Delphi Internals: Moving Up To 32-Bits

by Dave Jewell

As you'll have seen from the last issue's cover, the 32-bit version of Delphi is almost with us. This month's *Delphi Internals* column is devoted to a discussion of 32-bit issues. In particular, we'll be concentrating on how you can smooth the transition to 32-bits. Interestingly, if you want to start programming in 32-bits, it's possible to start ***now***. If you can't wait to get started, we'll be looking at some clever public domain code which lets you do just that.

There are a number of different ways in which you can smooth the transition from 32-bits. Bear in mind that Delphi32 is still in the beta test stage as I write this, so it's possible some things may change in the final release!

Using The Right Data Types

One of the most important changes you'll find in Delphi32 is support for Unicode. Today, there is a heavy emphasis on building support for foreign languages into your application. Unfortunately, when you look at all the world's languages together, there are far too many symbols to be represented with the 256 available "slots" in the ASCII character table. In fact some languages (Chinese being a notable example), have hundreds, possibly thousands, of special symbols that are specific to that language alone.

To cater for situations like this, Windows NT supports Unicode, a new 16-bit character format that encompasses many different languages and character sets. If the high-order byte of this 'wide character' is zero, the low-order byte contains an ANSI character.

There are now three different char types which are available:

- `ANSIChar` This is always one byte, just like the `Char` type in previous versions of Delphi.
- `Char` The same as `ANSIChar`. At some point in the future, `Char` might be equated to `WideChar`.
- `WideChar` Always 16-bits wide for full Unicode support.

If `Char` becomes equated to `WideChar` at some point, it could cause you a lot of grief if you've made assumptions in the past about `Char` being one byte long. Let's face it, we've probably all done this. Now would be a good time to re-check your code! Any time that you need to know the size of a character, be sure to use `Sizeof(Char)` rather than just assuming a value of 1.

The same caveats apply to the `Integer` type. This is a 16-bit quantity under Delphi 1.0x and 32-bits under Delphi32. The same is true of the `Cardinal` type which is an unsigned 16-bit number in Delphi 1.0x and an unsigned 32-bit number in Delphi32. If you need to use an integer value that's always 16-bits wide, then use either `SmallInt` or `Word` according to whether it needs to be signed or unsigned, respectively.

Just as characters can potentially be 32-bits wide, Pascal strings can now potentially consist of characters that are 32-bits wide. To accommodate this, a number of new string types have been introduced. These are summarised in Figure 1.

Type	Meaning
`AnsiString`	A dynamically allocated string of variable length, also called a "long string". Each element is one byte long.
`ShortString`	A non-dynamically allocated string with a maximum length of 255 characters. Exactly equivalent to `String` in Delphi 1.0x. Each character element is one byte long.
`String`	Either a `ShortString` or an `AnsiString`, depending on the value of the `$H` compiler directive.
`WideString`	A long string, with each element being of type `WideChar`.

➤ *Figure 1 New string types*

In the 32-bit version of Delphi, a string can now potentially be of indefinite length. Such strings (so-called "long strings") are dynamically allocated on the heap. When you alter the length of such a string, Delphi will (behind the scenes) re-allocate the memory being used. Like `PChars`, long strings are null terminated, so you can use them wherever you might pass a `PChar` to a Windows API call (for example). Because it would be inefficient to repeatedly perform memory allocation each time you add a character to the end of a long string, Delphi32 provides a new routine called `SetLength` which allocates memory for the string.

There is also a new compiler directive, `{H+}` or `{H-}` which determines whether the keyword `String` is interpreted as a `ShortString` or an `AnsiString` respectively. The latter interpretation is the default and all the components in Delphi32 use long strings (but not, currently, `WideStrings`).

No Low Level Assumptions

I've seen it suggested that a program which contains 16-bit

inline assembler code won't work properly under the new 32-bit development system. This may or may not be the case – it really depends on exactly what the 16-bit assembler code gets up to. It is perfectly legal (from the processor's point of view) to manipulate 16-bit registers while running in 'flat memory mode'. However, if your assembler code makes too many assumptions about the sort of environment its running in, then it will fall over.

One possible assumption concerns stack layout. When doing tricky things in assembler, it's tempting to modify parameters 'in-situ' on the stack. This simply won't work under Delphi32. Not only are all the parameters stretched to 32-bits (thus changing the stack layout anyway), but more importantly, Delphi32 uses a completely different calling convention to its 16-bit cousin. The latter always pushes every parameter on the stack. However, as a performance optimisation, Delphi32 will by default use the `EAX`, `EDX` and `ECX` registers to pass parameters to called routines. This is a technique which high-performance C/C++ compilers such as Watcom have used for some time. Passing parameters in registers speeds up performance by eliminating stack operations which are relatively slow. In cases where more than three parameters are passed, the extra parameters go onto the stack in the usual way.

As a general rule, it's probably a good idea to avoid inline assembler if at all possible. If you must use it, then try to confine it to one specific unit so as to minimise later porting problems. Also, bear in mind that the 32-bit version of Delphi will create executables that run on Windows NT – an operating system which is inherently portable to other non-Intel processor architectures. Therefore, to maximise the potential audience for your product, try to make as few assumptions as possible.

VCL Commitment

You're probably getting fed up of hearing me say this, but the single most important thing you can do to improve the portability of your code is to stick to the VCL library wherever possible. I have been able to re-compile a number of my Delphi programs for 32 bits – they all re-compiled and worked flawlessly, which left my head several sizes larger than it was to start with! I even re-compiled a 16-bit Delphi component of my design and, once again, it worked first time. The key to maximising portability is to use VCL calls rather than calling the Windows API directly. This is particularly true of Windows messages where the message layout (the arrangement of the `wParam` and `lParam` fields) often differs widely between the 16-bit and 32-versions). Use VCL if at all possible!

And If You Simply Can't Wait...

This is all very well, but maybe you can't wait to start making use of all those sexy new Windows 95 API functions. You know, the ones that are only available to 32-bit applications. Fortunately, there is a solution and, even better, it won't cost you a penny! An enterprising individual by the name of Christian Ghisler has written a set of routines (packaged as a Delphi unit) which allow a 16-bit program to directly call 32-bit routines in a DLL. Since the Windows API is itself DLL-based, this means you can call the entire 32-bit API and any other goodies such as (for example) the new 32-bit Common Controls DLL.

Christian's routines are entirely public domain which means that you can use them without restriction in your own applications. His work, in turn, is based upon the work of a chap called Peter Golde, who originally wrote a set of public domain routines in the form of a DLL so that 16-bit Visual Basic programs could make calls down to the 32-bit API. The routines, together with complete source code and a sample program, are contained in a file called CALL32NT.ZIP on the free disk included with this issue.

So how does it work? The `CALL32NT` code (which, despite the name, works quite happily with both NT and Windows 95) makes use of the Generic Thunking mechanism built into the 16-bit Windows subsystem that's present in both NT and Windows 95. It's not essential to know how it works at the nuts and bolts level, but we do need to know how to use it.

To call a 32-bit function in your 16-bit application, you need to take the following steps.

Firstly, add the unit name `CALL32NT` to the `Uses` clause of the unit you're working with.

Next, you need to declare all the 32-bit functions you're going to use. This is how you might declare the 32-bit `PolyBezier` routine:

```
var
  PolyBezier : function(
    hdc : longint;
    var points : tagPoint;
    count,id : Longint) :
    Longint;
```

Notice that this is a function variable. In other words, `PolyBezier` is effectively a pointer to a function of the specified type. Additionally, any parameters in the 32-bit routine must be followed by a final `LongInt` variable (which in this case is called `id`). This step is very important. You'll also notice that our `PolyBezier` routine has exactly the same name as the target API routine. In this case, there's no conflict because the `PolyBezier` routine doesn't exist in the 16-bit `WinProcs` unit. In cases where there *is* a conflict, (as is the case with most of the API), if you use the same name as an existing 16-bit routine, then Pascal's scoping rules will 'hide' the original 16-bit definition. If you want to be able to call both the 16-bit and 32-bit versions of, say, `GetDC`, then you'll have to invent a new name such as `GetDC32`. In this case, your routine declaration might look like this:

```
var
  GetDC32 : function(wnd,id :
    longint) : longint;
```

As before, don't forget the extra `id` parameter and notice that, in this case, the routine expects a window

handle. Since window handles are 32-bits wide in Win32, we've used a LongInt as a place holder for the handle. In Win32 all handles become 32-bits long and anything that was an integer in the 16-bit API likewise turns into a LongInt.

In the initialisation part of your unit, all these function variables must be pointed at a single, common routine called Call32. This routine is exported by the CALL32NT unit. You'd simply set up the above two function variables like this:

```
PolyBezier := @Call32;
GetDC32 := @Call32;
```

This step is very important – for obvious reasons! If you call through a function variable that hasn't been initialised, then your program will crash and burn in a spectacular manner.

The next job is to declare a uniquely named LongInt identifier that's associated with each required API routine. This is where those extra id parameters come into play. For the above two cases, we might define them like this:

```
var
  id_PolyBezier : LongInt;
  id_GetDC32 : LongInt;
```

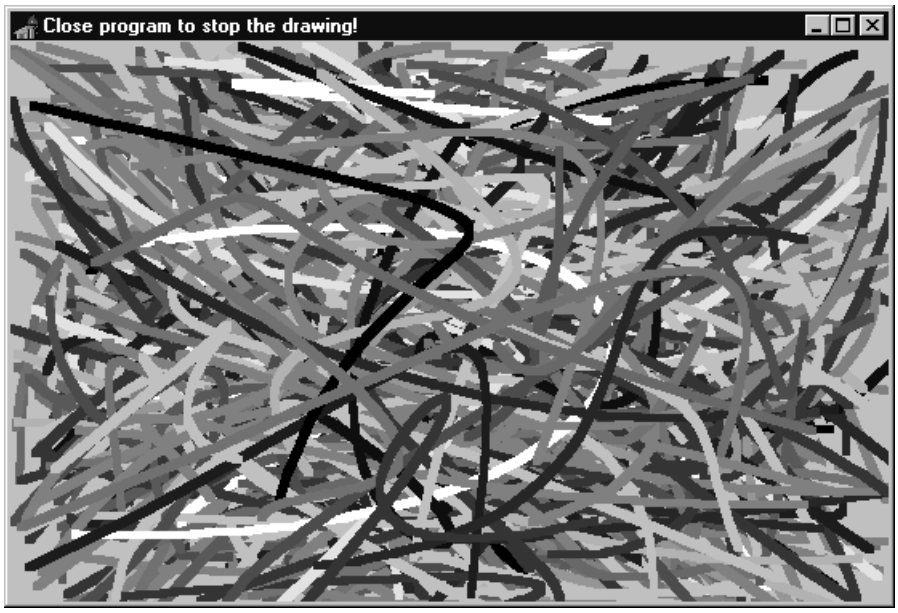
The last step (you'll be relieved to hear!) is to initialise these variables by calling the Declare32 routine. Again, this is another function exported by the CALL32NT unit. For the two examples here, we'd do something like this:

```
id_PolyBezier :=
  Declare32('PolyBezier,
  'gdi32', 'ipi');
id_GetDC32 :=
   Declare32('GetDC',
   'user32', 'w');
```

This code goes in the initialisation part of your unit along with the other initialisation previously described. The first parameter to the Declare32 routine is the name of the 32-bit API routine you're targeting. The Declare32 routine will expect to find this routine in the specified DLL (named as the second parameter) so you must take care to get this right. Christian reckons that the first parameter is case sensitive so be sure to get this right as well. The third parameter specifies the number and type of parameters expected by the routine. This information is used by the CALL32NT routines to push the appropriate parameters on the stack. For the PolyBezier routine, we've got a 32-bit integer (or handle), followed by a pointer, followed by another 32-bit integer.

The 'w' specifier (used in the second call, above) is rather special. Normally, when you pass a window handle to a Win32 routine, it's expected that you'll pass a 32-bit window handle. However, as an added convenience, the CALL32NT routine will allow you to pass 16-bit window handles. These window handles are then converted to the equivalent 32-bit window handle inside the unit. It's quite interesting to see how this is done: first a 16-bit SetCapture is used to set the window which receives the capture, and then a 32-bit GetCapture is used to read back the equivalent 32-bit handle. OK, so it's a kludge but it's a cunning one and it works!

All the foregoing may sound complex, but it's actually only a few minutes work to set up and it's a great way to access 32-bit API routines from within 16-bit Delphi.



➤ *Figure 2 The proof of the pudding...*

If you want to start using this technique now, I'd suggest that you use {$IFDEF} constructs to organise your code in such a way that you can easily remove the CALL32NT stuff once you get your hands on the 'real' 32-bit compiler.

As proof of the pudding, take a look at Figure 2. This is a 16-bit Delphi application running under Windows 95 and calling the PolyBezier routine: something that a 16-bit program can't normally do.

The complete source code of the Delphi program which calls the PolyBezier routine is on the disk, as file TESTW32A.PAS. You'll find that this code differs somewhat from that in the CALL32NT.ZIP file (in file TESTW32.PAS). Christian's code performed all its drawing inside the Form1.FormCreate method. I thought this was not really the proper Delphi way of doing things so I rewrote the code, using a Timer to initiate bezier drawing at regular intervals.

Dave Jewell is a freelance consultant, programmer and technical journalist specialising in systems level DOS and Windows work. He is the author of Instant Delphi published by Wrox Press. You can contact Dave on the internet as djewell@cix.compulink.co.uk