Beating the System: Deciphering The DCU, Part 2

by Dave Jewell

As you will know, last month's *Beating the System* was an introductory article on the internals of the DCU file format. This time round, without further ado, we'll carry on right where we left off. In this month's column, I'll give you the source code for a very simple Delphi unit, and we'll then work through some of the bits and bytes in the resulting DCU file. I should mention that throughout this article I'll be using the Delphi 2 compiler (no, I've not gone all retro on you, it's just easier for me to show you what's going on with Delphi 2 DCU files!), but, as I indicated last time round, you'll find that the DCU files produced by later versions have the same overall structure.

A Very Basic DCU File

For reasons that will become obvious later, we're going to begin by looking at the innards of a *very* simple unit. Try typing in the following code and compile it into a DCU using the Delphi 2 compiler:

```
unit Squit;
interface
const
  SomeNumber = $55AA;
implementation
end.
```

➤ *Listing 1*

```
procedure TForm1.FoundOne (const PathName: String);
var
  eof: Byte;
  S: String;
  Valid: Boolean;
  fs: TFileStream;
  Item: TListItem;
  Magic: array [0..3] 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);
    if (Magic [0] = D2Magic) and ((Magic [3] and $ff) <> 0) then begin
      ShowMessage (PathName + ' is invalid Delphi2 DCU.  Skipping...');
      Valid := False;
    end;
... etc ...
```

```
22A2:0000  48 53 50 50 90 00 00 00-3D A5 C2 26 00 70 09 73  HSPP....=%B&.p.s
22A2:0010  71 75 69 74 2E 70 61 73-29 A3 C2 26 00 64 06 53  quit.pas)#B&.d.S
22A2:0020  79 73 74 65 6D 00 00 00-00 63 25 0A 53 6F 6D 65  ystem....c%.Some
22A2:0030  4E 75 6D 62 65 72 8A 33-7E 45 D9 02 00 53 AD 02  Number.3~EY..S-.
22A2:0040  28 05 53 71 75 69 74 80-00 00 00 00 00 02 04 63  (.Squit........c
22A2:0050  44 00 04 00 06 00 FB FF-03 0C 40 00 00 00 44 00  D.....{...@...D.
22A2:0060  08 00 06 0F 00 00 00 80-0F FF FF FF 7F 00 6C 02  ..............l.
22A2:0070  C3 6D 04 00 03 06 02 04-06 90 02 16 00 91 02 02  Cm..............
22A2:0080  18 00 92 00 93 00 00 94-04 06 20 00 00 00 00 61  .......... ....a
```

➤ *Figure 1*

Next, create a hexadecimal dump of the DCU file. Once done, you should see something like Figure 1.

As you can see, the file begins with the `D2Magic` signature (see last month's article for an explanation of the various signatures) and ends with the `Tag_End` byte, value `$61`. I won't give a blow by blow account of the file structure that we've already discussed (highlighted in the hexadecimal dump in Figure 1) because you had that last month. Rather, let's cut straight to the chase by starting with the byte at location `$0C`.

It just so happens that the zero byte here is a peculiarity of Delphi 2 DCU files. The Delphi 2 compiler always writes a string to this location, and that string is always empty! Therefore, you can guarantee that if the signature is `D2Magic`, then the byte at location `$0C` should be zero. If it isn't, then you've got a corrupt file. This empty string was only ever present in Delphi 2, and disappeared in subsequent versions of the compiler.

If you look at Listing 1, you'll see how I've modified last month's code to cater for this. The `FoundOne` routine now reads 16 bytes from the beginning of each file, and performs a special check for Delphi 2 signatures. If it's dealing with a Delphi 2 DCU file, then the byte at location `$0C` in the file (corresponding to the low byte of the fourth entry in the `Magic` array) is checked to see if it's zero. If it is not zero, then the program displays a message identifying the invalid DCU file and doesn't add this file to the list of available DCUs in the `TListView` control.

The DFK Tag Quartet

The next byte, `$70`, is another record tag, just like `Tag_End`. It's defined like this:

```
const
  Tag_DFK_Source = $70;
```

In case you are wondering if I'm making up these identifier names (`Tag_End`, `Tag_DFK_Source` etc), the answer is that I'm not! These are the real identifier names you'd see in the actual source code for the Delphi compiler (which, incidentally, is written in C). The only reason that I've seen these names is because I've poked around inside several versions of DCC32.EXE, some of which were debug builds containing some debugging code. I can promise you that I have never seen a shred of Borland's compiler source.

I have a strong hunch that DFK stands for *Delphi File Kind*, and thus `Tag_DFK_Source` introduces a record type which describes a source file. This is indeed the case, because (referring to the above hex dump) we can see that the tag byte is immediately followed by `squit.pas` encoded as a Pascal string, ie preceded by a length byte. Conceptually, the `Tag_DFK_Source` byte introduces the name of a Pascal source file which is required to build the current DCU file. You might therefore be forgiven for thinking that there will only be one of these particular tags per DCU file, but in fact that's not the case. When working with a large Pascal unit, programmers sometimes split the file into several smaller files (by convention, with file extensions of .INC) which are included into the main Pascal unit by using the `{$I filename}` compiler directive. If you do this, each of the .INC files will naturally be regarded as an essential source file by the compiler, and each of these files will be referenced through its own `Tag_DFK_Source` record. Consequently, multiple `Tag_DFK_Source` records could be present.

In practice, Pascal source files are not the only 'source' files used to create a DCU. Years ago, any techie worth his salt wasn't happy unless his Pascal units included a smattering of .OBJ files, introduced to the compiler by using the `{$L filename}` directive. As old timers will appreciate, early versions of Turbo Pascal didn't include a decent inline assembler and it was therefore often necessary to include external assembler code into Pascal units. These days, associating an .OBJ file with a Pascal unit is relatively uncommon, but when it is done you will find a `Tag_DFK_Object` record in the DCU file for each referenced .OBJ file.

```
const
  Tag_DFK_Object = $71;
```

Much more common in Delphi is the need to associate a .RES file with a .DCU, this being done through the `{$R filename}`. As you will no doubt be aware, .DFM files are merely .RES files which contain a single, streamed, `TForm` resource; Delphi form units therefore include a compiler directive which looks like this:

```
{$R *.DFM}
```

Given a form file called SetupDlg.pas, this will include a reference to the corresponding SetupDlg.dfm file into the resulting DCU file. As with Pascal source and .OBJ files, another special tag is used to indicate a resource file:

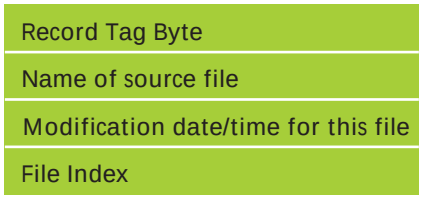
```
const
  Tag_DFK_Resource = $72;
```

There is one final tag value in what might loosely be called the 'DFK' tag quartet, and that's `Tag_DFK_TheAdr`. At this point, I have to confess that I don't know what this tag represents because, at the time of writing, I haven't spotted one inside a DCU file.

```
const
  Tag_DFK_TheAdr = $73;
```

So let's summarise. DCU files may contain four different 'DFK' record types, each of which represents a dependency needed to create that DCU file. The four record types describe dependencies on Pascal source files, .OBJ files, resource files, and *A N Other*, where the latter remains to be determined! Multiple copies of the same record type may be present in a DCU file because a particular unit might 'include' multiple .INC files, multiple .OBJ or .RES files, or whatever.

From this information, the automatic make system built into Delphi (actually, it's built right into the compiler DLL itself) is able to create a dependency list of all the files needed to manufacture the DCU. When you attempt to run a program from within the IDE, the compiler looks at every DCU file in the project, and for each DCU, walks the list of dependencies for that file, checking to see if the DCU needs to be rebuilt. Of course, this won't always be possible (you might not have the source code to a particular unit), but in those cases where the source is available, the compiler uses timestamp information to determine whether a particular source file is more recent than the DCU itself. If so, then the DCU is rebuilt.



➤ *Figure 2*

So where does this timestamp information come from? Each of the four DFK record types starts with a tag byte and, as we've seen, is then followed by the name of the source file in question. But that's not the end of the story. Immediately after the source filename comes a four-byte modification date/time which specifies when the file was built. It's this information which is used to determine whether or not the target DCU file needs to be recompiled. Finally, after the four-byte file modification time is another byte (actually it's not necessarily a byte: see later!) which you can think of as a file index. It's used to provide a file reference number by which this source file is subsequently identified within the DCU. Thus, each of the four DFK record types looks conceptually like Figure 2.

Putting It Into Practice

Take a look at Listing 2 which puts together everything that we've discussed so far. This is very much a 'delta' of last month's code, but you'll find complete, compilable sources on this month's companion disk.

Firstly, I've added a routine, `TreeListDblClick`, which is executed whenever you click on a displayed DCU pathname in the program's `TListView`. This routine begins by making absolutely certain that a particular `TListItem` is selected and, if not, exits stage left. Next, the full pathname of the relevant DCU file is retrieved from the list item, and used to create a

TFileStream object through which the DCU file is read into memory. Rather than messing about seeking backwards and forwards within the file, the entire DCU is loaded into memory. Well, let's face it, a DCU file is not that enormous compared to (say) a multi-megabyte graphics file. One of the biggest commonly encountered DCU files is WINDOWS.DCU which weighs in at around 400Kb: pretty small beer for today's modern hardware.

Once the DCU file has been loaded into memory and the file stream closed, the pointer p is positioned at the start of the file, or rather, at the start of the interesting data. Here again, we check for a Delphi 2 DCU and advance the pointer to skip over the null byte contained in such files. Next, the code goes into a loop, parsing the file one record at a time and case-ing out on the actual record type involved. Well, at least, that's the theory, but in practice we understand very few record types as yet. Thus, most of the time, the program will fall through into the else clause of the case statement, indicating that it has found an unknown tag value.

In fact, if you run the program as it stands on a Delphi 3 or Delphi 4 DCU file, the code won't even get as far as the DCUDumpDFKRecord call. In Delphi 2 DCU files, the 'DFK' records tend to immediately follow the zero-length string byte, but in later DCU files, they appear somewhat later.

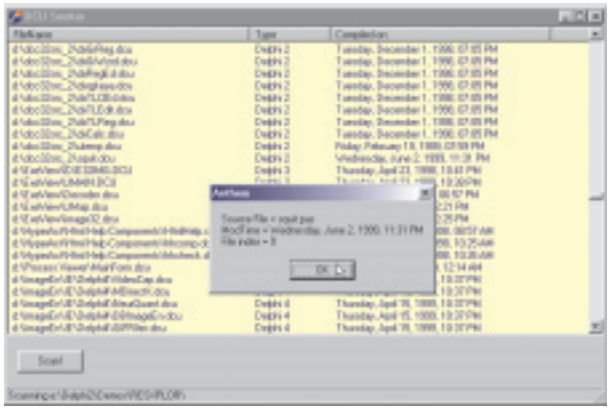
The idea here is that, as more DCU tag types are identified and documented, we'll be able to add more and more cases to the switch statement. In Listing 2, the code really only does anything worthwhile with the four DFK tags that we've discussed earlier. For each of these four possible tag values, the DCUDumpDFKRecord method is called. The first parameter is a string identifying the type of source file we're dealing with, and the second parameter allows us to pass the current pointer into the DCU image as a var parameter, the theory being that the pointer ends up pointing at the next record in the DCU image ready for the next trip round the while loop.

Moving on to DCUDumpDFKRecord, we see that it begins by calling a new routine called DCUReadString, which is intended to read a Pascal-style string from the current place in the DCU image, returning the string as the function result. As with higher-level routines, the DCUReadString code is written so as to pass the current pointer location as a var parameter, and the code has also been written so as to do the right thing when presented with an empty string. Using DCUReadString, the name of the source file is obtained and appended to the source type string passed as an argument to the routine.

Next, the code used a bit of pointer manipulation to obtain the next four bytes as a longint, taking care to update the pointer by four. This long int value is passed to the FileDateToDateTime and FormatDateTime routines, just as we did last month with the unit modification time found in the DCU file header. The result is likewise appended to the string, S, that we're building. Finally, the code calls another routine, DCUDecodeNum, in order to obtain the file index value for this source file. The result of all this is shown in Figure 3. Here, we've

➤ *Listing 2*

```
function TForm1.DCUReadString (var p: PChar): String;
var
  Len: Byte;
begin
  Result := '';
  Len := Ord (p^);  Inc (p);
  while Len <> 0 do begin
    Result := Result + p^;
    Inc (p);  Dec (Len);
  end;
end;
function  TForm1.DCUDecodeNum (var p: PChar): Integer;
const
  SizeNum: array [0..15] of Byte = ( 1, 2, 1, 3, 1, 2, 1, 4,
    1, 2, 1, 3, 1, 2, 1, 5 );
  ShiftNum: array [0..15] of Byte = ( 25, 18, 25, 11, 25,
    18, 25, 4, 25, 18, 25, 11, 25, 18, 25, 0 );
var
  Idx: Byte;
begin
  Idx := Ord (p^) and 15;
  Inc (p, SizeNum [Idx]);
  Result := PLongInt (p - 4)^ shr ShiftNum [Idx];
end;
procedure TForm1.DCUDumpDFKRecord(
  const Typ: String; var p: PChar);
var
  s: String;
  modtime: LongInt;
begin
  s := Typ + ' = ' + DCUReadString (p) + #10;
  try
    modtime := PLongInt (p)^;  Inc (p, 4);
    s := s + 'ModTime = ' +
      FormatDateTime('dddd, mmmm d, yyyy, hh:mm AM/PM',
      FileDateToDateTime(modtime)) + #10;
  except
  { Eat exceptions if modtime is invalid } ;
  end;
  s := s + 'File index = ' + IntToStr(DCUDecodeNum (p));
  ShowMessage (s);
end;
procedure TForm1.TreeListDblClick(Sender: TObject);
var
  Tag: Byte;
  Buff, p: PChar;
  Item: TListItem;
  fs: TFileStream;
begin
  Item := TreeList.Selected;
  if Item = Nil then
    Exit;
  fs := TFileStream.Create (Item.Caption, fmOpenRead);
  try
    GetMem (Buff, fs.Size);
    fs.Read (Buff^, fs.Size);
  finally
    fs.Free;
  end;
  // point at first byte of interest in DCU image
  p := Buff + 12;
  // Skip over Delphi 2's always-zero string
  if PLongInt(Buff)^ = D2Magic then
    Inc (p);
  try
    while True do begin
      Tag := Ord (p^);  Inc (p);
      case Tag of
        Tag_End          : Exit;  // All done!
        Tag_DFK_Source   : DCUDumpDFKRecord(
                             'Source File', p);
        Tag_DFK_Object   : DCUDumpDFKRecord(
                             'Object File', p);
        Tag_DFK_Resource : DCUDumpDFKRecord(
                             'Resource File', p);
        Tag_DFK_TheAdr   : DCUDumpDFKRecord(
                             'Tag_DFK_TheAdr ????', p);
        else begin
          ShowMessage(Format('Unknown tag $%x', [Tag]));
          Exit;
        end;
      end;
    end;
  finally
    FreeMem (Buff);
  end;
end;
```



➤ *Figure 3: With the code modifications provided this month, our testbed program can correctly interpret all the information which tells the Delphi compiler which source files are needed to build a particular DCU file.*

clicked on the SQUIT.DCU file, obtained by compiling the simple unit given at the beginning of this article, and our DCU testbed program has displayed information relating to the source file needed to build this DCU file. You'll notice that this entry has a file index of zero, which is what we'd expect for the one and only source file that's needed.

Incidentally, if you want to verify that things are working as I have described, you can try adding one or more 'dummy' (ie empty) include files to the SQUIT.PAS file using the `{$I filename}` compiler directive. It really doesn't matter whether you put these into the `interface` or `implementation` part of the unit. If you now recompile the unit (again, stick with Delphi 2 for now) and rerun the testbed program, you will find that information on all of the referenced source files is displayed, along with a file index that increments by one each time.

Numeric Encoding

OK, what is this weird `DCUDecodeNum` routine, and what do I mean when I say that the file index field of a DFK-type record is sort of a byte, and sort of not?

The unit that we've been examining so far is, of course, trivial. All this unit does is declare a single constant, `SomeNumber`, to which I've assigned an arbitrary (but easily recognisable) value. If you were to examine the resulting SQUIT.DCU file with your favourite debugging tool, you might be surprised to discover that the number `$55AA` does not actually appear anywhere within the file. Allowing for the way in which the Intel processor stores numbers, you won't find `$AA55` either. In fact, with this particular unit (compiled with the Delphi 2 command line compiler) you won't even find the single bytes `$55` or `$AA` anywhere within the DCU. So what's going on?

The short answer, of course, is that the aforementioned constant has been encoded in some way. This is not, as far as I know, a deliberate attempt by Borland to obfuscate the format of DCU files, although I may well be wrong here! Rather, the company has adopted a technique which significantly reduces the storage requirements of numeric constants in many cases. As we'll see, DCU files are heavily 'number-oriented' and by using this technique, DCU files can be made much more compact than they'd otherwise be.

The file index field in DFK-type records is a case in point. This is actually an integer value which has been encoded into a single byte, thus giving a saving of three bytes. Thus, although the file index appears in the DCU file as a simple byte, it's much more accurate to think of it as an encoded integer. For example, if you create a unit that includes another file, you'll find that (as mentioned above) the test-bed program reports that the second, included source file has a file index of 1. However, if you look

Free Pascal And Open Source Delphi

It is my fervent hope that by doing my best to publish what I've discovered about the DCU file format, I will provoke others to delve even deeper, until at some point we have a complete understanding of this important file format. Once this is achieved, it will make it possible to write utilities for 'upgrading' older DCU files to work with more recent versions of Delphi, and a lot more besides.

Eagle-eyed readers will have no doubt spotted the fact that I'm getting increasingly keen on the idea of open source software. You may or may not be aware of the fact that a well-respected Delphi clone called FPC (the Free Pascal Compiler) already exists, and can be used to create programs to run on a number of different processors and architectures, including 32-bit Windows and Linux. If you haven't encountered Free Pascal before, www.brain.uni-freiburg.de/~klaus/fpc is the official home page where you can find out more about the compiler, and the current state of the project. Be advised, though, that this isn't a particularly fast site, and if you want to download the software (either binaries or source) then you'd be better advised to go to one of the faster mirror sites which are linked from the FPC home page.

Because FPC is an open source project, details of the unit file format used by this compiler are freely available (see Figure 4) and you are positively encouraged to experiment with it, suggest enhancements and improvements and so forth. I'm specifically mentioning the FPC project here because it represents another good reason for getting a better understanding of DCUs. Wouldn't it be nice if FPC could be modified to create DCU files compatible with Delphi? Wouldn't it be great if you could take an existing Delphi DCU file and translate it into a form that was acceptable to the FPC linker? At the moment this is all pie-in-the-sky of course, but with even Microsoft hinting at making source code for parts of Windows freely available (it's amazing what concessions companies will make when the US government is closing in for the kill!) it can surely only be a matter of time.

at the actual bits and bytes in the file, you'll see that the number 1 is encoded into a byte with the value 2! In other words, you *must* use the `DCUDecodeNum` routine whenever an encoded integer is present. If you don't, you'll simply get nonsense results.

The `DCUDecodeNum` routine itself represents something of a descent into madness. It contains two arrays, `SizeNum` and `ShiftNum`. The first array codes for the size of the encoded data, while the second array represents the number of right shifts that are required in order to obtain the decoded result. Thus, if you imagine calling this routine with the argument `p` pointing at a zero byte, you can see that `Idx` will be set to zero. This will cause the first entry in the `SizeNum` array to be referenced, indicating (as we've already discovered) that the value zero encodes into a single byte. The corresponding entry in the `ShiftNum` array indicates that we need to take the preceding *four bytes*, treat them as a long integer and shift the result 25 places to the right. At first glance, this might seem bizarre, because there's only one byte there in the first place. The key point is that the other three bytes are filled with garbage that we're not interested in. By shifting 25 places to the right, the zero byte (which started off as the most significant byte) is shifted all the way down to the least significant byte, and we get a final result of zero.


A.3 The Header

The header consists of a record containing 24 bytes:

type
  TpuHeaderPacked = packed record
    Id   : array[0..3] of Char; { = 'PPU' }
    Ver   : array[0..3] of Char;
    Compiler : Word;
    Cpu   : Word;
    Target   : Word;
    Flags   : LongInt;
    Size   : LongInt; { size of the ppufile without header }
    Checksum : LongInt; { checksum for this ppufile }
  end;

The header is already read by the ppufile.pas constructor. You can access all fields using ppufile.header which holds the current header record.

field	description
id	this is always 'PPU'. can be checked with function ppufile.CheckPPUId:boolean;
ver	ppu version. currently '0.3'. can be checked with function ppufile.GetPPUVersion:longint; (returns 17)
compiler	compiler version used to create the unit. Doesn't contain the patchlevel. Currently 0.99 where 0 is the high byte and 99 the low byte
cpu	cpu for which this unit is created. 0 = i386,1 = m68k
target	target for which this unit is created, this depends also on the cpu! For i386: 0 Go32v1, 1 Go32v2, 2 Linux-i386, 3 OS/2, 4 Win32. For m68k: 0 Amiga, 1 Mac68k, 2 Atari, 3 Linux-m68k
flag	the unit flags, contains a combination of the uf_ constants which are defined in ppu.pas
size	size of this unit without this header
checksum	checksum of the interface parts of this unit, which determine if a

Figure 4: The Free Pascal compiler is an example of an open source Delphi clone which will run under Windows 32, Linux and other platforms. The comprehensive documentation includes full details on the file format of the unit files generated by Free Pascal.

```
In the range $00..$7F                 ➜     Encodes as a single byte
In the range $80..$3FFF               ➜     Encodes as a word (2 bytes)
In the range $4000..$1FFFFF           ➜     Encodes as three bytes
In the range $200000..$FFFFFFF        ➜     Encodes as four bytes
In the range $10000000..$FFFFFFFF     ➜     Encodes as five bytes
```

Figure 5

Now let's look at the situation for (say), the second `Tag_DFK_Source` record in a DCU file. As already indicated, the file index for this record will be encoded with the value 2. This gives a value of 2 for the `Idx` variable, and (as before) the pointer, `p`, will only be incremented by a single byte. However, since we're shifting by 25 places rather than by 24 (which would be exactly 3 bytes worth) the net effect is to not only get the most significant byte into the least significant position, but also to apply an initial division of 2. Thus 2 gets transmogrified into 1 which is what we'd expect to be the file index value in this case.

Figure 5 shows a quick overview of the 'range' of encodable values and the encoded byte size in each case.

Note that this compression scheme (for such it really is, of course) only really breaks down for very large numbers greater than `$10000000`. In such cases, Borland effectively use five bytes to encode a four-byte number. The 32-bit number is placed into the DCU file image, but preceded by a special 'escape' byte (`$0F`) which indicates to the `DCUDecodeNum` routine that the following four bytes should be taken 'as is'. If you work through the `DCUDecodeNum` routine once again, imagining that `p` is pointing at the value `$0F`, you will see what I mean: the pointer is advanced by five bytes and no shift is applied to the 32-bit quantity following the 'escape' byte.

Don't worry if the above leaves you reeling, it's not essential to an understanding of DCU file internals. Suffice to say that numeric constants within many different types of DCU record are encoded using this scheme, and that the `DCUDecodeNum` routine must always be used in such cases to recover the original value. Interestingly, within the actual C source code for the compiler, the `DCUDecodeNum` routine is implemented as a preprocessor macro in order to improve performance, but that's one luxury we don't have. Maybe in Delphi 5..., but probably not.

Conclusions

This month, I've given you the detailed structure of four different types of record within a DCU file. More importantly, I've structured the testbed program into a form whereby we can easily add new handlers for different types of record tags. Perhaps most importantly of all, I've described the numeric encoding scheme which is used to store numbers within a DCU file. Next month, we'll use this foundation as a basis for looking at several other types of record with a DCU file.

In case you are wondering, after August I will be moving onto other topics in this column, but I shall return to the DCU file format as and when I (or others) have managed to extract more useful information!

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