

The **Delphi** CLINIC

Edited by Brian Long

Problems with your Delphi project?

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

Explorer Dialog Problem

QI am writing a Delphi 3 project that uses DLLs to provide custom functionality. The name of the current DLL in use is stored in a database and I use a `TOpenDialog` to let the user select which DLL to use. My problem is that when the user's machine has the Hide files of these types: option selected in Windows Explorer, DLLs do not show up in a `TOpenDialog`. Is there any way to get the dialog to ignore the Explorer setting? This may not be possible but it would save me duplicating functionality by writing my own dialog.

A Since a `TOpenDialog` actually displays Windows Explorer, to allow you to navigate around the machine, then it could be difficult to accomplish what you want. Personally, I would use a custom extension, rather than *.DLL, something like *.MOD. Delphi 3 (and later) offer a project option to specify the file extension of the compiled output file. It corresponds to the `$E` compiler directive used in the project file.

Home Grown SQL Monitor

QIs there a documented way of getting at the same information that the SQL Monitor lists? I would like to have an SQL trace window in my program, rather than having to use the SQL Monitor program, which I am not allowed to distribute with my application.

AYou make an important point in your second sentence. On the one hand it might be more desirable to have a trace

window actually resident in your application, rather than in a separate one. However, on the other hand, if you want a trace of the SQL operations in a deployed application, you have little choice but to implement this yourself since Inprise do not permit you to distribute the SQL Monitor. One way around this legal problem would be to write your own SQL Monitor. Enough information is present in the VCL to work out how to do exactly this. More on this subject later...

Way back in 1996, I showed how the 16-bit BDE could be enticed into generating trace messages for all its SQL operations in the *Tracing SQL* entry from Issue 6's *Delphi Clinic*. This described setting would make the BDE generate debug strings that could be picked up with an appropriate notification tool, or a debug terminal.

The 32-bit BDE introduced a new callback specifically designed for picking up these trace messages. A BDE callback is a routine that you ask the BDE to execute whenever certain events happen. Delphi 2 introduced a `TBDECallback` class designed, as the name suggests, to wrap up a BDE callback. The new callback is described with the constant `cbTRACE`.

I talked about 16-bit BDE callbacks in Issue 5 in *Please Call Later... Callbacks in Windows and the Borland Database Engine (Part 2)*. 32-bit BDE callbacks use much the same principles, but have changed in various ways. The best way to deal with these callbacks is to use the `TBDECallback` class and feed it with appropriate information gleaned from reading the BDE API help file (which you should find in your BDE directory as file `BDE32.HLP`). Note that all the BDE

routines, types and constants are declared in the BDE unit in Delphi 2 onwards.

This help file stresses the following about using BDE callbacks: the routine that you ask the BDE to call when the event in question occurs must not call any other BDE routine: the BDE is not re-entrant during a callback.

As it turns out, if you digest the help file information, the callback is reasonably easy to set up. It also works rather nicely in applications developed in Delphi 3 and 4. Delphi 2 causes irritation when testing the application. The callback only works if you run the application from outside the environment, due to issues housed within its component library. We'll come back to why this happens later.

When you construct a `TBDECallback`, you pass it an owner object (which isn't used) and a BDE cursor handle (if relevant, which in this case it is not). You also pass the callback type (`cbTRACE`), and a callback descriptor. The descriptor is callback type dependent, and the memory for it is managed by your application. This descriptor is passed to your callback method when the BDE triggers it. Finally, you tell the callback object if you wish to replace any installed callback of this type, or augment it (chain onto it).

The signature of the callback method (or event handler) is dictated in the help. The event type is defined as:

```
TBDECallbackEvent =  
    function(CBInfo: Pointer):  
        CBRTYPE of object;
```

Many callbacks pass meaningful return values back to tell the BDE

```

//From DBTables.pas
type
  TTraceFlag = (tfQPrepare, tfQExecute, tfError, tfStmt, tfConnect,
    tfTransact, tfBlob, tfMisc, tfVendor, tfDataIn, tfDataOut);
  TTraceFlags = set of TTraceFlag;
//From BDE.pas
const
  traceQPREPARE = $0001; { prepared query statements }
  traceQEXECUTE = $0002; { executed query statements }
  traceERROR    = $0004; { vendor errors }
  traceSTMT     = $0008; { statement ops (i.e. allocate, free) }
  traceCONNECT  = $0010; { connect / disconnect }
  traceTRANSACT = $0020; { transaction }
  traceBLOB     = $0040; { blob i/o }
  traceMISC     = $0080; { misc. }
  traceVENDOR   = $0100; { vendor calls }
  traceDATAIN   = $0200; { parameter bound data }
  traceDATAOUT  = $0400; { trace fetched data }

```

► Listing 1

how to proceed, but the trace callback return value is ignored.

The sample application on this month's disk that shows the idea is called (unimaginatively perhaps) `SQLTrace.dpr`. It has a table, datasource and data-aware grid, all prepared to show a table from the sample InterBase database (thereby generating SQL instructions). A checkbox on the form allows you to open and close the table using the Active property, and of course you can edit the table with the grid, to provoke more SQL activity.

The BDE callback object is stored in a data field defined in the form called `FSQLTraceCallback`. The callback event handler is a routine called `SQLTraceFunction` and simply wants to add whatever string comes along into a listbox. The trace string is supplied by the BDE in the callback descriptor, whose address is passed along to the

event handler. The trace callback descriptor is a record of type `TRACEDesc`, defined in the BDE unit as:

```

TRACEDesc = packed record
  { trace callback info }
  eTraceCat : TRACECat;
  uTotalMsgLen : Word;
  pszTrace :
    array [0..0] of Char;
end;

```

This contains the category of the item described, the length that the trace message should be, and the message itself. Since you are responsible for allocating memory for this descriptor, whose last field can be as long as you choose to make it by allocating sufficient memory, the message length field can tell you if the string was truncated. The BDE help file recommends you allocate enough space so the last text field of the record has `DBIMAXTRACELEN (8,192)` characters to play with.

The program's call descriptor is stored in a data field called `FTraceBuffer`, and `GetMem` is used to allocate space for it, before giving it to the `TBDECallback`. Just before this happens, the Session object has its `TraceFlags` properties set. This property is a set property and can be used to specify which operations will generate trace messages, much like the trace options in the SQL Monitor.

The program duplicates the SQL Monitor checkboxes, each of which has a `Tag` property set with the corresponding constant defined in the BDE (as passed in the `eTraceCat` field of the `TRACEDesc` record). `GetTraceFlags` is a handy routine that loops through these checkboxes, which are sitting in a group box, and tots up the `Tag` property values of those which are checked. This gives the value that the BDE wants, which is a simple number made out of bit mask values. However, as was mentioned, the VCL uses a set property instead.

The values in the enumerated type used by the set have been laid out such that their representative values when used in a set directly match the corresponding BDE constants. Small sets are implemented using sequential bits in a word-sized storage area. In other words, the first enumerated type value matches the constant with a

► Listing 2

```

TTraceForm = class(TForm)
...
private
  FTraceBuffer: PTraceDesc;
  FSQLTraceCallback: TBDECallback;
  function GetTraceFlags: TTraceFlags;
  function SQLTraceFunction(CBInfo: Pointer): CBRType;
end;
...
procedure TTraceForm.FormCreate(Sender: TObject);
begin
  //Give listbox a horizontal scroll bar
  SendMessage(1stTrace.Handle, lb_SetHorizontalExtent,
    2000, 0);
  //Set session trace flags
  Session.TraceFlags := GetTraceFlags;
  //Initialise BDE before trying to install callback
  Session.Open;
  //Allocate callback descriptor
  GetMem(FTraceBuffer, SizeOf(TRACEDesc) + DBIMAXTRACELEN);
  //Install BDE callback
  FSQLTraceCallback := TBDECallback.Create(nil, nil,
    cbTRACE, FTraceBuffer, SizeOf(TRACEDesc) +
    DBIMAXTRACELEN, SQLTraceFunction, True);
end;
procedure TTraceForm.FormDestroy(Sender: TObject);
begin
  //Uninstall BDE callback
  FSQLTraceCallback.Free;
  FSQLTraceCallback := nil;

  //Deallocate descriptor
  FreeMem(FTraceBuffer);
  FTraceBuffer := nil;
end;
procedure TTraceForm.chkTableOpenClick(Sender: TObject);
begin
  Table1.Active := chkTableOpen.Checked;
end;
function TTraceForm.GetTraceFlags: TTraceFlags;
var
  I, TraceValue: Integer;
begin
  TraceValue := 0;
  //Get Tag values of checked checkboxes
  for I := 0 to TraceCategories.ControlCount - 1 do
    if TraceCategories.Controls[I] is TCheckBox then
      if TCheckBox(TraceCategories.Controls[I]).Checked then
        Inc(TraceValue, TraceCategories.Controls[I].Tag);
  //Turn number into set
  Result := TTraceFlags(Word(TraceValue));
end;
function TTraceForm.SQLTraceFunction(CBInfo: Pointer):
  CBRType;
begin
  //Set a result to avoid warning, even though it is ignored
  Result := cbrUSEDEF;
  1stTrace.Items.Add(StrPas(PTraceDesc(CBInfo).pszTrace));
end;

```

value of 1 (or 2^0). The second one matches the constant whose value is 2 (or 2^1). The third matches the one valued 4 (or 2^2), and so on (see Listing 1).

Each time a checkbox is checked, the `GetTraceFlags` routine is called again to update the BDE session's `TraceFlags` property.

Listing 2 shows all the other code that has been described so far, which doesn't really leave anything else in the program to explore. Figure 1 shows a screenshot of the application running, tracing an application's manipulation of an InterBase table.

In addition to this application's custom BDE callback, the VCL installs several callbacks of its own. Any `TSession` object installs a login callback, a server call callback (to set the SQL cursor when connecting to a server), a cached update callback, and an SQL trace callback of its own (which is worth further investigation). If the session is in a DLL, a dedicated detachment notification callback is also installed.

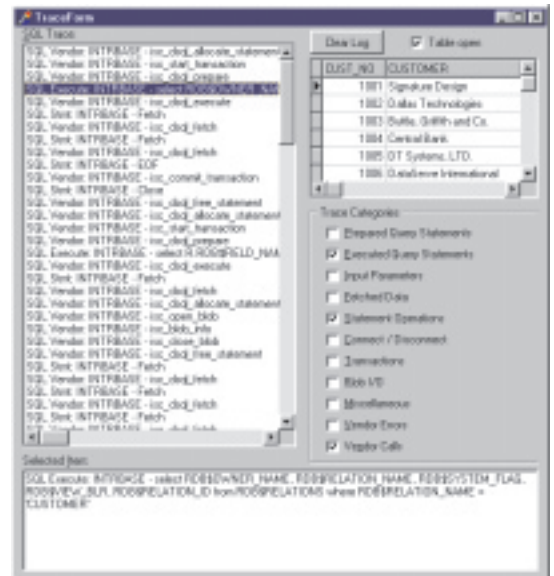
Going back to the session's own SQL trace callback, you might wonder what it does. Well, it is what makes the SQL Monitor work. The SQL Monitor cannot plug in to your application's BDE session, so your program locates the SQL Monitor (if running) and sends it all the messages to display.

The SQL Monitor uses a small DLL (`SMClient.Dll`) that is accessed in one way or another by Delphi applications to get the information to display. Delphi 2 relies upon the fact that this DLL sets up a memory

mapped file called `SMClientLib` containing the fully qualified path to itself. A Delphi 2 app reads the DLL name, loads it into memory, locates the `RegisterClient` routine exported by the DLL and calls it. This tells the DLL that a new application is registered and also gives the SQL Monitor a routine to call when the user changes the trace option checkboxes. The routine simply resets the session's `TraceFlags` property to match what the user asks for. When `RegisterClient` returns, the application is given the address of a routine in the SQL Monitor to call with trace messages.

One minor issue worth noting is that you will not be able to trace messages with a custom callback written in a Delphi 2 application, if Delphi 2 is running. The component library appears to open up `SMClient.Dll` and keep it open. This allows the program to locate the memory mapped file, locate the DLL path, reload the DLL and get a procedure address for passing trace messages to. However, when the program tries to call the routine, it will fail since the DLL was not loaded by the SQL Monitor itself. The failure causes the VCL to set the session's `TraceFlags` property to an empty set.

Delphi 3 introduced Delphi COM support and `SMClient.Dll` was rewritten as an in-process COM server. A COM object in the DLL implements the `ISMClient` interface defined in the `SMIntf` unit (present in the VCL source directory). The unit also defines the COM object's class ID to enable Delphi

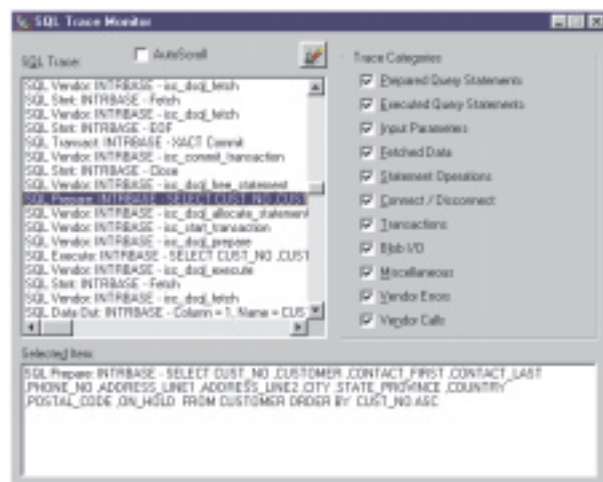


➤ Figure 1: A BDE callback allows us to trace SQL operations.

applications to get access to its interface. This interface defines both the `RegisterClient` routine and also the `AddStatement` method, which is called to pass trace messages to the SQL Monitor. A Delphi 3 (and upwards) application checks that the COM server DLL has set up a memory mapped file called `SMBuffer` (although it doesn't check the contents), then connects to the COM object contained within.

With all this information, you should be able to see that it is now more than possible to write a standalone SQL trace application, in exactly the same way as the SQL Monitor works. All you need is a main application that allows the user to set trace options, and can list trace messages, and a COM server DLL that implements the `ISMClient` interface.

To prove the point (more to myself than anyone else really), I spent a while developing just such an application. The design goals were to provide functionality similar to Borland's original SQL Monitor, but it should be able to pick up trace messages from any 32-bit BDE application. In other words, it needs to provide an appropriate COM object for Delphi 3 and later, as well as standard exported routines for Delphi 2. In addition, it should have no



➤ Figure 2: You can write a new SQL Monitor without writing a BDE callback.

limitations on its distribution, unlike the original that it is based upon.

The SQL trace application is called `Monitor.dpr` and the support DLL is called `MonLib.dpr`. The DLL is explicitly loaded by `Monitor.Exe` when it starts up, and unloaded when the program closes down. The DLL, for its part, sets up the two previously mentioned memory-mapped files. `SMClientLib` has the path to the DLL placed in it, and `SMBuffer` is used to store a trace message in, before telling the application to add it to its listbox. The application and the DLL

communicate with each other via Windows messages.

In order for the DLL and the monitor program to communicate, the DLL has a non-visible form in it to receive messages. The DLL broadcasts a message to announce its presence when it is loaded by any application (including the monitor) passing along this form's window handle. The monitor picks up this message and sends a message back to the specified window handle indicating its own main form handle.

Whenever a client application's BDE signals a trace message is

available through its callback, the DLL copies the message into the `SMBuffer` memory mapped file and sends a message to the monitor to prod it into adding it to its message list. When the user changes the state of any of the trace option checkboxes, the monitor broadcasts a dedicated message around (there are potentially many DLLs that need to know) and each DLL calls the signal procedure in each of its clients. This signal routine was passed in when the client registered itself.

The DLL exports the required `RegisterClient` routine, and if it is

Windows Functionality

When Windows 95/98 displays a dialog offering you the chance to restart Windows (for example, after changing some Control Panel settings, or when choosing Start menu, Shut Down...), holding down Shift whilst pressing the Yes button will usually cause Windows to close and restart without the usual reboot.

To access the Properties dialog of any item being displayed in Windows Explorer (which includes items sitting on your Windows desktop), as well as the usual right-clicking and choosing Properties, you can hold down Alt and either press Enter or double-click.

When you drag an item from one folder to another in Windows Explorer, depending upon the type of item and where you are dragging from or to, you tend to find that the item may be copied, moved, or a shortcut to it might be created. If you want to dictate the end result of your drag, bear in mind that Ctrl+drag will copy, Shift+drag will move, whilst Shift+Ctrl+drag will create a shortcut.

On a similar vein to the above point, if you drag an item with the right mouse button instead of the left mouse button, you get a popup menu of options that allows you to choose which operation will occur. The one that *would have* occurred (with a left drag) will be shown in bold (the default menu item).

Right clicking on a file in Windows Explorer gives a menu, typically starting with Open. This menu item will open the file in the default way, typically using the application associated with that type of file. If you hold Shift down when you right click, you should get an extra Open With... item that allows you to choose which application will open the file. To give a simple example of why you might want to use this, you could open a Delphi unit in Notepad, instead of waiting for the default associated application (ie Delphi) to start up and open it.

On a related theme to the above item, when you right click on a file displayed by Windows Explorer, there is a Send To menu. In this menu, there are a number of variably useful menu items. It is easy to add new entries to this menu, any application can be added such that it will be passed the file name as a command-line parameter. For example, you may wish to add Notepad in there, so you can easily load any text file (regardless of file extension) into Notepad. Locate the Send To folder, which can be found under your Windows folder, and make a shortcut to your target application (NotePad.exe in this case). If you want to rename it, then do

so. Now when you right click on a file and choose Send To, your target program will now be listed and ready for use. Note that on Windows 95, Send To is directly under Windows. On Windows 98 and NT, you will need to explore a bit further. For example, on NT, Send To and Desktop are all replicated for each user under the Profiles subdirectory.

A variant on the previous item's theme is to make Notepad be automatically on the context menu for any file with or without a default file association. This means that a non-standard file can be sent to Notepad without even going to Send To.

Launch RegEdit.Exe and navigate to HKEY_CLASSES_ROOT\Unknown. Add a new key called Shell under Unknown. Add another key under Shell called Notepad, and another key under Notepad called Command. In the Command key, there will be a string item called Default. Double-click it and give it a value of NOTEPAD %1. A .REG file with the following contents would have the same effect if you double-clicked it in Windows Explorer:

```
REGEDIT4
[HKEY_CLASSES_ROOT\Unknown\shell\Notepad\
  Command]
  "Notepad %1"
```

If, like me, you are not blessed with the ability to touch type, you probably get irritated when you accidentally turn Caps Lock on, without noticing. Of course it is not too much of a problem if you are typing in Delphi's editor at the time, as you can use Ctrl+0, Ctrl+U to toggle the case of a selected block. Leaving out this specific case, there is more help at hand from 32-bit Windows. In Control Panel, choose Add/Remove Programs and use the Windows Setup tab to ensure Accessibility Options are installed. Now choose Accessibility Options from Control Panel. On the Keyboard tab, make sure Use ToggleKeys is checked. Now, whenever Caps Lock, Num Lock or Scroll Lock is toggled on or off, your PC will make a high or low-pitched beep respectively.

I hope you find these tips useful: Windows can usually do much more than we ever get around to finding out!

Windows Key Plus...	Function	Equivalent Way To Access Functionality
E	Windows Explorer	Start Programs, Windows Explorer
F	Find Files or Folders	Start Find, Files or Folders...
Ctrl+F	Find Computer	Start Find, Computer...
M	Minimise All	Right-click on task bar, Minimize All Windows
Shift+M	Undo Minimise All	Right-click on task bar, Undo Minimize All
R	Run dialog box	Start Run...
F1	Windows Help	Start Help
Tab	Cycle through taskbar buttons. Press Enter to select the application you want	None, but similar to Alt+Tab
Break	System Properties dialog box	Right-click on My Computer, Properties, or use Control Panel

► Table 1

compiled in Delphi 3 or 4 also includes units needed to define and implement the COM object. This implements the interface as specified in the `SMIntf` unit, and duplicates the class ID declared in that unit. The implementation of the COM object maps down to calls to the `RegisterClient` routine and the routine that is given to Delphi 2 clients to call when a new trace message needs to be added to the list.

I'll leave any more ambitious examination of the code to yourselves, and will just mention that if the DLL project is compiled in Delphi 2, it will service Delphi 2 clients. If compiled in Delphi 3 (or later), will service Delphi 3 (or later) clients.

Do remember that you will need to register the COM server DLL (in


Delphi 3 or later). This can be done from the Run menu, or using the `TRegSvr` command-line tool supplied with Delphi. Figure 2 shows the application running with trace messages coming in from another application.

Windows Miscellany

Q This is not a Delphi question, but I thought you might know the answer anyway. What is the point of that irritating Windows key on my keyboard? All it seems to do for me is to take focus away from the application I am typing into, and bring up the Start menu when I accidentally press it instead of pressing Alt or Ctrl.

A This question has arisen on a number of the training courses that I have conducted and so I feel it is worth looking at

briefly. After all, Windows users really should have the benefit of knowing what is on offer such that they can then make a balanced judgement of whether to use or to ignore the available facilities.

As you have already seen, pressing and releasing the Windows key (the one with the  picture on it) on its own will pop up the Start menu. However it can also be used in conjunction with other keys as well. You can hold down the Windows key, press another key, then release that key and finally release the Windows key (in other words it is used in a similar way to Ctrl and Alt, which probably explains why it is placed where it is on the keyboard). Table 1 shows the keys that it works with on Windows 95, Windows 98 and Windows NT 4 and above. You can see that it generally provides a shorthand mechanism to navigating through the Start menu, or manipulating the task bar.

I cannot confirm this next point, not having the relevant software available to me, but I have read that if Microsoft IntelliType is installed, the Windows key also works in conjunction with the keys shown in Table 2.

Before leaving the subject of lesser known Windows functionality, see the sidebar included on page 62 which contains some of my favourite Windows usage tips.

► Table 2

Windows Key Plus...	Function
L	Log off Windows
P	Print Manager
C	Control Panel
V	Clipboard Viewer (if installed)
K	Keyboard Properties dialog box
I	Mouse Properties dialog box
A	Accessibility Options (if installed)
Space	Displays the list of IntelliType Hotkeys
S	Toggles CAPS LOCK on and off