The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

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

File I/O Error

Q I am writing an application that opens a text file and displays it in a memo component. When I run this project from Delphi 3, the file opening statement causes an error code 32. I don't want to write to the file, I just want to open and read it. What's happening here, and how might I be able to get around this problem?

A Any 32-bit Delphi routines that open files ultimately result in calls to Windows API routines. If you get an error, this will probably be a Win32 error code (see later for how to decide whether it is or not). More than likely, the error will be reported as an `EInOutError` exception, whose `ErrorCode` field contains the error number in question. A handful of error codes generate meaningful strings, but most use the generic message as shown in Figure 1.

To examine individual I/O related errors, your file manipulation code can be enclosed within a `try..except..end` statement. You can then use a `case` statement for the `ErrorCode` field. Alternatively, if you use compiler directives to turn I/O checking off, you can use a `case` statement around a call to the `IOResult` function. Of course it is usual to desire constant symbols rather than literal values such as 32, so where do we find them?

The answer is in the `Windows` unit, or at least that is where we find most of them. If you search through the unit you will find many error constants that all start with an `ERROR_` prefix. Listing 1 shows a small section of this error list, including the constant `ERROR_SHARING_VIOLATION`, which has a value of $20, or 32. This suggests that some other application (or indeed other code in the same application) has the file open in a manner that excludes you opening the file for reading.

You can also find these codes listed in the Win32 help file that comes with Delphi. In the Win32 Programmer's Reference help file that ships with Delphi 5 (and I would imagine also in earlier versions) look up the string *error codes* and, if you get a list of topics to choose from, choose *Error Codes (Win32 Programmer's Reference)*.

The comments in Listing 1, which are also used as descriptions in the help file's alphabetical list of error codes, represent the textual descriptions of the problems. A call to `SysErrorMessage` from the `SysUtils` unit will translate a Win32 error code into its corresponding textual description.

As well as Win32 error codes, Delphi reserves certain ranges of error codes for itself. Error codes 100 to 149 are Delphi I/O error codes. Delphi 3 (and later) lists them if you look up the string *I/O Errors* in the help. Error codes 200 to 255 are Delphi fatal error codes as found in the help under *Fatal Errors* (for example, runtime error 216 is an Access Violation). The `EInOutError` exception has specific descriptive messages that are used for certain error codes, as shown in Table 1.

➤ *Figure 1: Default EInOutError exception dialog.*

Now that we have more information about these I/O errors we can get back to the business of finding more information about them when they arise. Going back to the idea of wrapping I/O code in an exception handling statement, Listing 2 shows a piece of code that opens a text file (or tries to) and then closes it. Listing 3 shows how extra code can be added to make the `EInOutError` exception's message more informative (the code comes from the blindingly simple IOError.Dpr project). Figure 2 shows the new exception dialog that comes up (assuming the modified `EInOutError` exception is not handled), giving more information than that shown in Figure 1.

Having identified the error code, whether it is a Windows code or a Delphi code, and what it means, you are now better placed to write code that handles a particular error, if you so wish.

If the problem is being caused by another process having the file open in a manner that excludes you from opening it read-only, then you have little choice but to

➤ *Listing 1*

```
{ The system cannot read from the specified device. }
ERROR_READ_FAULT = 30;
{ A device attached to the system is not functioning. }
ERROR_GEN_FAILURE = 31;
{ The process cannot access the file because it's being used by another process. }
ERROR_SHARING_VIOLATION = $20;
{ The process cannot access the file because
  another process has locked a portion of the file. }
ERROR_LOCK_VIOLATION = 33;
```

➤ *Listing 2*

```
procedure TForm1.Button1Click(
  Sender: TObject);
var TF: TextFile;
begin
  AssignFile(TF, 'c:\SomeFile.Txt');
  Reset(TF);
  try
    //Blah blah
  finally
    CloseFile(TF)
  end //try..finally
end;
```

I/O Error Code	Win32 Constant	Win32 Description	EInOutError Message
2	ERROR_FILE_NOT_FOUND	The system cannot find the file specified	File not found
3	ERROR_PATH_NOT_FOUND	The system cannot find the path specified	Invalid filename
4	ERROR_TOO_MANY_OPEN_FILES	The system cannot open the file	Too many open files
5	ERROR_ACCESS_DENIED	Access is denied	File access denied
100	N/A	N/A	Read beyond end of file
101	N/A	N/A	Disk full
106	N/A	N/A	Invalid numeric input

➤ *Table 1*

wait until the file is closed by that process. However, if your own program has the file open elsewhere, then you can probably do something about the problem, by closing the file temporarily or opening it with appropriate sharing flags.

Sharing flags can be assigned to the `FileMode` global variable for non-text file variables. They can also be passed to the `FileOpen` function if you are using file handles, and to a file stream object's constructor. For example, assuming typed or untyped file variables are being used, this sets the `FileMode` variable to specify read-only access, and not to exclude anyone else having read-only access:

```
FileMode := fmOpenRead or
  fmShareDenyWrite;
```

Similarly, for a file stream object, this opens it in a similar mode:

```
var
  FS: TFileStream;
...
FS := TFileStream.Create(
  'TheFile', fmOpenRead or
  fmShareDenyWrite)
```

Type Library Corner Cutting

Q I need to write a driver for an application, and to do this I need to implement a pre-defined COM interface in a COM object. As well as being documented on paper, the interface is also defined in a type library. Since the Delphi type library editor has a tendency to type in interface methods in the implementing class while you define a *new* COM interface, I wondered if it can do the same with an *existing* interface. This would save me entering a lot of methods in the type library editor by hand.

A The type library editor, while a little rough in places (such as where you enter parameters), is a very flexible piece of software. It can indeed be coaxed into typing in a whole number of methods from a pre-defined interface from another type library. In fact there are two ways of doing what you want, one of which was already discussed in a recent issue of *The Delphi Magazine*. In Issue 49, Steve Teixeira briefly ran though some steps in his *COM Corner* column demonstrating how to successfully turn an `IDispatch`-based interface defined in your type library into an interface based upon some arbitrary interface defined in another type library.

The other approach is slightly different. It involves telling the type library editor that your COM object implements the target interface in question, thereby avoiding the need to have another interface defined.

Before looking at this in detail, let's set the scene. As mentioned, the questioner has a type library that defines an interface that she/he wishes to implement in a COM object. To try and match this arrangement, for demonstration purposes we will focus our attention on the Microsoft Office 97 type library and the `CommandBar` interface. Of course, you are unlikely to

➤ *Listing 3*

```
procedure BetterIOError(E: EInOutError);
var EIO: EInOutError;
begin
  if Assigned(E) then begin
    EIO := EInOutError.Create(E.Message);
    EIO.ErrorCode := E.ErrorCode;
    case E.ErrorCode of
      1, 6..99: EIO.Message := Format('Win32 API error %d: '#13#13'%s',
        [E.ErrorCode,SysErrorMessage(E.ErrorCode)]);
      2..5, 100, 101, 106: { As descriptive as can be right now };
      102: EIO.Message := 'File variable not assigned a name with AssignFile';
      103: EIO.Message := 'File not open';
      104: EIO.Message := 'File not open for input';
      105: EIO.Message := 'File not open for output';
      107..149: EIO.Message := Format('Delphi I/O error %d', [E.ErrorCode]);
    end; //case
    raise EIO at ErrorAddr
  end
end;
procedure TForm1.Button1Click(Sender: TObject);
var TF: TextFile;
begin
  AssignFile(TF, 'c:\SomeFile.Txt');
  try
    Reset(TF);
    try
      //Blah blah
    finally
      CloseFile(TF)
    end //try..finally
  except
    on E: EInOutError do
      BetterIOError(E)
  end //try..except
end;
```

➤ *Figure 2: A more descriptive EInOutError exception dialog.*

ever want to implement this interface, but it comes from a readily available type library and will serve as an example that you can duplicate if needed.

If Office 97 has been installed on your machine you will already have its type library installed on your machine and registered in the Windows registry. In the questioner's case, the type library may well not be registered, but it needs to be. To register a type library called XXXXXXXX.YYY (where YYY will typically either be TLB or OLB, unless it has been compiled into an EXE or DLL) use the TREGSVR tool that comes with Delphi:

```
TRegSvr -t xxxxxxxx.yyy
```

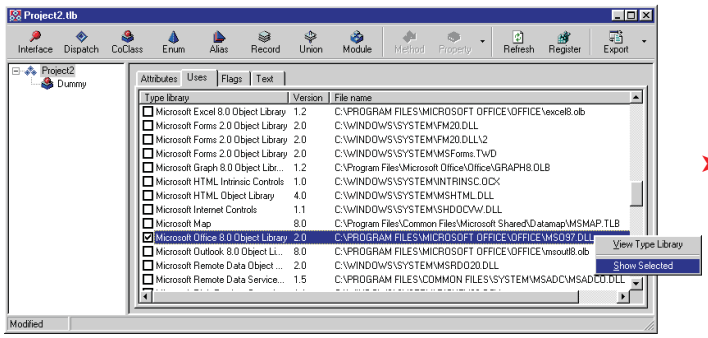
Now that the type library is definitely registered, we can proceed. To start with, you need a new application or a new ActiveX Library from the `File | New...` dialog. You can now import the type library into your project (`Project | Import Type Library...`) to get a Pascal unit containing Pascal versions of all the interfaces (and other bits and pieces) defined in the type library. Office 97's type library will make an import unit called Office_TLB. Delphi 5 already has the Office type libraries imported, with compiled versions placed in the Imports directory and the source files in OCX\Servers, however the unit name has been changed to `Office97`.

You should take a moment to look at the definition of your target interface and identify if it is based upon `IDispatch` (or some other interface that is `IDispatch`-based, which poses other issues as we shall shortly see) or not. If the interface is directly or indirectly `IDispatch`-based, you will need a new Automation object from the `ActiveX` page of the `File | New...` dialog. Alternatively, if `IDispatch` does not come into it, you should choose a new COM object (not available in Delphi 3), but make sure you specify the need for a type library.

The COM/Automation object wizard dialog will ask you for a class (or coclass) name. Enter whatever you want here. I will work on the basis that you entered `Dummy`. This causes Delphi to manufacture a type library in your project, define a coclass called `Dummy` and an interface called `IDummy`, implemented in a class called `TDummy`. Since we will not be needing the `IDummy` interface, delete it from the type library editor. Also, remove `IDummy` from the `TDummy` class definition, since it will no longer be implementing that interface. This leaves the `TDummy` class looking quite unspectacular, as per Listing 4.

When examining the interface to see if it is based upon `IDispatch`, you should be careful if it is indirectly `IDispatch`-based. If your target interface is based on another interface which is based upon `IDispatch` you will need to implement that interface as well. In the case of the `CommandBar` interface there are several other interfaces that need to be implemented. `CommandBar` is based upon `_IMsoOleAccDispObj`, which is based upon `IAccessible` which is then based upon `IDispatch`. So we will need to implement `_IMsoOleAccDispObj` and `IAccessible` in addition to `CommandBar`.

The next step is to make your application's type library reference the type library that defines the interface (or interfaces). To do this, in the type library editor's object list pane (the tree structure on the left),



➤ *Figure 3: Your type library can use other type libraries.*

```
TDummy = class(TAutoObject)
protected
  { Protected declarations }
end;
```

➤ *Listing 4*

select the type library node (the top of the hierarchy, which defaults to the name of the project). The `uses` page on the right lists all the other referenced type libraries. Right-click on it and choose `Show All Type Libraries`. This adds in all the other registered type libraries, allowing you to check the Microsoft Office 8 type library (or whichever one defines the target interface). Figure 3 shows what this looks like.

Now select the `Dummy` coclass in the object list pane and go to the `Implements` page on the right. This allows you to dictate which interfaces will be implemented by `TDummy`. Right-click and choose `Insert Interface` and you will get a list of interfaces defined in this type library and all referenced type libraries. Choose the target interface(s) and click `OK` to add it/them to the list of implemented interfaces. To ensure the COM object still works correctly make sure that the `Default` column says `True` only for the top-level interface (by right-clicking and choosing `Default` if necessary). Figure 4 shows the outcome in this example's case.

Now the `Refresh` button will do the hard work for you. Once pressed, the `TDummy` class will have declarations and stub implementations of all the methods in the interface(s). The only thing left to do now is to add the other type library's import unit into your COM class unit's `uses` clause. In this example's case, this means adding Office_TLB or Office97 to the `uses` clause. The code in Listing 4 should now look like Listing 5.

Customised Docking

Q I am writing a Delphi 4 program that incorporates a form which floats above the main form. If the user wants to dock this form, they can press a button and

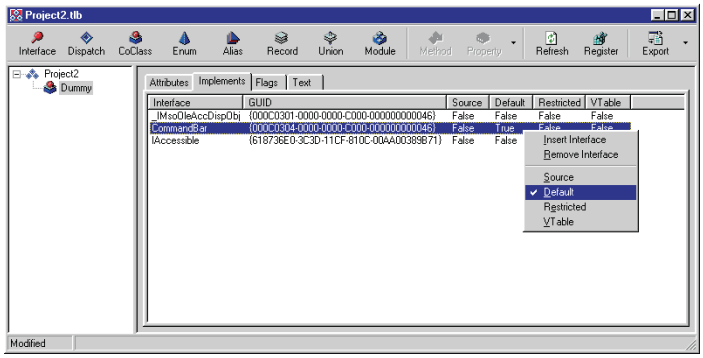
the form docks itself into a panel on the form below, thanks to a call to the floating form's `ManualDock` method. However, I cannot find a way to turn off the double line and miniature `X` button that appear when the form is docked (the same as those seen above the Code Explorer when it is docked in the code editor). I don't need them, as the form is always docked and undocked via a separate button. Is there any way of removing them?

I have tried turning off the docking manager for the panel (by setting its `UseDockManager` property to `False`) but when I call `ManualDock` the form simply vanishes and doesn't appear docked at all.

A This question gives us a chance to look a little into the docking support added into Delphi 4, of which I have seen little in print. Before getting into those two lines (called grabber lines) and the cross (the close button), it's worth seeing why the `UseDockManager` property didn't work as expected.

When a docking operation takes place, there needs to be some control over where and how the docked item will be drawn in the dock site. Delphi uses a dock manager object to give default control over all the docking operations. Setting `UseDockManager` to `False` means that you have to supply your own dock manager, or take full control of docking yourself.

The dock manager of any windowed control can be any object that implements the `IDockManager` interface, defined in the `Controls` unit. Delphi's default dock manager is the `TDockTree` class. Each dock site needs its own dock manager, and has a `DockManager` property (of type `IDockManager`) that refers to it. Assuming the dock site control has its `DockSite` and `UseDockManager` properties set to `True`, a default dock manager is created as needed and assigned to the `DockManager` property.



➤ *Figure 4: A Delphi COM object can implement predefined interfaces.*

As mentioned, the dock manager (`TDockTree` by default) takes responsibility for both placing a docked control in a dock site and also for how it gets drawn. `TDockTree` has a pair of methods that cause the two grabber lines and the small close button to be drawn, allowing the user to undock the control. `AdjustDockRect` is used to shrink the docked control's rectangle to make room for the grabbers and close button, whilst `PaintDockFrame` draws them in. If the docking/undocking is all program-controlled, then we need to find some way of disabling these methods.

There appear to be two ways of achieving the questioner's goal. One way is to create a custom dock manager just for the panel that the second form gets docked into. This would leave all other dock sites operating as they normally do. The alternative is to replace the default dock manager used by all `TWinControl`-based components with one that stops the grabbers and button being drawn.

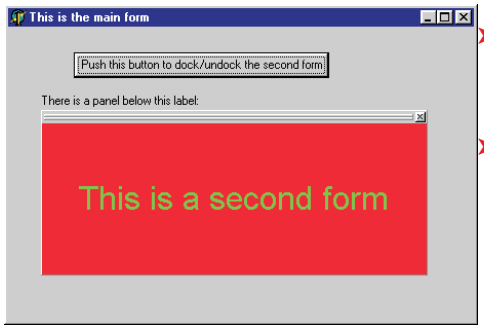
Fortunately, since the two methods discussed above are both pleasingly defined as virtual (which may surprise some of the more cynical Delphi developers among you), the first way is made rather easier. You define a new dock manager class inheriting from `TDockTree` with overridden versions of these two methods that do nothing, causing docked controls not to be shrunk, and not to get the grabber/button adornments. An instance of this class is then given to the panel's `DockManager` property and we can see the results in two screenshots. Figure 5 shows a small red form docked into a panel in the default way with no changes. Figure 6 shows the same form docked, but using the new dock manager.

The second solution, where the default dock manager is completely replaced with a new dock manager, works in much the same way as with customised tooltip windows (see *Hints With Attitude* in Issue 16). The `Controls` unit defines a class reference type called `TDockTreeClass`, defined to represent the `TDockTree` class, or any class inherited from it. It also declares a class reference variable called `DefaultDockTreeClass` of type `TDockTreeClass` which is initialised to `TDockTree`. To completely change the default dock manager, you can assign a class inherited from `TDockTree` to this class reference variable in some unit's initialisation section.

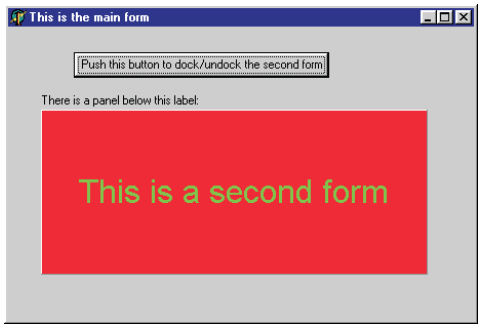
When a component creates a default dock manager, it uses the

➤ *Listing 5*

```
TDummy = class(TAutoObject, _IMsoOleAccDispObj, CommandBar, IAccessible)
protected
  function accHitTest(xLeft, yTop: Integer): OleVariant; safecall;
  function accNavigate(navDir: Integer; varStart: OleVariant): OleVariant;
    safecall;
  function FindControl(Type_, Id, Tag, Visible,
    Recursive: OleVariant): CommandBarControl; safecall;
  function Get_accChild(varChild: OleVariant): IDispatch; safecall;
  ...
  // Many methods removed to keep this listing short
  ...
  procedure Set_NameLocal(const pbstrNameLocal: WideString); safecall;
  procedure Set_Position(ppos: MsoBarPosition); safecall;
  procedure Set_Protection(pprot: MsoBarProtection); safecall;
  procedure Set_RowIndex(piRow: SYSINT); safecall;
  procedure Set_Top(pypTop: SYSINT); safecall;
  procedure Set_Visible(pvarfVisible: WordBool); safecall;
  procedure Set_Width(pdx: SYSINT); safecall;
  procedure ShowPopup(x, y: OleVariant); safecall;
  { Protected declarations }
end;
```



Left, top: Figure 5
A docked form with visible grabbers and close button.



Left, bottom: Figure 6
A docked form with no adornments.

class reference variable to do so. If your replacement class has been assigned, this means that an instance of your dock manager class will be created.

In much the same way, to get a more functional tooltip window, you assign a class inherited from `THintWindow` to the `Forms` unit `HintWindowClass` class reference variable which is initialised to `THintWindow`.

Listing 6 shows some code that covers both solutions (from the DockEg project on the cover disk) thanks to some conditional compilation directives. Depending on whether the conditional symbol `CompletelyReplaceDefaultDockManager` is defined, either the panel's dock manager, or the global default dock manager, is replaced with a customised version. To define the conditional symbol, either use `Project | Options...` (or `Ctrl+Shift+F11` in Delphi 4 and later), go to the `Directories/ Conditionals` page and type it into the `Conditional defines:` area, or alternatively enter this towards the top of the unit:

```
{$define
 CompletelyReplaceDefaultDockManager}
```

CreateProcess Alert

Thanks to a keen-eyed reader who contacted me recently, I have learnt that all calls to the Win32 API `CreateProcess` that I have ever made have been somewhat lacking. This includes calls made to it in a number of *Delphi Clinic* entries over the years. The problem is a resource leak (at least whilst the program that calls `CreateProcess` is running). Since this is the first time anyone has raised this issue with me, I feel that others may also be falling foul of the same *faux pas* and so will spend a little time going over it.

`CreateProcess` launches a program as specified by its parameters and fills up a `TProcessInformation` record that you pass in. This record ends up containing the launched program's process ID and primary thread ID along with a handle to the thread and a handle to the process. It is these handles that are at issue here.

When a process is launched, the Windows kernel allocates a process object for the program and a thread object for the primary thread. These objects are reference counted, like COM objects, and remain in existence until the reference count goes back down to zero. The launching of the process increments the process object reference count to 1. The creation of the process's primary thread similarly causes the thread object's reference count to be incremented to 1. Part of the job given to `CreateProcess` is to open a handle to the process and open another handle to the primary thread. This causes the count of each kernel object to go up to 2.

When the launched process (the launch*ee*) terminates, both reference counts are decremented to 1. So the program has gone away, but the program that called `CreateProcess` (the launch*er*) still has open handles to it. These kernel objects remain in existence until the handles are closed. If the launcher is terminated, the handles *will* be closed (one of the nice things about 32-bit Windows is that when a process closes down, Windows closes all its open handles and frees up any memory associated with it).

The point being made here is that if the launcher has no need to

Listing 6

```
type
  TMyDockTree = class(TDockTree)
  protected
    procedure AdjustDockRect(Control: TControl; var ARect: TRect); override;
    procedure PaintDockFrame(Canvas: TCanvas; Control: TControl;
      const ARect: TRect); override;
  end;
procedure TMyDockTree.AdjustDockRect(Control: TControl; var ARect: TRect);
begin
  //Do nothing to change the control's boundaries as
  //there will be no dock grabber lines or close button
end;
procedure TMyDockTree.PaintDockFrame(Canvas: TCanvas; Control: TControl;
  const ARect: TRect);
begin
  //Do nothing as we do not want dock grabber lines or close button
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
{$ifndef CompletelyReplaceDefaultDockManager}
  { Use this code to change just panel's dock manager. We do not need to free this
    object since property is actually an interface reference and will auto-free }
  Panel1.DockManager := TMyDockTree.Create(Panel1)
{$endif}
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Form2.Floating then
    Form2.ManualDock(Panel1)
  else
    Form2.ManualDock(nil)
end;
initialization
{$ifdef CompletelyReplaceDefaultDockManager}
  //Only use this code to change all docking to draw no adornments
  DefaultDockTreeClass := TMyDockTree
{$endif}
end.
```

refer to the handles, it should immediately close them, to avoid keeping the kernel objects alive longer than necessary. In fact, the Win32 API help file says: '*The created process remains in the system until all threads within the process have terminated and all handles to the process and any of its threads have been closed through calls to* `CloseHandle`. *The handles for both the process and the main thread must be closed through calls to* `CloseHandle`. *If these handles are not needed, it is best to close them immediately after the process is created.*'

In his book *Advanced Windows*, Jeffrey Richter also makes the point quite emphatically. In a box marked as *Important* and with an exclamation mark next to it, he says: *'Don't forget to close these handles. Failure to close handles is one of the most common mistakes developers make and results in a system memory leak until the process that called* `CreateProcess` *terminates.'*

So if you simply wish to launch some application, code like Listing 7 is the resource-friendly way of doing it. Of course sometimes you do need to refer to the process handle or thread handle after launching the process. For example, you may need to pass it to `WaitForSingleObject` to either wait for the process to terminate, or find out if it has terminated. If this is the case, then remember to close the handles as soon as you are done with them.

The other Win32 API that launches programs is `ShellExecuteEx`. The `TShellExecuteInfo` record that acts as its only argument does not give you a thread handle, however it does return a process handle. Fortunately, by default, this handle is always 0 (representing a closed handle). However, if the `fMask` field includes the value `see_Mask_NoCloseProcess` then the handle will be a valid, open handle to the process. This means that if you use that flag, you have the responsibility of closing the process handle.

A slightly more capable version of Listing 7 can be seen in Listing 8. By more capable, I mean that in addition to programs, this new version can also take a file name and launch the associated program. Both these routines are in the `RunCmd` unit on this month's disk.

Acknowledgements

Thanks are due to Geoff Lawrence for improving my Windows API knowledge this month.

➤ *Listing 7: A resource-friendly command executor.*

```
procedure RunCommand(const Cmd, Params: String);
var
  SI: TStartupInfo;
  PI: TProcessInformation;
  CmdLine: String;
begin
  //Fill record with zero byte values
  FillChar(SI, SizeOf(SI), 0);
  SI.cb := SizeOf(SI);      //Set mandatory record field
  //Ensure Windows mouse cursor reflects launch progress
  SI.dwFlags := StartF_ForceOnFeedback;
  //Set up command line
  CmdLine := Cmd;
  if Length(Params) > 0 then
    CmdLine := CmdLine + #32 + Params;
  //Try and launch child process. Raise exception on failure
  Win32Check(CreateProcess(nil, PChar(CmdLine),
    nil, nil, False, 0, nil, nil, SI, PI));
  //Wait until process has started its main message loop
  WaitForInputIdle(PI.hProcess, Infinite);
  //Close process and thread handles
  CloseHandle(PI.hThread);
  CloseHandle(PI.hProcess);
end;
```

➤ *Listing 8: An alternative command-line executor.*

```
uses ShellAPI;
...
{ Extended version of RunCommand which can
  handle file associations }
procedure RunCommandEx(const Cmd, Params: String);
var SEI: TShellExecuteInfo;
begin
  //Fill record with zero byte values
  FillChar(SEI, SizeOf(SEI), 0);
  SEI.cbSize := SizeOf(SEI);  //Set mandatory record field
  //Ask for an open process handle
  SEI.fMask := see_Mask_NoCloseProcess;
  //Tell API which window error dialogs should be modal to
  SEI.Wnd := Application.Handle;
  //Set up command line
  SEI.lpFile := PChar(Cmd);
  if Length(Params) > 0 then
    SEI.lpParameters := PChar(Params);
  SEI.nShow := sw_ShowNormal;
  //Try and launch child process. Exit on failure
  if not ShellExecuteEx(@SEI) then Exit;
  //Wait until process has started its main message loop
  WaitForInputIdle(SEI.hProcess, Infinite);
  //Close process handle
  CloseHandle(SEI.hProcess);
end;
```