Beating the System: Deciphering The DCU, Part 1

by Dave Jewell

The DCU file is one of several reasons for the extraordinary compilation speed advantage of Delphi over more conventional systems using OBJ files. DCU files allow the compiler to read symbol table information very quickly. The .OBJ file was designed back in the days when many microprocessor systems (not PCs, they came later) used a paper tape punch and reader as mass storage devices! Consequently, every record in an .OBJ file had an associated checksum byte to allow for those frequent times when a paper tape reader misread a byte of data.

With the advent of high speed, high capacity hard disks and huge amounts of RAM, Borland were able to adopt a much more efficient approach in the design of TPU, and later DCU, files. In essence, you can think of a DCU file as a tear-off, re-locatable chunk of compiler symbol table. Viewed in this way, it's no surprise that the technology is far faster and more efficient than the old .OBJ file approach. The compiler finds a `USES Widgets` statement, inserts the various symbol definitions from the WIDGETS.DCU file into the symbol table, and then continues on its merry way.

As you will undoubtedly be aware, one of the (few) irritating aspects of Delphi development is the way in which the format of DCU files appears to change every time Borland update the development system. What this means in simple terms is that if your application is reliant on third party components (and many applications are), then you've either got to purchase the source code along with each third party control you buy, or else you've got to hope that the tool vendor in question will still be around when the time comes to upgrade to a more recent version of Delphi. The former option can be expensive while the latter option is obviously risky, particularly when dealing with small outfits.

In practice, most reputable component suppliers are aware of this problem and are happy to provide source code to their components, often as standard. I'm thinking of companies such as TurboPower (www.turbopower.com) who *always* supply full source code to the products they sell. I'm not anticipating that TurboPower are likely to disappear anytime soon but if, for the sake of argument, the company was to disappear tomorrow, all its existing customers should have relatively few problems in moving existing software to (say) Delphi 5, because they've got full source code.

By contrast, consider what would happen if you purchased the PixelGraphics image processing library from Peter Beyersdorf (www.beyersdorf. com). In case you're not aware of this product, PixelGraphics is a brilliant set of image processing components for Delphi/C++Builder, besides which my own humble efforts of last month pale rapidly into insignificance. In the past, I've waxed lyrical about the PixelGraphics tools, and I'd go so far as to suggest that this is still the best set of native VCL image processing components around. But on the negative side, Peter still hasn't provided official support for Delphi 4, despite the fact that (by the time you read this) Delphi 4 will have been around for almost twelve months. Moreover, Peter doesn't provide source code to the PixelGraphics library and (at the time of writing) he hasn't been seen in the PixelGraphics newsgroup for a couple of months. It doesn't require Mensa membership to get the impression that Peter has lost interest in his own brainchild, and if you were to purchase the graphics library at this point in time you might have serious difficulties when Delphi 5 does appear. (If you are reading this Peter, I'd love you to prove me wrong: bite the bullet, release a Delphi 4 version, make the source code available as an optional extra, show up in the newsgroup now and again, and I'll be delighted to publish a retraction.)

So Why Decipher The DCU?

If source code is available from most component writers, why bother deciphering the format of DCU files at all? Let me begin by stating that I'm intending nothing illegal here. You'll see frequent requests for the DCU file format in the Delphi-related newsgroups, and I get the impression that many people think that an understanding of the DCU file format would enable them to 'sourcify' third party Delphi units without bothering to purchase the source code. This is absolutely not the case, and it certainly isn't my intention here.

Any Delphi programmer worth his salt will fully understand that code can be disassembled much more easily by compiling it into a package than would ever be the case with a standalone DCU file. A DCU file typically 'exports' a number of public symbols such as types, variables, classes and methods. At the same time, it obviously 'imports' a large amount of information too, every time you call the `Length` function, some behind the scenes compiler magic will transmogrify this into a call to the appropriate (depending on the type of string involved) library routine in the `System` unit. These relocatable references obviously complicate the process of disassembly, but by compiling the unit into a package, all these fixups are

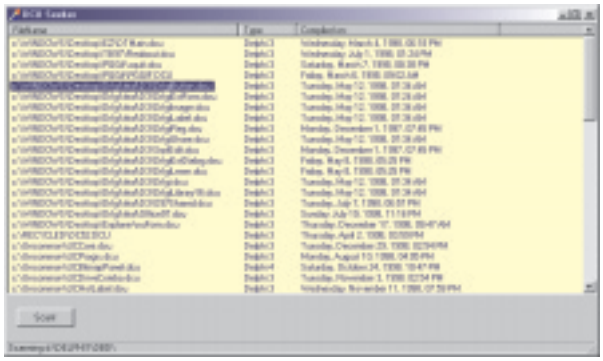

Figure 1: This program (included on this month's disk) scans an entire hard disk for DCU files, displaying the version and compile date/time data for each file found.

automatically done for you. A good disassembler such as IDA-Pro (www.datarescue.com) will show all the exports from the package, and the latest version even understands the format of Delphi and C++Builder RTTI information! This is why I commented that Borland's decision to distribute the Delphi 4 IDE in packaged form was like manna from heaven to those who like to see what's going on under the hood.

But I digress. Hopefully, the above will have convinced you that for serious disassembly, packages are the only way to travel. My motivation in discussing the internals of DCU files is simply an effort to precipitate the same thing that happened with the Windows .HLP file format. Back in the September and October issues of *Dr Dobb's Journal*, Pete Davis described the hitherto undocumented format of Windows .HLP files. Although the initial information was incomplete, it encouraged others to get involved in unravelling the file format, the end result being that the format is now pretty well understood, with all manner of shareware, freeware and commercial utilities around which can directly access .HLP files.

This is what I'd like to see happen with DCU files. I can't claim to have anything more than a rudimentary understanding of DCU file internals yet, but it's my hope that the information presented here will encourage others to work on the problem and come up with more detailed information, thereby making it possible to write new classes of Delphi programming tools and IDE add-ins. As an example, I've from time to time seen newsgroup messages asking for information on how to hook into Delphi's 'Kibbitz' system (the internal name for the code completion stuff which displays parameter information and possible method calls while you type). By parsing the content of a DCU file from within a Delphi expert, a similar effect could be achieved.

But What About Version Problems?

But what about versioning problems, I hear you cry! Don't Borland claim that they keep the DCU file format private specifically because the format changes with each new release of the compiler?

Now and again, I've suggested that Borland should consider making the DCU file format available, but this has been pretty much ignored. At other times, I've suggested that even if the company doesn't wish to make the file format available, they might release a conversion utility, thereby allowing, for example, Delphi 3 DCUs to be upgraded for compatibility with Delphi 4. Again, this has provoked no response. The fact that Borland doesn't seem willing to release a conversion utility (surely a simple undertaking for those with full details of the DCU changes that result from one version change to the next?) suggests that the company must have some other motivation in not making this information available, and that the versioning excuse is simply that: an excuse. Well, I've got no axe to grind, but from what I've learned about the DCU file format so far, the overall structure of the file format has *not* changed radically since Delphi 2. Make of that what you will...

Some years back, Borland actually promised that they would freeze the DCU file format, making it possible to reuse existing object files with later versions of the development system. However, for whatever reason, this promise was never fulfilled.

DCU File Format Basics...

Right then, I've tortured you enough. Let's roll up our sleeves and make a start.

Every DCU file begins with a 'magic' four-byte signature. If this signature isn't present, then we can be pretty sure that we're not dealing with a valid DCU file. Borland have cunningly changed this magic signature value from one version of Delphi to the next. The signature values that I'm aware of are as follows:

```
D2Magic  =          $50505348;
D3Magic  =          $44518641;
D4Magic  =          $4768A6D8;
B3Magic  =          $475896C8;
```

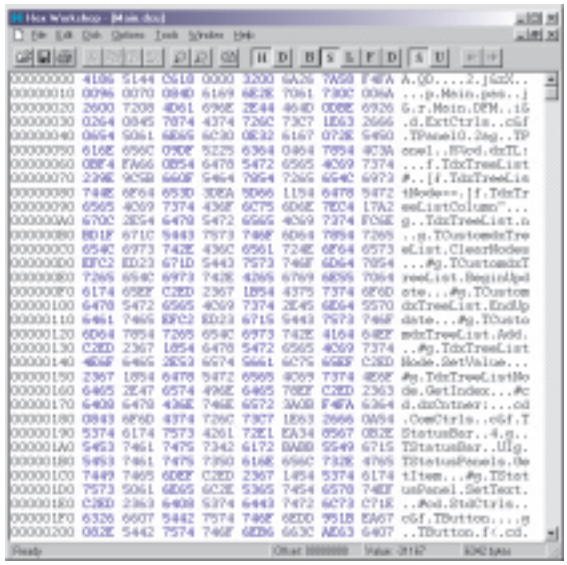

Figure 2: It looks like complete junk, doesn't it? But if you compare this hexadecimal dump of a DCU file to the information mentioned in this month's article, you should be able to see the magic signature, file length and compile date/time.

Thus, D2Magic, D3Magic and D4Magic are the signature values for Delphi 2, 3 and 4 DCU files. The B3Magic signature relates (I think!) to the DCU files supplied with C++Builder 3. I have several thousand DCU files in various places on my hard disk, and the only place where I discovered files with the B3Magic signature was in the \Lib\Obj directory of C++Builder 3.

Following the four-byte magic signature is a 32-bit integer that specifies the size of the DCU file in bytes. This file size is inclusive (ie it includes the preceding magic signature) and corresponds directly to the file size as reported by DOS, again, you should check that it does. As a third and final check, if you seek to the end of a DCU file, you should find that the last byte in the file has a value of $61. This represents a sort of end of file marker for DCU files and is present in every DCU file I've encountered.

Next, after the file length specifier is another 32-bit quantity which indicates the date and time when the unit was compiled. Generally speaking, this will be the same as the DOS modification date/time of the DCU file itself, but this need not be so. If you use a hex editor to fiddle around with the innards of a DCU file, use a 'touch' utility to alter the modification time, or whatever, then the internal compile date/time will obviously differ from the file's modification date/time. Why is it necessary for the development system to store the unit's compile time into the DCU file itself? The reason, of course, is to do with Delphi's own internal 'make' algorithm. When you compile the current application, Delphi examines each unit to determine whether it needs to be recompiled. If unit A is dependent on unit B, and it turns out that unit B was compiled after unit A, then the compiler will automatically recompile unit A as well. By embedding the compilation time into the DCU, the make system is immune to user modifications of the file date/time.

The 32-bit compile time is in standard DOS format: a packed 16-bit time field, followed by a packed 16-bit date field. This can be converted to a standard TDateTime quantity using the FileDateToDateTime routine in the SysUtils unit.

There is one strange wrinkle here. Borland-supplied DCUs frequently store the value $FFFFFFFF into this date/time field. It's best to check for $FFFFFFFF because if this value is inadvertently passed to the FileDateToDateTime routine,

➤ *Listing 1*

```
unit Main;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ComCtrls, ExtCtrls;
const
  // Magic signatures
  D2Magic  =          $50505348;
  D3Magic  =          $44518641;
  D4Magic  =          $4768A6D8;
  B3Magic  =          $475896C8;
  // DCU record tags
  Tag_End =          $61;
type
  TForm1 = class(TForm)
  Scan: TButton;
  StatusBar1: TStatusBar;
  TreeList: TListView;
  procedure ScanClick(Sender: TObject);
  private
    Scanning: Boolean;
    procedure ScanDrive (const Path: String);
    procedure FoundOne (const PathName: String);
  public
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.ScanClick (Sender: TObject);
var
  p: PChar;
  szBuff: array [0..255] of Char;
begin
  Scanning := not Scanning;
  if Scanning then begin
    Scan.Caption := 'Stop Scan!';
    Screen.Cursor := crHourGlass;
    TreeList.Items.Clear;
    TreeList.Items.BeginUpdate;
    try
      p := szBuff;
      GetLogicalDriveStrings (sizeof (szBuff), szBuff);
      while Scanning and (p^ <> #0) do begin
        if GetDriveType (p) = Drive_Fixed then
          ScanDrive(p);
        Inc(p, 4);
      end;
    finally
      Scanning := False;
      Scan.Caption := 'Scan!';
      Screen.Cursor := crDefault;
      TreeList.Items.EndUpdate;
    end;
  end;
end;
procedure TForm1.FoundOne (const PathName: String);
var
  eof: Byte;
  S: String;
  Valid: Boolean;
  fs: TFileStream;
  Item: TListItem;
  Magic: array [0..2] of LongInt;
begin
  fs := TFileStream.Create (PathName, fmOpenRead);
  try
    fs.Read (Magic, sizeof (Magic));
    fs.Position := fs.Size - 1;
    fs.Read (eof, sizeof (eof));
    Valid := (Magic [1] = fs.Size) and (eof = Tag_End);
  finally
    fs.Free;
  end;
  if Valid then begin
    Item := TreeList.Items.Add;
    Item.Caption := PathName;
    case Magic [0] of
      D2Magic :  S := 'Delphi 2';
      D3Magic :  S := 'Delphi 3';
      D4Magic :  S := 'Delphi 4';
      B3Magic :  S := 'C++ Builder 3';
      else       S := '???' + IntToHex (Magic [0], 8);
    end;
    Item.SubItems.Add (S);
    if Magic [2] = $ffffffff then
      S := 'Invalid date/time'
    else
      S := FormatDateTime('dddd, mmmm d, yyyy, hh:mm AM/PM',
        FileDateToDateTime (Magic [2]));
    Item.SubItems.Add (S);
  end;
end;
procedure TForm1.ScanDrive (const Path: String);
var
  Res: Integer;
  SR: TSearchRec;
begin
  Application.ProcessMessages;
  StatusBar1.Panels [0].Text := 'Scanning ' + Path;
  Res := FindFirst (Path + '*.*', faAnyFile, SR);
  try
    while Scanning and (Res = 0) do begin
      if SR.Name [1] <> '.' then begin
        if UpperCase (ExtractFileExt(SR.Name)) = '.DCU' then
          FoundOne (Path+SR.Name)
        else if ((SR.Attr and faDirectory) <> 0) then
          ScanDrive(Path + SR.Name + '\');
      end;
      Res := FindNext (SR);
    end;
  finally
    FindClose (SR);
  end;
end;
end.
```

ScanLine Secrets: Matters Arising And More Ideas...

I don't know about you, but by the time I've finished writing some new Delphi application, I can often think of a faster, more efficient way of getting the same job done. Very similar things could be said about writing magazine articles on programming! As far as I know, there weren't actually any bugs in last month's code (famous last words!) but since I wrote that article, a few other things have occurred to me which I think are worth mentioning.

Firstly, you'll remember that I replaced the Borland supplied definition of `TRGBQuadArray` with a new definition that looks like this:

```
TRGBQuadArray = array [Word] of TRGBQuad;
```

This was done because the Borland code used a `Byte` range as the index of the array, effectively limiting the `TRGBQuadArray` to working with bitmaps no more than 256 pixels wide, not much use with today's hardware. I don't expect that you'll come up against a pixel width limit of 65536 any time soon, but if you really want to future-proof that code, then you can rewrite that declaration to look like this:

```
TRGBQuadArray = array [0..0] of TRGBQuad;
```

If this declaration leaves you somewhat baffled, suffice it to say that declaring an array with upper and lower bounds of zero is a venerable old technique which goes right back to the days of DOS-based Turbo Pascal and may even have been implemented in the old UCSD Pascal compiler. By declaring an array in this way, you're essentially telling the compiler that you want to disable range-checking on the supplied array subscript, and that you want to be able to pass any integer (variable or constant) as an array index. Normally, if you've got runtime range checking disabled, the compiler will only complain if you pass an out-of-bounds constant as an array subscript, but with this technique, you can pass a constant value equal to `MaxInt` (2,147,483,647) and the compiler will be perfectly content. Now that's what I call a big bitmap!

Secondly, for reasons of space, I didn't include a routine for greyscaling a bitmap. However, converting a bitmap to greyscale is actually extremely easy. Here's how to do it: you simply need to walk through each pixel of the bitmap, using the `ScanLine` property as demonstrated last month. For every pixel in the bitmap, add up the sum of the red, green and blue components and then divide the result by three to get the overall brightness level of that particular pixel. Then, set the red, green and blue components of the pixel to this value. This technique effectively preserves the relative *brightness* levels of each pixel, while removing all the *colour* information. Because the RGB components of each pixel are always the same, you just get lots of different shades of grey, hence, a grey-scaled bitmap. I haven't included any code to do this here, but if you've read last month's column, you'll appreciate that it's trivial to implement.

Finally, I've been messing around with PhotoShop over the last few days. Devotees of this excellent graphics program will be aware that a variety of plug-ins are available to increase the variety of special effects implemented by PhotoShop. In particular, I've been playing with a set of tools called NVR BorderMania 3.0 which I found at http://www.mediaco.com/nvr/filters.html. These plug-ins are freeware, can be used without restriction and can be used to put shaded borders around an image so as to give it a 3D 'look'. An example of this is the Delphi 3 splash screen shown below. Here, you can see that shaded borders have been used to create a bevelled three-dimensional effect. It occurred to me that you could use the brightness-tweaking code from last month's article to programmatically achieve the same result, increasing the brightness of the top and left borders and reducing the brightness of the bottom and right borders. Maybe some enterprising individual would like to write a new Delphi button component which takes a bitmap and automatically tweaks the border brightness to create a nice effect? Let me know if you succeed...

then you'll get an exception. What's the significance of storing `$FFFFFFFF` into the date/time field? I'm not sure, I haven't experimented here, but it's possible that doing so will disable the internal 'make' logic.

Desperately Seeking DCUs...

Earlier, I stated that I'd searched through several thousand DCU files and only been able to detect the `B3Magic` signature in those belonging to C++Builder 3. Do you really think I'm such an anorak that I'd do this by hand? Well, no, I'm not quite *that* bad! Instead, I developed a small DCU searching program, which you can see running in Figure 1. It basically searches your entire hard disk (including all the different DOS partitions) for any DCU files it can find, validating them using the checks that I've mentioned earlier.

The source code to the DCU Seeker program is shown in Listing 1. Basically, it consists of a `TListView` component operating in 'report' mode, along with a status bar control to display the progress of the disk seeking operation. You can interrupt the recursive file search at any time; if you do so the listview will simply display information on the DCUs that have been found so far. The program is very straightforward, using the nifty `GetLogicalDriveStrings` API call to provide a list of logical disk drives, from out of which we scan only the fixed disks via the `ScanDrive` routine.

Inside `ScanDrive`, the status bar caption is updated, the disk is searched for DCU files, and `ScanDrive` calls itself recursively every time that a subdirectory is sound. Every encountered DCU file is passed to the `FoundOne` routine in order to be validated. Inside `FoundOne`, we open the DCU file by creating a `TFileStream` object and then read the first 12 bytes of the file, corresponding to the signature, file size and compile-time quantities that I've discussed above. The code then seeks to the end of the file, reads the final byte of the DCU and checks that it's equal to `Tag_End` while also

checking that the internal file size specifier agrees with the physical length of the file.

If all this works out ok, the DCU signature is then checked, and the appropriate file type string is added to the list item. Finally, I perform an explicit check to see if the internal compile time/date equals `$ffffffff`. If not, then `FormatDateTime` is called to return a prettily-formatted date/time for the tree-list display.

Conclusions

But what lies after those twelve bytes that I've discussed so far? For that, I'm afraid you'll just have to wait for next month's column! This month, I was keen to spend some time explaining my motivation for deciphering the DCU file format and effectively laying the ground work for what's to follow.

Next month, I'll begin by presenting the source code for a small Delphi unit, and then we will take a walk through the DCU file that's produced by compiling this unit, examining the many wonders contained therein!

Finally, I'll extend the DCU Seeker program so that when you double click a file in the list, it will display a form containing a lot more information on the file in question. See you then!

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows. He is the Technical Editor of *Developers Review* which is also published by iTec. You can contact Dave at TechEditor@itecuk.com