Beating the System: The Private Life Of A WinHelp File (Part 3)

by Dave Jewell

Last month we covered data compression in Windows help files and added a number of additional utility routines to the HELPINFO unit. Up until now, we've only really examined the |SYSTEM subfile. This month, we're going to look principally at the phrase information contained in a typical help file. This month's disk includes a program, HELPDECO, which can access the information inside.

In last month's explanation of data compression, I described how the Microsoft compression algorithm searches a 'sliding window' for sequences of bytes that it's already seen. If it finds any, then it simply adds a reference to the previous byte sequence by adding a length/distance pair to the output data stream. What might surprise you is that this compression technique is replicated within the help system through a higher-level mechanism that searches for individual phrases within the help text being compiled. These two mechanisms, a high-level text oriented scheme and the low-level byte oriented compression, together allow the compiler to create the smallest possible help files.

To see how this works, imagine that the help text contains:

"In order to export a document from Super-Wombat, you should first open the document, and then select Export from the File menu."

In this imaginary fragment of help text, the word "document" appears twice: a fact that will certainly be noticed by the compiler. This process isn't restricted to single words, the compiler will try to find long phrases of many words because, naturally, the more words in a phrase, the greater the benefit in terms of help file size reduction. Some phrases can be several sentences long if the author has 'cut and pasted' material when creating the help file. However, a phrase is limited to 512 bytes in size.

With this in mind, you should begin to appreciate why it is that the help file compiler is so painfully slow! This process of detecting duplicate phrases is potentially one of the most time-consuming aspects of the compilation process. Bear in mind that the compiler must, at all times, have access to the entire help text in order to perform phrase de-duplication and with some help files, the amount of text involved can be enormous: think about the MSDN CD!

As duplicate phrases are detected, they are removed from the help text and replaced by a phrase reference. The duplicate phrase is then added to a subfile of phrases which is constructed during the compilation process. The phrase reference is essentially an index into this subfile of phrases. Under early versions of the help compiler, the phrases subfile was given the name |Phrases (remember that system-level subfiles are always preceded by a | character). Later versions of the help compiler store the phrases in a subfile called |PhrImage and place phrase indexing information in a file called |PhrIndex. If the help file is very small or no duplicate phrases are found the compiler doesn't bother to create phrase file information.

The phrase file information is compressed using the same data compression algorithm that we discussed last time and of course the phrase data is itself compressed in the same way. From all this, you can see that a lot of extra complexity is added through the requirement to make the help file as small as possible.

Here Be Dragons

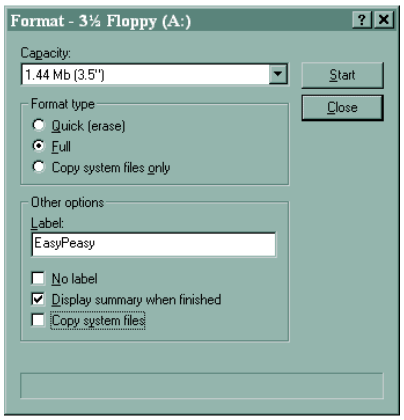
Although this might seem relatively straightforward, there are a number of other considerations associated with phrase data. Like other aspects of the help file, things have sometimes changed from one release of the compiler to the next. Back in the days of Windows 3.0, the Mark 1 format of the |Phrases file looked like this:

```
type
  PHRASEHDR = record
    PhraseCount: Integer;
    Phrase256: Integer;
  end;
```

The first field specifies the number of phrases in the file. This is immediately followed by Phrase256 which always has a value of $100. Immediately after the header, there is an array of 16-bit words, PhraseCount+1 of them to be precise. These words are offsets into the phrase data itself, which follows the list of offsets. Each phrase string is stored without any separator information: there are no C-style nulls after each string, nor are there any Pascal-style length bytes ahead of each string. The text of one string is immediately followed by the next. In order to determine the length of a particular phrase, you need to compare its offset with the offset of the next string: the length of phrase N is the offset of phrase N+1 minus the offset of phrase N.

This, of course, is the reason that there is one more offset in the offset array than there are phrases: we need to be able to determine the size of the final phrase!

Under Windows 3.0, none of the above is compressed: the 3.0 version of the compiler never compresses the |Phrases subfile. However, under Windows 3.1, the |Phrases subfile is always compressed, irrespective of whether or not compression is applied to the entire help file. In order to cater for



➤ *Figure 1*

this, an extra field appears in the Mark 2 PHRASEHDR record. This field, PhraseSize, is a 32-bit long integer which fits between the Phrase256 field and the start of the array of offsets. PhraseSize gives the total size of all the phrases after decompression; it's useful for allocating a buffer into which the phrases will be decompressed.

Still with me? That's not the end of the story by any means. Early versions of the compiler (up to and including Windows 3.1) were limited to a maximum of $700 (1792) phrases. At some point Microsoft removed this limit and they therefore had to revise the PHRASEHDR structure yet again. The new Mark 3 structure looked like this:

```
type
  PHRASEHDR = record
    Phrase2048: Integer;
    PhraseCount: Integer;
    Phrase256: Integer;
    PhraseSize: LongInt;
    PhraseJunk:
      array [0..29] of Byte;
  end;
```

In this version of the header the first field, Phrase2048, always has the value of $800. An old version of the Windows help engine would interpret this as a value of $800 for the PhraseCount, which would be interpreted as excessive and the help file would be rejected. More recent help file readers will recognise $800 as the signature of a Mark 3 header and get the real phrase count from the next field. For some reason, this version of the |Phrases subfile also sets aside 30 bytes of

A Disk Formatting Postscript

Many of you will remember the disk formatting code I developed some months back. This code was intended specifically to allow 16-bit Pascal applications to format floppy disks under Windows 3.1, Windows 95 and NT. If you're writing a 32-bit application, there are ways of formatting disks directly, but the necessary code is somewhat messy.

In time honoured-fashion, Microsoft developed a simple, easy to use disk formatting API call – but they forgot to tell anyone else about it. The routine is called SHFormatDrive and as far as I know, it isn't mentioned in any official documentation. You may remember that I mentioned this routine some time ago but at the time I confessed that I didn't know how to drive it. Well, Wilbert van Leijen (101573,3345 on CompuServe) has done the necessary spade-work and discovered how to drive this routine. Many thanks to Wilbert.

Here's what the function prototype would look like in Pascal:

```
function SHFormatDrive(
  Wnd : HWnd; Drive, Size, Options: Integer): Integer;
```

The first parameter is a window handle which is used as the parent window for any dialogs and message boxes which might need to be invoked by the routine. The second parameter specifies the drive to format. This is interpreted as A=0, B=1, C=2, etc. Incidentally, you need to take care with this routine because Wilbert reckons that it can be used to format a hard disk! Needless to say, I didn't test out his hypothesis – take care.

The next parameter, Size, is used to specify a size for formatting the drive. The value of this will depend on the type of floppy disk you're formatting: for a 1.44Mb drive, a value of 5 will produce a 720Kb disk whereas 6 corresponds to a 1.44Mb disk. A value of zero corresponds to the "default" value for the drive. Finally, the Options parameter determines the type of formatting that takes place: zero indicates a quick format, 1 specifies a full format, while 2 simply copies the system files to the designated floppy disk (this is equivalent to using the DOS SYS command).

With the exception of the Drive parameter, the various parameters need to be taken with a pinch of salt: they are really initial values which can subsequently be altered by the user. When you call SHFormatDrive, the dialog box shown in Figure 1 appears, complete with the various options you've selected. The user is free to change the initial values and type in an optional volume label before hitting the fateful Start button.

On the negative side, the SHFormatDrive call is presently only available under Windows 95, it may or may not be implemented in the release version of NT 4.0 which (at the time of writing) is expected to land on my doormat very soon. The ridiculously short program below (which I compiled with Borland Pascal 7.0) will perform a full format of your floppy drive A: and you could easily incorporate this code into a Delphi application if you so wished. The code is included on this month's cover disk along with the executable file: a mere 1,536 bytes in size!

```
program FormDemo;
uses
  WinTypes, WinProcs;
function SHFormatDrive(Wnd: hWnd; Drive, Size, Options:
  Integer): Integer; far; external 'SHELL';
{ Full format of drive A }
begin
  SHFormatDrive (0, 0, 0, 0);
end.
```

unused space before the phrase offset information; presumably this was reserved for future expansion.

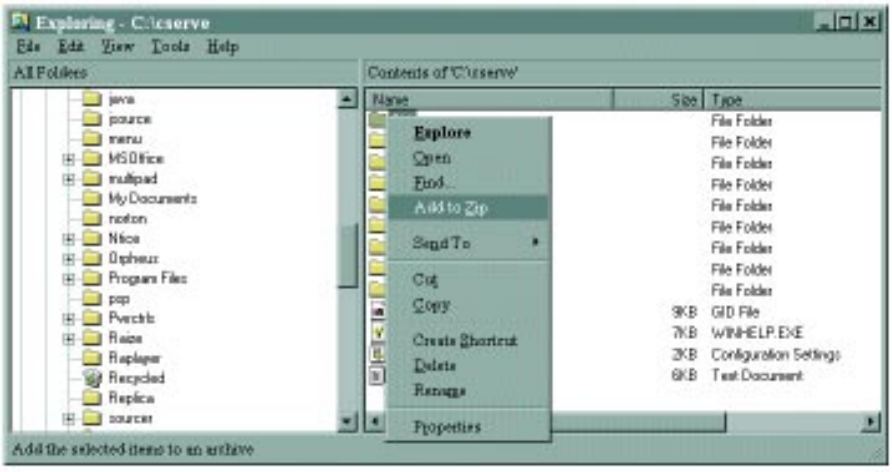
Things become progressively more complex when we move to Windows 95. At this point, Microsoft decided to separate out the phrase offsets from the phrase data itself. As mentioned previously, two subfiles are now involved: |PhrIndex and |PhrImage. The |PhrImage file simply contains the phrase data as before, with no intervening characters between each phrase. This file is typically compressed. But doesn't have to be: you can find out by looking at the other file, |PhrIndex. The Mark 4 version of the header is shown below:

```
type
  PHRASEHDR = record
    Unknown1: LongInt;
    PhraseCount: LongInt;
    CompressedSize: LongInt;
    PImageSize: LongInt;
    CompPImageSize: LongInt;
    AlwaysZero: LongInt;
    BitFlags: Word;
    Unknown2: Word;
  end;
```

The first field, Unknown1, typically has the value $4A01 but can sometimes be $0001. Its precise meaning is unknown, but it's not needed to extract phrase information from the help file. The next value, PhraseCount, stores the total number of phrases as before (notice that this is now a long value). The CompressedSize field simply stores the size of the |PhrIndex file itself while PImageSize and CompPImageSize contain the uncompressed and compressed sizes of the phrase data found in the |PhrImage file. If these two fields differ in size, then you can assume that |PhrImage is a compressed file. The AlwaysZero field is (as the name suggests) always zero and can be ignored.

Just to make things a little more interesting, Microsoft used another compression scheme to compress the phrase index information in the |PhrIndex file. Compression scheme is perhaps not the right word: it's just a clever hack that's used to reduce the size of the offset data. When the compiler builds the help file, it determines the minimum number of bits required to hold a phrase offset and all offsets are stored only in that number of bits. Interestingly, a phrase offset still ends up as only a 16-bit offset (just as in earlier versions of the help file) which presumably means that there are some real restrictions on the maximum number of phrases that can be stored: no more than 64Kb of data in total. This strikes me as something of an anachronism.

The lower four bits of the BitFlags field determine the bit count used to decode each phrase offset. If you want to see, in detail, how this works, look at the HELPDECO source code. This brings me onto...



➤ *Figure 2: Programs like WinZIP add their own custom context menu items to the Windows 95 Explorer. Look at the highlighted Add to ZIP menu item shown here. Next month, I'll show you how...*

Introducing HELPDECO

This month I've decided to cheat! I recently came across a public domain help file disassembler which is far more complete than anything I could do in the time available, so I've included this on the disk rather than create my own source code for taking apart help files. The only bad news is that it's written in C, 6,500 lines of C to be precise, the good new is that a compiled version is included too. There's a few other helpful goodies on this month's disk as well, such as a very detailed description of the internals of a Windows help file. The HELPDECO program itself can be used to poke around inside a help file and can reconstitute the various components of a help file into a form ready for subsequent modification and recompilation using one of the standard Windows Help file compilers. The author of this software, a German developer called M Winterhoff, should be sincerely thanked for putting this fascinating code into the public domain and especially for making the source code available.

Next Time

In next month's column I'll be demonstrating how to use Delphi to extend the functionality of the Windows 95 Explorer through the use of context menus. If you have a copy of the excellent WinZip shareware program, you'll know that it uses context menus to provide access to new functionality from inside the Explorer (see Figure 2). Next month, we'll be seeing exactly how you can do this from inside your own programs.

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is the author of "Instant Delphi Programming" published by Wrox Press. You can contact Dave as DaveJewell@msn.com, DSJewell@aol.com or 102354,1572 on CompuServe.