Please Call Later...

Callbacks in Windows and the Borland Database Engine (Part 1)

by Brian Long

A callback is a routine in your application that you arrange to be called by someone else. Generally this "someone else" is Windows, but in a Delphi application it can also be the Borland Database Engine (BDE), among others. Reasons for using callbacks vary, depending on why the support for them was implemented in the first place. However, here are just a few examples:

- ➣ You want to find all the top-level windows in the system, and perform some action for each of them (Windows `EnumWindows`);
- ➣ You want to find all the child windows of a particular window, and perform some action for each of them (Windows `EnumChildWindows`);
- ➣ You want to be notified when a Paradox table is modified, or periodically through a database operation, so you can show a progress report (BDE `DbiRegisterCallBack`);
- ➣ You want to be notified when a Windows API call is supplied an invalid parameter (ToolHelp `NotifyRegister`);
- ➣ You have some hardware and need to service it reliably every twentieth of a second (multimedia library's `timeSetEvent`);
- ➣ You have a desire to trap GPFs yourself, instead of letting Delphi's exception handling do it (ToolHelp `InterruptRegister`).

Before we look at implementing callbacks, we need to examine some theory and history.

The core set of Windows APIs has many routines that call callbacks, or cause callbacks to be called, of which `EnumWindows` and `EnumChildWindows` are just two – try searching Delphi's Help for "Callback functions (3.1)". There are three kinds of callback routines and it is important to understand the difference, since two kinds need to be given special attention.

The first type of callbacks will be called whilst your application is the current task, usually immediately following the callback-oriented API call. `EnumWindows` and `EnumChildWindows` work like this: you call the API passing an appropriate routine and it is immediately called once for each window in question. The second callback category is called when your task is not necessarily the active task. The table modification and parameter validation callbacks operate like this – when a table is modified, or an invalid parameter passed, it can be from any task in the system. There is no guarantee which task will be running when the callback gets called.

The last category of callbacks are interrupt level callbacks – they are called from interrupt handlers. An example of this type is the Windows Multimedia library's high resolution, accurate timer callback set in motion by the `timeSetEvent` API (the multimedia help file is `\DELPHI\BIN\MMSYSTEM.HLP`). The `ToolHelp` library also invokes a callback upon certain interrupts occurring, and this is initiated with `InterruptRegister` (although only one of these callbacks is allowed per task, and the `SysUtils` unit already installs one for the hardware exception handling).

So some callbacks are called when your task is active, some are called when any task is active and others are called when certain interrupts are generated, again when any arbitrary task is active.

There is another type of callback that is really reserved for the propeller heads among us. Those people wishing to implement protected mode interrupt handlers in Windows, for interrupts that occur while the processor is in real mode, can make use of DPMI services to use real mode callbacks. The idea is that you pass the interrupt handler's address to the appropriate DPMI interrupt call, and it returns you the address of a real mode callback. When the real mode interrupt triggers the real mode callback switches to protected mode and calls your handler. The terminology is a bit different than with Windows callbacks. We would expect the real mode callback, generated by the DPMI call, to be called a mode thunk, and your interrupt handler to be called the callback, but such is life. Anyway, I don't think we'll be coming back to this subject, although if it is of interest it is examined in detail with examples in Section 5.10 of my Addison-Wesley book *The Borland Pascal Problem Solver*.

For a callback to work, it needs to be made exportable. This means that the first occurrence of the procedure or function header must be followed by the `export` keyword. The first occurrence will either be a declaration of a public callback in the `interface` section of a unit, or the start of the definition of a private callback in the `implementation` section of a unit, or in the project source code. Whether you actually export the callback, by adding it to the `exports` statement, is irrelevant with respect to its correct functioning. Exporting a routine simply puts information about it in the header of the executable file. This is fairly pointless in programs, as the API that reads such information, `GetProcAddress`, claims to not find exported routines from programs. If you export a callback from a DLL, it makes it visible to other applications as a normal subroutine.

Historically, the use of Windows callbacks has been a thorny issue and cause for many a problem. There is a pair of Windows APIs, called `MakeProcInstance` and `FreeProcInstance`, that needed to be called for things to work okay in programs, but weren't needed in DLLs (although it didn't matter if you used them), and the whole subject was not well understood by Windows programmers. The basic issue was that when Windows was executing and about to call the callback in your program, the data segment register (DS) was set to a Windows data segment. By the time the callback executes, DS needs to be looking at the program's data segment, so any references to your global variables etc will be valid operations.

At first glance you may think that the entry code for the callback (function prolog code) should be able to set DS up, but this is not so. Remember that if you run multiple instances of an application, the code is only loaded the first time. Each invocation of the application uses the same code, but has its own separate data segment. One piece of code could set DS up correctly for one instance, but not for all instances (although this is exactly what happens in DLLs where each copy uses the same data segment).

This is where MakeProcInstance comes in: it returns a pointer to a tiny piece of code, called a thunk (or more specifically an instance thunk) that in conjunction with the callback prolog will set DS up correctly. MakeProcInstance takes two parameters, a callback function address and an instance handle (the HInstance system variable in a Delphi program). The instance thunk (which can be freed by FreeProcInstance) moves HInstance (which in Windows is the data segment value) into AX and then calls the callback. The callback prolog moves AX into DS, providing the callback is exportable, finishing the job. So the thunk's code looks like this:

```
asm
  mov ax, SavedHInstance
  jmp SavedCallBack
end;
```

and the prolog code for a function in an application would look something like this by the time Windows has loaded it (in fact the compiler generates slightly different code which is changed by Windows as it is read in):

```
nop
nop
nop
inc bp
push bp
mov bp, sp
push ds
mov ds, ax
```

Because an exportable routine loads DS from AX, it is unsafe to call such a routine directly if the data segment will be executed at any point during its lifetime – AX won't have been set up with the data segment value in it. Instead you should always call the code address returned from MakeProcInstance (although this isn't necessary in DLLs – see later).

So hopefully that makes the subject of callbacks as clear as a reasonably clear thing. Unfortunately the tale does not stop there. There is more, much more. Recall that most of the above was marked as history. With Delphi we can almost forget about all the MakeProcInstance stuff, because every Delphi application has smart callbacks enabled by default. Smart callbacks are set by the {$K+} directive in the project source file, or in the Options | Project... | Compiler page. The intention of smart callbacks is to remove the need for MakeProcInstance by taking advantage of a Windows implementation detail. Whenever you call a DLL, the stack it uses is the caller's. So when you are calling Windows APIs or the BDE or any other DLL, SS points to your stack segment. If you know anything about the layout of a Windows application in memory, you will know that the stack lives in the data segment.

So, smart callbacks are smart because the prolog code generated for exportable callbacks loads the correct segment into DS, by moving SS into AX, which then later in the prolog gets moved into DS. This means the instance thunk is redundant: whatever it stores in AX is overwritten by the smart callback prolog. So no more MakeProcInstance. This is great news for Delphi programmers who don't even need to know the existence of such an API, but it is also good for code compatibility with Windows 95. The Win32 API doesn't make any use of MakeProcInstance, although a stub function is there for code that's being ported to 32-bit. The smart callback prolog looks like this:

```
mov ax, ss
nop
inc bp
push bp
mov bp, sp
push ds
mov ds, ax
```

Smart callbacks also remove the earlier warning about calling exportable callbacks directly. Since the DS setup is all packaged in the prolog now, instead of being split over a thunk and a prolog, callbacks compiled with the smart callbacks option can be called from almost anywhere.

Unfortunately, smart callbacks are not a solution to all callback problems. They only work with callbacks that are called when your task is active. Consider what would happen if a callback in your application was triggered when another program was active. Because a task switch does not occur (your callback is called in the context of the active task) that program's SS register would be set as your DS. Any access you made to the data segment would examine or modify someone else's data. Oh dear. Of course if your callback doesn't cause any reference to DS, then there will be no problem.

When we come across this problem and eventually work out what is going on, our first reaction seems, judging by talk on CompuServe, to be to try and turn smart callbacks off for the extent of the callback definition, using compiler directives. This is sadly doomed to failure since the smart callbacks option is a global setting, not something that can be turned on and off at will. It is a shame that the compiler doesn't object to spurious {$K-} and {$K+} directives found in various units – it would prevent much confusion.

A general solution for this problem for both the second and third category of callbacks is to implement them in DLLs where DS gets correctly set up without recourse to confusing APIs or funny compiler options, but it is a chore to manufacture a new project and results in another file to distribute. What would be desirable would be to resolve the problem in the application project. Turning off smart callbacks completely is not a viable option – various parts of the VCL have been coded with the assumption that smart callbacks are on. If you turn them off, those sections will not work.

One approach that does something akin to the desired job is to use manually written assembler prolog to enter the callback. Instead of referring to the callback directly or to an instance thunk from MakeProcInstance, when calling the callback related API, you can refer to a dedicated assembler routine like this:

```
procedure CallBackThunk;
  assembler;
asm
  { Cause DS to be set up
    correctly }
  mov ax, seg @Data
  { Bypass the smart callback
    instruction }
  jmp CallBackProc + 3
end;
```

What this does is to jump over the main smart callback instruction (MOV AX, SS and the following NOP opcode, which together take three bytes) after having set up AX with an appropriate value. This is not an all-singing, all-dancing solution, but it allows you to get the desired effect. The thing you need to bear in mind is that it is the sort of code I earlier questioned the omission of in normal prolog code, and hence has the same limitation – it will only work with the first instance of an application. Subsequent instances will access the data of the first one. Also the callback has to be careful what it calls. For example, if it calls the VCL ShowMessage routine, the program will crash. It must not cause any exportable routines to be called, since that routine's prolog will load SS into DS again. All other callbacks will still have smart callback prolog and are still prone to the inherent problem when being called in the context of another task.

This approach is fine if you only want one instance. It poses a good question about how to detect if your program is a second (or subsequent) instance and, if it is, how to behave correctly. The generally accepted definition of correct behaviour when launching a second copy of a multiple instance app is to set focus to the original instance. Many proposed solutions don't work with all combinations of Delphi main window properties. We'll come back to this problem, and indeed the problem of the setting up the right data segment for multiple instances when we look at inter-task callbacks, next time.

The third category of callbacks generally need to be implemented in DLLs to prevent system crashes anyway (InterruptRegister callbacks are an exception), which is convenient since a DLL-based callback will work regardless. The reason for this DLL requirement is that code executed at interrupt time needs to be in fixed segments (ones which won't be paged out to the swap file), but fixed segments in executables are treated as moveable (and so potential candidates for being paged out – note this is paging as distinct from normal segment discarding) when read in by Windows. This is examined further by Matt Pietrek in *Windows Internals*, published by Addison-Wesley (Chapter 2, the section entitled "Fixed Versus MOVEABLE Segments").

Okay, now that we have seen the theory, let's try putting it into practice. We'll go through taking each of the callbacks listed above in turn and see if we can make something out of it.

Firstly, the two window enumerators. Listing 1 shows some code from the ENUMWND.DPR project, that puts information about each top-level window into a listbox on the form when the form starts up. The information can be updated at any time by pressing an alphanumeric key. To invoke the callback, EnumWindows is called and is passed two parameters. The first is the address of the callback to call for each top-level window in the system, the second is any long integer we want passed to the callback, simply for our own reference. If we have no information we want passed in, we can just pass zero. In this case, I am passing the object reference for the listbox on the form (WindowBox).

Notice that I have been browsing the source for the WinCrt unit to identify how to future-proof my code for Win32 as much as possible before the launch of Delphi32. WinCrt shows that exportable routines need to be marked stdcall instead of export in Delphi32, and also that Delphi32 will define a conditional symbol WIN32 to allow one source file to be maintained across the two platforms.

Because the callback is a standard subroutine, not a method, it can't directly access the listbox without de-referencing Form1 (ie Form1.WindowBox). Arranging for the listbox object reference to be passed into the callback is an alternative to doing this. The callback itself is very simple: it sets the return value to True, indicating that the enumeration should not terminate when this particular invocation of the callback has exited. After this, it adds a string of information about the current window, obtained from GetWindowInfo, into the listbox. GetWindowInfo uses a variety of Windows APIs to find the window class, window caption, owning module and whether the window is hidden or not, and builds a representative string out of them, in a format similar to that used by WinSight. The result for my Windows 95 desktop can be seen in Figure 1. It's interesting to see that this 16-bit application can see 32-bit application Windows in Windows 95. The reasoning behind this can be found in a Sept 1994 article by Matt Pietrek in *Microsoft Systems Journal*, "Stepping Up To

```
unit Enumwndu;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    WindowBox: TListBox;
    procedure FormCreate(Sender: TObject);
    procedure WindowBoxKeyPress(
      Sender: TObject; var Key: Char);
  end;
var Form1: TForm1;

implementation
{$R *.DFM}

function GetWindowInfo(Wnd: HWnd): String;
var
  Caption: array[0..255] of Char;
  ClassName: array[0..255] of Char;
  ExeName: array[0..255] of Char;
  Instance: Cardinal;
const
  Visibility: array[False..True] of String[8] =
    ('(hidden)', '');
begin
  GetWindowText(Wnd, Caption, SizeOf(Caption));
  GetClassName(Wnd, ClassName, SizeOf(ClassName));
  {$ifndef WIN32}
  Instance := GetWindowWord(Wnd, gww_HInstance);
  {$else}
  Instance := GetWindowLong(Wnd, gwl_HInstance);
  {$endif}
  GetModuleFileName(Instance, ExeName, SizeOf(ExeName));
  Result := '{' + StrPas(ClassName) + '} ' +
    StrPas(ExeName) + ' “' +
    StrPas(Caption) + '" ' +
    Visibility[GetWindowLong(Wnd, gwl_Style) and
      ws_Visible <> 0];
end;

function EnumWindowsProc(
  Wnd: HWnd; UserData: Longint): Bool;
{$ifndef WIN32} export;{$else} stdcall;{$endif}
begin
  Result := True;
  TListBox(UserData).Items.Add(GetWindowInfo(Wnd));
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  EnumWindows(@EnumWindowsProc, Longint(WindowBox));
end;

procedure TForm1.WindowBoxKeyPress(
  Sender: TObject; var Key: Char);
begin
  WindowBox.Clear;
  EnumWindows(@EnumWindowsProc, Longint(WindowBox));
end;
end.
```

➤ *Listing 1*

32 Bits: Chicago's Process, Thread, and Memory Management."

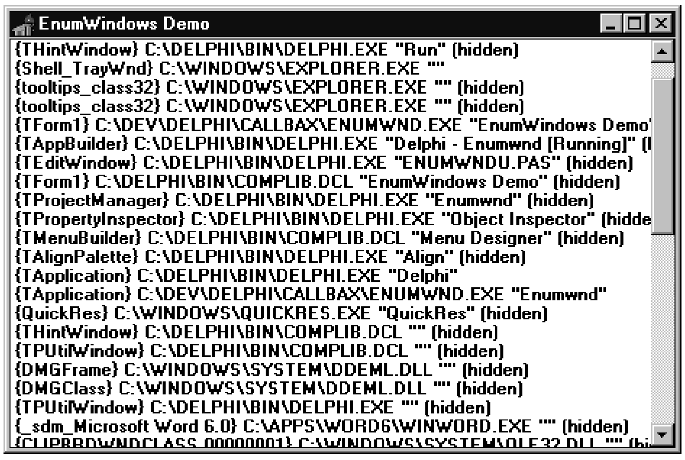
Moving onto `EnumChildWindows`. The example program for this, ENUMCHLD.DPR does a similar job to the previous example, but uses both `EnumWindows` and `EnumChildWindows` to populate an outliner (also called `WindowBox`) instead of a listbox. Figure 2 shows this in action. The important parts of the code are shown in Listing 2.

You can see that for each top-level window, the code calls `EnumChildWindows` with the given window handle. Thus, for each top level window we can find all its children and add them in the outline as child nodes (this is achieved in the child window callback by preceding each entry with a space character, (#32, ASCII character 32).

In the next issue, we'll look at the inter-task callbacks available from Windows and the BDE and see what they can do for us when we can work out how to call them...

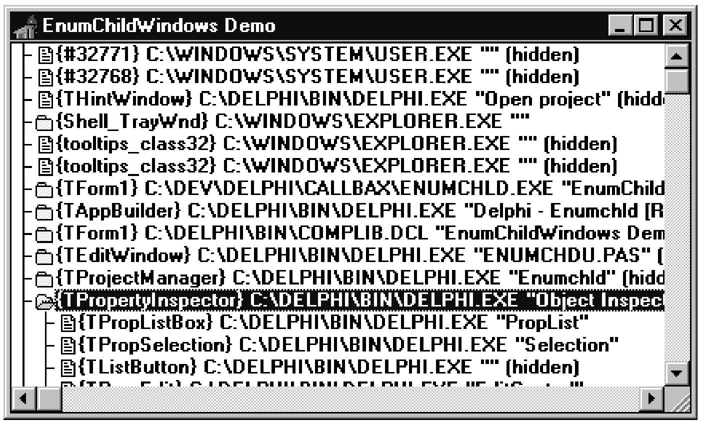
Brian Long is an independent consultant and trainer specialising in Delphi. His email address is 76004.3437@compuserve.com

➤ *Figure 1*



```
EnumWindows Demo
{THintWindow} C:\DELPHI\BIN\DELPHI.EXE "Run" (hidden)
{Shell_TrayWnd} C:\WINDOWS\EXPLORER.EXE ""
{tooltips_class32} C:\WINDOWS\EXPLORER.EXE "" (hidden)
{tooltips_class32} C:\WINDOWS\EXPLORER.EXE "" (hidden)
{TForm1} C:\DEV\DELPHI\CALLBAX\ENUMWND.EXE "EnumWindows Demo"
{TAppBuilder} C:\DELPHI\BIN\DELPHI.EXE "Delphi - Enumwnd [Running]" (
{TEditWindow} C:\DELPHI\BIN\DELPHI.EXE "ENUMWNDU.PAS" (hidden)
{TForm1} C:\DELPHI\BIN\COMPLIB.DCL "EnumWindows Demo" (hidden)
{TProjectManager} C:\DELPHI\BIN\DELPHI.EXE "Enumwnd" (hidden)
{TPropertyInspector} C:\DELPHI\BIN\DELPHI.EXE "Object Inspector" (hidde
{TMenuBuilder} C:\DELPHI\BIN\COMPLIB.DCL "Menu Designer" (hidden)
{TAlignPalette} C:\DELPHI\BIN\DELPHI.EXE "Align" (hidden)
{TApplication} C:\DELPHI\BIN\DELPHI.EXE "Delphi"
{TApplication} C:\DEV\DELPHI\CALLBAX\ENUMWND.EXE "Enumwnd"
{QuickRes} C:\WINDOWS\QUICKRES.EXE "QuickRes" (hidden)
{THintWindow} C:\DELPHI\BIN\COMPLIB.DCL "" (hidden)
{TPUtilWindow} C:\DELPHI\BIN\COMPLIB.DCL "" (hidden)
{DMGFrame} C:\WINDOWS\SYSTEM\DDEML.DLL "" (hidden)
{DMGClass} C:\WINDOWS\SYSTEM\DDEML.DLL "" (hidden)
{TPUtilWindow} C:\DELPHI\BIN\DELPHI.EXE "" (hidden)
{_sdm_Microsoft Word 6.0} C:\APPS\WORD6\WINWORD.EXE "" (hidden)
{CLIPBRDWNDCLASS 00000001} C:\WINDOWS\SYSTEM\OLE32.DLL "" (hi
```

➤ *Figure 2*



```
EnumChildWindows Demo
├ {#32771} C:\WINDOWS\SYSTEM\USER.EXE "" (hidden)
├ {#32768} C:\WINDOWS\SYSTEM\USER.EXE "" (hidden)
├ {THintWindow} C:\DELPHI\BIN\DELPHI.EXE "Open project" (hidd
├ {Shell_TrayWnd} C:\WINDOWS\EXPLORER.EXE ""
├ {tooltips_class32} C:\WINDOWS\EXPLORER.EXE "" (hidden)
├ {tooltips_class32} C:\WINDOWS\EXPLORER.EXE "" (hidden)
├ {TForm1} C:\DEV\DELPHI\CALLBAX\ENUMCHLD.EXE "EnumChild
├ {TAppBuilder} C:\DELPHI\BIN\DELPHI.EXE "Delphi - Enumchld [R
├ {TForm1} C:\DELPHI\BIN\COMPLIB.DCL "EnumChildWindows Dem
├ {TEditWindow} C:\DELPHI\BIN\DELPHI.EXE "ENUMCHDU.PAS" (
├ {TProjectManager} C:\DELPHI\BIN\DELPHI.EXE "Enumchld" (hidd
├ {TPropertyInspector} C:\DELPHI\BIN\DELPHI.EXE "Object Inspec
│  ├ {TPropListBox} C:\DELPHI\BIN\DELPHI.EXE "PropList"
│  ├ {TPropSelection} C:\DELPHI\BIN\DELPHI.EXE "Selection"
│  ├ {TListButton} C:\DELPHI\BIN\DELPHI.EXE "" (hidden)
```

➤ ***Below:*** *Listing 2*

```
function EnumChildWindowsProc(Wnd: HWnd; UserData: Longint): Bool; export;
begin
  Result := True;
  TOutline(UserData).Lines.Add(#32 + GetWindowInfo(Wnd));
end;

function EnumWindowsProc(Wnd: HWnd; UserData: Longint): Bool; export;
begin
  Result := True;
  TOutline(UserData).Lines.Add(GetWindowInfo(Wnd));
  EnumChildWindows(Wnd, @EnumChildWindowsProc, UserData);
end;
```