Moving Up: Visual Basic

by Dave Jewell

Key issues for Visual Basic developers moving up to Delphi

The first thing that needs to be said is this: is you've had much experience using Visual Basic, then it's good news all the way!

You've got a terrific head start on understanding Delphi's component based, RAD, philosophy. A 'component' is just Delphi-speak for what Visual Basic developers habitually refer to as a 'control', and RAD is the new buzz-word for Rapid Application Development: the process of building an application by choosing components from a palette, arranging them on a form and writing the necessary code to 'glue' the whole thing together. Both Delphi and Visual Basic are component-based RAD systems.

A Better Development Environment

VB developers will instantly understand the purpose behind Delphi's forms, Component Palette and Object Inspector. They'll also soon realise how much better thought out the Delphi development environment is. Visual Basic hasn't really evolved over the last few years and its showing its age in all sorts of ways.

For example, the Component Palette gives you instant access to a lot of components without clutter. The same is true of the Object Inspector which allows you to have nested, hierarchical properties such as the Font property. If you try double-clicking on the small '+' sign immediately to the left of the Font property name, you'll find that Delphi will open up another layer of sub-properties. This approach can be taken to an arbitrary number of levels: if you click the Style sub-property, you'll see a set of sub-sub-properties relating to the style of the wanted font! One of the key characteristics of a good user interface is that it should provide maximum flexibility with minimum clutter.

Regarding the form window, Delphi contains all sorts of facilities for aligning and sizing components with respect to one another, something that's long been missing from VB. Of course, you can always buy Visual Basic add-ons such as ToolThings, and so forth, but the cost soon mounts up; it's great to have all this functionality where it ought to be: in the development environment.

Another particularly nice feature of Delphi is the number of 'container' components that are included with the product. The `NoteBook`, `Panel` and `GroupBox` are all examples of 'containers' which can hold one or more other components. This allows you to almost create mini-forms within a form, easily moving different groups of controls around to get the effect you want.

Compilers Versus Interpreters

Most Visual Basic developers appreciate the fact that VB programs use an interpreted system – that's why you need to have the VBRUN300.DLL file somewhere on your hard disk, which contains the interpreter and the VB run-time library.

Every time you make a Visual Basic call such as `STR$`, `MID$`, etc, what gets executed is a small, fast, machine-code routine deep inside VBRUN300.DLL. Of course, a huge number of small, fast routines adds up to a lot of code: the VB 3.0 version of this file is almost 400Kb in size. In fairness, it has to be said that Delphi programs aren't exactly petite either – but at least with Delphi there are various approaches available to reducing the size of your executable file. With Visual Basic, you don't have the option, the VBRUN300.DLL file includes the code for every single possible Visual Basic call, whether or not your application uses those calls.

A more significant difference concerns the way in which interpreters and compilers work. Microsoft's Visual Basic interpreter uses a very cunning technique called 'threading' which allows it to tokenise your VB application as you're typing it in.

If you're not familiar with the idea of tokenisation, just think of it as a technique whereby the interpreter converts your program into a more efficient, internal representation. The reserved word `FOR`, for example, is represented by just a single byte, meaning that VB can run your program as fast as possible, rather than laboriously examining each character of source code that you've typed in.

This is the reason why Visual Basic instantly complains if you type something invalid: the parsing and tokenisation is happening all the time, behind the scenes, as you add code to your application.

The Delphi approach is quite different. It uses a true compiler which converts your program directly into machine code. There is no intermediate, tokenised form. This has advantages and disadvantages. By compiling to machine code, the program runs far faster than it ever could under an interpreted system. Secondly, there's no need for an interpreter – you don't need to carry the Delphi equivalent of VBRUN300.DLL around with your applications. On the negative side, compilation is more complex – and therefore slower – than tokenisation. This means that you don't get the 'instant take off' response of Visual Basic, but Delphi's Pascal compiler isn't exactly a slouch, it's reckoned to be just about the fastest in the industry and it gives you virtually interpreter-like response with none of the disadvantages of an interpreted language.

If you are used to VB, you might find it strange that you can type complete garbage into a code window without Delphi complaining! Delphi will only syntax check your code when you actually invoke the compiler. Having said that, you'll notice that Delphi uses 'syntax highlighting', just like VB. As you type in a code window, language constructs such as comments, identifiers and reserved words are recognised and highlighted using the colour scheme you've chosen.

A Language With a Difference

Without question, learning Object Pascal is the most challenging aspect of the move to Delphi.

When I first learnt BASIC, the `GOSUB` statement had only just arrived on the scene and the idea of passing parameters to subroutines was a long way in the future. Your average BASIC program listing consisted of miles of spaghetti `GOTO` statements, interspersed with arcane two-letter variable names which were almost entirely meaningless – and no I'm not talking exclusively about my own code here! Fortunately, those days are far behind us In recent years, BASIC dialects such as Visual Basic have become far more 'structured' and Visual Basic is far closer to Pascal and C/C++ than it's ever been before.

Ironically, recently this trend has reversed somewhat: Visual Basic's new 'variant' variables are certainly convenient to use, but they're anathema to a strongly-typed language like Pascal.

With Pascal you must declare all variables 'up front' before you use them and you have to specify the type of each variable. You can't use something as if it were an integer one minute and a character the next. The Pascal compiler rigidly checks every operation on a variable to see if it makes sense for that variable's type. It would be nonsense, for example, to use a floating point variable as an index into an array.

Delphi does away with the notion of assigning certain types to variables depending on the final character of the variable name. With Visual Basic, a '$' character at the end of a variable name always implies a string variable. With Delphi, everything depends on the variable declaration. Table 1 lists some common Visual Basic variable names and their Delphi equivalents.

If you really miss Visual Basic's variant records, here's a neat little trick you can use in Delphi. It exploits a peculiarity of the Pascal syntax called the 'free union'. I first learned this trick many years ago when programming in UCSD Pascal on Apple][computers, and it still works with Delphi Pascal today !

To create a free union, create a new type definition like the one in Listing 1. You can now declare your own variables of type `Variant` and access them using the associated field names. For example, try this:

```
var
  v: Variant;
begin
  v.IsBool  := True;
  if v.IsInteger = 1 then
    { always beeps! }
    MessageBeep (0);
  {...}
```

You can add your own types to the `Variant` type definition so that (for example) you could assign to a floating point `Double` and then use an array of bytes to examine the contents of the `Double` on a byte by byte basis. This mechanism obviously won't let you perform automatic type conversion (which is at the heart of the VB Variant mechanism) but nevertheless, it's a useful way of side-stepping Pascal's strong type-checking when you want to.

Stringing You Along...

Just to make things a little more complicated, Delphi actually uses two different types of string variables. There's the 'classical Pascal' `String` type and there's the newer `PChar` type.

The reason for having two is historical. The `String` type, which has always been part of the Pascal language definition, is implemented as a single length byte followed by that number of characters so that, for example, the string 'Fred' would be implemented as a length byte of 4, followed by the actual string data. Variables of this type are quick to manipulate (since you know immediately how long the string is) but they're limited to a maximum of

➤ *Table 1*
Common Visual Basic variable names and Delphi equivalents

Variable Type	Visual Basic	Delphi
Integer	Count%	Count : Integer;
Long Integer	BigCount&	BigCount : LongInt;
Single-Precision Floating Point	SmallNum!	SmallNum : Single;
Double-Precision Floating Point	BigNum#	BigNum : Double;
String	Str$	Str : String;

➤ *Listing 1*

```
type
 Variant = record case Integer of
   0: (IsChar: Char);       { now I'm a signed character ! }
   1: (IsByte: Byte);       { now I'm an unsigned byte ! }
   2: (IsInteger: Integer); { now I'm an integer! }
   3: (IsWord: Word);       { now I'm an unsigned word }
   4: (IsBool: Boolean);    { now I'm a Boolean! }
 end;
```

255 characters in length for obvious reasons.

The PChar type corresponds to a 'C' style string and is the type used when calling any Windows API routine that expects a string. It consists simply of a pointer to the characters in the string, which are followed by a terminating zero byte (Hex 00). Such strings have the advantage that they can be very long, but string operations are less efficient. Concatenating one PChar string to another, for example, involves scanning both strings for their corresponding zero bytes before the actual string data is copied.

Previous versions of Borland's Pascal-based Windows development system used PChar extensively and included routines for converting from String to PChar and vice-versa. However, with the advent of Delphi, Borland have de-emphasised PChar variables:

> *"Delphi's VCL library uses String types almost exclusively and you're only likely to need to convert to a PChar when you want to 'hit the metal' and call a Windows API routine directly."*

Porting your Existing Code

Because of the similarities between Delphi and Visual Basic regarding forms, event handling and property names, it should be possible, in principle, to develop an automated porting tool designed to copy the bulk of an existing VB application over to Delphi.

An enterprising American company, Earth Trek Inc, have developed an application called Delphi Conversion Assistant which does just that. At the time of writing, this product is still in beta test, but it looks like it will greatly simplify the business of moving up to Delphi. The program doesn't understand the format of binary .FRM files (Microsoft have never released details of this format), so forms must be saved as text files before beginning the conversion process.

For more details on Delphi Conversion Assistant, you can reach Earth Trek by email at 72321.742@compuserve.com. *[We plan to carry a review of Conversion Assistant when it's ready. Editor]*

Finally, here's another little tip for those Visual Basic programmers who make heavy use of VB's DoEvents call. I've heard several ex-VB developers bemoaning the fact that there seems to be no equivalent of the DoEvents routine under Delphi. It turns out that Delphi does have an equivalent call, but it's fairly well hidden! What you need to do is call Application.ProcessMessages instead. While you're at it, take a look at the other properties and methods of the Application object: there's a lot of good stuff hidden away in there.

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