Beating the System: A Delphi Version Reporting Component

by Dave Jewell

A few weeks ago, I happened to come across a collection of shareware Delphi components including some very interesting and innovative controls. However, there's often a runt in the litter and I wasn't especially impressed with the version component which was contained therein. The idea was that of a non-visual component with fields such as build number, company name and so forth. You set these properties at design-time and, hey presto, the same information could be accessed at runtime for display in an `About` box or whatever. Ho-hum...

I'm afraid I found this concept rather underwhelming. If you're going to implement a component just to store a few strings and not do much else, then you're not really bringing anything new to the party. It occurred to me (but maybe not to the author of this anonymous control), that the Delphi IDE already provides the ability to embed a standard Windows `VERSION` resource into your executable. (Just go to `Project|Options`, select the `VersionInfo` tab, click the `Include Version Information In Project` checkbox, and off you go). I felt that the best way to implement a Version control would be to use the Delphi IDE to set up the various fields in the version resource and then use the component at runtime to provide quick and easy access to the version resource.

I also felt that it would be nice if the version component could display the same information at design-time (fools step in, etc!) and I therefore decided to write the component in such a way that the version information could also be viewed from the Object Browser, thus providing visual confirmation to the programmer that things had been set up as intended.

Pilgrim's Progress...

I started building the skeleton of this new component and things went well, until I began using the Windows versioning API routines to extract information from the `VERSION` resource to display it in the Object Inspector at design-time. I discovered that Microsoft's routines execute so slowly that, with the form designer window open in the Delphi IDE and the versioning component selected, my 300MHz Pentium II machine was reduced to a quivering wreck. I'm not sure what it is in the Delphi IDE which causes the Object Inspector to refresh the currently displayed property set of the selected component, but it was happening far too quickly for the system to cope.

In an effort to reduce the overhead, I decided to build a `TTimer` control into my versioning component. The timer is only created when the component is being used at design-time: it simply lurks in the background and refreshes the version data once every so often rather than every time one of the property access routines is called. Surprisingly, I found that, even with this enhancement, the versioning API was still slowing things down more than I would have liked.

I just couldn't figure out why Microsoft's implementation of the versioning API was so incredibly slow. In the end, I decided to bite the bullet and reached for my trusty disassembler in order to find out what was going on. It soon became clear that the versioning API has been written by someone with a real sense of humour.

As you may know, the 32-bit implementation of the versioning calls reside in a small DLL called VERSION.DLL. It turns out that this is no more than a wrapper DLL. It takes each of the 32-bit calls and 'thunks' down to the old VER.DLL library which handles versioning calls for 16-bit clients. The only API calls which don't immediately thunk down to 16-bit DLL are those routines which take a filename, `GetFileVersionInfoSize` being a good example. Since 16-bit DLLs don't understand long filenames, it was necessary for Microsoft to convert the passed filename into the equivalent short filename string (complete with all those wonderful ~ characters) before passing the call down to the 16-bit library. Sigh...

But it gets better. The 16-bit VER.DLL code in turn calls down to LZEXPAND.DLL, the 16-bit data decompression library which Microsoft provide. Trust me, I'm not making this up! At this point, I began wondering if maybe the decompression library pulls in some multimedia code, or maybe Internet Explorer, or... It was clear that Microsoft were using a sledgehammer to crack a peanut.

In the light of all this, I decided to put Microsoft's versioning code where it belongs, in the small round filing cabinet under my desk. I rewrote my component to directly load the version resource into memory, parse the data *in situ* and return the information as component properties, all without touching VERSION.DLL or its playmates. Moreover, I modified my timer routine so that it periodically checks the file modification time of the executable in question, and only fetches the version resource information if it detects the file modification time has changed. I reckon that this represents just about the simplest, lowest overhead approach to the problem.

Understanding VERSION

Now maybe you are thinking, Dave, if you are going to access the

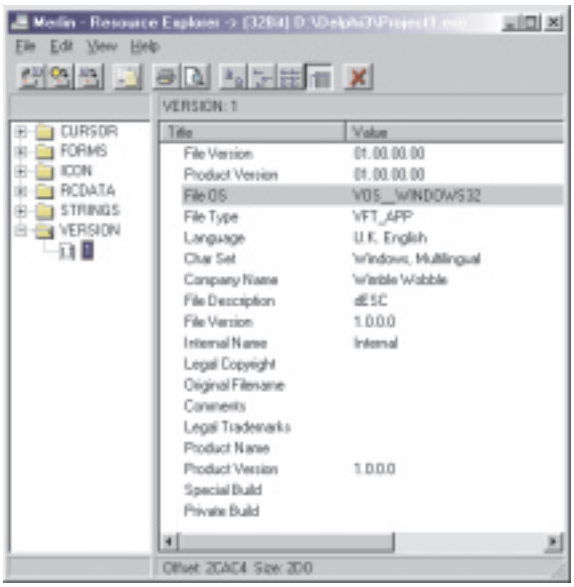
version resource directly, then isn't your code going to break if/when Microsoft ever change the format of the resource? Well, there's a measure of truth in that, but it's not really a big issue. For starters, the only version resource that we're interested in is the one built into our own executable! Consequently, when this component is deployed, any subsequent changes that Microsoft make aren't going to affect us. Indeed, one could argue that by removing the need for VERSION.DLL, we're actually in a safer position than we would be otherwise! Having just come through the Delphi 4 `ItemIndex` debacle, you'll understand just what I mean. Secondly, there's a measure of extensibility built right into the version resource even as it stands. As you may know, if you right-click on the Key/Value grid of the IDE's `VersionInfo` `Project|Options` dialog, the IDE will give you the option of adding a new custom key to the resource. The code presented here will cope with these new custom keys (it just comes out in the wash) and with this inherent extensibility, I don't really expect Microsoft to change the version resource format any time soon.

OK, so how is the resource organised? Once you tell the IDE that you want a version resource and then rebuild your application, the resource will immediately be compiled into your executable. If you look at Figure 1, you can see the Merlin Resource Explorer examining a version resource block. This program takes a relatively high view of life and doesn't show us the nuts and bolts view of the file that we need.

Fortunately, all the necessary information is provided by the Microsoft SDK. If you look up the string `VS_VERSION_INFO` in the WIN32.HLP file (look in your \Delphi3\Help directory) then you'll see something like that shown in Figure 2, which gives an overview of the version resource format. As you can see, there's a certain amount of preamble which is then followed by another data structure of type `VS_FIXEDFILEINFO` (this one appears in the WINDOWS.PAS file as `TVSFixedFileInfo`, although strangely the initial data structure doesn't). Following the file header is a list of `StringFileInfo` records, each of which comprise one or more `StringTable` records, which themselves comprise one or more `String` records! That's right folks, more Microsoft humour at work!

As it happens, this information actually makes things look somewhat more complex than the real-life situation. In reality, the size of the various initial strings is fixed and therefore we can make some assumptions about the byte offsets of data within the version info block. Moreover, the version resource created by Delphi seems to always be a degenerate case, in the sense that there is only ever one `StringFileInfo` record comprising a single `StringTable` record. Again, this means that the code is very much simpler than it would otherwise be.

That said, I give you no warranties, express or implied! As stated earlier, my code will certainly break when Microsoft change the format of the version resource and Inprise modify the IDE to generate the newer type of resource. However, I can't see this happening any time soon and at least with my code you know that you don't need a whole slew of other DLLs in order for your application to run. I mean: LZEXPAND, I ask you!



➤ *Figure 1: Here's a typical version resource as seen using the Resource Explorer utility that comes with Merlin. This utility presents a high-level view of the version data. To see what's happening below the scenes, we need to consult the SDK...*

How It Works

The complete source code for my `TVersionInfo` control is given in Listing 1. Normally, Delphi programmers make use of units such as `TOOLINTF`, `EXPTINTF` and the like when creating add-in experts for the Delphi IDE. Unintuitive as it may seem, these units are also available to the component writer provided (and it's a big proviso!) that you only want to use them at design-time. If you think about it, this is a pretty obvious restriction. When your component is running at design-time within the IDE, it's got essentially the same runtime environment as an IDE wizard or add-in. By contrast, when your application is compiled and executing, any IDE packages are unavailable to the application.

I put this fact to use in the constructor of `TVersionInfo`. Here, you can see that I check to see if the `ToolServices` variable is `Nil`. This variable, defined in the `EXPTINTF` unit, will be `Nil` if we're running standalone, so to speak, and it will be non-zero if our component is sitting on a form at design-time.

If you find this confusing, just bear in mind that the code of a Delphi component is 'running' just as much at design-time as it does when it's compiled into an executing program. As soon as you add a component to a form, an instance of that component is created and it starts executing. The conventional way of distinguishing between the two cases is to test the `csDesigning` bit in the `ComponentState` property. However, since we're actually going to make a call through the `ToolServices` interface, we may as

well use that to test whether this is design-time or runtime.

Having established that `ToolServices` is `Nil`, we know that this is the normal runtime case. This being so, the name of the executable file containing the version resource must be `Application.ExeName`. The code simply sets up the private string variable that holds the file name and then calls the `Refresh` method to do the real work of loading and parsing the version data. In the runtime case, we only ever call `Refresh` once, here in the constructor.

But what if we're sat on an IDE form designer window at runtime? In this case, `ToolServices` is not `Nil`. Instead, we create a `TTimer` in the usual way, set the `Interval` property to 500 (500ms is twice per second) and point the `OnTimer` event at the `TimerRefresh` method. Since we're not going to do anything in the `TimerRefresh` method if the file modification date/time hasn't changed, a 500ms interval is plenty fast enough for us. Even with Delphi, you'd be hard pushed to get your compile-debug-edit cycle down to half a second!

The only question remaining here is: where do we get the executable file name from at design-time? Here's where `ToolServices` comes in, we can just use the regular `GetProjectName` method to obtain the name of the project and then replace any file extension with .EXE. That's right, I'm assuming that you're not building a DLL or an OCX here, although it would be relatively straightforward to modify the code for these other cases. Assuming you've saved your project, the `GetProjectName` method will return the pathname of the saved project file. If you've just started a new project, then you'll find that you get a default project name and path. In the case of my Delphi 3 system, an unsaved project will result in a value of `D:\Delphi3\Project1.exe` being stored into the `fExecutableFileName` variable.

From this example, you should be able to see that we can potentially make use of *any* `ToolServices` methods from a design-time component. Like I said, this is a powerful, though non-obvious technique. Remember, you saw it here first. ☺

Of course, you could argue that the correct way of doing all this would be to install a notifier, such that the notifier calls the component every time we've done a compile. Well, fair enough, that would be another way of doing things: the `ToolServices` variable certainly gives us access to the notifier code at design-time. However, I'm a great believer in the KISS (Keep It Simple Stupid!) philosophy and it's certainly easier to set up and manage a `TTimer` than it is to establish a custom notifier.

This brings us onto the `TimerRefresh` routine, called by the timer. Here, the code retrieves the current file age (date and time of last modification) and compares it with the `fFileAge` variable. If they differ, then the executable has been rebuilt since the last time around the loop. The `Refresh` method is called and the `fFileAge` variable is reset. Note carefully what will happen if the executable hasn't yet been created or (for whatever reason) has been deleted. If the `FileAge` routine can't find the specified file, then it returns a value of -1. This is why I initialise `fFileAge` to -1 in the constructor. If the executable isn't available at any time, then the various properties revert to an 'unavailable' state.

Much of the real work, of course, is done via the `Refresh` method. Here, any existing version data is first de-allocated and the code then checks to see if we've got a valid pathname and if we can load the version resource from the executable. The `Refresh` routine then wends it way through the usual tedious sequence of API routines needed to load a resource into memory (oops, did I say that?) and finally copies the data into our in-memory buffer, calling the `ParseVersionData` method to convert the version data into a more accessible format. Along the way, there's also a sanity check call to ensure that we recognise this type of version data.

The rubber hits the road in the `ParseVersionData` method. This routine isn't perhaps a shining example of the programmer's art, but at least it gets the job done. Referring back to the earlier discussion on the format of the version resource, you'll see that I've hard-coded three constants which are absolute byte offsets into the version data. Armed with this information, the code skips through the version resource, finding key strings and adding them to the `fVerStrings` list along with any

➤ *Listing 1 (below and facing)*

```
unit VersionInfo;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms,Dialogs, ExtCtrls, ExptIntf, ToolIntf;
type
  TVersionInfo = class(TComponent)
  private
    fTimer: TTimer;
    fFileAge: Integer;
    fVersionData: PChar;
    fVerStrings: TStringList;
    fExecutableFileName: String;
    procedure ReadOnlyStringProperty(Index: Integer;
      const Value: String);
    procedure ReadOnlyIntegerProperty(
      Index, Value: Integer);
    procedure SetVersionStrings (Value: TStringList);
    function GetIndexStringProperty(Index: Integer): String;
    function GetIndexIntegerProperty(
      Index: Integer): Integer;
    procedure TimerRefresh (Sender: TObject);
    procedure Refresh;
    procedure ParseVersionData;
    function GetKey (const KeyName: String): String;
  protected
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    property Key[const KeyName: String]: String read GetKey;
  published
    property ExecutableFileName: String index 0 read
      GetIndexStringProperty write ReadOnlyStringProperty;
    property CompanyName: String index 1 read
      GetIndexStringProperty write ReadOnlyStringProperty;
    property FileDescription: String index 2 read
      GetIndexStringProperty write ReadOnlyStringProperty;
    property FileVersion: String index 3 read
      GetIndexStringProperty write ReadOnlyStringProperty;
    property InternalName: String index 4 read
      GetIndexStringProperty write ReadOnlyStringProperty;
    property LegalCopyright: String index 5 read
      GetIndexStringProperty write ReadOnlyStringProperty;
    property LegalTrademarks: String index 6 read
```

```
      GetIndexStringProperty write ReadOnlyStringProperty;
    property OriginalFilename: String index 7 read
      GetIndexStringProperty write ReadOnlyStringProperty;
    property ProductName: String index 8 read
      GetIndexStringProperty write ReadOnlyStringProperty;
    property ProductVersion: String index 9 read
      GetIndexStringProperty write ReadOnlyStringProperty;
    property Comments: String index 10 read
      GetIndexStringProperty write ReadOnlyStringProperty;
    property Keys: TStringList read fVerStrings
      write SetVersionStrings;
    property FileVersionHigh: Integer index $30 read
      GetIndexIntegerProperty write ReadOnlyIntegerProperty;
    property FileVersionLow: Integer index $34 read
      GetIndexIntegerProperty write ReadOnlyIntegerProperty;
    property ProductVersionHigh: Integer index $38 read
      GetIndexIntegerProperty write ReadOnlyIntegerProperty;
    property ProductVersionLow: Integer index $3C read
      GetIndexIntegerProperty write ReadOnlyIntegerProperty;
  end;
procedure Register;

implementation

constructor TVersionInfo.Create (AOwner: TComponent);
begin
  Inherited Create (AOwner);
  fFileAge := -1;
  fVersionData := Nil;
  fVerStrings := TStringList.Create;
  if ToolServices = Nil then begin
    // Run-time case
    fExecutableFileName := Application.ExeName;
    Refresh;
  end else begin
    // Design-time case
    fTimer := TTimer.Create (Self);
    fTimer.Enabled := True;
    fTimer.Interval := 500;
    fTimer.OnTimer := Self.TimerRefresh;
    fExecutableFileName := ToolServices.GetProjectName;
    if fExecutableFileName <> '' then
      fExecutableFileName :=
        ChangeFileExt (fExecutableFileName, '.exe');
  end;
end;

destructor TVersionInfo.Destroy;
begin
  if ToolServices <> Nil then fTimer.Free;
  if fVersionData <> Nil then FreeMem (fVersionData);
  fVerStrings.Free;
  Inherited Destroy;
end;

procedure TVersionInfo.Refresh;
var
  pSrc: PChar;
  hMod: hModule;
  Res: hRsrc;
  lRes: hGlobal;
begin
  // Trash the existing version data buffer
  if fVersionData <> Nil then FreeMem (fVersionData);
  fVersionData := Nil;
  // Now get the updated stuff....
  if fExecutableFileName <> '' then begin
    hMod := LoadLibraryEx(PChar(fExecutableFileName), 0,
      Load_Library_As_DataFile);
    if hMod <> 0 then try
      Res := FindResource (hMod, PChar (1), rt_Version);
      if Res <> 0 then begin
        lRes := LoadResource (hMod, Res);
        if lRes <> 0 then begin
          pSrc := LockResource (lRes);
          if pSrc <> Nil then begin
            // Sanity check time!
            if PWideChar(pSrc+6) = 'VS_VERSION_INFO'
              then begin
              GetMem(fVersionData,
                SizeofResource(hmod, Res));
              Move(pSrc^, fVersionData^,
                SizeofResource(hmod, Res));
              ParseVersionData;
            end;
          end;
        end;
      end;
    finally
      FreeLibrary (hMod);
    end;
  end;
end;

procedure TVersionInfo.ReadOnlyStringProperty(
  Index: Integer; const Value: String);
begin
  // Read-only property
end;

procedure TVersionInfo.ReadOnlyIntegerProperty(
  Index, Value: Integer);
begin
  // Read-only property
end;

procedure TVersionInfo.SetVersionStrings(
  Value: TStringList);
begin
  // Read-only property
end;

procedure TVersionInfo.TimerRefresh (Sender: TObject);
var AgeNow: Integer;
begin
  AgeNow := FileAge (fExecutableFileName);
  if AgeNow <> fFileAge then begin
    // Executable has been freshened, newly created or deleted
    fFileAge := AgeNow;
    Refresh;
  end;
end;

procedure TVersionInfo.ParseVersionData;
const
  //** ACHTUNG!  Don't change these constants unless the
  //** format of the VERSION resource is altered.
  vSFIStart = $5C;       // Start of StringFileInfo block
  vSTStart  = vSFIStart + $24;  // Start of String table block
  vSStart   = vSTStart + $18;   // Start of String table proper
var
  p: PChar;
  pw: PWord absolute p;
  StringFileInfoLen, ThisEntryLen: Word;
  Key, Val: String;

  function Align32 (p: PChar): PChar;
  var pp: LongInt absolute p;
  begin
    pp := (pp + 3) and $fffffffc;
    Result := p;
  end;

begin
  // You can never have too many sanity checks...
  if PWideChar(fVersionData + vSFIStart + 6) <>
    'StringFileInfo' then
    raise Exception.Create ('Unrecognised version block');
  // Looks good - parse the version strings
  fVerStrings.Clear;
  p := fVersionData + vSTStart;
  StringFileInfoLen := pw^;
  // Point at first entry
  p := fVersionData + vSStart;
  while p < (fVersionData+vSTStart+StringFileInfoLen)
    do begin
    ThisEntryLen := pw^;
    Key := PWideChar(p + 6);
    Val := PWideChar((Align32(p+6+((Length(Key)+1)*2))));
    fVerStrings.Add (Key + '=' + Val);
    p := Align32 (p + ThisEntryLen);
  end;
end;

function TVersionInfo.GetKey(const KeyName: String): String;
var
  S: String;
  Index, nPos: Integer;
begin
  if fVersionData = Nil then
    Result := '--not available--'
  else begin
    for Index := 0 to fVerStrings.Count - 1 do begin
      S := fVerStrings [Index];
      nPos := Pos ('=', S);
      if Copy (S, 1, nPos - 1) = KeyName then begin
        Result := Copy (S, nPos + 1, MaxInt);
        Exit;
      end;
    end;
    Result := '';
  end;
end;

function TVersionInfo.GetIndexStringProperty(
  Index: Integer): String;
const
  PropName: array [1..10] of String = (
    'CompanyName',      'FileDescription',
    'FileVersion',      'InternalName',
    'LegalCopyright',   'LegalTrademarks',
    'OriginalFilename', 'ProductName',
    'ProductVersion',   'Comments');
begin
  case Index of
    0 :     Result := fExecutableFileName;
    1..10 : Result := GetKey (PropName [Index]);
  end;
end;

function TVersionInfo.GetIndexIntegerProperty(
  Index: Integer): Integer;
begin
  if fVersionData = Nil then
    Result := -1
  else
    Result := PInteger (fVersionData + Index)^;
end;

procedure Register;
begin
  RegisterComponents ('The X Factor', [TVersionInfo]);
end;

end.
```

Figure 2: The Microsoft documentation reveals that the version resource data can have a somewhat baroque format. Fortunately, the Delphi IDE generates a straightforward variation of this data format.

accompanying key-value information, the two fields being separated by a = character. Woe betide you if you happen to use this character as part of your version data! In a commercial version of this code, it might be better to use a non-printable character as a delimiter or else to store the key-value data in the `Objects` part of the list. Another wrinkle in this code is Microsoft's odd stipulation that certain fields need to be aligned on 32-bit boundaries. This explains the use of the `Align32` routine within this method.

Using TVersionInfo

Once we've grabbed the key-value information from the version data, the rest is relatively plain sailing. The component maintains a list of ten distinct property names, (`CompanyName` through to `Comments` in the source code) each of which is a 'standard' field within the version resource. These ten property names are assigned index values (you know by now that I just love indexed properties!) and they all map down onto the `GetIndexStringProperty` routine. Here, the index is used to obtain the appropriate property name from a constant array of strings, and the appropriate call is then made to the `GetKey` routine in order to obtain the value part of the string. You will also notice that the `GetIndexStringProperty` routine is shared with the `ExecutableFileName` property.

I mentioned earlier that the version resource is inherently extensible. You can add your own custom keys by right-clicking the version resource's key-value grid in the IDE `Project Options` dialog. In order to make the `TVersionInfo` component, I needed a way to access custom keys in addition to the ten pre-defined string fields mentioned above. I chose to implement this through an array property, `Keys`, which takes a key name as the array index. Thus, assuming that you've got a `TVersionInfo` component called `Version`, you can access a custom key like this:

```
DeveloperName.Caption :=
  Version.Keys['DeveloperName'];
```

This code fragment assumes that you've got an `About` box in which you want to display version information about your program. On the `About` box form, you've got a `TLabel` called `DeveloperName` which gives the name of the proud programmer/s. All that's necessary is for you to add a custom key-value pair to your version resource, and off you go.

In practice, you'll most likely be using my `TVersionComponent` to display `About` box information, although there's nothing to stop you using it for other nefarious purposes. As an example, you could use a registration utility to patch encrypted user name and registration key information into custom placeholder key-value pairs within the version resource. The program would then examine this data at runtime to see whether it's a registered copy and, if so, provide full functionality to the end-user. The benefit of this approach is that a registered executable remains registered even when copied to floppy disk or moved to another computer: it doesn't depend on secret settings in the registry. Most users will be dissuaded from giving copies to their friends when they see their names appear in the `About` box, and any attempt to 'massage' this information would effectively de-register the application.

Finally, you'll notice that I've also added a few `FileVersionHigh/Low` and `ProductVersionHigh/Low` properties to the component, these fields simply returning the version information as integer values rather than strings. This might be more convenient for the developer under certain circumstances.

Figure 3: Here's my component strutting its stuff. All the properties are read-only. This is primarily a reporting component which allows an application to easily interrogate its own version resource information at runtime.

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level work. He is Technical Editor of *Developers Review*, also published by iTec. You can reach Dave at Dave@HexManiac.com