Back To Basics: A Reintroduction To Properties, Part 2

by Dave Jewell

In last month's property primer, we looked at the fundamentals of properties, with a particular emphasis on how they provide a sort of syntactic shorthand, allow you to do work behind the scenes whenever a property is read or written to and, most importantly, provide an elegant mechanism to decouple the interface of an object from its internal implementation. This month, we're going to delve deeper, covering issues such as Object Inspector compatibility, read-only components, the innards of RTTI, and default properties. We'll also be taking a peek at how properties are implemented in the .NET camp.

Properties Meet The Object Inspector

In order for a property to appear in the Object Inspector, it needs to be declared in the `published` section of the containing class. When the compiler sees that a property is published, it emits a special chunk of data called RTTI, or Run-Time Type Information. This RTTI fully describes the property: it includes such information as the type and name of the property, the maximum and minimum values allowed for the property (as in a scalar type, for example), whether the property has a default value, and what that value might be, etc. The technically curious and anoraks amongst us (those like me, who prefer to see the bits and bytes!) might want to take a look at the boxout entitled *RTTI, Under The Hood* to see how this stuff works.

If you've been getting to grips with Microsoft's .NET initiative, you will have seen all sorts of new buzzwords bandied around such as 'assembly', 'metadata' and so forth. From a Delphi perspective, there's absolutely nothing to be afraid of here: the language may have changed (pun strictly intentional) but the concepts certainly haven't. In the .NET scheme of things, metadata corresponds directly to Borland's RTTI information, although it's more than this, as we shall see. An assembly, on the other hand, is a specialised DLL which can contain one or more classes and/or components. Yes, you've guessed it, an assembly is equivalent to a Delphi package! I'm not trying to downplay the significance of .NET here, or pour scorn on the innovative features of the product. I think .NET is going to be extremely important in the next few years, and it's great to know that Borland are hard at work creating .NET-compatible versions of Delphi and C++Builder. But, on the other hand, I think that Delphi developers are entitled to award themselves a certain self-satisfied smirk when they see dyed-in-the-wool Microsoft developers excitedly rushing around saying how wonderful it is that component details no longer need to be stored in the registry, metadata is stored with the executable, etc, etc. Been there, done that, got the T-shirt...

One of the few irritating aspects of Delphi's property programming model is the way in which the Object Inspector refuses to display read-only properties. You can make a property read-only simply by omitting the `write` clause from the property declaration, so:

```
property Version: Integer
  read fVersion;
```

A version number is, of course, an excellent candidate for a read-only property, and it's perfectly reasonable to want to display a component's version number, status, vendor details, etc, at design-time. However, if the Object Inspector sees that a property's `SetProc` field is set to zero (see *RTTI, Under The Hood*), then it will stubbornly refuse to play ball.

Fortunately, it's easy to get around this, and here's how:

```
property Version: Integer
  read fVersion
  write fDummyInteger;
```

Since the Object Inspector wants to see a `write` clause, we'll supply one! Simply point the `write` clause at a private 'dummy' field of the appropriate type, and that's it. Whenever an attempt is made to write to this property, all that happens is that the value of the dummy field gets changed. If you want to implement several read-only properties all of the same type (eg three read-only integer properties) you can reduce memory requirements by pointing the `write` clause of all three properties at the same dummy field. If you want to implement one or more read-only `String` properties, for example, then you'll need to add a dummy `String` field, and so on.

The Object Inspector isn't just picky about read-only properties. It's picky period. In addition to the aforementioned fix for read-only properties, the Object Inspector also refuses to display array properties. If you look at this month's *Beating The System* column, you will see that I have implemented a Delphi component, `TDesktopManager`, which allows an application to programmatically manipulate the Windows desktop. There are four `published` properties in this component, and three array properties that are not published, but merely `public`. There are two reasons why the array properties aren't published. Primarily (as mentioned in the article), I wanted to distinguish these properties from the others because they aren't accessible unless the `Active` property is first set. However, even

if the three array properties *were* published, they still wouldn't show up in the Object Inspector. The format of the RTTI information generated by the Delphi compiler is adequate for simple types, but it's not adequate to describe more complex types such as array properties. Aside from user interface issues, this is the biggest reason why array properties aren't supported by the Object Inspector. If you implement a class that publishes properties which the compiler doesn't regard as being 'Object Inspector compatible' then it silently demotes those property declarations to public. In other words, no RTTI is generated.

Default Properties

Having told you that Object Inspector doesn't support array properties, there is one rather neat feature of Object Pascal that relates to array properties. Last month, I mentioned the `default` keyword, and described how it can be used to specify the default value of a property. This same keyword can also be used in a somewhat different context to indicate that a specific array property is to be treated as the default property of the class.

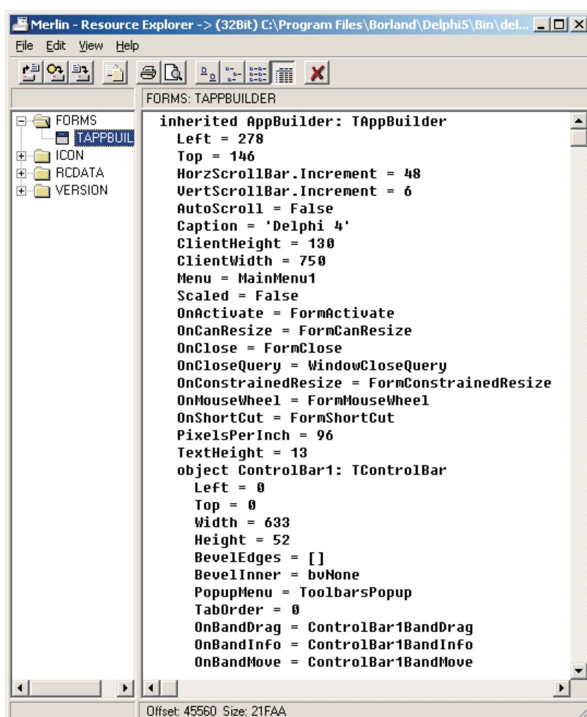
Let's look at an example to make this clearer. Once again, I'll refer to the `TDesktopManager` component in this month's *Beating the System*. Here you'll find a property declaration that looks like this:

```
property Caption[
  Index: Integer]: String
  read GetCaption
  write SetCaption;
```

Suppose I decided that `Caption` was the most important property of this class, and that 99% of the time, this is the property that other developers would wish to access. If this were the case, then it would make sense to make `Caption` the default property of the class:

```
property Caption[
  Index: Integer]: String
  read GetCaption
  write SetCaption;
  default;
```

You might think that the compiler would confuse this with the 'default value' usage of the same keyword. However, there are at least three reasons why this can't happen! From a syntactic point of view, the 'default property' usage of the keyword follows the property declaration proper; that is, after a semicolon, as I have shown above. More importantly, only an array property can be specified as being the default property, and only non-array properties can be given a default value, so you will see that the argument against syntactic ambiguity is actually pretty watertight!



➤ *Figure 1: Property persistence, though totally transparent most of the time, is crucial to the operation of Delphi. A .DFM-sniffing tool such as Merlin (as is shown here) illustrates how the non-default state of numerous properties is stored in the .DFM stream.*

OK, but what does it do? Without the `default` clause, we would have to refer to `Caption` like this:

```
MyString :=
  Desktop.Caption[Idx];
Desktop.Caption[Idx] :=
  MyString;
```

If `Caption` is the default property, we can simplify these assignments to:

```
MyString := Desktop[Idx];
Desktop[Idx] := MyString;
```

In other words, the compiler spots the square brackets, realises that an array operation is being used, and automatically assumes the default property, `Caption`. Default properties are used by Borland to great effect in the `TList` and `TStringList` classes: you will realise this now, even if you didn't before!

Another big issue is property storage, also known as persistence. The Delphi development system is so successful at implementing property persistence that novice developers are hardly aware that it's happening, the process is completely transparent to the average programmer. Consider a simple design-time form: when you change the colour or position of the form and then close the project, you'd naturally expect that, next time you open the project, the form has the colour and position that it had when you saved it!

The properties of a form are, of course stored in the associated .DFM file. Any components that reside on the form will also have their properties stored in the same file, and this can extend to several levels where container components such as group boxes, panels and toolbars are concerned. Because Delphi must work transparently with third-party components, there must be some mechanism for establishing which properties get streamed out to the .DFM file when the state of a component is 'persisted'. This is where the `stored` keyword comes into play.

RTTI, Under The Hood

For those who like to delve a little deeper, here's a simple example of how RTTI is implemented. Let's suppose you define a `boolean` property like this:

```
property Enabled: Boolean
  read fEnabled write SetEnabled default True;
```

Assuming that this declaration appears in the published part of a class, the compiler will obligingly spit out a chunk of RTTI which looks as below. This example is taken from a real-live Delphi component, but obviously the address shown (left-hand column) is arbitrary. Where an object has more than one published property, an array of these data structures effectively exists in memory:

```
00406F5C      dd        offset Boolean        ; PropType
00406F60      dd        0FF000018h            ; GetProc
00406F64      dd        offset SetEnabled     ; SetProc
00406F68      dd        1                     ; StoredProc
00406F6C      dd        80000000h             ; Index
00406F70      dd        1                     ; Default
00406F74      dw        0                     ; NameIndex
00406F76      db        7,'Enabled'           ; Name
```

The first 32-bit quantity is a pointer to the type of the property. In this case, it points at the `boolean` type which is defined inside `SYSTEM.PAS`. Most crucially, a property's RTTI also contains a couple of 32-bit values which specify how to read or write the property. They are shown as `GetProc` and `SetProc` in the above RTTI description. For a simple field, these values are conceptually pointers to a specific byte offset within the class structure; this byte offset is the location of the field. Where a getter or setter routine has been declared, these values point directly to the routine itself.

The `GetProc` value shown here is $FF000018; in this particular case, we can interpret it as a reference to the byte at location $18 within any object instance of this class. In other words, it corresponds to the `fEnabled` field as shown in the property declaration. The `SetProc` value, however, is a straight 32-bit pointer that refers directly to the `SetEnabled` method that's used to set the value of the property.

This is actually a very sneaky design since the most significant byte of the `GetProc`/`SetProc` fields acts as a sort of 'escape code'. If this byte contains the value $FF, then the remaining 24 bits are interpreted as a byte offset to the field we're interested in, relative to the start of the object instance. Incidentally, this imposes an absolute limit on the instance size of a Delphi object of 2^{24} or 16Mb. This isn't likely to be a significant limitation most of the time!

The astute reader will no doubt be thinking, what about virtual methods? It's perfectly possible to have a property which uses a getter or setter routine that is itself a virtual method. The advantage of this is that a derived class can implement a new, overridden getter/setter method, and thus easily override the internal implementation of the property. How is this encoded in the above scheme? The answer is that, here again, the 'escape code' is used. As we've seen $FF indicates a byte offset from the start of the class instance, ie a simple field. In a similar way, $FE is used to indicate an offset that's relative to the object's VMT or virtual method table. In other words, the remaining 24-bits indicate which 'slot' in the VMT refers to the virtual getter/setter. If the most significant byte is neither $FF nor $FE, then it's assumed to be a straight pointer to a non-virtual method. Under Win32, it's impossible to have a valid process address which has a high byte of $FF or $FE, and consequently no ambiguity can arise.

For more information on all this, take a look at `TYPINFO.PAS` which includes the definition of `TPropInfo`, corresponding to the data structure shown above. Do bear in mind though, that `TPropInfo` is an inherently variable sized record because of the Pascal-style string that holds the property name. While you're at it, try and find a copy of Danny Thorpe's excellent *Delphi Component Design* published by Addison-Wesley. Sadly, this book is no longer in print, and it does need updating for more recent versions of Delphi, but it's still a great reference for those who like to get into the nitty-gritty.

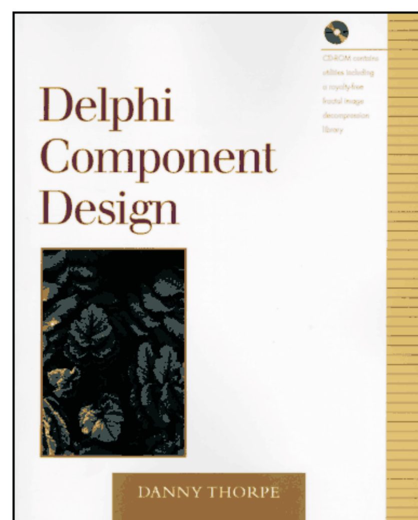
Note that I'm focusing on .DFM files and design-time issues here, but the same argument obviously applies to a compiled executable. At runtime, the .DFM file is replaced by a `FORM` resource which is contained within the program's EXE file:

```
property ItemCount: Integer
  read GetItemCount
  write fDummyInteger
  stored False;
```

The property declaration above is taken, once again, from this month's *Beating The System*. In this case, `ItemCount` is a property which returns the total number of items on the Windows desktop. This value is obviously dynamic, in other words, it depends on which PC we're running the program on, and it may change from one minute to the next as the user adds and removes desktop items. It's a value which can only be determined on-the-fly, and it therefore makes absolutely no sense to make this property persistent. For this reason, the property declaration includes a `stored` clause which tells the compiler to generate RTTI specifying that `ItemCount` shouldn't be stored.

A note for the anoraks: referring to the *RTTI, Under The Hood*

➤ *Figure 2: If you see this in a car boot sale, then buy it. Danny Thorpe's excellent treatise on component design is a real Delphi classic, sadly now out of print.*



```
public int Position
{
  get { return Seek(0, StreamOrigin.soFromCurrent); }
  set { Seek(value, StreamOrigin.soFromBeginning);  }
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
[DefaultValue(null)]
[Description("The image associated with the control")]
[Category("Appearance")]
public Image ActiveImage {
  get {...}
  set {...}
}
```

boxout, you'll see that, by default, the compiler sets the `StoredProc` field of a property's RTTI to `1`, indicating that the property should be persisted. If you specify '`stored False`' as part of the property declaration, then this field will be set to zero. Any other value is taken as a pointer to a boolean method which returns `True` or `False` according to whether or not the property should be stored.

In the same way, my desktop manager component has two other properties which are not stored because they are dynamic quantities. Judicious use of the `stored` directive can minimise the size of .DFM files by eliminating the storage of otherwise redundant information. You'll notice that the array properties in `TDesktopManager` don't have `stored False` as part of their declaration. The reason is that Delphi doesn't persist array properties anyway, and therefore we don't need to tell the compiler not to do so!

.NET: The View From The Other Side

I'd originally planned to spend some time talking about property editors, the new (and in my view, poorly designed) Delphi 5 support for property categories, and so on. However, this is a pretty big subject in its own right and perhaps we'll return to this in a future article, if the Editor can be prevailed upon.

In the remainder of this article, I thought you'd be interested in seeing a quick description of how properties are implemented from the perspective of a .NET programmer using the C# language, and how this compares with the Delphi approach. Here's the declaration for the `Position` property of the `TStream` class, taken from CLASSES.PAS:

```
property Position: Longint
  read GetPosition
  write SetPosition;
```

Expressed in C#, the equivalent property declaration would look like Listing 1.

You'll probably appreciate that the `get` and `set` clauses correspond directly to the `read` and `write` clauses in a Delphi property declaration. Omit either one, and you'll end up with a read-only or write-only property. You'll also see that the 'getter' and 'setter' methods essentially disappear and are replaced with inline code that appears as part of the property declaration.

Well, you might think that, but you'd be wrong! It's interesting to note that the C# compiler actually generates a pair of hidden, private, methods, `get_Position`, and `set_Position`, and these contain the (supposedly) inline code shown above. These methods are automatically called whenever the property is referenced. Even in the degenerate case where a public property simply reads and writes the value of a private field, the compiler still generates these private access methods. Personally, I like the way in which the getter/setter code appears as part of the property declaration: I think this is neater than the Delphi approach where the access code for a property can be located some distance away from the property declaration.

Where C# really shines is in the use of attributes which make it easy to control the design-time behaviour of your component. For example, consider the Listing 2 snippet taken from the .NET documentation.

In this case, three attributes (always introduced by square brackets) have been applied to the property `ActiveImage`. The first attribute, `DefaultValue`, tells the system that the default value of this property is null. In terms of property persistence, it obviously has a similar role to the `stored` keyword that I mentioned earlier. More interesting is the `Description` attribute; users of Visual Basic, for example, will know that as you

.NET Framework Developers Guide

This is preliminary documentation and subject to change.
Send feedback on this topic.

Common Attributes for Properties and Events

The commonly used attribute classes from the **System.ComponentModel** namespace are listed below. These classes can be extended.

Attributes for Properties and Events

Note: Attributes for events are applied to the **AddOn**<*EventName*> method that attaches event delegates to the event. When the attribute is provided, the form designer is informed how the event named <*EventName*> should be displayed in the events window.

BrowsableAttribute
Specifies whether a property or an event should be displayed in the property window. To hide a property or event from the property window, set **BrowsableAttribute.No**.
[C#]
[Browsable(false)]

CategoryAttribute
Specifies the name of the category that a given property or event should be grouped in. When categories are used, component properties and events can be displayed in logical groupings in the property window.
[C#]
[Category("Appearance")]

DescriptionAttribute
Defines a small block of text to be displayed at the bottom of the property window when the user selects a property.
[C#]
[Description("The image associated with the control")]

Attributes for Properties

BindableAttribute
Specifies whether a property is appropriate to bind data to.
[C#]
[Bindable(true)]

DefaultPropertyAttribute
Specifies the default property for the component.

DefaultValueAttribute
Sets a simple default value. **DefaultValueAttribute** results in the following:

- There is visual indication in the property window if a property has been modified.
- The user can right-click a property and choose **Reset** to restore the property to its default value.
- More efficient code is generated.

➤ *Figure 3: .NET provides a number of standard attributes which make properties easier to use and simplify the life of the component developer. Let's hope all this stuff is surfaced in Delphi.NET!*

select different items in VB's Property Inspector, a small hint area at the bottom of the window provides a brief description of the property. This, of course, is exactly what the `Description` attribute is used for: it adds the supplied descriptive string to the metadata associated with the component such that the description string is displayed when the property is active. Finally, the `Category` attribute ensures that this property is added to the `Appearance` category of properties. You don't need me to tell you that this is much quicker, neater and cleaner than the clunky property categorisation support which Borland added to Delphi 5. And all the more so when you realise that .NET's attribute mechanism is extensible, you can create your own custom attributes to handle such diverse issues as licensing, custom property editors and a whole lot more.

No, I'm not telling you this to depress you, quite the opposite. .NET is an exciting new development environment which offers unprecedented opportunities for component developers. My fervent hope is that the forthcoming implementation of Delphi.NET (or whatever it might be called) will fully exploit all these high-end features such as the aforementioned attribute mechanism. Exciting times lie ahead for your favourite programming language!

Dave Jewell is the Technical Editor of *The Delphi Magazine*. Contact Dave as TechEditor@itecuk.com