The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project? Just email Brian Long, our Delphi Clinic Editor, on 76004.3437@compuserve.com or write/fax us at The Delphi Magazine

FileMode Failure

Q Based on your series of file handling articles, I am trying to import data from an old DOS package we use into Paradox files. The problem is locking. I specify

```
FileMode := fmOpenReadWrite +
  fmShareDenyNone
```

so it can import a snapshot even if other users are using the package. However, Delphi 2's Reset throws an exception with I/O error 32 (*ERROR_SHARING_VIOLATION*). Am I doing something wrong?

A No. Delphi 2 has a bug where all the sharing bits of FileMode are stripped off and ignored. 2.01 also has the same problem, however there is a patch available, which is included on this month's disk as SYSTEM.ZIP in the CLINIC directory. It's also on the CompuServe BDELPHI32 forum. Unzip this into your Delphi 2 LIB directory and re-compile your application. This should fix things.

If you are a source junkie, this replacement version of the System unit changes the _ResetFile routine in Delphi 2's SOURCE\RTL\SYS\OPENFILE.ASM file as shown, with my comments, in Listing 1 (you could verify this with Turbo Debugger, or as I did with Delphi's undocumented CPU view described in Issue 13's *Delphi Clinic*). This unit also changes quite a lot of GETMEM.INC to fix some internal memory leaks.

System Modal Forms

Q How can I set a form up so that the user is forced to complete the bits of it that I want them to before doing anything else. This includes preventing them switching to other applications, so ShowModal is not sufficient. I want something like TForm.ShowSystemModal, but of course that doesn't exist.

A TForm.ShowModal causes a form to be application modal. You want the form to be system modal. There is a Windows API, that can be used in 16-bit Delphi programs, which turns a normal form into a system modal form. If you call SetSysModalWindow(Handle) in a form's OnCreate event handler, it will be a system modal form. Being system modal means the form can't be moved, minimised or maximised so it would be advisable to set the form's BorderIcons property accordingly or set the BorderStyle property to bsDialog.

If you wish to be able to toggle a form between being system modal and modeless then you need to record the return value from SetSysModalWindow, as this will need to be used when you go back to being modeless. A sample project MODAL.DPR on the disk shows this use of the API and also caters for changing the form's border style. A checkbox on the form toggles between system modality and non-modality. The important code is in Listing 2.

System Error Message

Q I have discovered a routine SysErrorMessage in the SysUtils run-time library unit. This is very convenient for getting a textual description of an error as reported by the GetLastError API, but has a limitation. If you use CreateProcess to run a file *[See the next Clinic entry for details on how to do this. Editor]* that is not an executable file (say \DELPHI\README.TXT), the API returns

➤ *Listing 1*

```
Before:
//if FileMode  2 then
//  FileMode := 2
  mov cl, FileMode
  cmp cl, 2
  jbe @@skip
  mov cl, 2
@skip:
After:
//if FileMode and 3 = 3 then
//  FileMode := 2
//in other words, check this is
//not the illegal file mode
//combination
//  fmOpenWrite + fmOpenReadWrite
  mov cl, FileMode
  and cl, 3
  cmp cl, 2
  jbe @@skip
  mov cl, 2
@skip:
```

➤ *Listing 2*

```
procedure TForm1.chkModalClick(Sender: TObject);
const
  OldModalWnd: THandle = 0;
begin
  if chkModal.Checked then begin
    if chkUpdateUI.Checked then
      BorderStyle := bsDialog;
    OldModalWnd := SetSysModalWindow(Handle);
  end else begin
    if chkUpdateUI.Checked then
      BorderStyle := bsSizeable;
    SetSysModalWindow(OldModalWnd);
  end
end;
```

False. GetLastError returns an error number of 193 (*ERROR_BAD_EXE_FORMAT*) and SysErrorMessage gives a string of *%1 is not a valid Win32 application*. Clearly the file name should be used as a place holder but SysErrorMessage doesn't know that. Can you fix it?

A Win32 is much more pleasant in the way it gives error information. Many APIs return a Boolean, where False means a problem has occurred and GetLastError tells us the error number. The SysErrorMessage uses the FormatMessage API to produce a textual description of the error. Having looked at the source for SysErrorMessage I have modified it to take an extra parameter that can be a file name, which will be substituted for the %1 if it appears. Listing 3 shows the new routine (located in the RUNWAITU.PAS unit on this month's disk) along with a sample call to it. This routine is used in the next Clinic entry for the purposes of giving a descriptive text string to an exception object.

Waiting For Termination

Q How do I tell if a program I launch from my Delphi 2 app has terminated? The old 16-bit approach of calling GetModuleUsage doesn't work in Win32.

A Fortunately the Windows API has been improved here. It is easy to wait until a process has completed – provided you have its process handle. If you still use WinExec or ShellExecute to launch programs (as a hang-over from Delphi 1) then you're not going to get very far. You need to use the Win32 APIs CreateProcess or ShellExecuteEx instead. Listing 4 shows two routines that can be used to launch an application, one for each API. Both functions return the process handle, or raise an exception on failure. Two versions of another routine are shown in Listing 5 that wait until the specified process has finished executing using the WaitForSingleObject API.

This API is nice and simple but the calling program hangs whilst it

```
function SysErrorMessageParam(ErrorCode: Integer; Param: String): String;
var
  Len: Integer;
  Buffer: array[0..255] of Char;
  ArgArray: array[1..1] of PChar;
begin
  ArgArray[1] := PChar(Param);
  Len := FormatMessage(Format_Message_From_System or
    Format_Message_Argument_Array, nil, ErrorCode, 0, Buffer,
    SizeOf(Buffer), @ArgArray);
  while (Len > 0) and (Buffer[Len - 1] in [#0..#32, '.']) do Dec(Len);
  SetString(Result, Buffer, Len);
end;
...
ShowMessage(SysErrorMessageParam(GetLastError, FileName);
```

➤ *Listing 3*

```
type
  EExecAppError = class(Exception);
...
function ExecApp(AppName, Params: String): THandle;
var SI: TStartupInfo;
    PI: TProcessInformation;
begin
  FillChar(SI, SizeOf(SI), 0);
  with SI do begin
    cb := SizeOf(TStartupInfo);
    dwFlags := StartF_UseShowWindow;
    wShowWindow := sw_ShowNormal;
  end;
  if not CreateProcess(nil, PChar(AppName + ' ' + Params),
    nil, nil, False, 0, nil, nil, SI, PI) then
    raise EExecAppError.Create(SysErrorMessageParam(GetLastError, AppName));
  Result := PI.HProcess;
end;

function ExecApp2(AppName, Params: String): THandle;
var SEI: TShellExecuteInfo;
begin
  FillChar(SEI, SizeOf(SEI), 0);
  with SEI do begin
    cbSize := SizeOf(SEI);
    fMask := see_Mask_NoCloseProcess;
    Wnd := Application.Handle;
    lpFile := PChar(AppName);
    lpParameters := PChar(Params);
    nShow := sw_ShowNormal;
    if not ShellExecuteEx(@SEI) then
      {this API requires the ShellAPI unit }
      raise EExecAppError.Create(
        SysErrorMessageParam(GetLastError, AppName));
    Result := SEI.HProcess;
  end;
end;
```

➤ *Listing 4*

```
type
  TWaitThread = class(TThread)
  private
    FProcess: THandle;
  public
    constructor Create(HProcess: THandle);
    procedure Execute; override;
  end;
...
constructor TWaitThread.Create(HProcess: THandle);
begin
  FProcess := HProcess;
  inherited Create(False)
end;

procedure TWaitThread.Execute;
begin
  WaitForSingleObject(FProcess, Infinite)
end;

procedure WaitForApp(HProcess: THandle; Event: TNotifyEvent);
begin
  with TWaitThread.Create(HProcess) do begin
    FreeOnTerminate := True;
    OnTerminate := Event
  end;
end;

procedure WaitForApp2(HProcess: THandle);
begin
  while WaitForSingleObject(HProcess, 100) = Wait_TimeOut do begin
    Application.ProcessMessages;
    if Application.Terminated then
      Break
  end
end;
```

➤ *Listing 5*

is waiting. Therefore, you either have to repetitively wait for short periods, calling the usual `ProcessMessages` in between, or use a thread to do the waiting. Notice that the thread-based routine takes a method as a parameter, suitable for use as an `OnTerminate` event handler for the thread. These routines are supplied in a unit on the disk called RUNWAITU.PAS, and a sample project RUNAPP.DPR makes use of them all. The important code from the form unit of the project is shown in Listing 6.

Single Instance Only

Q How can I ensure that my program is restricted to a single instance? In other words, if a user tries to invoke my app a second time, it should not start a second copy. Delphi 1 allowed me to simply check `HPrevInst` against zero. Delphi 2 has removed this variable.

A There's usually two sides to this question. Firstly, how to restrict the application to being single instance and secondly how to switch focus back to the first instance when a new instance is invoked. In 16-bit programming, `HPrevInst`, the instance handle of the previous instance, makes the first step easy. In Win32, all applications have their own address space and get loaded (typically) at the same address. Since an instance handle is really just the address of the application's data segment all instance handles can be the same value – in other words not very useful. All this means that we have to find our own mechanism to achieve the goal.

Some people elect to use atoms, which are system-supplied numbers related to a supplied string, or atom name. They will create a global atom (`GlobalAddAtom`) upon application start-up (using `Application.ExeName` as the atom name) and delete it on exit (`GlobalDeleteAtom`). Before creating the atom, they will check if it exists with `GlobalFindAtom`, whereupon they know if a previous instance is running. It's worth knowing that the global atom table can only hold 37 atoms. Every Delphi 2 application makes use of one atom and all Delphi 1 apps use two each.

An alternative approach, as used here, is with a semaphore. These Win32 devices are often used to limit the number of threads using a resource. We will cheat a little by not using the semaphore API *per se*, but just find out if a semaphore already exists.

That deals with detecting the previous instance, so now we need to switch focus to it. The approach used in 16-bit was covered in Issue 5 in my article *Please call later... Callbacks in Windows and the Borland Database Engine (Part 2)*. In order to switch back, you had to locate a window in the original instance, bring it to the front of all other windows and restore it if it was minimised. In the article the window was located using `EnumWindows` in conjunction with a few little tests. As an alternative, the solution presented here involves using the `GetWindow` API to iterate through the windows on the desktop.

Listing 7 shows the `initialization` section of a unit (ONEINST.PAS) that can be added to any project (`File | Add to Project...`) or simply added to the uses clause of a unit in a project. The code has conditional compilation directives to ensure successful operation in Delphi 1 and 2. Having used the unit somewhere in your project, the program will only allow single instances, as the `initialization` section of the unit executes all the code described above. There is a trivial demo project supplied on the disk that uses the `OneInst` unit, called INST_EG.DPR.

Thanks to Paul Broadfield for fixing the first version of my semaphore call and to Roy Nelson for the approach to switch back to the original instance.

Thunking Error

Q In Issue 13 you followed up your Issue 12 article on calling 16-bit DLLs from 32-bit by supplying a routine in the *Tips & Tricks* column to make calling 16-bit code a very simple affair. This little beauty served me very well until I tried to use it with a routine that took a pass by reference parameter. Even though the 16-bit code would modify the parameter, the 32-bit variable would still be in its pre-call state, unmodified. I had a good step through your code (which has taught me a good few useful bits – thanks) and realised that the `Call16BitRoutine` function

➤ *Listing 6*

```
procedure TForm1.ThreadTerminate(Sender: TObject);
begin
  Button1.Enabled := True
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  { This is the one that uses a thread. The button normally gets
    re-enabled in the thread's OnTerminate event handler. If there's
    a problem, we'll do it here }
  Button1.Enabled := False;
  try
    WaitForApp(ExecApp(Edit1.Text, Edit2.Text), ThreadTerminate);
  except
    Button1.Enabled := True;
    raise
  end
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  { This one doesn't use a thread, so we'll }
  { make sure we re-enable the button here  }
  Button2.Enabled := False;
  try
    WaitForApp2(ExecApp2(Edit1.Text, Edit2.Text));
  finally
    Button2.Enabled := True
  end
end;
```

throws my modified value away before we have chance to do anything useful with it. It seems that the routine is forgetting to do something.

Apart from that I must thank you for a very nice chunk of code which has saved me weeks. After playing with the Microsoft Thunk Compiler, I am *very very glad* your article reared its head when it did. *[These are genuine comments, Brian hasn't made them up. Editor]*.

A You're right. The routine copies the 32-bit parameters into blocks of memory accessible by 16-bit code, calls the 16-bit code and then doesnt do anything with them. Any parameters that might have been modified (`var`, pointer and structured `const` parameters have a potential of being altered) should be copied back to the 32-bit variables.

A replacement set of files from the article appears on this month's disk. Thanks are due to Russ Garner who spotted the problem and suggested a fix.

```
procedure EnsureSingleInstance;
var
  Wnd: HWnd;
  WndClass, WndText: array[0..255] of char;
begin
 {$ifdef Win32}
  { Try and create a semaphore. If we succeed, then check if the semaphore
    was already present. If it was then a previous instance is floating
    around. Note the OS will free the returned semaphore handle when the
    app shuts so we can forget about it }
  if (CreateSemaphore(nil, 0, 1,
    PChar(ExtractFileName(Application.ExeName))) <> 0) and
    (GetLastError = Error_Already_Exists) then
 {$else}
  if HPrevInst <> 0 then
 {$endif}
  begin
    Wnd := GetWindow(Application.Handle, gw_HWndFirst);
    while Wnd <> 0 do begin
      { Look for the other TApplication window out there }
      if Wnd <> Application.Handle then begin
        { Check it's definitely got the same class and caption }
        GetClassName(Wnd, WndClass, Pred(SizeOf(WndClass)));
        GetWindowText(Wnd, WndText, Succ(Length(Application.Title)));
        if (StrPas(WndClass) = Application.ClassName) and
          (StrPas(WndText) = Application.Title) then begin
          { This technique is used by the VCL: post a message then bring
            the window to the top, before the message gets processed }
          PostMessage(Wnd, wm_SysCommand, sc_Restore, 0);
         {$ifdef Win32}
          SetForegroundWindow(Wnd);
         {$else}
          BringWindowToTop(Wnd);
         {$endif}
          Halt
        end
      end;
      Wnd := GetWindow(Wnd, gw_HWndNext)
    end
  end
end;
initialization
  EnsureSingleInstance
end.
```

➤ *Listing 7*