Please Call Later...

Callbacks in Windows and the Borland Database Engine (Part 2)

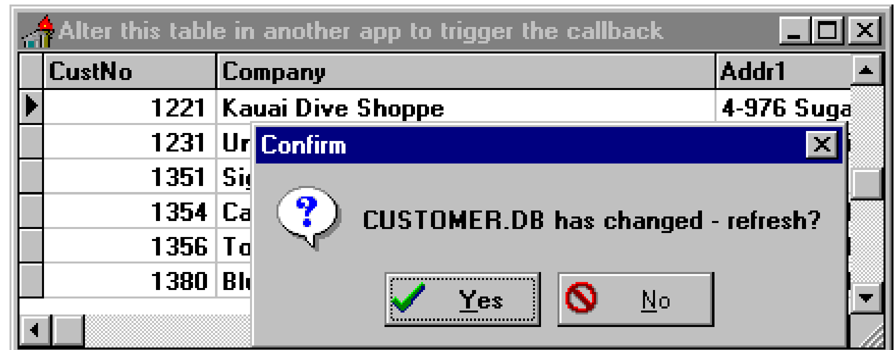
by Brian Long

In issue 4 we looked at Windows callbacks that were called when our application was the current task. This time we will look at the more interesting cases of callbacks that are called when any task may be active. To start with we are going to focus on some Borland Database Engine (BDE) callbacks which are installed using the BDE's `DbiRegisterCallBack` routine.

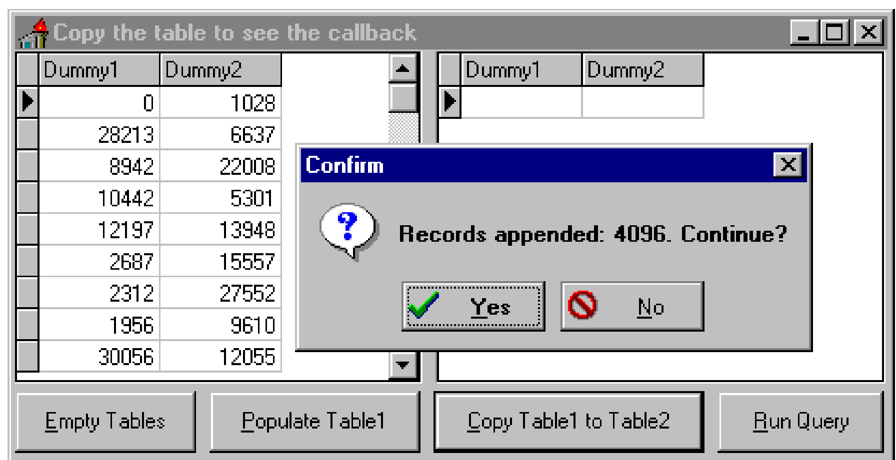
If you have not got hold of the BDE documentation, you might care to download the BDE help file which can be found in BDEHEL.ZIP (217Kb) in Library 6 of the DELPHI forum on Compuserve. Also, in case you are unaware, an updated version of the BDE itself (which shipped with Visual dBASE for Windows 5.5 and the Delphi 1.02 update) is available as file BDEDEMO.ZIP in Library 4 of the BDEVTOOLS forum on CompuServe, where the BDE is normally supported.

The interface sections of the units that declare all the BDE stuff can be found in the directory \DELPHI\DOC, as files DBIPROCS.INT, DBITYPES.INT and DBIERRS.INT. In DBITYPES.INT there is a callbacks section from lines 959 to 1065. The available BDE callbacks are listed in the `CBType` enumerated type; although one that is marked as reserved, `cbReserved21`, later turns out to be redefined as `cbServerCall`. This callback type is actually used in any Delphi application using the DB unit (ie applications using the database components); it is used in the `Session` component to set up the SQL mouse cursor for server-based operations.

The new 32-bit BDE coming from Borland adds new callbacks which are triggered when a login occurs, when a delayed update occurs, when fields get recalculated and upon a trace event. However, Delphi 2.0's BDE callback support is rather changed (and markedly improved) so we'll stick to 16-bits only in this article.



➤ *Figure 1: Catching table changes*



➤ *Figure 2: The general progress callback*

The first callback we will investigate is `cbTableChanged`. If we install one of these for a Paradox table, any modifications made to the table by any other BDE-based application cause the callback to be executed. This allows your application to refresh its view of a table only when it is needed, rather than in the normal way, which is periodically via a timer component.

There are two versions of the program that implement this callback (see Figure 1) on the disk: CHANGE.DPR and CHANGE2.DPR. The former uses the assembly prolog which was discussed in the previous article and is consequently tied to only running correctly for the first instance. Subsequent invocations would talk to the first one's data segment instead of their own. To allow this program to be a valid item to add on the disk, I have added some code based on a technique from Borland UK's Roy Nelson to restrict it to only running once. If you try and launch it again, it detects that there is a previous instance and switches to it instead of continuing. The important point about this code is that many suggestions of similar routines do not handle all Delphi startup possibilities correctly.

To expound, the general functionality of a single instance application is that a second instance should cause the first instance's main window to appear on the desktop. This means

```
unit Changeu;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls, Grids,
  DBGrids, DB, DBTables;
const wm_TableChanged = wm_User + 57;
type
  TForm1 = class(TForm)
    Table1: TTable;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    procedure FormDestroy(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private   { Private declarations }
    FOldCallBack: TCallBack;
    FChangeFunctionThunk: TFarProc;
  public    { Public declarations }
    procedure WMTableChanged(var Msg: TMessage);
      message wm_TableChanged;
  end;
var Form1: TForm1;
implementation
uses  DbiTypes, DbiProcs;
{$R *.DFM}

function FindPrevInstanceProc(Wnd: HWnd; UserData:
  Longint): Bool; export;
var WndClass, WndText: array[0..255] of char;
begin
  Result := True;
  { Concentrate solely on our EXE }
  if GetWindowWord(Wnd, gww_HInstance) = HPrevInst
  then begin
    GetClassName(Wnd, WndClass, Pred(SizeOf(WndClass)));
    GetWindowText(Wnd, WndText,
      Succ(Length(Application.MainForm.Caption)));
    { Normally first window will be Application's but if
      app started minimised, it will be main form's }
    if (StrPas(WndClass) = Application.ClassName) or
       ((StrPas(WndText) = Application.MainForm.Caption)
        and IsIconic(Wnd)) 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);
      BringWindowToTop(Wnd);
      Halt;
    end;
  end;
end;

function ChangeFunction(ecbType: CBType; iClientData:
  Longint; var CbInfo: Pointer): CBRType; export;
begin
  Result := cbrUseDef;
  if ecbType = cbTableChanged then
    PostMessage(Application.MainForm.Handle,
      wm_TableChanged, 0, iClientData);
  with Form1.FOldCallBack do
    if ChainedFunc <> nil then Result :=
      pfDBICallBack(ChainedFunc)(cbTableChanged,
        Data, Buffer)
end;

procedure ChangeFunctionThunk; assembler;
asm
  mov ax, seg @Data
  { Bypass the smart callback instruction }
  jmp ChangeFunction + 3
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  if HPrevInst <> 0 then
    EnumWindows(@FindPrevInstanceProc, 0);
  FChangeFunctionThunk := @ChangeFunctionThunk;
  with FOldCallBack do
    DbiGetCallBack(nil, cbTableChanged, Data, BufLen,
      Buffer, @ChainedFunc);
  DbiRegisterCallBack(Table1.Handle, cbTableChanged,
    Longint(Table1), 0, nil,
    pfDbiCallback(FChangeFunctionThunk));
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  DbiRegisterCallBack(Table1.Handle, cbTableChanged,
    0, 0, nil, nil);
end;

procedure TForm1.WMTableChanged(var Msg: TMessage);
begin
  if MessageDlg(TTable(Msg.LParam).TableName +
    ' has changed - refresh?', mtConfirmation,
    [mbYes, mbNo], 0) = mrYes then
    TTable(Msg.LParam).Refresh;
end;
end.
```

➤ *Listing 1*

bringing it to the front if it is behind other windows, but also restoring it if it is minimised. Normally, when a Delphi application is minimised, the icon is actually the iconised version of the normally hidden `Application` object's window: all other windows are hidden. However, if the application starts off life with the main form having a `wsMinimized` `WindowState` property, the icon you see when you launch the application is the main form's icon (you can use WinSight to verify all this). This explains why, in Windows 95, when you restore a minimised Delphi application, it just appears rather than enlarging from the icon bar: the windows are being un-hidden rather than restored. The different icons cause headaches for the previous instance code, but this version seems to take account of all obvious possibilities.

The source for the unit for the single instance version is in Listing 1. The code to register the callback is in the form's `OnCreate` event handler after the check for another instance, and it is un-registered in the form's `OnDestroy` handler. This ensures it is active throughout the useful part of the program's lifespan. `FormCreate` ensures we don't block any existing handler by storing its details in a form data field called `FOldCallBack`, just like the VCL `Session` object does. Using `FOldCallBack` allows the old callback to be chained onto from our callback routine. This field's type, `TCallBack`, has already been declared in the DB unit.

The actual registration specifies the handle of the table we wish to be kept informed about, a 32-bit value that we wish to be passed to the callback for our own purposes (the table object reference in this case) and some reference to the callback routine itself. You may recall from the previous article that unlike many callbacks, those that can be called when any arbitrary task is active cannot be referred to simply by their name. This is because the data segment will not be set to that required by the routine, but that of the active task (due to the smart callback option, which needs to be on in a VCL application). The remedy used here is the aforementioned assembler thunk. The code in `ChangeFunctionThunk` causes `DS` to be set up by placing the correct value into `AX` and jumping over the callback prolog instruction which would otherwise write the wrong value into `AX`.

Now onto the callback itself. It sets the return value to a safe value defined in DBITYPES.INT and then

checks that it is the appropriate type of callback.

The next thing it does might seem a bit odd. It sends a message to the main form window, which is picked up by a message handler. The point of this is that I want to put up a message box to the user, informing them that the table has been updated by someone, and check whether they wish to refresh the table (see Figure 1). As mentioned in the last issue, causing other code to execute that has been made exportable with the `export` keyword is a bad idea. `ShowMessage` and `MessageDlg` do this. The problem is that their prolog code will still contain the instruction which will set the data segment to that of the active task, undoing the work of the thunk code. The usual result of calling `MessageDlg` in this type of callback is a GPF, since some code at some point in the VCL will refer to something in the data segment (the wrong data segment). Sending a message to the form will cause the right data segment to be set up all round because a task switch will occur before the message handler starts. The message has the user-defined callback data passed along to it as the long parameter, which means that the message handler can access the table object in question by typecasting `Msg.LParam` back to a `TTable`.

During a BDE callback, no BDE operations should be performed. This is why the message (which possibly causes a refresh) is sent with `PostMessage` instead of `SendMessage` – `SendMessage` causes the message to be handled immediately, whereas `PostMessage` doesn't, allowing the callback to finish first.

Remember that this type of callback is called in the context of whatever task is running. This means that if the callback causes a GPF (eg by calling `ShowMessage` directly), it will be reported as coming from the application that was active at the time the callback was started, not from your application, which was the real culprit.

The CHANGE2 project is not restricted to single instances. It

```
uses
  ThunkU, ...
...
procedure TForm1.FormCreate(Sender: TObject);
begin
  FChangeFunctionThunk := NewMakeProcInstance(@ChangeFunction, HInstance);
...
```

➤ *Listing 2*

```
unit ThunkU;
{$ifndef WINDOWS} 'This is suitable only for 16-bit Windows' {$endif}
interface
uses  WinTypes;
function NewMakeProcInstance(Proc: TFarProc; Instance: THandle): TFarProc;
procedure NewFreeProcInstance(Proc: TFarProc);
implementation
uses  WinProcs, SysUtils;
type
  { Used by NewMakeProcInstance and NewFreeProcInstance }
  PProcInst = ^TProcInst;
  TProcInst = record
    Mov: Byte; DSeg: Word;        { mov ax, seg @Data }
    Jmp: Byte; ProcAddr: Pointer; { jmp Addr + 3 }
    DataSelector: Word;           { saved data selector }
  end;
function NewMakeProcInstance(Proc: TFarProc; Instance: THandle): TFarProc;
begin
  Result := GlobalAllocPtr(gmem_Share or gmem_Fixed, SizeOf(TProcInst));
  with PProcInst(Result)^ do begin
    Mov := $B8;
    DSeg := Instance;
    Jmp := $EA;
    ProcAddr := Proc;
    Inc(PtrRec(ProcAddr).Ofs, 3);
    DataSelector := PtrRec(Result).Seg;
    { Note Delphi help is out of date, shouldn't free the code selector }
    PtrRec(Result).Seg := AllocDSToCSAlias(DataSelector);
  end;
end;
procedure NewFreeProcInstance(Proc: TFarProc);
begin
  PtrRec(Proc).Seg := PProcInst(Proc)^.DataSelector;
  GlobalFreePtr(Proc);
end;
end.
```

➤ *Listing 3*

uses a different way of setting up the data segment, by using what is effectively a re-implementation of `MakeProcInstance` and `FreeProcInstance`, defined in the ThunkU.PAS unit. Yes, I know I said before that `MakeProcInstance` and all that goes with it was history, but to put inter-task callbacks into an application rather than a DLL requires getting your hands dirty. Listing 2 shows the bits in CHANGE2U.PAS that are different from CHANGEU.PAS, and Listing 3 shows the ThunkU unit. What `NewMakeProcInstance` does is to allocate a block of memory big enough to hold the thunk and fill it up with appropriate instructions to set up the right data segment and jump past the "bad" (in the context of inter-task callbacks) smart callback instruction. `NewFreeProcInstance` will free the thunk memory. You may notice that one of `NewMakeProcInstance`'s parameters is an instance handle, as with `MakeProcInstance`; in Windows 3.x an instance handle is the data segment value. When we call the callback registration routine, we pass the address of the thunk instead of the address of the target routine, with an appropriate typecast.

The other BDE callback we will look at is the general progress callback. When performing a long job, like a batch move (that the BDE has control over: in other words on local data tables rather than remote SQL tables), the BDE calls a `cbGenProgress` callback periodically with information on how the job is getting along.

```
unit Progresu;
interface
uses
  ThunkU, SysUtils, WinTypes, WinProcs, Messages,
  Classes, Graphics, Controls, Forms, Dialogs, StdCtrls,
  Grids, DBGrids, DB, DBTables, DbiTypes, DbiProcs;
const wm_GenProgress = wm_User + 58;
type
  TForm1 = class(TForm)
    Table1: TTable;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    FillBtn: TButton;
    CopyBtn: TButton;
    Table2: TTable;
    DBGrid2: TDBGrid;
    DataSource2: TDataSource;
    EmptyBtn: TButton;
    QueryBtn: TButton;
    Query1: TQuery;
    procedure FormDestroy(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure EmptyBtnClick(Sender: TObject);
    procedure FillBtnClick(Sender: TObject);
    procedure CopyBtnClick(Sender: TObject);
    procedure QueryBtnClick(Sender: TObject);
  private  { Private declarations }
    FOldCallBack: TCallBack;
    FProgressBuf: CBProgressDesc;
    FProgressFunctionThunk: TFarProc;
  public   { Public declarations }
    procedure WMGenProgress(var Msg: TMessage);
      message wm_GenProgress;
  end;
var  Form1: TForm1;
implementation
{$R *.DFM}
{$S-} { turn stack checking off }
function ProgressFunction(ecbType: CBType;
  iClientData: Longint; var CbInfo: Pointer):
  CBRType; export;
begin
  Result := cbrUseDef;
  if ecbType = cbGenProgress then
    Result :=
      CBRType(SendMessage(Application.MainForm.Handle,
      wm_GenProgress, 0, Longint(@Form1.FProgressBuf)));
  with Form1.FOldCallBack do
    if ChainedFunc <> nil then Result :=
      pfDBICallBack(ChainedFunc)(cbGenProgress,
        Data, Buffer)
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  try
    Table1.Exclusive := True;
    Table1.Open;
    Table2.Exclusive := True;
    Table2.Open;
  except
    on EDatabaseError do
      MessageDlg('Can''t find those tables', mtError,
        [mbOk], 0);
  end;
  FProgressFunctionThunk :=
    NewMakeProcInstance(@ProgressFunction, HInstance);
  with FOldCallBack do
    DbiGetCallBack(nil, cbGenProgress, Data, BufLen,
      Buffer, @ChainedFunc);
  DbiRegisterCallBack(nil, cbGenProgress, 0,
    SizeOf(FProgressBuf), @FProgressBuf,
    pfDbiCallBack(FProgressFunctionThunk));
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  DbiRegisterCallBack(nil, cbGenProgress,
    0, 0, nil, nil);
  NewFreeProcInstance(FProgressFunctionThunk);
end;
procedure TForm1.WMGenProgress(var Msg: TMessage);
var Progress: String;
begin
  with pCBProgressDesc(Msg.LParam)^ do
    if iPercentDone <> -1 then
      Progress := IntToStr(iPercentDone)
    else
      Progress := StrPas(szMsg);
{$define INTERACTIVE}
{$ifdef INTERACTIVE}
  case MessageDlg(Progress + '. Continue?',
    mtConfirmation, [mbYes, mbNo], 0) of
    mrYes :  Msg.Result := Longint(cbrContinue);
    mrNo  :  Msg.Result := Longint(cbrAbort);
  end;
{$else}
  Caption := Progress;
{$endif}
end;
procedure TForm1.EmptyBtnClick(Sender: TObject);
begin
  Table1.EmptyTable;
  Table2.EmptyTable;
end;
procedure TForm1.FillBtnClick(Sender: TObject);
var   Loop: Longint;
const NumRecs = 5000;
begin
  Screen.Cursor := crSQLWait;
  with Table1 do begin
    DisableControls;
    for Loop := RecordCount + 1 to RecordCount + NumRecs
    do begin
      Append;
      Fields[0].AsInteger := Random(High(SmallInt));
      Fields[1].AsInteger := Random(High(SmallInt));
      Post;
      Caption := 'Adding record ' + IntToStr(Loop) +
        ' of ' + IntToStr(NumRecs);
    end;
    First;
    EnableControls;
  end;
  Screen.Cursor := crDefault;
  Caption := 'Copy the table to see the callback';
end;
procedure TForm1.CopyBtnClick(Sender: TObject);
begin
  Table2.BatchMove(Table1, batAppend);
  Table1.First;
end;
procedure TForm1.QueryBtnClick(Sender: TObject);
begin
  Query1.Close;
  Query1.Open;
end;
end.
```

➤ *Listing 4*

The PROGRESS.DPR project on the disk contains an application that shows this working in a couple of ways. For the application to run, you will need to copy the empty example tables, TABLE.DB and TABLE2.DB, into your directory \DELPHI\DEMOS\DATA (ie the alias DBDEMOS). Listing 4 shows the code for this project's unit. You'll notice that apart from a few more interface components, it is very similar to the previous example. There is an extra data structure for a `cbGenProgress` callback that needs to be given to the callback registration routine. The `CBProgressDec` data field of the form, `FProgressBuf`, will be filled with information for the callback to use.

Figure 2 shows the program running. You'll notice that there is a button to fill the first table with 5000 records of random values, a button to batch move the first table to the other and another button to empty both tables again. The idea is that periodically through the batch move, the callback gets triggered. Additionally, there is a button to execute a query and again, during the processing of this, the callback is invoked. The callback sends a message to the form to do some user-interface interaction, checking if the user wishes to carry on with the operation. If yes, the message handler returns a value understood by the BDE to mean 'carry on', if not it returns a value understood to mean 'stop this operation'. The callback itself receives this message return value as the result of the call to `SendMessage` and sets its own return value accordingly.

With this program the message parameters do not include the table object, because the progress

callback is registered with a value of `nil`, rather than a particular table handle. This registers the callback for the entire BDE session rather than one specific table. The information passed along with the message this time is the address of the progress buffer in the long parameter. The message handler can get access to the information given by the BDE by typecasting `Msg.LParam` into a `CBProgressDesc` pointer and de-referencing it. It is possible for a progress callback to get either numeric information (a percentage) or textual information (an absolute value) in the `CBProgressDesc` record. As I understand it, the former is only obtained for operations where the BDE knows how much work there is to do right from the start (it would make sense that way).

The program has conditional compilation to change what happens in the callback's associated message handler. Currently it is set up to interrogate the user when the callback fires, but removing the `$DEFINE` directive will automate what goes on, simply writing the information to the form's caption.

It is important to bear in mind that, in a similar fashion to DLL routines operating on the caller's stack, your callback will also execute on someone else's stack (usually the active task's). Some system callbacks, like the `ToolHelp` notification callbacks (which we have yet to investigate), in particular the segment freeing and module freeing notifications, are called from very sensitive places in Windows and you can end up with next to no stack to play with. On top of the small stack you may be operating on, it is important to ensure stack checking is not on whilst the callback is executing: it's not your stack so the stack check code is not appropriate.

With all the considerations for inter-task callbacks and bouncing back to the first instance where needed, it seems I have run out of room. Perhaps we'll come back to look at 32-bit BDE callbacks when Delphi 2.0 is shipping, but next time we'll look into the 16-bit Windows inter-task callbacks (there aren't any inter-task callbacks in the Win32 API, so far as I know). Experiment with care, and ThunkU for your time...

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

Copyright ©1995 Brian Long
All rights reserved.