Delphi Internals: Undocumented Secrets, Part 1

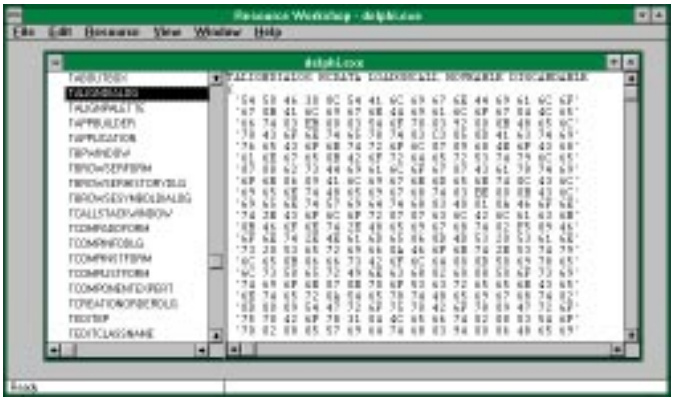
by Dave Jewell

OK, I lied. In the previous *Delphi Internals* column I discussed DLLs in some detail. I faithfully promised that this month I'd write about creating Delphi DLLs which could be used by applications written in other languages. Well, I got so carried away writing about the stuff for the column for this issue that I completely forgot what I did previously. Sorry about that. I'll take up the DLL story again next time round. I'll put it down to premature senility. Can't you just hear those arteries hardening?

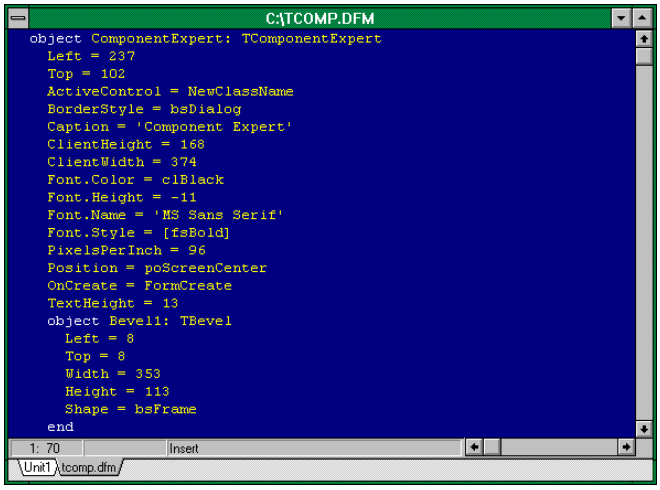
Delphi On Delphi: A Walk On The Inside

As most Delphi developers are aware, the Delphi system was actually developed using... Delphi! If anyone ever suggests to you that a Delphi program is likely to slow down to a crawl when you've got more than a dozen forms in your application, then just show them Delphi itself! Delphi actually contains seventy or eighty different forms and runs just as speedily as any other Windows program.

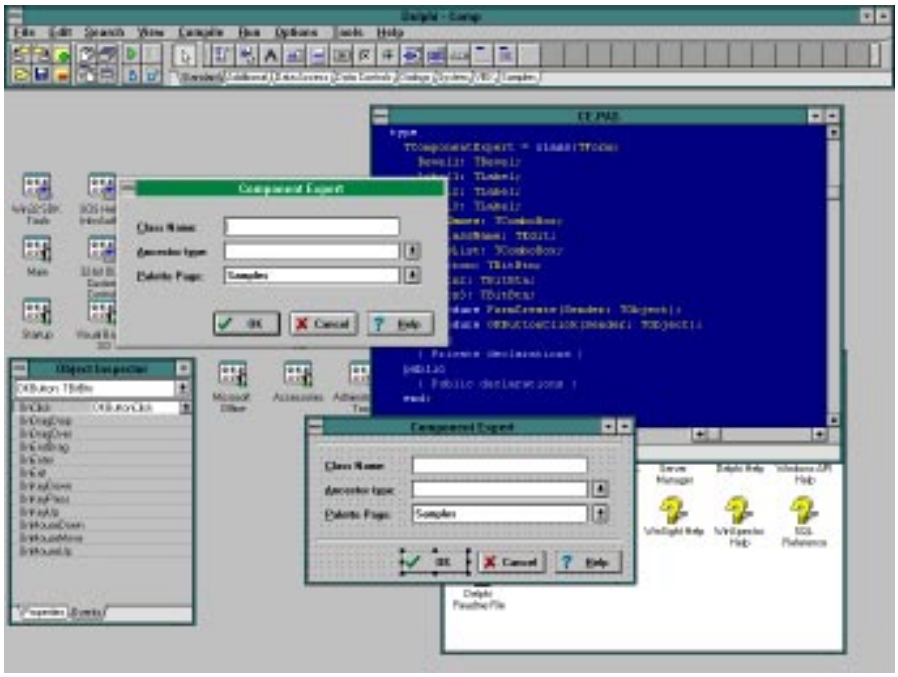
Seeing, as they say, is believing, and I thought it would be interesting to have a peek inside a Delphi application to see how the application's forms actually get stored. If you're a seasoned Windows developer, you might expect each form to be held as a `DLG` resource



➤ *Figure 1 Resource Workshop showing all the form resources inside Delphi, identifiable from the resource names.*



➤ *Figure 2 Here's the Component Expert form loaded up in a Code Editor window. Of course, you don't get the source code as well, but what do you want out of life...?*



➤ *Figure 3 Is this a Component Expert window I see before me? With a little additional massaging, you can see the Component Expert form in all its glory. Obviously the same principles can be applied to any form in any compiled Delphi executable.*

inside the application. However, you need to bear in mind that not all forms are dialogs – a Delphi form can be either a dialog or a regular application window. DLG resources are therefore out.

Delphi actually stores each form as an RCDATA resource, where the name of the resource corresponds directly to the Delphi form type. For example, a form of type TComponentExpert would be stored as an RCDATA resource with the name TCOMPONENTEXPERT. You can see this in Figure 1. The TComponentExpert form, of course, corresponds to Delphi's Component Expert accessible from the File menu.

Let's say that we wanted to extract the Component Expert form from Delphi so as to take a closer look at it. Using Borland's excellent Resource Workshop (Oh, *why* didn't Borland include the Resource Workshop with Delphi? *[They've now made it available to Delphi users in the new 'RAD Pack' add on. Editor]*) or a similar resource editor, we can copy the appropriate resource, create a new .RES file and paste the form into this resource file.

What might really surprise you here is that you've just created yourself a form file! That's right: a Delphi form file (extension .DFM) is actually nothing more or less than a plain vanilla .RES file with a different file extension. If you rename your new resource file to TCOMP.DFM (for example) and then load it up under Delphi, you'll see the result shown in Figure 2. Pretty neat, huh?

Note: If you don't know how to do this, just click Open File *from the* File *menu and then select* DFM files *from the* File Type *combo box. You'll then be able to load any form file into a new Code Editor window.*

Using this technique, it should, in principle, be possible to extract any form resource from an application, load it up as a text file, make some alterations and then plug the resource back into the executable file. I can't off-hand see much benefit in doing this since (naturally) you can't get access to the code associated with the form.

On a more serious note, the above discussion should highlight the fact that Delphi has an extremely open architecture. I don't doubt that as soon as someone has figured out the exact internal format of a Delphi form resource, (not a very difficult proposition from what I've seen of it – maybe I'll cover this in a future *Delphi Internals* column), we'll see one or more resource editors specifically designed to work with Delphi applications, allowing you to re-arrange a form, change captions and component properties even within the compiled and linked executable file.

As pointed out earlier, there are definite restrictions on how far you can go down this route. For example, changing the Sorted property of a listbox from True to False might break the application if the code made assumptions about the order in which items might appear in the listbox.

Personally, I feel that there are two legitimate reasons for wishing to poke around with a compiled application's form resources. Firstly, I've recently seen a utility which can scan an executable program's dialog boxes looking for user interface errors. By this I mean such misdemeanours as spelling mistakes in dialog box items, missing or duplicated Alt-key assignments, duplicated control IDs and so forth. A utility like this would be equally useful in the Delphi marketplace but would require carnal knowledge of the form resource layout.

A second reason is far more pragmatic and seems to have been overlooked so far by both Borland and the developer community. I refer, of course, to the thorny issue of program internationalisation. With conventional C/C++ or Pascal development, this is all very easy. If you've been a good boy or girl and stored all your program's string constants in string resources, then you can easily modify those resources with Resource Workshop (or similar) to create a foreign-language version of your product. In the same way, all dialogs (stored as DLG resources) are eminently modifiable with a resource editor. This is clearly very advantageous to companies wishing to send a copy of their software to a foreign distributor for conversion for the foreign marketplace. All conversion can be done without losing control of the product's source code. But what about Delphi forms? Oh dear! Yes – I think you'll find that a Delphi form editor will appear on the market quite soon...

Joining The Union

In the *Moving Up* column in this issue I looked briefly at the 'free union' capability built into the Pascal programming language. Let's now look at this same facility in rather more detail.

If you're familiar with C and C++, you'll know that these languages have both struct and union keywords. The struct keyword corresponds to Delphi Pascal's record facility, but there is no direct equivalent to the C/C++ union.

This is rather a shame, since unions are very useful things to have. In essence, a union allows you to overlay one or more different variables at the same location in memory. They allow you to treat a data structure as if it were of more than one different type. As an example, the SDK documentation tells you that Windows bitmap files (.BMP files) can start with one of two different file headers. By reading the file header information into a union type, you can make a quick determination of which file header type is involved and then access it using the appropriate part of the union.

It turns out that Pascal does have this capability, but it's very well hidden. For example, try experimenting with the type declaration shown in Listing 1 (next page).

Technically speaking, this type of record is called a discriminated union and has a tag variable (called Sel) which discriminates between the different parts of the union. Here, Sel is a Boolean value which means that we're restricted to two alternative data layouts within the union. However, you could equally

```
type
  Union = record case Sel: Boolean of
    False : (AsInteger: Integer);
    True  : (AsBytes: array [0..1] of Byte);
  end;
```

➤ *Listing 1*

```
type
  PShape = ^Shape;
  Shape = record case ShapeType: Integer of
    Rect   : (UpperLeft, LowerRight: TPoint);
    Circle : (Centre: TPoint; Radius: Integer);
    Line   : (StartPt, EndPt: TPoint);
    ...etc...
```

➤ *Listing 2*

```
procedure DrawShape(s: PShape);
begin
  case s^.ShapeType of
    Rect:   { code to draw rectangles }
    Circle: { code to draw circles }
    Line:   { code to draw lines }
    ... etc ...
  end;
```

➤ *Listing 3*

```
Union = record case Boolean of
  False : (AsInteger: Integer);
  True  : (AsBytes: array [0..1] of Byte);
end;
```

➤ *Listing 4*

well use an Integer (as I've done elsewhere in the *Moving Up* column) which will obviously give you a lot more!

Where might you use this sort of data structure? Well, imagine that you're creating a drawing program and you want to represent each different object with the same `Shape` data structure. In other words, circles, rectangles, ellipses, straight lines and so on are all encapsulated by the `Shape` data type. You might choose to implement the `Shape` data type something like the example in Listing 2.

In this example, the `Rect`, `Circle` and `Line` symbols are assumed to be constants which you've defined previously. Equally, you could use an enumerated type for your tag variable (the `ShapeType` field) and just use the different values of the enumerated type as selectors in the union. Provided that the tag variable is a scalar (eg not a floating point!) then the compiler will be happy. With the above data structure, you might implement the `DrawShape` procedure of our hypothetical drawing program like the example in Listing 3.

At this point, any OOP devotees out there will no doubt be yelling 'polymorphism'! Certainly, the effect is very similar.

Discriminated unions aren't the only types of unions that Pascal allows. We can also implement a non-discriminated or 'free' union by removing the tag variable entirely. In this case, our initial type definition would look something like that in Listing 4. Because the tag field no longer exists, the size of the structure is reduced: it's now equal to the size of whichever variant part happens to be the largest.

A more modern innovation (as far as Borland's particular flavour of Pascal is concerned) is the `absolute` keyword. Using `absolute`, you can tell the compiler to locate two variables at an identical storage space, or even place a variable at a specific segment and offset (this latter facility is more relevant to real-mode DOS programming than Delphi). For instance, here's how we would achieve a similar effect to the above `Union` declaration using the `absolute` clause:

```
var
  AsInteger: Integer;
  AsBytes: array [0..1] of
    Byte absolute AsInteger;
```

Sadly, there are restrictions on where the `absolute` specifier is located. It would be nice, for example, if you could put it inside a type definition like this:

```
Union = record
  AsInteger: Integer;
  AsBytes: array [0..1] of
    Byte absolute AsInteger;
end;
```

If you try this, you'll get a syntax error: it's not allowed. You can see, then, that the free union trick is most relevant to type declarations, whereas the `absolute` clause is used for variable declarations. The techniques are complementary.

Incidentally, if you find this sort of thing interesting, then I'd recommend you track down a copy of Peter Grogono's *Programming in Pascal*. (Addison Wesley, 1979, ISBN 0-201-02473-X). This book is now out of print, but it's well worth trying to find a second-hand copy. You obviously won't find any Borland extensions mentioned, but it's the best Pascal reference I've ever seen.

Dave Jewell is a freelance consultant and Windows developer. He is the author of *Instant Delphi* published by Wrox Press. You can contact Dave on the internet as djewell@cix.compulink.co.uk