Beating the System: Deciphering The DCU, Part 3

by Dave Jewell

In last month's column, I gave further details of the DCU file format and I also explained the operation of the encoding scheme which Borland use to compress numeric data within the file. This time round, we'll take up the story directly where we left off. Just to refresh your memory, I've reproduced last month's hexadecimal dump of that very simple Pascal unit we were examining, see Figure 1. Based on the information in the last two instalments of this column, you should now understand the purpose of all the bytes that are shown in red.

➤ *Figure 1*

```
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
```

Dissecting USES

The next tag encountered has a value of `$64`. It's defined as shown below:

```
const
  Tag_Int_Use = $64;
```

`Tag_Int_Use` introduces a (potentially complex, hierarchical) record which essentially encapsulates all the types, symbols, procedures, etc, that are *imported* by a unit and referenced within the `interface` part of that unit. Thus, for example, if you were to define a variable, `Fred`, of type `Integer` within the `interface` part of a unit, then this would have a `Tag_Int_Use` clause which (amongst other things) would indicate that the type `Integer` was being imported from the `System` unit. You might not expect that low-level built-in types such as `Integer`, `Boolean`, etc have to be explicitly imported from the `System` unit but, behind the scenes, that's exactly what happens.

In actual fact, our 'dummy' unit doesn't import anything from the `System` unit, but it references `System` nevertheless. Incidentally, if you want to build a DCU file parser, you should be aware that `System` isn't the only unit which can appear with a 'null' `Tag_Int_Use` record. If you add `Dialogs` (for example) to the list of units in a `uses` clause, but then neglect to reference anything from the `Dialogs` unit, you'll probably end up with an empty `Tag_Int_Use` record for the `Dialogs` unit in the resulting DCU file.

If you think about it, things have to work this way. Why? Because, in the past, we've all built 'faceless' units which contain nothing in their `interface` clause but which nevertheless perform behind-the-scenes magic simply through the mere act of being included in a program, the unit's `initialization` and `finalization` clauses being executed at the appropriate time to do the real work. Thus, you'll understand that even though nothing appears to be referenced from a unit, the presence of the unit in the `uses` clause cannot be ignored by the compiler.

If you examine the hex dump in Figure 1, you'll see that the tag byte is followed by the name of the unit (`System`) and then by a double-word date/time stamp which provides versioning information as described last month. Next comes information on any referenced identifiers present in the preceding unit name, in this case none. Finally, the record is terminated by `Tag_End_Record`, another special tag whose job is to mark the end of the record and 'pop' us back up to the previous level.

```
const
  Tag_End_Record = $63;
```

This is where we really come up against a limitation of our dummy unit because it's just giving us an empty `Tag_Int_Use` record, we need something meatier to look at! If we were to define (for example) an integer variable in the interface part of our unit, then the `Tag_Int_Use` record might look like that shown in Listing 1.

```
001D    64                         ; tag = TAG_INT_USE
001E    06 53 79 73 74 65 6D       ; references 'SYSTEM' unit
0025    00 00 00 00                ; date/time stamp

0029    66                         ; tag = TAG_TYPE_USE
002A    07 49 6E 74 65 67 65 72    ; references 'Integer' type
0032    FA 6C 80 42                ; -- magic --

0036    63                         ; tag = Tag_End_Record
```

➤ *Listing 1*

This hex dump introduces you to another new tag, `Tag_Type_Use`, (value `$66`) which has a fairly obvious meaning, it specifies that a particular type is being referenced. In looking at the record layout given above, you need to appreciate that the unit name can potentially be followed by multiple `Tag_Type_Use` sub-records, and that these can then be followed by the name of another unit which introduces another bunch of sub-records containing references to the second unit name, and so on.

As explained above, the `Tag_Int_Use` tag essentially groups

together all the references to information in other units that are being used within the `interface` part of the current unit. However, what happens if you employ other types and variables in the `implementation` part of your unit? In order to cope with this, Borland defined another tag, `Tag_Imp_Use`.

```
const
  Tag_Imp_Use = $65;
```

The format of this record is exactly the same as for `Tag_Int_Use`. Once again, it can potentially be made up of a number of references to other units, each of which is further divided up into sub-records that define the individual information.

At this point, you're probably pondering the meaning of the 'magic' field in the above hex dump, which I've indicated is part of the `Tag_Type_Use` record. I'll come to this later in this month's instalment. In addition to `Tag_Type_Use`, there's another very similar record type, `Tag_Sym_Use`, which is used to refer to symbols defined in another module.

```
const
  Tag_Sym_Use = $67;
```

Procedure Declaration

There are a host of other record types that can occur within a DCU file. Let's look at some of the more interesting ones. Firstly, there's the `Tag_Proc` tag which, as the name suggests, identifies a procedure defined within the DCU file:

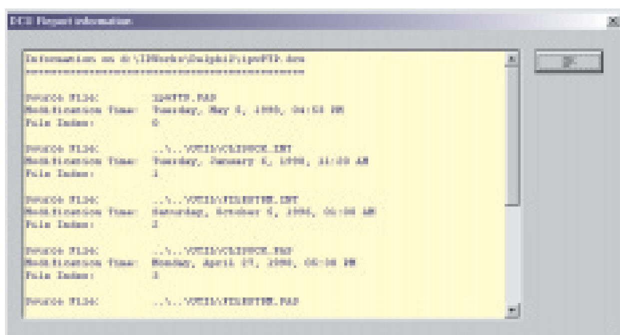
```
const
  Tag_Proc = $28;
```

As with most tags, this is immediately followed by the name of the procedure. Next comes an integer (encoded as I described last month) which represents a set of bit flags. The defined bit flags are shown in Table 1.

The meaning of most of these fields will become clear as you examine the innards of one or more DCU files, comparing the source code with what gets generated by the compiler. For example, the `System` unit contains a global variable called `hInstance` which, as all good Delphi programmers know, is an instance handle for the currently running program. This instance handle can be used to (for example) retrieve resource data from your application's EXE file using the somewhat baroque API calls provided by Microsoft.

If you locate the 'declaration' for `hInstance`, you'll find that the `Value` flag is set, as well as `Assignable`, meaning that `hInstance` is an assignable variable, although you'd be ill-advised to change its value! The `Memory` bit flag is set, indicating that this variable resides in memory rather than in a register, and the `Address` and `Exported` bit flags will also be set. The `Exported` flag is perhaps the most important, indicating that this symbol is being 'exported' from this unit. Be careful not to confuse this meaning of 'exported' with the `export` keyword which, in Delphi, is used to export functions from a dynamic link library.

Hang on a minute, Dave: I thought you were talking about the `Tag_Proc` record type? Is `hInstance` a procedure or is it a variable? Well, it's obviously the latter, but the bit flags described in Table 1 tend to crop up in many different contexts within a DCU file, as we shall see: constants, variables, types and procedure all have the same set of bit flags.



➤ *Figure 2: Some DCUs contain many dependencies on other files. As you can see, this one (taken from the popular IPWorks package) includes at least two other .INT files and two other .PAS files.*

Bit Flag	Meaning
$01	Value
$02	Assignable
$04	Constant
$08	Register
$10	Memory
$20	Address
$40	Exported
$80	Link or Qualifier

➤ *Table 1*

Following the flags field, the `Tag_Proc` record then continues with a four-byte magic number, similar to the magic number used by the `Tag_Type_Use` record. These magic numbers are always unencoded (ie a straight four-byte quantity within the DCU file). As we work through this month's code, you'll see that pretty well everything inside the DCU file also has a magic number, *provided* that the item is exported. And that's the key to what this magic number is. I'm not totally sure how this works, but I believe the magic number is some sort of auto-generated value (possibly derived from running a hashing function on the identifier name) which is used to identify symbols internally. In other words, rather than continually referring to symbols by name, it's more convenient to reference them through a 32-bit identification number, the magic value for `Integer` being `$42806CFA` under Delphi version 2.0. You can think of these magic numbers as the DCU equivalent of a COM object's GUID!

After the procedure's magic number come three encoded integers. I'm not sure what the first of these represents, but the second corresponds to the amount of generated code (in bytes) required for the procedure, whereas the third integer represents the function result type, if any. If we're dealing with a procedure rather than a function, this integer is still present, but is mapped to what might be defined as a 'null' type.

Following this is a list of sub-records that define all the parameters required by the procedure, together with the various local variables defined in the procedure. The `Tag_Proc` record is then terminated by a `Tag_End_Record` marker, which we've already encountered. Thus, you can see that sub-records may appear in several different contexts (we've already seen them inside `Tag_Int_Use` and `Tag_Imp_Use`) and a hierarchical record constructed in this way ends with a `Tag_End_Record` byte.

Each parameter to a procedure is introduced by means of a `Tag_Param` record:

```
const
  Tag_Param = $21;
```

The tag byte is followed, in the usual way, by a string giving the name of the parameter. There then follows an encoded integer containing the flag bits previously described. Because parameters of exported routines are not themselves 'exported', there is no accompanying magic number in a `Tag_Param` record. However, you'll often find that the `Register` bit flag is set for `Tag_Param` records because, as you'll probably appreciate, the default calling convention in Delphi is to pass arguments via registers. Let's try and put this into concrete terms. Suppose we define a function `Add`, like this:

```
function Add(x, y: Integer):
  Integer;
```

This will result in the generation of two `Tag_Param` sub-records (within the enclosing `Tag_Proc` record) that looks like Listing 2.

```
Param:      x
ParamFlags: $0000000B[value, assignable, reg]
ParamType:  1
ParamLoc    0
Param:      y
ParamFlags: $0000000B[value, assignable, reg]
ParamType:  1
ParamLoc    1
```

➤ *Listing 2*

The last two fields in each `Tag_Param` record are used to define the type and location of the parameter. The first of these, `ParamType`, is used to encode the type of the parameter in some way that I don't yet understand, probably using an internal table of types that are constructed for each DCU file on a per-unit basis. The meaning of the second field, `ParamLoc`, is more obvious and indicates either the assigned register number (for register-based parameters) or the stack frame offset (for stack-based parameters). The EAX register is encoded as zero, the EDX register as 1, and the ECX register as 2, these are the order in which registers are assigned when using the default calling conventions. Thus, imagine a function declaration like:

```
function Add(x, y, z, p, q:
  Integer): Integer;
```

In such a case, the `x`, `y`, `z` parameters will all have the `Register` bit flag set, and they'll have `ParamLoc` values of `0`, `1`, `2` respectively. However, the `p`, `q` parameters must be pushed on the stack, and they will therefore have the `Memory` bit flag set instead. The `p` parameter will have a `ParamLoc` value of `12`, corresponding to a stack frame offset of 12 bytes, whereas `q` will have a stack frame offset of 8. Thus, if you were to peek at the generated code, you'd see these parameters being addressed as [bp+12] and [bp+8].

Clearly, all of this information has to be 'up front' in the interface part of a unit. If you have a unit `X`, which calls routines in unit `Y`, the code generator within the compiler needs to know how to call all the exported routines in unit `Y`'s code during the compilation of `X`. The code generator can't 'see' the code corresponding to the exported routines, but it can see all the `Tag_Proc` records belonging to unit `Y`, and the information in these is used to generate the calling code sequences.

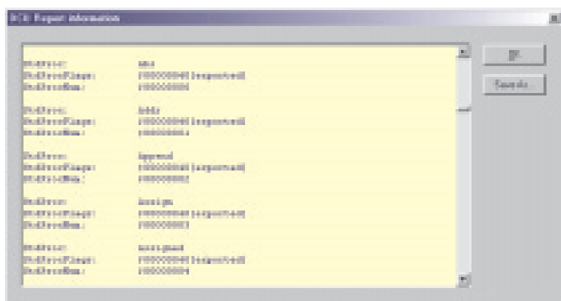
Understanding Standard Procedures

As you'll no doubt appreciate, the Delphi language contains many built-in 'intrinsic' routines such as `Abs`, `Addr`, `Assigned`, `Chr`, `Ord` and so on. From the viewpoint of a programmer, these look just like ordinary routines, and you might imagine that they all result in a call on the runtime library. Funnily enough, *none* of the five routines I mentioned map onto a call. Rather, you can think of them as the equivalent of C++ macros or inline functions: no subroutine call is involved. The correct term for these routines is 'standard procedures'. Some standard procedures map onto a runtime library call and some don't. `Random`, `Assign`, `Insert`, `Delete` are all examples of standard procedures that do involve a library call.

In order to implement standard procedures in Delphi Pascal, Borland implemented each of them as a pseudo-entry within the `System` unit, and this is one of the reasons why `System` plays such a crucial role in Delphi. You might be surprised to know that, in Delphi 2, there are no less than ninety-one of these standard procedure entries in SYSTEM.DCU! You can see some of them in Figure 3.

Each standard procedure definition is introduced by a special tag called `Tag_StdProc`:

➤ *Figure 3: Here's a view of some of the standard procedure definitions inside SYSTEM.PAS. Many of these aren't actually procedures at all, in the real sense of the word, but this magic is needed in order to keep the front-end parser happy!*



```
const
  Tag_StdProc = $29;
```

As far as I know, only the `SYSTEM` unit contains these special entries and I suspect that there's no way of getting the standard Delphi compiler to generate them. I seem to remember reading somewhere that it was necessary to hand-assemble the SYSTEM.DCU file for each new release of the development system, and I wouldn't be surprised if this wasn't still the case.

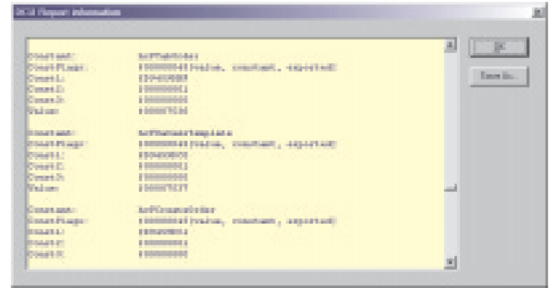
Internally, the structure of a `Tag_StdProc` record is very simple; the name of the standard procedure follows the tag byte, and this in turn is followed by a standard set of flags as previously described. Next comes a procedure number by which the standard procedure is identified, and that's it! Within the compiler, whenever a standard procedure is referenced, the compiler essentially references this internal procedure name, using it to decide how to handle a particular standard procedure. Some standard procedures, notably `Read`, `Readln`, `Write` and `Writeln` require a substantial amount of extra code to parse, and may require calls to several runtime library routines.

Updating Anthem

Time to put all of this together! Since last month's article, I've made a number of changes to Anthem, the DCU scanning and browsing utility that we've been playing with over the last couple of months. In particular, I hived off all the low-level DCU 'sniffing' code into a separate form, `TReportForm`, which uses a `TMemo` component to provide a scrolling view of an arbitrarily large chunk of data. You can see a typical view in Figure 2.

`TReportForm` is contained in a unit called REPORT.PAS, the source code to which is shown in Listing 3. REPORT.PAS in turn uses a small file, DCUDEFS.PAS, in which I've placed all the tag values that have been discussed so far: see Listing 4. If you look carefully, you'll see that Anthem uses a few tags that I haven't discussed so far, one of which is `Tag_DLL_Import`. This tag is used to describe symbols that have been imported from a DLL. For obvious reasons, you'll most often find this tag used when peeking inside the WINDOWS.DCU file. In fact, if you use Anthem to load up the Delphi 2 version of WINDOWS.DCU, you'll probably think the program has crashed: it hasn't, but it will take it several seconds to digest the huge number of Windows API calls that are provided through the `Tag_DLL_Import` record, the format of which is identical to `Tag_Int_Use` and `Tag_Imp_Use`. When examining DLL declarations stored in a `Tag_DLL_Import` record, you'll also notice that the corresponding 'magic' field is always zero, because of course this is only applicable when resolving symbols between different DCU files.

The REPORT.PAS code is based around a few simple routines, `PutStr`, `PutField`, etc, which add lines of text to the `TMemo` component. I've also added a `Save As...` button so that you can optionally write a DCU information dump out to a text file. The real work is done inside the `FormShow` method which works its way through the DCU file until an unrecognised tag is encountered, at which point the program bottles out just like last month's code. However, I've enhanced things slightly by printing out the hexadecimal file offset at which the unknown tag was found, thus making it easier to (for example) fire up a hex file editor and see the offending data for yourself.

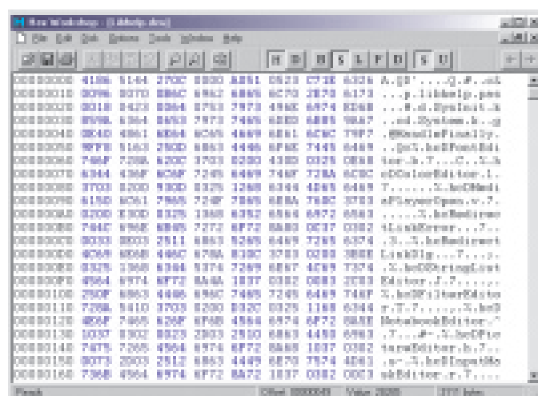


➤ *Figure 4: Despite our inability to recognise all tags, Anthem can still do useful work. If you want to create specialised property editors but use predefined help contexts in the IDE help file, a peek inside LibHelp.DCU will show you the magic numbers required.*

Compatibility Issues: Delphi 3, 4... And 5!

If you've been following through this series, and trying out Anthem with various DCU files, you'll know that previous versions of the program weren't terribly happy with Delphi 3 and 4 files. As I stressed last time round, this *doesn't* mean that the format of Delphi 3 and 4 DCUs is fundamentally different, because it isn't. My understanding was that the problem was caused by unrecognised tags right at the beginning of the DCU file, immediately after the 12-byte header.

In fact, I was wrong. Another keen DCU investigator (who wishes to remain anonymous!) told me that Delphi 3 and 4 DCUs store a 32-bit quantity at location `$0C` in the DCU, effectively adding 4 more bytes to the fixed 12-byte header we've discussed in previous months. Initially, I thought he



➤ *Figure 5: If you want to do any serious DCU investigation of your own, you'll need a decent hexadecimal editor such as Hex Workshop, shown here. By now, you should be able to recognise all the various tags and record types in this DCU file!*

➤ *Facing page: Listing 3*

```
unit Report;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, DCUDefs;
type
  TReportForm = class(TForm)
    OKButton: TButton;
    Info: TMemo;
    SaveDialog1: TSaveDialog;
    Button1: TButton;
    procedure FormShow(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    p: PChar;
    Buff: PChar;
    Unknown: Boolean;
    Version: TDCUVersion;
    procedure PutStrUnderlined (const S: String);
    procedure PutStr (const S: String);
    procedure PutField (const Name, Val: String);
    function  DCUReadString: String;
    function  DCUDecodeNum: Integer;
    procedure DCUUnknown (Tag, Offset: Integer);
    procedure DCUDumpDFKRecord (const Typ: String);
    procedure DCUTypeSymUse (const Typ: String);
    procedure DCUDumpUsesRecord (const Typ: String);
    procedure DCUProcDeclaration;
    procedure DCUStdProcDeclaration;
    function  DCUGetSymFlags (Flags: Integer): String;
    procedure DCUParamDeclaration;
    procedure DCUVariableDeclaration;
    procedure DCUConstDeclaration;
    procedure DCUTypeDeclaration;
    procedure DCUVMTDeclaration;
    procedure DCUTypedConstantDeclaration;
    procedure DCUThreadVarDeclaration;
    procedure DCUPutMagic (Flags: Integer);
    procedure DCUIncrementLevel;
    procedure DCUDecrementLevel;
    procedure DCUUnitFlags;
  public
  end;

implementation

{$R *.DFM}

procedure TReportForm.PutStr (const S: String);
begin
  Info.Lines.Add (S);
end;

procedure TReportForm.PutStrUnderlined (const S: String);
var
  Str: String;
begin
  PutStr (S);
  Str := '';
  while Length (Str) < Length (S) do
    Str := Str + '=';
  PutStr (Str);
  PutStr ('');
end;

procedure TReportForm.PutField (const Name, Val: String);
const
  Offset = 20;
var
  S: String;
begin
  S := Name;
  while Length (S) < Offset do
    S := S + ' ';
  PutStr (S + Val);
end;

procedure TReportForm.DCUUnknown (Tag, Offset: Integer);
begin
  Unknown := True;
  PutField('Unknown tag:', Format ('$%x at offset $%x',
    [Tag, Offset]));
end;

function TReportForm.DCUReadString: 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  TReportForm.DCUDecodeNum: 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;

function TReportForm.DCUGetSymFlags (Flags: Integer):
  String;
begin
  Result := '[';
  if (Flags and 1) <> 0 then
    Result := Result + 'value, ';
  if (Flags and 2) <> 0 then
    Result := Result + 'assignable, ';
  if (Flags and 4) <> 0 then
    Result := Result + 'constant, ';
  if (Flags and 8) <> 0 then
    Result := Result + 'reg, ';
  if (Flags and 16) <> 0 then
    Result := Result + 'mem, ';
  if (Flags and 32) <> 0 then
    Result := Result + 'adr, ';
  if (Flags and 64) <> 0 then
    Result := Result + 'exported, ';
  if (Flags and 128) <> 0 then
    Result := Result + 'link or qual, ';
  if Length (Result) > 1 then
    SetLength (Result, Length(Result) - 2);
  Result := Result + ']';
end;

procedure TReportForm.DCUParamDeclaration;
var
  Flags: Integer;
begin
  PutField ('Param:', DCUReadString);
  Flags := DCUDecodeNum;
  PutField ('ParamFlags:', '$' + IntToHex (Flags, 8) +
    DCUGetSymFlags (Flags));
  PutField ('ParamType:', IntToStr (DCUDecodeNum));
  PutField ('ParamLoc', IntToStr (DCUDecodeNum));
  PutStr ('');
end;

procedure TReportForm.DCUTypedConstantDeclaration;
var
  Flags: Integer;
begin
  PutField ('TypedConstant:', DCUReadString);
  Flags := DCUDecodeNum;
  PutField('ParamFlags:', '$' + IntToHex(Flags, 8) +
    DCUGetSymFlags(Flags));
  DCUPutMagic (Flags);
  PutField('typedconst1:', '$'+IntToHex(DCUDecodeNum,8));
  PutField('typedconst2:', '$'+IntToHex(DCUDecodeNum,8));
  PutStr ('');
end;

procedure TReportForm.DCUConstDeclaration;
var
  Flags: Integer;
begin
  PutField ('Constant:', DCUReadString);
  Flags := DCUDecodeNum;
  PutField('ConstFlags:', '$' + IntToHex(Flags, 8) +
    DCUGetSymFlags(Flags));
  PutField('Const1:', '$' + IntToHex(PLongInt (p)^, 8));
  Inc (p, 4);
  PutField ('Const2:', '$' + IntToHex (DCUDecodeNum, 8));
  PutField ('Const3:', '$' + IntToHex (DCUDecodeNum, 8));
  PutField ('Value:', '$' + IntToHex (DCUDecodeNum, 8));
  PutStr ('');
end;

procedure TReportForm.DCUIncrementLevel;
begin
  PutStr ('Increment Level:');
  PutStr ('');
end;

procedure TReportForm.DCUDecrementLevel;
begin
  PutStr ('Decrement Level:');
  PutStr ('');
end;

procedure TReportForm.DCUUnitFlags;
begin
  PutField ('Unit Flags:', 'Flags = $' +
    IntToHex(DCUDecodeNum, 8));
  if Version in [D4, D5, B3] then
    PutField ('Unit Flags:', 'Priority = $' +
      IntToHex (DCUDecodeNum, 8));
  PutStr ('');
end;

procedure TReportForm.DCUPutMagic (Flags: Integer);
begin
  // Magic is only present for exported symbols.
  if (Flags and 64) <> 0 then begin
    PutField ('Magic:', '$' + IntToHex (PLongInt (p)^, 8));
    Inc (p, 4);
  end;
end;

procedure TReportForm.DCUThreadVarDeclaration;
var
  Flags: Integer;
begin
  PutField ('ThreadVar:', DCUReadString);
  Flags := DCUDecodeNum;
```

{ *** CONTINUED ON NEXT PAGE *** }

```
{ *** CONTINUED FROM PREVIOUS PAGE *** }
  PutField('ThreadVarFlags:', '$' + IntToHex(Flags, 8) +
    DCUGetSymFlags(Flags));
  DCUPutMagic (Flags);
  PutField('threadvar1:', '$'+IntToHex(DCUDecodeNum,8));
  PutField('threadvar2:', '$'+IntToHex(DCUDecodeNum,8));
  PutStr ('');
end;
procedure TReportForm.DCUVariableDeclaration;
var Flags: Integer;
begin
  PutField ('Variable:', DCUReadString);
  Flags := DCUDecodeNum;
  PutField ('VarFlags:', '$' + IntToHex (Flags, 8) +
     DCUGetSymFlags (Flags));
  DCUPutMagic (Flags);
  PutField ('VarType:', '$' + IntToHex (DCUDecodeNum, 8));
  PutField ('VarLoc:', '$' + IntToHex (DCUDecodeNum, 8));
  PutStr ('');
end;
procedure TReportForm.DCUTypeDeclaration;
var
  Flags: Integer;
begin
  PutField ('Type:', DCUReadString);
  Flags := DCUDecodeNum;
  PutField ('TypeFlags:', '$' + IntToHex (Flags, 8) +
    DCUGetSymFlags (Flags));
  DCUPutMagic (Flags);
  PutField ('type1:', '$' + IntToHex (DCUDecodeNum, 8));
  PutStr ('');
end;
procedure TReportForm.DCUVMTDeclaration;
var Flags: Integer;
begin
  PutField ('VMT:', DCUReadString);
  Flags := DCUDecodeNum;
  PutField ('VMTFlags:', '$' + IntToHex (Flags, 8) +
    DCUGetSymFlags (Flags));
  DCUPutMagic (Flags);
  PutField ('vmt1:', '$' + IntToHex (DCUDecodeNum, 8));
  PutField ('vmt2:', '$' + IntToHex (DCUDecodeNum, 8));
  PutStr ('');
end;
procedure TReportForm.DCUStdProcDeclaration;
var Flags: Integer;
begin
  PutField ('StdProc:', DCUReadString);
  Flags := DCUDecodeNum;
  PutField ('StdProcFlags:', '$' + IntToHex (Flags, 8) +
    DCUGetSymFlags (Flags));
  PutField('StdProcNum:', '$'+IntToHex(DCUDecodeNum,8));
  PutStr ('');
end;
procedure TReportForm.DCUProcDeclaration;
var Flags: Integer;
begin
  PutField ('Procedure:', DCUReadString);
  Flags := DCUDecodeNum;
  PutField ('Proc Flags:', '$' + IntToHex (Flags, 8) +
    DCUGetSymFlags (Flags));
  DCUPutMagic (Flags);
  PutField ('proc1:', IntToStr (DCUDecodeNum));
  PutField('Code Size:', IntToStr(DCUDecodeNum)+' bytes');
  PutField ('ResultType:', IntToStr (DCUDecodeNum));
  PutStr ('');
  while not Unknown do begin
    Tag := Ord (p^);  Inc (p);
    case Tag of
      Tag_End_Record :   break;  // All done!
      Tag_Param      :   DCUParamDeclaration;
      Tag_Variable   :   DCUVariableDeclaration;
    else
      DCUUnknown (Tag, p - Buff - 1);
    end;
  end;
end;
procedure TReportForm.DCUDumpUsesRecord (const Typ: String);
var
  S, UnitName: String;
  modTime: LongInt;
begin
  PutStrUnderlined (Format ('USES (%s)', [Typ]));
  UnitName := DCUReadString;
  PutField ('UnitName:', UnitName);
  modtime := PLongInt (p)^;  Inc (p, 4);
  if modtime = 0 then
    S := '00000000'
  else
    try
      S := FormatDateTime('dddd, mmmm d, yyyy, hh:mm AM/PM',
        FileDateToDateTime (modtime));
    except
      { Eat exceptions if modtime is invalid } ;
    end;
  PutField('Modification Time:', S);
  while not Unknown do begin
    Tag := Ord (p^);  Inc (p);
    case Tag of
      Tag_End_Record :   break;  // All done!
      Tag_Type_Use   :   DCUTypeSymUse ('Used Type:');
      Tag_Sym_Use    :   DCUTypeSymUse ('Used Symbol:');
    else
      DCUUnknown (Tag, p - Buff - 1);
    end;
  end;
  PutStr ('');
end;
procedure TReportForm.DCUTypeSymUse (const Typ: String);
var TypName: String;
begin
  TypName := DCUReadString;
  PutField(Typ, TypName+' (Magic: $'+
    IntToHex(PLongInt(p)^,8)+')');
  Inc(p, 4);
end;
procedure TReportForm.DCUDumpDFKRecord (const Typ: String);
var
  modtime: LongInt;
begin
  PutField (Typ + ':', DCUReadString);
  try
    modtime := PLongInt (p)^;  Inc (p, 4);
    PutField ('Modification Time:', FormatDateTime(
      'dddd, mmmm d, yyyy, hh:mm AM/PM',
      FileDateToDateTime(modtime)));
  except
    { Eat exceptions if modtime is invalid } ;
  end;
  PutField ('File Index:', IntToStr (DCUDecodeNum));
  PutStr ('');
end;
procedure TReportForm.FormShow(Sender: TObject);
var
  fs: TFileStream;
begin
  fs := TFileStream.Create (Caption, fmOpenRead);
  try
    PutStrUnderlined(Format('Information on %s',[Caption]));
    Caption := 'DCU Report information';
    GetMem (Buff, fs.Size);
    fs.Read (Buff^, fs.Size);
  finally
    fs.Free;
  end;
  if Buff <> Nil then try
    p := Buff;
    // Get version number in an easily usable form
    case PLongInt (p)^ of
      D2Magic :  Version := D2;
      D3Magic :  Version := D3;
      D4Magic :  Version := D4;
      D5Magic :  Version := D5;
      B3Magic :  Version := B3;
    end;
    // point at first byte of interest in DCU image
    Inc (p, 12);
    // If this isn't a Delphi 2 file, then there's an
    // unknown 32-bit field to skip..
    if Version <> D2 then
      Inc (p, 4);
    // Now skip the ever-empty string field
    DCUReadString;
    while not Unknown do begin
      Tag := Ord (p^);  Inc (p);
      case Tag of
        Tag_End          : break;     // All done!
        Tag_Int_Use      : DCUDumpUsesRecord('Interface');
        Tag_Imp_Use      : DCUDumpUsesRecord(
                             'Implementation');
        Tag_DLL_Import   : DCUDumpUsesRecord('DLL Import');
        Tag_DFK_Source   : DCUDumpDFKRecord('Source File');
        Tag_DFK_Object   : DCUDumpDFKRecord('Object File');
        Tag_DFK_Resource : DCUDumpDFKRecord(
                             'Resource File');
        Tag_DFK_TheAdr   : DCUDumpDFKRecord(
                             'Tag_DFK_TheAdr ????');
        Tag_Proc         : DCUProcDeclaration;
        Tag_StdProc      : DCUStdProcDeclaration;
        Tag_Const        : DCUConstDeclaration;
        Tag_VMT          : DCUVMTDeclaration;
        Tag_Type         : DCUTypeDeclaration;
        Tag_StructConst  : DCUTypedConstantDeclaration;
        Tag_Variable     : DCUVariableDeclaration;
        Tag_ThreadVar    : DCUThreadVarDeclaration;
        Tag_Unit_Flags   : DCUUnitFlags;
        Tag_Inc_Level    : DCUIncrementLevel;
        Tag_Dec_Level    : DCUDecrementLevel;
      else
        DCUUnknown (Tag, p - Buff - 1);
      end;
    end;
  finally
    FreeMem (Buff);
  end;
end;
procedure TReportForm.Button1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then
    Info.Lines.SaveToFile(SaveDialog1.FileName);
end;
end.
```

was in error but, on rechecking the code in DCC32.EXE, he turned out to be correct. Moreover, the empty string at location `$0C`, which I'd supposed only to be present in Delphi 2 DCU files is *always* there, but it gets shifted down to location `$10` in files after version 2.0 because of the preceding 32-bit field I've just alluded to.

To summarise then: all 32-bit DCU files begin with a magic signature, file length and 32-bit timestamp. For versions of Delphi after 2.0, this is then followed by a mystery 32-bit number, whose significance isn't yet understood. All versions then have an empty string, which equates to a zero byte, and this is then followed by the tags proper.

While on the subject of minor incompatibilities, you'll notice that one of the tags handled by the updated Anthem program is called `Tag_Unit_Flags`. This record didn't exist at all in Delphi 2, but is used in Delphi 3 onwards. The format of the record, which consists of either one or two encoded integers, is version-sensitive, because the second, priority, field didn't exist before Delphi 4. The code in Listing 3 has been modified to reflect these minor tweaks, and it now returns useful data from Delphi 2, 3 and 4 DCU files. However, do bear in mind that our investigation of possible tag types is incomplete and that sooner or later the program will discover an unrecognised tag as it works its way through a DCU.

Did I Say 5?

And now for the question you've been dying to ask: yes, Anthem works fine with Delphi 5 as well.

All that's necessary is to get the program to recognise the magic signature for Delphi 5 DCU files (`$F21F148B`, in case you were wondering!) and away we go... More than anything else, this really amounts to an eloquent demonstration of the fact that the DCU format really *hasn't* changed much from one version of Delphi to the next. Yes, I accept that later versions contain newer tags for implementing stuff such as default parameters, function overloading, and so on, but surely it wouldn't be *too* difficult for Borland to write a utility capable of 'upgrading' older DCUs to a newer format? Come on guys: we know you can do it!

Of course, the aim of *Beating The System* (and indeed, the whole of *The Delphi Magazine*) is to provide you with practical, real-world code and techniques that you can immediately put to good use in your own applications. This mini-series on the DCU file format doesn't quite come into this category but, then again, a new-born baby is rarely of much practical use!

My hope, as I said last time round, is that other folks will build upon these articles and that eventually, we'll have a complete understanding of what's going on inside the mysterious DCU. I'm personally committed to an ongoing investigation of the DCU file until such time as Borland themselves release the necessary information and, accordingly, we'll be returning to the Anthem program on an occasional basis as more is discovered.

```
unit DCUDefs;
interface
const
  // Magic signatures
  D2Magic  =          $50505348;
  D3Magic  =          $44518641;
  D4Magic  =          $4768A6D8;
  D5Magic  =          $F21F148B;
  B3Magic  =          $475896C8;
  // DCU record tags
  Tag_Variable     =         $20;  // variable definition
  Tag_Param        =         $21;  // parameter definition
  Tag_VarPar       =         $22;  // VAR parameter definition
  Tag_ResPtr       =         $23;
  Tag_AbsVar       =         $24;
  Tag_Const        =         $25;  // constant definition
  Tag_VMT          =         $26;  // VMT definition
  Tag_StructConst  =         $27;  // Typed constant definition
  Tag_Proc         =         $28;  // procedure definition
  Tag_StdProc      =         $29;  // stdproc definition
  Tag_Type         =         $2A;  // type definition
  Tag_Label        =         $2B;  // label definition
  Tag_Field        =         $2C;
  Tag_Method       =         $2D;
  Tag_Constructor  =         $2E;
  Tag_Destructor   =         $2F;
  Tag_Property     =         $30;
  Tag_ThreadVar    =         $31;  // thread variable definition
  Tag_ResString    =         $32;  // resource string definition
  Tag_ExtProc      =         $33;
  Tag_End          =         $61;  // end of file marker
  Tag_End_Record   =         $63;  // end of compound record
  Tag_Int_Use      =         $64;  // external definitions used by INTERFACE
  Tag_Imp_Use      =         $65;  // external definitions used by IMPLEMENTATION
  Tag_Type_Use     =         $66;  // reference to external type
  Tag_Sym_Use      =         $67;  // reference to external symbol
  Tag_DLL_Import   =         $68;  // reference to DLL-imports
  Tag_Inc_Level    =         $6A;  // increment level
  Tag_Dec_Level    =         $6B;  // decrement level
  Tag_DFK_Source   =         $70;  // required Source file (.PAS / .INC)
  Tag_DFK_Object   =         $71;  // required Source file (.OBJ)
  Tag_DFK_Resource =         $72;  // required Source file (.RES)
  Tag_DFK_TheAdr   =         $73;  // required Source file (.???)
  Tag_Unit_Flags   =         $96;  // unit flags information
type
  TDCUVersion = ( B3, D2, D3, D4, D5 );
implementation
end.
```

➤ *Listing 4*

In the meantime, if you have any DCU-related insights of your own, you can contact me at the email address given below. Obviously, full credit will be given for all discoveries received.

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. You can contact Dave at TechEditor@itecuk.com