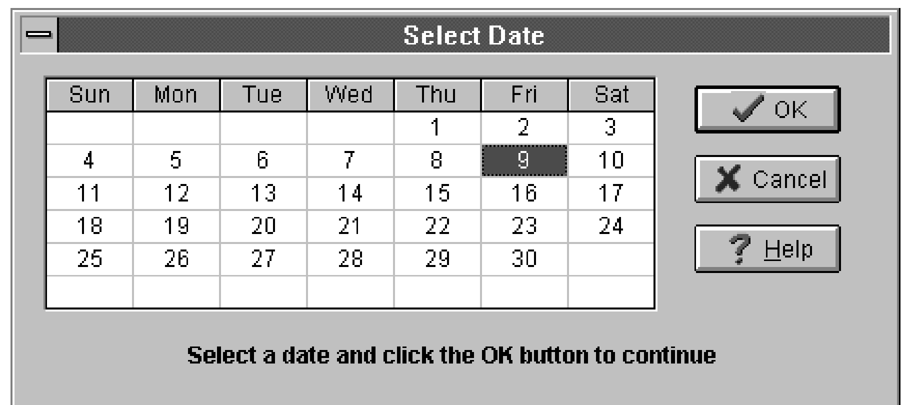
Delphi Internals:
Through The Language Barrier

by Dave Jewell

This month, Dave concludes his look at how to create DLLs using Delphi and, in the process, shows us how to invoke Delphi forms from other applications.



➤ *Here's our fresh-faced Delphi dialog box. Little does it know that it's about to be transmogrified into a DLL and called from Pascal, C and even – horrors – Visual Basic!*

In Part 1 of this two-part look at Delphi DLL programming (back in Issue 1), we examined the basics of DLL programming using Object Pascal. We saw how to build a DLL and how to export routines from the DLL so they could be called from an application written in C, C++, Pascal or even Visual Basic.

That was OK was far as it went, but this month we're going to look at the more exciting stuff: how to actually package up a Delphi user interface into a DLL so that it can be called from another application.

This is very relevant to developers who already have a major investment in another language. For example, I recently had an enquiry from a scientific user who has a large application written in FORTRAN. He wants to keep his existing back-end number crunching code and port the user interface side of his application across to Delphi. A DLL implementation is ideal for this sort of approach.

The Date Selection Dialog

Let's get straight down to brass tacks then, and put together a simple DLL. For the purposes of this discussion, we'll build a DLL which implements the dialog box shown in Figure 1.

Although this is the only dialog box we're going to be discussing, it needs to be said that if you're using the DLL approach it's a good idea to package many dialog boxes into the same DLL. Any VCL dialog pulls in a minimum of 100–150Kb of VCL code and it makes no sense to replicate this overhead across several DLLs. Don't adopt a one-dialog-per-DLL approach unless you have very good reasons for doing so!

As you can see from the screen shot, our sample dialog box implements a simple date selection mechanism. A `TCalendar` component (taken from the Samples page of the Delphi component palette) is used in conjunction with three `BitBtn` controls. The user can select a date and click the `OK` button or else double-click directly on the date, which will automatically dismiss the dialog. Clicking `Cancel` will likewise dismiss the dialog with no selection being made and the `Help` button does what you'd expect.

In a real world application, you'd most likely need something more sophisticated, but these are issues that we're not going to concern ourselves with here – our mission is to package the dialog into a DLL and show how to access it.

Planning Your DLL Interface

Examining the Select Date dialog box, we can see that we need to provide the following information as input to the DLL:

- ➢ A Caption String for the form itself (so we can display application-specific strings such as "Select Next Appointment" etc);
- ➢ The Name of a Windows Help file and help context number for when the `Help` key is clicked. This information should not be hard-wired into the DLL. A DLL must be as general-purpose as possible.

As for as getting information out of the DLL, we need to know:

- ➢ If the user selected a date (clicking `OK` or double- clicking a date) or clicked `Cancel`;
- ➢ If a date was selected, we need to know what it was; after all, that's the whole raison d'être of this dialog box!

If we were implementing this dialog as a reusable Delphi component, we'd just create properties that correspond to the various inputs and outputs of the dialog box. For example, we might have a property called `HelpContext` which stores the help context that's invoked when the `Help` button is clicked.

However, we can't use that approach here – one of the most crucial things to bear in mind about a DLL interface is that it's strictly procedural. We can't make any object-oriented calls from the outside world to the DLL. You might wonder why this is so. The reason is that object-oriented

languages (both C++ and Object Pascal) always have rather more going on behind the scenes than meets the eye! For example, most VCL components (including forms themselves) have a `Hide` method. When you call this method, it appears as though it has no parameters:

```
MyForm.Hide;  { Hide our form }
```

However, every method of an object has an implicit parameter, `Self`, which is a pointer to the instance data of the object being referenced. Thus, the `Hide` method actually takes a hidden object pointer as its single parameter. That pointer, which references a dynamically allocated VCL object, is completely meaningless to a C++ application. For just the same reason, calling a C++ method from Delphi code is a decidedly non-trivial exercise (even C++ compilers from different vendors aren't compatible in this respect!). It's for this reason that we must stick to a strictly procedural interface.

In the end, I came up with the following three interface routines which constitute our DLL interface to the outside world:

```
procedure DateSelSetCaption(
  Caption: PChar); export;
procedure DateSelSetHelpInfo(
  HelpFile: PChar;
  HelpContext: LongInt);
  export;
function DateSelDialog(
  var TheDate: Integer): Bool;
  export;
```

The first, `DateSelSetCaption`, is used to set an optional, custom caption string for the form. If this routine is never called, then the form's caption remains set to "Select Date", which is how I set the `Caption` property in Delphi's form designer. The second routine, `DateSelSetHelpInfo`, is used to set up the name of the Windows help file and a help context number. These values are used when the user clicks the `Help` button. If this routine is never called, then the Help button is greyed out and unavailable.

The third routine, `DateSelDialog`, is the real "business end" of the DLL. It's responsible for invoking the form and getting user input. When the caller returns from this routine, the form will already have been closed. A return value of `True` indicates that a selection was made and `False` indicates that the user clicked `Cancel`. When `DateSelDialog` is called, an integer variable is passed as a `var` parameter to the routine and its value will be set to the selected day of the month.

A couple of things to note. Firstly, all three DLL interface routines use the special `export` specifier. I discussed the use of this keyword last time. Suffice it to say that you must be sure to use the `export` keyword when declaring any routines that are going to be exported from the DLL. The second thing to note is the use of the `PChar` variable type when passing strings into (and potentially out of) the DLL. It should go without saying that when creating an interface between the Delphi DLL and the host program, you must only use data types which are compatible with both programming languages. Thus, you can't pass Pascal strings through a DLL interface to a C or C++ program. Similarly, you can't use Pascal sets as part of the DLL interface. Sets are a foreign concept to C/C++ compilers!

Before we move on, here's one further point. I've implemented our DLL interface using three separate routines, but you could pass all this information using just one routine with lots of parameters. However, I'd caution you against this: it's unwieldy and error prone. My own philosophy is to provide a simple interface which implements essential functionality (`DateSelDialog`, in this case) and then provide other routines to extend that functionality where required. The infamous `GetPrivateProfileString` Windows API routine is a good example to dwell upon! As ever, the KISS adage (Keep It Simple, Stupid!) applies...

Inside The DLL Source

The source to the form unit is shown in Listing 1. As you can see, the beginning of the unit is quite conventional with the class definition of `TDateSelector` as produced by the Delphi form designer. This is immediately followed by procedure declarations for the three exported routines that we just examined.

Four constants are defined in the `implementation` part of the unit. These constants, (used to store the caption, help information and selected date), are used by the code that follows. Bear in mind that you'd normally store this sort of form-specific information as properties of the form but this can't be done here because we need to access these variables from the procedural (non-OOP, non-VCL aware!) exported routines.

The `FormCreate` handler is very straightforward. It looks to see if a custom caption has been set up and, if so, sets the form's caption to the new custom caption string. It also checks to see if there's a valid help file name and help context. If there isn't, then the `Help` button is disabled. As an alternative, you could arrange for the `Help` button to disappear altogether by calling its `Hide` method. This would probably be a neater approach.

The `OKButtonClick` handler is equally straightforward. I've set this up as a shared event handler so that it's called by both the `OK` and `Cancel` buttons. It's also used as the `OnDblClick` handler of the `TCalendar` component, so that a double-click on a date closes the dialog. Alternatively, you could implement the button-clicking functionality using the `ModalResult` mechanism. If anything other than the `Cancel` button is clicked, the `iDate` variable is set to the currently selected date in the `Calendar` control.

The next event handler, `HelpButtonClick`, merely calls the Windows Help engine with the help information that's been obtained from the host application via a call to `DateSelSetHelpInfo`. It should be obvious that this handler will only get called if the help button is enabled which in turn means that there must have been a call to `DateSelSetHelpInfo`, so we don't bother to include another check here. The help file name which is

```
unit Dateform;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, Grids, Calendar,
  StdCtrls, Buttons;
type
  TDateSelector = class(TForm)
    Calendar1: TCalendar;
    OKButton: TBitBtn;
    CancelButton: TBitBtn;
    HelpButton: TBitBtn;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure OKButtonClick(Sender: TObject);
    procedure HelpButtonClick(Sender: TObject);
  private    { Private declarations }
  public     { Public declarations }
  end;

procedure DateSelSetCaption(Caption: PChar); export;
procedure DateSelSetHelpInfo(HelpFile: PChar;
  HelpContext: LongInt); export;
function DateSelDialog (var TheDate: Integer):
  Bool; export;

var DateSelector: TDateSelector;

implementation
{$R *.DFM}
const
  CustomCaption: String = '';
  HelpFileName: PChar = Nil;
  dwHelpContext: LongInt = -1;
  iDate: Integer = 0;

procedure TDateSelector.FormCreate(Sender: TObject);
begin
  { Set custom caption if one has been supplied }
  if CustomCaption  '' then Caption := CustomCaption;
  { If Help stuff not set up disable Help button }
  if (HelpFileName = Nil) or (dwHelpContext = -1) then
    HelpButton.Enabled := False;
end;

procedure TDateSelector.OKButtonClick(Sender: TObject);
begin
  { If this wasn't a Cancel, set the iDate global }
  if Sender  CancelButton then iDate := Calendar1.Day;
  Close;
end;

procedure TDateSelector.HelpButtonClick(Sender: TObject);
begin
  { Call the Help Engine with the info we've been given }
  WinHelp (Handle, HelpFileName, Help_Context,
    dwHelpContext);
end;

procedure DateSelSetCaption (Caption: PChar);
begin
  { StrPas doesn't check for Nil strings, so watch it! }
  if Caption  Nil then CustomCaption := StrPas (Caption);
end;

procedure DateSelSetHelpInfo (HelpFile: PChar;
  HelpContext: LongInt);
begin
  { If we've previously been called, clear old string }
  if HelpFileName  Nil then StrDispose (HelpFileName);
  { Now set up the new stuff }
  HelpFileName := StrNew (HelpFile);
  dwHelpContext := HelpContext;
end;

function DateSelDialog (var TheDate: Integer): Bool;
begin
  iDate := 0;     { no date selected yet }
  { Create the form instance }
  DateSelector := TDateSelector.Create (Application);
  try
    DateSelector.ShowModal;  { Display the dialog }
  finally
    DateSelector.Free;  { Destroy the form instance }
    Result := iDate  0;
    if Result then TheDate := iDate;
  end;
end;
end.
```

➤ *Listing 1*

passed to the DLL can be a fully qualified pathname if you wish.

The three exported interface routines come next in the listing. `DateSelSetCaption` simply uses the `StrPas` routine to copy the zero-terminated caption string into a Pascal-style string. It's a good idea to check for a `NIL` argument here since `StrPas` doesn't bother and a GPF will result if the host application passes a `NIL` value. The `DateSelSetHelpInfo` code uses `StrNew` to store the passed help file name. This allows us to store an arbitrarily long pathname, but it is important to dispose of any prior allocated file name in case we get called more than once.

Finally, the `DateSelDialog` routine is where the rubber meets the road! The routine first initialises `iDate` (again, this is in case we get called more than once) and then creates an instance of the `TDateSelector` form. The `ShowModal` method is called to actually display and process the form's user interactions and the `finally` clause ensures that the dialog is cleaned up no matter what befalls. The function result is set to `True` or `False` according to whether or not a valid date was set up and if so its value is passed back to the caller as a `var` parameter.

The code in Listing 2 is the corresponding Delphi Project (.DPR) file. Delphi has no built-in expert for creating DLLs *[But now you have, of course, courtesy of Bob Swart's article on page 35! Editor]*. There are four simple steps in the process as outlined in the Borland documentation:

- Change the initial keyword `program` to `library`.
- Remove `Forms` from the `uses` clause of the file.
- Remove everything between the `begin` and `end` statements.
- Add an `exports` entry for each exported function.

As you can see in the listing, I have added `exports` entries for the three routines that constitute our DLL interface. I've also given them ordinal numbers, although this is optional if you're happy to import by name (see Part 1 of this article for a fuller discussion of this).

Making It All Happen

The proof of a DLL, of course, is in the calling! At this point, I had a 226Kb file, DATESEL.DLL, and I needed some way of testing it. To do this, I knocked up a couple of small host applications, one in Pascal and one in C. These are shown in Listings 3 and 4. It's worth pointing out that the Pascal host application is a prodigious 1536 bytes in size: a definite case of the tail wagging the dog!

The results of running these two small host applications are shown in Figures 2 and 3. As you can see, it all works as advertised. I didn't bother knocking up a Visual Basic application to prove the point, but I'm confident that it would also work equally well. There are a number of improvements that suggest

themselves for our sample dialog. Perhaps the most necessary improvement (and certainly the biggest shortcoming of the code presented here) is the thorny issue of multiple users. As soon as you package up code into a DLL, this is an issue that you really need to think about. At the moment, our sample dialog wouldn't work correctly if used simultaneously by more than one application since those `const` variables in the form unit are effectively global variables. If you need to create a dialog that's going to be used by multiple callers, I'd suggest that you package up all this *state variable* information into a dynamically allocated block of memory. A handle to this memory block is then passed around between the host application and the DLL and allows the DLL to retrieve and set all state information pertaining to the current call. This isn't difficult to implement and since my brief was to explain how to package up Delphi forms into DLLs rather than covering DLL programming in general, I feel absolutely no guilt in leaving this as an exercise for you!

Dave Jewell is a freelance consultant and programmer, specialising in systems-level work under Windows and DOS. You can contact Dave on the internet as djewell@cix.compulink.co.uk

```
library Datesel;
uses Dateform in 'DATEFORM.PAS' {DateSelector};
{$R *.RES}
exports
  DateSelSetCaption index 1,
  DateSelSetHelpInfo index 2,
  DateSelDialog index 3;
begin
end.
```

Listing 2

```
program DateProg;
uses WinProcs, WinTypes;
procedure DateSelSetCaption (Caption: PChar);
  far; external 'DATESEL' index 1;
procedure DateSelSetHelpInfo (HelpFile: PChar; HelpContext: LongInt);
 far; external 'DATESEL' index 2;
function DateSelDialog (var TheDate: Integer): Bool;
  far; external 'DATESEL' index 3;
var  i: Integer;
     buff: array [0..127] of Char;
begin
  DateSelSetCaption ('Calling Delphi from Vanilla Pascal');
  lstrcpy (buff, 'You made no choice');
  if DateSelDialog (i) then wvsprintf (buff, 'You chose %d !', i);
  MessageBox (0, buff, 'Pascal Caller', mb_ok);
end.
```

Listing 3

```
#include  <windows.h>
#ifdef __cplusplus
extern "C" {
#endif
void WINAPI DateSelSetCaption (LPCSTR lpCaption);
void WINAPI DateSelSetHelpInfo (LPCSTR HelpFile, DWORD HelpContext);
BOOL WINAPI DateSelDialog (LPINT TheDate);
#ifdef  __cplusplus
}
#endif
int i;
char buff [128];
int PASCAL WinMain (HINSTANCE hInst, HINSTANCE hPrev,
               LPSTR lpCmdLine, int nCmdShow)
{
    DateSelSetCaption ("Calling Delphi from Vanilla C");
    lstrcpy (buff, "You made no choice");
    if (DateSelDialog (&i)) wvsprintf (buff, "You chose %d !", &i);
    MessageBox (0, buff, "C Language Caller", MB_OK);
    return (0);
}
```

Listing 4

Figure 2 Calling the DLL from Borland Pascal 7; you'll notice the help button is greyed: I didn't bother to set up help information

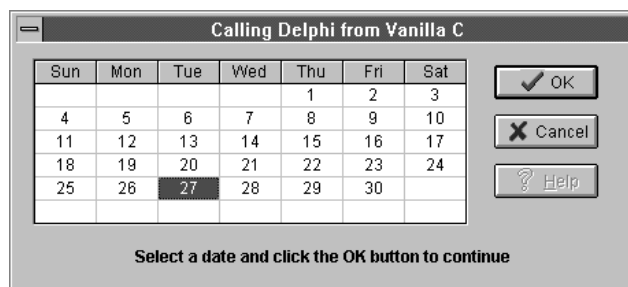


Figure 3 And here's the same dialog again, being called from a C application