# Please Call Later...

## *Part 3 of a series on using Windows and BDE callbacks*

***by Brian Long***

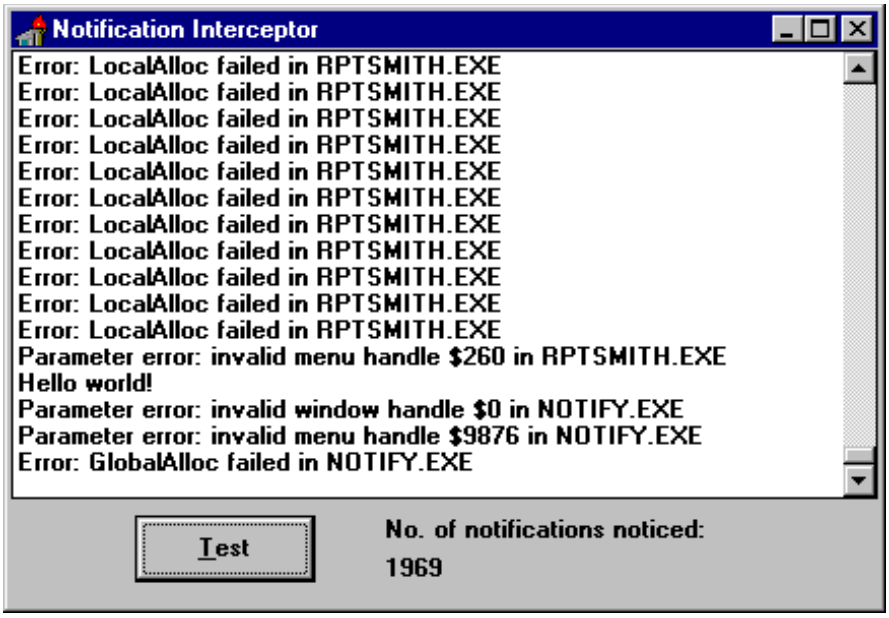
Having checked out some of the BDE's callback capabilities in the last issue, let's move on to the Windows `ToolHelp` library.

One of its APIs sets up a callback to be called whenever certain notifications are generated. There are a variety of notifications, such as programs exiting, invalid API parameters being passed and debug strings being issued. The full run down can be found by looking at the online help for `NotifyRegister`.

For the example on this month's disk (NOTIFY.DPR), we will focus on catching debug strings, parameter validation errors and general API errors. It should be noted at this point that the Win32 API does not support these notification callbacks – this is 16-bit only. However, there is some mileage for this program even in these enlightened days of Windows 95. Firstly, many of us are not only still running 16-bit applications but also developing them and so notifications will still be generated. Also, many Windows 95 APIs thunk down to 16-bit equivalents and so we may get a sniff of a notification from a 32-bit application anyway.

To register a notification callback we call `NotifyRegister` (which is declared in the `ToolHelp` unit) and to un-register it we call `NotifyUnRegister`. These APIs are called in the initialization section and an exit routine respectively of your form unit (see NOTIFYU.PAS in Listing 1).

Whenever a notification happens in the system, the callback is triggered and passed two pieces of information: a notification code and a long integer parameter whose meaning depends on the notification. If the notification represents a debugging string, the long integer is a `PChar` which points to the string. If the notification occurred because of an API failure, the extra parameter is a pointer to a `TNfyLogError` structure, whose only useful value is another code specifying what caused the failure. If it is a parameter validation notification we get a `TNfyLogParamError` record, which similarly contains an identification code, but also an address inside the API which objected to the parameter, along with the bad value itself.




➤ *Figure 1*

The notification handler itself doesn't do much except increment a notification counter (which is written to the screen periodically by a timer) and check that the notification received is one we are interested in. If it is not, the code exits immediately. If it is, it saves the task handle of the task in which the notification happened, and then sends a message to the form (causing a task switch and ensuring the data segment is set up okay for any code to execute) passing along the id code and the additional long integer. The message handler then adds an entry to the form's list box, describing the cause of the notification.

Since there are many possible causes for general API errors and API parameter errors, the routines which parse the information for these two notifications are in a separate unit (NFYUTIL.PAS, see Listing 2). In short they do a look-up and generate a descriptive string which is added to the list box. However they also identify the executable file name of the task which was running using a couple of additional `ToolHelp` APIs, to make the text more informative.

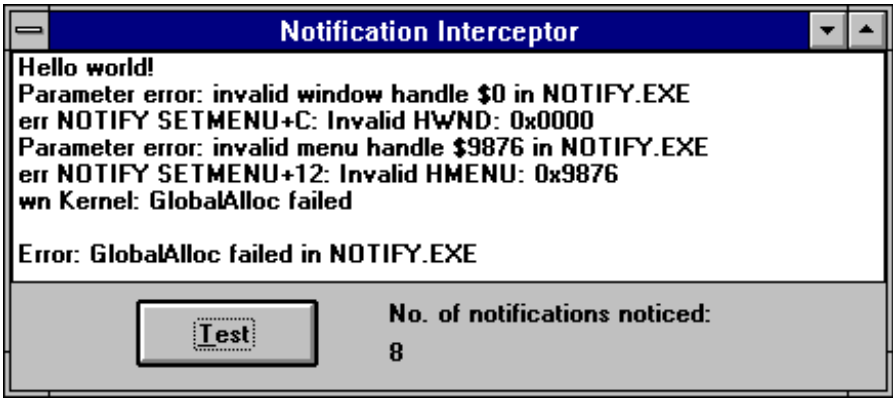
There is a button on the form which tests how well the notifications are being trapped. It issues a debug string using the API call `OutputDebugString`, calls `SetMenu` twice, once with a bad window handle (0) and once with a bad menu handle ($9876) and then asks 16-bit Windows to allocate more memory than it can offer (approximately 100Mb).

NOTIFY.EXE is shown running in Windows 95 in Figure 1 and in Windows 3.11 Debug Version in Figure 2. The first screen shot was taken after running ReportSmith, loading the CROSST.RPT report, closing ReportSmith and pressing the `Test` button on the Notify

program's form. You'll notice that there are nearly two thousand notifications generated by doing this. Most are sent as Windows loads segments of various modules that ReportSmith needs, but quite a few are generated for API errors of one description or another, as shown. The screen shot from the debugging version of Windows 3.11 doesn't include any ReportSmith information – I couldn't get ReportSmith to successfully load under that operating environment. However, even by just pressing the Test button you can see that Windows generates debug strings representing the parameter validation errors and general API errors that occur. It generates them with more detail than our project alone can do: it tells you which API had an invalid parameter.

It is possible to extend our program to find this information out, even in retail Windows. We are given the failure address and in theory can work backwards to find which module owns the code segment in the address and then which API lives at that address. Unfortunately in practice it is rather complicated: you have to do a lot of low-level mucking about with data structures both documented and undocumented, since there are no convenient APIs to do it for you. I suspect it is not worth going through the drudgery, particularly given the impact of Windows 95, where notifications are gone. However, if there is sufficient demand for this I will cover it in a later article.

Similarly, it is feasible to wrap this sort of stuff into a component which can be used for debugging purposes. The component can notice a notification error, walk the stack to find which application function passed the wrong API parameter and raise an exception at that point. But again, stack tracing and all that goes with it to produce such a component is




➤ *Figure 2*

➤ *Listing 1*

```
unit Notifyu;
{$ifdef WIN32} 'This is Windows 3.x-specific {$endif}
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls,
  ExtCtrls, ThunkU;
const wm_DoNotification = wm_User + 65;
type
  TForm1 = class(TForm)
    Panel1: TPanel;
    ListBox1: TListBox;
    TestBtn: TButton;
    Label1: TLabel;
    NumNfysLbl: TLabel;
    Timer1: TTimer;
    procedure TestBtnClick(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    procedure WMDoNotification(var Msg: TMessage);
      message wm_DoNotification;
  public
  end;
var Form1: TForm1;

implementation
uses ToolHelp, NfyUtil;
{$R *.DFM}
const NumNfys: Longint = 0;

function NotifyFunction(wID: Word; dwData: Longint):
  Bool; export;
begin
  Result := False;
  Inc(NumNfys);
  if not (wID in [nfy_OutStr, nfy_LogError,
    nfy_LogParamError]) then
    Exit;
  ProblemTask := GetCurrentTask;
  SendMessage(Form1.Handle, wm_DoNotification,
    wID, dwData);
end;

procedure TForm1.TestBtnClick(Sender: TObject);
begin
  OutputDebugString('Hello world!');
  SetMenu(0, 0);
  SetMenu(Handle, $9876);
  GlobalAlloc(0, 100000000);
end;

procedure TidyUp; far;
begin
  NotifyUnregister(GetCurrentTask);
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  NumNfysLbl.Caption := IntToStr(NumNfys);
end;

procedure TForm1.WMDoNotification(var Msg: TMessage);
begin
  with Msg, Listbox1.Items do
    case WParam of
      nfy_OutStr: Add(DoString(LParam));
      nfy_LogError: Add(DoLogError(LParam));
      nfy_LogParamError: Add(DoLogParamError(LParam));
    end;
end;

var NotifyFunctionThunk: TFarProc;

procedure TForm1.FormCreate(Sender: TObject);
begin
  if DebuggerRunning then
    MessageDlg(
      'Debugger active, some notifications will be lost',
      mtWarning, [mbOk], 0);
end;

initialization
  NotifyFunctionThunk :=
    NewMakeProcInstance(@NotifyFunction, HInstance);
  NotifyRegister(GetCurrentTask,
    TNotifyCallBack(NotifyFunctionThunk), nf_Normal);
  AddExitProc(TidyUp);
end.
```

rather messy. If there is demand, we can oblige...

One thing that is worth pointing out is that the program works best when not under the control of a debugger. Debuggers use notifications to identify when programs are starting and stopping and when code segments are discarded and reloaded. Consequently they use a notification callback. However, many notifications are not passed on and so you don't see debug strings and several others (although this does not affect the parameter validation or general API error notifications). Where Figure 1 shows over 1900 notifications when run outside the debugger, the same steps taken with NOTIFY.EXE being run within the debugger cause a rather smaller 350 notifications to be reported. The debugger has swallowed over 1600 of them.

To help warn of the problem, I have called `DebuggerRunning` from NFYUTIL.PAS in the `OnCreate` event handler for the form's, which checks if the program is running under the debugger. If it is, it puts up a message box reminding us of that fact. The way we can identify if the debugger has its hooks into us is to check a field in the debugger interface record, which is stored somewhere in the data segment of an application, though not of a DLL (hence the check for `PrefixSeg`, which is 0 in a DLL). The address of the record is stored at offset $24 in the data segment, and the layout of it is given in two files in the \DELPHI\SOURCE\RTL\SYS directory (providing you have the source code, of course): EXCP.ASM and SE.ASM.

The idea is to check the low word of the `dhDebugHooked` field, which will be non-zero if the debugger is active. In Listing 4 I've used a rather more cryptically coded version of the test (see later).

The other `ToolHelp` callback is designed for catching interrupts or exceptions. `InterruptRegister` and `InterruptUnRegister` allow us to install and remove a callback which is triggered when one of a certain number of interrupts occurs (see the online help for full details). There is a problem here in that, like the notification handler, only one such callback can be installed per task. Nothing wrong with that, you might think, until you realise that the `SysUtils` unit installs one of these callbacks in order to trap such errors as General Protection Faults, so it can turn them into the less severe Delphi software exceptions we know and love. It is possible to remove this callback by calling:

```
EnableExceptionHandler(False);
```

but this then suggests we have to take the trouble of re-implementing all the handling it was doing. A better idea is to hook into this callback, using a mechanism designed for just this purpose (though not documented). Inside the interrupt callback in `SysUtils`, the code checks a system-defined pointer called `ProcessorExceptHook`. If it is not `nil`, it jumps to the specified

➤ *Listing 2*

```
unit NfyUtil;
interface
uses WinTypes;
var  ProblemTask: THandle;
function DoString(Str: Longint): String;
function DoLogError(Block: Longint): String;
function DoLogParamError(Block: Longint): String;
function DebuggerRunning: Boolean;
implementation
uses ToolHelp, SysUtils, WinProcs;
var
  TaskEntry: TTaskEntry;
  ModuleEntry: TModuleEntry;
function DoString(Str: Longint): String;
var Loop: Byte;
begin
  Result := StrPas(PChar(Str));
  if Length(Result) > 0 then
    for Loop := Length(Result) downto 1 do
      if Result[Loop] in [#10, #13] then
        Delete(Result, Loop, 1);
end;
function DoLogError(Block: Longint): String;
begin
  with PNfyLogError(Block)^ do begin
    case wErrCode of
      { SEE FILE NFYUTIL.PAS ON DISK FOR DETAILS }
    end;
    TaskEntry.dwSize := SizeOf(TaskEntry);
    TaskFindHandle(@TaskEntry, ProblemTask);
    ModuleEntry.dwSize := SizeOf(ModuleEntry);
    ModuleFindHandle(@ModuleEntry, TaskEntry.hModule);
    Result := Format('Error: %s in %s', [Result,
       ExtractFileName(StrPas(ModuleEntry.szExePath))]);
  end;
end;
function DoLogParamError(Block: Longint): String;
const
  MsgType: array[False..True] of String[7] =
    ('error', 'warning');
var BadParam: Longint;
begin
  with PNfyLogParamError(Block)^ do begin
    case wErrCode of
      { SEE FILE NFYUTIL.PAS ON DISK FOR DETAILS }
    end;
    case (wErrCode and err_Size_Mask) of
      err_Byte: BadParam := Byte(lpBadParam);
      err_Word: BadParam := Word(lpBadParam);
      err_DWord: BadParam := Longint(lpBadParam);
    end;
    TaskEntry.dwSize := SizeOf(TaskEntry);
    TaskFindHandle(@TaskEntry, ProblemTask);
    ModuleEntry.dwSize := SizeOf(ModuleEntry);
    ModuleFindHandle(@ModuleEntry, TaskEntry.hModule);
    Result := Format('Parameter %s: %s $%x in %s',
      [MsgType[wErrCode and err_Warning <> 0], Result,
      BadParam, ExtractFileName(
      StrPas(ModuleEntry.szExePath))]);
  end;
end;
type
  PDebugRec = ^TDebugRec;
  TDebugRec = record
    dhMagic1, dhZero, dhMagic2, dhHookProc,
      dhDebugHooked: Longint;
    dhKind: Word; { Use TExceptionKind enumerated type above }
      dhAddr, dhCookie, dhNameLen, dhName, dhMsgLen,
      dhMsg, dhWantException, dhDoneExcept: Longint;
  end;
const
  DebuggerHook  = $24; { Offset in DS of pointer to debugger data }
function DebuggerRunning: Boolean;
begin
  Result := (PrefixSeg <> 0) and
    (LoWord(PDebugRec(Ptr(DSeg,
DebuggerHook)^)^.dhDebugHooked) <> 0);
end;
end.
```

address. We can set this pointer to point to a routine in our form unit and do what we wish to do there. The function type which needs to be used for your routine is defined in the SysUtils unit as TFaultHandler, which returns a value of type TFaultResponse, an enumerated type. Listing 3 is a snippet from SysUtils.

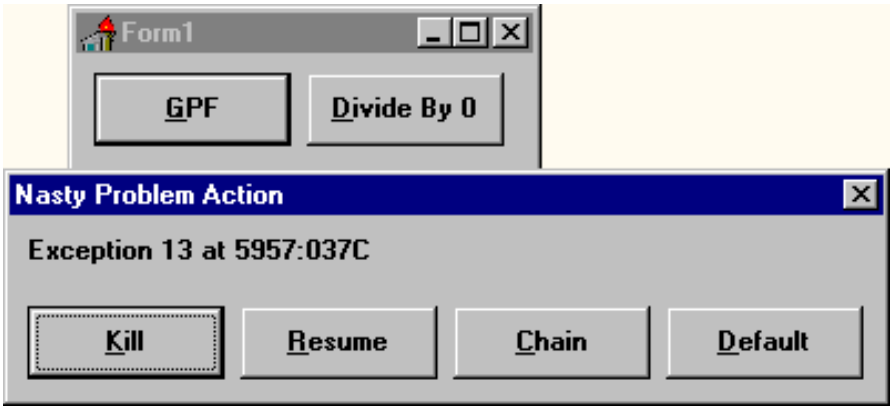
The function you set up has a number of options for terminating, as per the InterruptRegister online help. It can terminate the application, resume execution at the offending instruction (which rather assumes you have fixed the problem that caused the exception in the first place) or chain onto the default exception handler which Windows provides.

Another possibility is to do whatever Delphi would have normally done (ie turn the hardware exception into a software exception, causing the usual dialog message to appear). This can be achieved by calling a procedure DefaultExceptHandler and passing the exception number (fault id) and fault address.

The sample project which implements a processor exception hook routine is on the disk as INTRUPT.DPR and the source for the main form's unit is in Listing 4.

When you run it, it presents two buttons which generate exceptions. The first one causes a GPF by trying to access memory that doesn't belong to it and the second causes a divide by zero exception by dividing a number by zero. In order to hook the exception, the form's OnCreate handler assigns a custom routine, named IntHandler, to ProcessorExceptHook. When

➤ *Figure 3*




➤ *Listing 3*

```
{ Fault handler response }
  TFaultResponse = (frKill, frResume, frChain);
{ Fault handler function type }
  TFaultHandler = function(FaultID: Word;
    FaultAddress: Pointer): TFaultResponse;
{ Processor exception hook }
const
  ProcessorExceptHook: TFaultHandler = nil;
```

➤ *Listing 4*

```
unit Intruptu;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    GPFBtn: TButton;
    DivByZeroBtn: TButton;
    procedure FormCreate(Sender: TObject);
    procedure GPFBtnClick(Sender: TObject);
    procedure DivByZeroBtnClick(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
uses
  IntrupU2;
{$R *.DFM}
function IntHandler(FaultID: Word;
  FaultAddress: Pointer): TFaultResponse; far;
begin
  Result := frKill;
  with TActionFrm.Create(Application) do
    try
      Label1.Caption := Format('Exception %d at %p',
        [FaultID, FaultAddress]);
      case ShowModal of
        100: Result := frKill;
        101: Result := frResume;
        102:
          if Bool(PrefixSeg) and
            Bool(PWordArray(MemL[DSeg:36])^[8]) then
            MessageDlg('Do not choose this option ' +
              'whilst the debugger is active - in ' +
              'Windows 3.x you will drop to DOS; in ' +
              'Windows 95 you will hang the system. ' +
              'Terminating application instead',
              mtError, [mbOk], 0)
          else
            Result := frChain;
      else
        DefaultExceptHandler(FaultID, FaultAddress);
      end;
    finally
      Free;
    end;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  ProcessorExceptHook := IntHandler;
end;
procedure TForm1.GPFBtnClick(Sender: TObject);
begin
  Mem[0:0] := 0;
end;
procedure TForm1.DivByZeroBtnClick(Sender: TObject);
const
  F: Byte = 0;
begin
  F := F div F;
end;
end.
```

IntHandler is invoked, from the interrupt callback in SysUtils upon an exception occurring, it launches another form (see Figure 3) which gives the user a choice of actions. Depending on the action chosen, the form returns one of the three fault response values or calls DefaultExceptHandler.

To determine which button on the second form was pressed, each one has a different ModalResult property value. When buttons with non-zero ModalResult values are pressed on a form launched with ShowModal, they close the form and cause ShowModal to return their ModalResult value. The case statement in IntHandler checks for each value and performs its appropriate action.

One special case worth mentioning is the chain response. When you chain onto the default exception handler, Windows draws the white system modal message box, giving you the chance to close the application or ignore the problem. If your application is running under the debugger, trying to close the application has dire consequences. In Windows 3.x Windows will drop straight to DOS, losing all unsaved data. In Windows 95, when you push the offered button marked Shut Down, the whole system will freeze forcing a reboot. The code in IntHandler checks for the debugger using the logic discussed above and if it finds a compromising situation, elects to take a safe option instead, terminating the application.

Having done normal callbacks, inter-task callbacks (including BDE callbacks) and now interrupt level callbacks we have at last covered all the ground (except for real mode callbacks that I mentioned in the first article, but I'm avoiding those).

---

Brian Long is an independent consultant and trainer specialising in Delphi. His email address is 76004.3437@compuserve.com