErrorMessage routine.*

But how do you get the string resources linked in with the OS? Is there some registration required? It's mentioned often enough on Microsoft's website that you can do it, but they never explain how.

**A** When Win32 API calls fail, they typically call SetLastError, passing a Windows error number, and then return a value of False. See Issue 51's *Delphi Clinic* for more details on Windows error numbers.

Application programmers can spot the False return value, call GetLastError to find what the error value was (which clears the last error) and act accordingly. Delphi programmers can pass this value to SysErrorMessage to get a string describing the problem.

Alternatively, Delphi programmers can call RaiseLastWin32Error instead of calling GetLastError. This routine calls GetLastError, passes it to SysErrorMessage and raises an EWin32Error exception with the error description as its message. One last option is to pass the API's Boolean return value to Win32Check. This will also raise an appropriate exception (through a call to RaiseLastWin32Error), assuming the API return value is False, otherwise it simply returns True. Win32Check and RaiseLastWin32Error were introduced in Delphi 3.

The question asks how we can get DLL routines of our own to make their error descriptions available. To find out, we need to look at how SysErrorMessage works. The implementation in the SysUtils unit uses the rather cumbersome FormatMessage API, as is shown in Listing 6.

FormatMessage can be used to get a Windows system error message, format a string containing placeholders (rather like the VCL's Format) or, according to the API help file, get a custom string from the message-table resource in a specified module.

So it looks like a DLL routine could call SetLastError with an error number and have the description of the error sitting in a message-table resource. This message-table can be in the same DLL or in a resource-only DLL if preferred. In the simple example we will go through here, the message-table will be in the same DLL. When a programmer calls the DLL routine, if it returns False, they could call FormatMessage with the appropriate flags and module handle to get the custom message.

That's the theory according to the Win32 API help. Let's see if it works. The first thing we need to do is make a new DLL, with at least one routine that returns a Boolean value. If it returns False, it will be assumed to have called SetLastError. The next thing we need to do is to work out how to create a message-table resource.

The downside to this story is that message-tables are not catered for at all by Delphi. Instead, you need the Microsoft Message Compiler, MC.EXE. This tool is not supplied with Delphi, but does come with the Microsoft's Platform SDK. You can find this on one of the many CDs that come in MSDN Professional or MSDN Enterprise.

Even with the Message Compiler, things are far from rosy. Borland's resource compiler, BRC.EXE, does not understand the resource script terms that refer to message-table resources. So again we have to go to the Platform SDK and get Microsoft's resource

compiler, RC.EXE. With MC.EXE and RC.EXE, we can continue with the job.

The simple test DLL will have a routine that accepts one integer parameter. The value of the integer is expected to be between 0 and 1,000. If the number is less than zero, one error will be indicated (negative values not supported). If the value is greater than 1,000, a different error will be reported (number too big). The DLL project is on the disk as TheDll.Dpr and the implementation of its key unit, TheDLLUnit.Pas, is shown in Listing 7.

You can see that two constants are being used, `ERROR_NUMBER_NEGATIVE` and `ERROR_NUMBER_TOO_BIG`. These are defined in the `ErrorMsgs` unit, which we will look at shortly. The DLL project file simply exports the `DoSomething` routine, to make it available to calling applications.

The next step is to implement the message-table. According to the Platform SDK, a message table source file has an .MC file extension. Assuming you are just interested in writing English messages, you need to have repeated sections in the file, one for each message. Each section specifies the message number, the severity and symbolic name. This is followed by an indication of the message's language and the message text. The section is terminated by a single dot.

Additional languages would be defined at the top of the file. The severity indication can either be `Success`, `Informational`, `Warning` or `Error`, each being pre-defined symbols. You can also define additional severity types in the file header. Since we are defining strings to describe errors, it makes sense to mark them with `Error`. The sample message-table script (ErrorMsgs.MC) is in Listing 8.

The file can now be compiled with the Message Compiler using a command-line of:

```
MC ErrorMsgs
```

This produces three new files. The first is a resource script (ErrorMsgs.RC) that must be compiled with RC.EXE to produce a .RES file (ErrorMsgs.Res). The second is a binary file containing the compiled message-table. The default file name for English binary messages is MSG00001.BIN (which again can be changed by adding an entry to the header section of the message-table script file). This file is referred to in the resource script file and will be compiled into the .RES file.

The final file is a C header file (ErrorMsgs.H) which would initially appear to bear no relevance to the Delphi programmer. Initial appearances are misleading here, as you need to look in this file to get the error message ID values. Whilst Listing 8 specifies ID numbers of 1 and 2, these are not the final values. These initial values are combined with the severity value and an optional facility value (ignored in this case) to produce a final message ID. The important section of the header file can be seen in Listing 9 (the rest is made up of informative comments).

This tells you that the two message IDs are `$C0000001` and `$C0000002` respectively. Since MC.EXE does not generate a Pascal unit as well as the C header, this must be done by hand. The result is ErrorMsgs.Pas, the unit we skipped past earlier (see Listing 10). You can see that, as well as defining the two error number constants, the unit also links in the

➤ *Listing 7: A DLL routine setting custom errors.*

```
interface
uses
  Windows;
function DoSomething(Value: Integer): Bool; stdcall;
implementation
uses
  Dialogs, DLLConstUnit;
function DoSomething(Value: Integer): Bool;
begin
  Result := False;
  if Value < 0 then
    SetLastError(ERROR_NUMBER_NEGATIVE)
  else if Value > 1000 then
    SetLastError(ERROR_NUMBER_TOO_BIG)
  else begin
    Result := True;
    ShowMessageFmt('Value passed was %d', [Value])
  end
end;
```

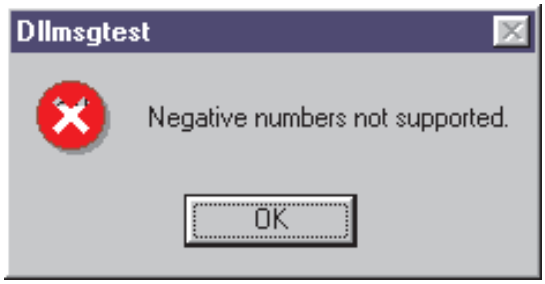
➤ *Listing 8: A message-table script.*

```
MessageId = 1
Severity = Error
SymbolicName = ERROR_NUMBER_NEGATIVE
Language=English
Negative numbers not supported
.
MessageId = 2
Severity = Error
SymbolicName = ERROR_NUMBER_TOO_BIG
Language=English
The number was too big
.
```

➤ *Listing 9: The C header created by MC.EXE.*

```
// MessageId: ERROR_NUMBER_NEGATIVE
//
// MessageText:
//
//  Negative numbers not supported
//
#define ERROR_NUMBER_NEGATIVE            0xC0000001L
//
// MessageId: ERROR_NUMBER_TOO_BIG
//
// MessageText:
//
//  The number was too big
//
#define ERROR_NUMBER_TOO_BIG             0xC0000002L
```


➤ *Figure 4: Normal action of the DLL routine.*




➤ *Figure 5: One of the DLL routine's custom error messages.*


➤ *Figure 6: Another custom error message.*

message-table resource file. With this unit written, the DLL project should compile. The resulting DLL will have a message-table built into it.

Now we can write a program to use the DLL and see how we can access these messages. A test project is supplied on the disk, called DLLMsgTest.Dpr. This has a simple form with an edit control and a button on it. The user can enter a number into the edit and press the button. The button's event handler translates the edit contents to a number and calls the DLL routine, passing the number. If the number is acceptable to the DLL routine, it displays a nice message box and returns `True` (Figure 4). However, if the number is outside acceptable ranges, the DLL will call `SetLastError`, passing one of our two message constants.

The required task is to call the DLL routine and, if `False` is returned, get the last error number and translate it into a descriptive string (one of those stored in the message table). To accomplish this, the project implements new versions of `Win32Check`, `RaiseLastWin32Error` and `SysErrorMessage`. To accommodate custom DLL error numbers and messages, the new routines are called `Win32DLLCheck`, `RaiseLastWin32DLLError` and `DLLErrorMessage`. Each takes an extra `HModule` parameter, to indicate which loaded DLL to look in for the message strings.

`Win32DLLCheck` is a copy of the `Win32Check` routine. The call it makes to `RaiseLastWin32Error` has been replaced with a call to `RaiseLastWin32DLLError` and the module handle is passed along to it. `RaiseLastWin32DLLError` has the call to `SysErrorMessage` replaced with a call to `DLLErrorMessage`, with the module handle passed through. `DLLErrorMessage` uses the `FORMAT_MESSAGE_FROM_HMODULE` flag and passes the module handle in to allow `FormatMessage` to get the required message string from the DLL. For more details of these replacement utility routines, see the code on the disk.

Listing 11 shows the simple call to the DLL routine wrapped in a call to `Win32DLLCheck`. The `GetModuleHandle` API is used to find the module handle of the DLL in question. Figure 5 and Figure 6 show the application raising exceptions with the custom message strings, to prove that the mechanism works as described.

➤ *Listing 10: A Delphi translation of Listing 9.*

```
unit ErrorMsgs;
{$R ErrorMsgs.Res}
interface
const
  ERROR_NUMBER_NEGATIVE = $C0000001;
  ERROR_NUMBER_TOO_BIG  = $C0000002;
implementation
end.
```

➤ *Listing 11: Calling the DLL routine and checking for custom errors.*

```
function DoSomething(Value: Integer): Bool; stdcall;
  external 'TheDLL.DLL';
procedure TForm1.btnCallDLLClick(Sender: TObject);
begin
  Win32DLLCheck(GetModuleHandle('TheDLL.Dll'),
    DoSomething(StrToInt(edtNumber.Text)))
end;
```

## Update To File I/O Error

In Issue 51, I discussed a couple of ways to get information on Windows error codes. I mentioned looking them up in the Win32 SDK Reference help file shipped with Delphi, or by browsing through the Windows unit. Thanks are due to Frank Hagenson who reminded me that Windows has a command-line tool that also gives help on Windows error codes.

The error code in Issue 51's Clinic question was 32. To get a description of this code you can run this Windows NT command-line:

```
net helpmsg 32
```

or this Windows 95 or 98 command-line:

```
net help 32
```

Running the command on Windows 95 produces the description: *'Error 32: The specified file is in use by another process. Try again later'.*