Windows Undocumented File Formats

Pete Davis and Mike Wallace

R&D Books
Lawrence, KS 66046

Acknowledgments

Pete Davis

There are so many people to thank. I must start immediately with Andrew Schulman (of "Undocumented Windows" fame) and Ron Burk (editor of *Windows Developer's Journal).* Andrew got me started writing professionally, so all blame should really go to him. Ron helped immensely with the reverse-engineering work on WinHelp and I would hazard a guess that he did more than half of it. But more importantly, he has helped me out in more ways than I can count since we first met. He has published several of my articles and been a valuable help in my writing and my understanding of Windows. Thank you both so much, for everything.

The following people provided extra special help. I don't mean to leave anyone out, and there were so many people involved, I'm sure I will, but these are the ones that come to mind right away. Wolfgang Beyer, Carl Burke, Stefan Olson, and Lou Grinzo provided lots of help on the WinHelp .HLP file format. Clive Turvey provided all the information on the W4 file format and also helped out with the LE and W3 file formats. Skip Key provided great insights on the LE file format, making up for my lack of understanding of executable files.

The long list begins: Dave Bakin, Kevin Burrows, Jon Erickson *(Dr. Dobb's Journal)*, Mike Floyd *(Dr. Dobb's Journal)*, Jim Hague, Dale Lucas, Nico Mak (of WinZip fame), Duncan Murdoch, Andrew Pargeter, Matt Pietrek, Steve Scanzoni, and Brian Walker. All of these people have contributed in one way or another to making this book happen. Most provided information or checked my information to make sure it was correct. Thank you all so much.

There are a few people who made considerable contributions but, unfortunately, must remain anonymous. Most of them are people who are working on projects that they don't want certain companies to know about. None of these people had anything to gain by providing me with their hard-earned information, but simply did it out of the idea that people should have access to this information.

I'd like to thank Steve Wexler, Julianne Sharer, Brett Foster, Ted Epstein, and all the others I've worked with at WexTech Systems, Inc. They provided me with a job that gave me time to write a majority of this book.

I'd like to thank everyone I work with (or have worked with) at Moffet, Larson, and Johnson, my current employer. They've continued to provide me with a challenging and exciting work environment that has continued to feed my brain. In particular, Enrique Lenz, Dan Cox, Gyuri Grell, Alex Reyes, Carol Kelly, Garri Yackobson, Walt Constantine, Paul Modzelowski, and Theresa Alford.

I'd like to thank my roommate and buddy, Mike Wallace, for putting up with me quitting my regular job to write and not being able to help pay the rent as timely as he'd like. I'm really glad you worked on the book with me. You've been a good friend and made it possible for me to pursue some of my dreams. Wow, we finally finished it, huh!

I'd like to thank my parents, who have always pushed me to do what I love and believed in me when I didn't. They've been an incredible support in every way.

I'd also like to thank my grandmother, who thought I'd been famous since I wrote my first article. She's been such an ardent supporter and read all my writings, even though she didn't understand a word.

And finally, I'd like to thank Berney Williams, our patient (oh, how patient) and understanding editor, and Robert Ward, our patient and understanding publisher, for making this book happen. Thanks for putting up with the delays (and delays, and delays, and delays).

Other acknowledgments: I must thank Jake (my cat) and Naoise (Mike's cat: pronounced "Neesha") for keeping me company while working on the book. Naoise has been a quick study at the art of stomp-typing and paper shuffling. Jake, as always, an expert at playing dead all day long, every day. The vet claims he's alive, so I guess I'll keep him.

We'd like to thank the crew at Bardo Rodeo, who let us blow off steam playing pool and drinking their fine home-brewed beer 'til the wee hours of the night.

And finally, thanks to all the fine musicians and bands who gave me something to listen to while I wrote this book.

Mike Wallace

The person I want to thank the most is Pete Davis. This book was entirely his idea, and he is more than capable of writing the entire thing by himself. It was my good luck to have him for such a good friend when he decided to tackle this project. He gave me my first exposure to the joys of writing professionally. There were times I thought I would never get some of the programs working, but Pete had the patience to let me work at my own schedule. He was far more understanding than I think most co-authors would be.

Next, I want to thank my family. My parents always let me pursue my own interests and never discouraged me when things didn't work out as planned. I also couldn't have asked for two nicer sisters, plus the coolest nephews, nieces, and brothers-in-law anyone could hope to have.

I'd like to thank the friends who helped me get through the last several months: Ed Reyes, Robin Carey, Carol Kelly, Jennifer Campbell, and the fine employees of Bardo Rodeo for letting me play pool long after they closed more than once, when I was in danger of burning out. If there's anyone I forgot, I'm sorry. I'm too forgetful to remember more than a handful of names at any given time!

I want to thank our editor and publisher for being so patient with us, and not saying anything when we missed our deadline.

Finally, there's the people I've never met, but provided inspiration in one form or another: Neil Young, Robert Pirsig, Henry David Thoreau, Toad the Wet Sprocket, Bela Fleck and Steve Howe.

Table of Contents

Chapter 1

Introduction and Overview

How It Began

This book, we feel, is a long time in coming. Before we started reverse-engineering file formats, we would go through bookstores looking for just this book. We never found it, obviously. The reason we began reverse-engineering file formats had less to do with a need for the information as it had to do with a curiosity about them. We're just the kind of people that like to know how everything works.

The works in this book began with a plea from Andrew Schulman (of *Undocumented Windows* fame) and Ron Burk (editor of *Windows Developer's Journal)* for someone to reverse-engineer the WinHelp .HLP file format. At the time, my interest in professional writing had just begun. I decided, "How hard can it be?" It seemed like a perfect opportunity to get my name in print. My initial idea was just to help Ron Burk out and hopefully get my name mentioned in his article when it was written.

Well, the .HLP file format turned out to be much more complex than I could have imagined. Although I did take my initial work to Ron and we did end up doing the entire project together (Ron probably did more than half the work), Ron left the articles to me (published in Andrew Schulman's "Undocumented Corner" column, *Dr. Dobb's Journal,* September and October 1993). I am to blame for all of the mistakes and the incomplete nature of the work.

At this point, I think it's important to make one thing clear. Although the original articles for the .HLP file format were printed in his column, and despite my thanks to him, Andrew Schulman was not directly involved in any aspect of this book. I mention this not to take any due credit from him, but because Andrew published the "Undocumented ..." series of books with Addison-Wesley, and this book is not connected with that series at all.

What's in This Book?

This book is partially based on previous work, which I will briefly mention here and in a more detailed bibliography at the end of this book. In particular, there was a two-part article on the WinHelp .HLP file format I wrote for *Dr. Dobb's Journal,* an article in *Windows Developer's Journal (WDJ)* on the .SHG and .MRB file formats, and another article in *WDJ* on the file format and LZ77 algorithm used by COMPRESS.EXE. The .ANN file format appeared in *PC Magazine* (Volume 14, No. 15) in an article I co-wrote with Jim Mischel on advanced WinHelp techniques.

Two chapters in this book came from work done by other people. The .RES file format was originally uncovered by Dmitry M. Rogatkin and Alex G. Fedorov and published in *Dr. Dobb's Journal* in August 1993. The .PIF file format was reverse-engineered by Michael P. Maurice and was also published in *Dr. Dobb's Journal* in July 1993.

The W4 file format presented in Chapter 9 was provided by Clive Turvey. He and several other people figured out the format, and Clive, as far as I know, was the one that figured out the compression algorithm. He was the first I heard of to write his own routines for decompressing W4 files, whereas other people were calling directly into the Double Space decompression routines.

The rest of the files covered in this book were solved by Mike and I and have not been previously published.

Information on the .HLP file format in this book has been completely revised from the original articles. A lot of corrections and pieces of missing information have been filled in.

In total, 10 file formats are covered in this book (actually, considering all the variations, it's more like 13 or 14). The next 8 chapters are organized as follows:

- **SECTION 1**
  - **Chapter 1** you're reading now, if you didn't know.
  - **Chapter 2** describes the multiresolution bitmap (.MRB) file format.
  - **Chapter 3** continues from Chapter 2 with the SHED (.SHG) file format, which is an extension of the .MRB file format.
  - **Chapter 4**, describes the complex WinHelp .HLP file format.
  - **Chapter 5** discusses the Annotation (.ANN) and Bookmark (.BMK) file formats used by WinHelp.
- **SECTION 2**
  - **Chapter 6** lays out the file format used by COMPRESS.EXE, EXPAND.EXE, and the LZEXPAND.DLL library for file compression.
  - **Chapter 7** is adapted from Dmitry M. Rogatkin and Alex G. Fedorov's articles to describe the .RES file format.

- **Chapter 8** uses information from Michael P. Maurice's article and talks about the .PIF file format.
- **Chapter 9** explains the W3 and W4 file formats used by Windows 3.x and Windows 95.
- **Chapter 10** provides an in-depth description of the LE (linear executable) file format used by VxDs.

Why All the Undocumented File Formats?

This is a common question. Why doesn't Microsoft document this stuff? I think there are different reasons for different file formats.

With SHED and MRBC (Multiresolution Bitmap Compiler), I'm not really sure what the problem is, exactly. Microsoft has released documentation for the .SHG file format, but it is completely inaccurate and misleading. I honestly believe this was an error and not intentional, because I was asked by someone at Microsoft to write correct documentation for them to release to the public. However, due to some disagreements, that never happened.

I have a pretty good idea why the WinHelp file formats were never released. For one thing, the formats have changed drastically with every version of Windows. Microsoft probably doesn't want to be restricted by having to maintain backward compatibility. More importantly, though, I think it's because the file format is a complete mess. The problem is that WinHelp has changed hands at Microsoft quite a few times. A lot of the people who maintained it in the past failed to pass down documentation to the new people. From what I hear, maintenance of WinHelp is a nightmare. For a while it was a solo project, so only one programmer at a time worked on it and no consistent group decisions were made. This could also account for the major changes between versions. Luckily, Windows 4.0 did not introduce major changes in the file format. Most of the changes involve additional internal files, but not many changes in the format itself.

As for the formats used by the compression utilities, I can't imagine why Microsoft would hide the documentation. It's certainly not to discourage competitors. Microsoft doesn't make any money off of the compression software, and LZEXPAND.DLL is redistributable. And it's not like they have the best compression algorithm out there. So again, I think it's just a matter of not feeling that the general public "needs" to know.

Why Are We Picking on Microsoft?

It may seem that we're picking on Microsoft. After all, every file we cover in this book is a Microsoft file. This was not the intention. Plenty of other companies have their own proprietary formats for their files, after all, but the main difference is that many of the Microsoft files are pieces of Windows itself. Everyone has these files. If we were going to cover other undocumented file formats, we might go after WordPerfect for Windows, Lotus, and others. We decided to stick to the files that we believe have the largest user base and therefore will be most useful to developers. Our hope is that other file formats, from Microsoft and others, will be included in later editions of this book. A lot of that will depend on demand.

The Future

We have every intention of keeping this book up-to-date and releasing revised editions when a significant number of changes have been made to the existing formats. We may also find that we want to cover formats not discussed in this edition in later editions. Either way, we want this book to remain current, so that you, the ordinary developer, will have access to the information you need and deserve.

Because we plan to keep this book up-to-date, we'd love some feedback. We want to know what you like about it, what you don't like about it, what files you think we should cover in future editions, and so on. What we need most is reports of errors or updates to any unknown fields in the formats we've provided. We feel we've done a really good job of covering the file formats that appear in this book. We've bent over backwards to find every field we can, and we've talked to a lot of people who have used this information to help us get it as accurate as possible.

Still, this is all "undocumented". We don't have access to source code; we can't be positive about some of the fields and structures. For example, in the .SHG and .MRB files, we have come up with what we think are the structures. It's possible that Microsoft's structures differ. What we might consider one structure, Microsoft might consider two or vice versa. We've done the best we can to see that structures are as logical as possible. Sometimes Microsoft doesn't afford us that luxury by the nature of the files, but we've done what we can.

There are already files we'd like to consider adding to future versions, including the Word for Windows 2.0 and 6.0 formats, the Paradox file formats, and the OLE 2.0 Docfile format. The work in this book is the best we could accomplish in a reasonable time period. These other file formats are complex and will require a lot of time, but if demand for this book is high enough, we will put out future editions. At some point, it would be nice if this book was considered "The Encyclopedia" of undocumented file formats.

How to Reverse-Engineer File Formats

Basically, you need to have two things to reverse-engineer file formats: good eyes and a good pair of glasses. That may seem like a contradiction, but let me elaborate. First, you need good eyes; when you're done, you will need a good pair of glasses. Nothing will strain your eyes and brain like staring at hex dumps for 8 hours straight.

Begin by just staring at hex dumps of the file, and if you're not making progress, stop thinking and keep staring. After about 8 hours, leave it and don't think about it. The next day, go back to it, but this time it will start to make sense. Why? I think the subconscious mind is a bit keener than the conscious mind. So without doing much more than staring when you're stuck, your brain starts running a background task (to use geek speak), trying to figure it out, so that you don't have to worry yourself about it.

Does this always work? Not always, but it certainly does a lot of times. The rest of the time, you add, divide, multiply, subtract, left-shift, right-shift and so on until numbers start to mean something. Sometimes things get worse and you break the hex numbers down to bit streams (ugh).

None of it is a particularly enjoyable experience. Unlike most programming jobs, there's very little satisfaction 95% of the time. The last 5% of the time is when things start to fall in place and you start to feel good about it. Getting through the first 95% is the hard part.

DUMP and SETVAL

DUMP will give you a simple hex dump of a file. I usually pipe the output to a file and then print it. There's nothing special about DUMP. It's as simple as they come and about as complex as you need. Staring at hex dumps is the best way to reverse-engineer a file format.

Take a look at an example. Figure 1.1 shows a simple text file. Figure 1.2 shows a hex dump of the text file compressed with Microsoft's COMPRESS.EXE.

Figure 1.1 TEST.TXT textfile.

```
This is a test. This is only a test.
This is not important information.
```

Figure 1.2 Results of DUMP of compressed TEST.TXT

```
Offset              Hex Values                                  Ascii

0x00000000: 53 5A 44 44 88 F0 27 33 41 00 4C 00 00 00 DF 54  SZDD^.'3A.L....T
0x00000010: 68 69 73 20 F2 F0 61 20 DF 74 65 73 74 2E EF F6  his ..a .test...
0x00000020: 6F 6E DB 6C 79 F7 F5 0D 0A F0 F5 6E 6F FF 74 20  on.ly......no.t
0x00000030: 69 6D 70 6F 72 74 FB 61 6E 20 00 6E 66 6F 72 6D  import.an .nform
0x00000040: DF 61 74 69 6F 6E 13 00 0D 0A                    .ation....
```

A first look shows a few things of interest. The first 4 bytes produce the letters SZDD. This can be seen in all files compressed by COMPRESS.EXE. It's usually a safe assumption that the first few bytes are some sort of magic number or header that identifies the file type. This is especially true if they represent letters. What does SZDD mean? It is probably the initials of the people who wrote the compression software; two people, I'd guess in this case. Again, it's speculation and you rarely find out the truth on something like that.

Look for more important information, though. Nothing really sticks out until bytes 11 to 14. The 0x4C followed by three zeros happens to be the same as the file size of TEST.TXT. This is something you'd look for, because it's pretty safe to assume that an original file size is going to be in there somewhere. After that, it appears that some of the original text is intact but mangled a bit here and there. This is where the hard work begins.

I'll skip ahead a few days into my work and look at the byte immediately after the file size, 0xDF. If I convert that to binary, I get 11011111. I don't see a correlation here yet, but if I turn it around, I get 11111011; that is, five 1's, followed by a 0, then two 1's. Here's something odd: The first five characters from our text file are fine, but then the word "is" (with the space following it) is missing, with 2 bytes, 0xF2 and 0xF0, instead, followed by two more characters from the text file. So, it seems that the binary 1's mean a normal character, whereas the binary 0's mean a 2-byte code. As I said, this occurred to me a few days after I had begun the project. These things don't just pop out at you immediately all the time.

I won't go into the format here; I'll save that for Chapter 6, but this is how you get started. You start looking for numbers you expect, look for patterns, break bytes down to bits, and do a lot of staring.

Sometimes you need more, though. I've included another program called SETVAL.EXE. It will change a byte value in a file. Simply pass the offset and byte value (both in hex), and it will change the byte at a given offset to the new value. I usually use this when I'm down to a few unknown fields. I can change those values and see what effect the change has. Sometimes the changes cause GP Faults and sometimes memory allocation errors, but sometimes they lead you to exactly what the field does. Between these two utilities, you're armed and ready to tackle some serious file formats.

The next thing to do is to create lots of samples. The best way to do this is take a single small file and make minor modifications. Get dumps of each change and see what values change. For example, with the .SHG and .MRB file formats, change the image size a little and see which values change, or move a hotspot over a bit and see what changes there.

Usually you just need to try every option the program has available and see what those options change in the file. This is very tedious work, because you only want to change one thing at a time and you need to save it and get a dump each time. If you change more than one thing between dumps, you don't know which of the changes caused which values to change.

It would be nice if there was an easier way than this, but there really isn't. Sometimes, once you've gotten started, you can build custom tools for specific files. When working on the WinHelp file format, Ron Burk and I wrote a program called HELPDUMP (a variation by the same name was released with my article on the `.HLP` file format). HELPDUMP started out as a custom hex-dump program. Instead of hex-dumping an entire `.HLP` file, though, once we figured out the internal `.HLP` file system, we could dump individual internal files within the `.HLP` file. Then as we figured out each of those internal files, we wrote a piece of code to handle them. If we had unknown fields, we'd have it print the values so that we could test specific fields of different test files.

It really helps to have a good knowledge of data structures and algorithms. I didn't have as good a knowledge as I originally thought. I certainly didn't know anything about compression when I started working on the different compression file formats, and I didn't remember much about b-trees when I started on the `.HLP` file format. So, I read up on them (see the annotated bibliography).

To sum up, you need good eyes, good glasses, lots of time, `DUMP.EXE`, `SETVAL.EXE`, and a good library of data structures and algorithm books. Now you're really armed to the teeth.

Listing 1.1 is the code for DUMP. The program is pretty straight forward. What I usually do is pipe the results to a file, so I can either print the file or examine it from an editor.

Listing 1.2 is the source code for SETVAL. Again, it's a very simple program but is invaluable in the art of reverse-engineering file formats.

Getting in Touch with Us

If we've screwed something up or you've figured out an unknown field, or if you have suggestions about how we can improve future editions of this book, we'd really like to hear from you. As far as we're concerned, this book is a living document and will continue to evolve as new information comes our way.

To contact us, send e-mail to `peted@mnsinc.com` or `mwallace@mlj.com`.

A lot of work has gone into producing this book. We really hope you find it useful. We look forward to hearing your comments, suggestions, and yes, even complaints (if you've paid for the book, you're entitled to them).

Listing 1.1 DUMP.C — Produces a simple hex dump of a file.

```
/**********************************************************************
 *
 * PROGRAM: DUMP.C
 *
 * PURPOSE: Produces a simple hex dump of a file.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 1, Introduction and Overview, from Undocumented Windows
 * File Formats, published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 **********************************************************************/

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef unsigned char     BYTE;

/****************************************************
  Performs a Hex/ASCII dump of a file.
****************************************************/
void HexDump(FILE *File) {

   char         Buffer[16];
   long         counter, FileSize;
   int          BytesToPrint, Index;

   fseek(File, 0, 2);
   FileSize = ftell(File);
   fseek(File, 0, 0);
   printf("Offset                  Hex Values");
   printf("                          Ascii\n");
   printf("-------------------------------------");
   printf("----------------------------------\n");

   for (counter = 0; counter < FileSize; counter+=16) {

      printf("0x%08lX: ", counter);
      BytesToPrint = fread(Buffer, 1, 16, File);
      for (Index=0; Index < BytesToPrint; Index++)
          printf("%02X ", (BYTE) Buffer[Index]);
      for (Index=0; Index < 16-BytesToPrint; Index++)
          printf("   ");
      for (Index=0; Index < BytesToPrint; Index++)
          putchar( isprint( Buffer[Index] ) ? Buffer[Index] : '.' );
      putchar('\n');
   }
}
```

Listing 1.1 (continued)

```
/***************************************************
  Show usage.
***************************************************/
void Usage() {

  printf("Usage:\n");
  printf(" DUMP filename\n\n");
  printf("   filename - Name of file to dump\n");

}

/***************************************************
  Open the file and dump it.
***************************************************/
int main(int argc, char *argv[]) {

    char filename[40];
    FILE *File;

    if (argc != 2) {
       Usage();
       return EXIT_FAILURE;
    }

    strcpy(filename, argv[1]);
    _strupr(filename);

    if ((File = fopen(filename, "rb")) == NULL) {
       printf("%s does not exist!", filename);
       return EXIT_FAILURE;
    }

    HexDump(File);
    fclose(File);

    return EXIT_SUCCESS;
}
```

Listing 1.2 SETVAL.C — Modifies a byte in a file.

```
/************************************************************************
 *
 * PROGRAM: SETVAL.C
 *
 * PURPOSE: Modifies a byte in a file.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 1, Introduction and Overview, from Undocumented Windows
 * File Formats, published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 ************************************************************************/

#include <stdio.h>

void usage()
{
  printf("Usage: SETVAL fn addr val\n");
  printf("  Where addr is the hex location of the byte\n");
  printf("  in the file to modify and val is the byte (in hex)\n");
  printf("  to set at that location.\n\n");
  printf(" Example:  SETVAL LIST.TXT 4B3 FF\n\n");
}

int main(int argc, char *argv[])
{
  char filename[256];
  char address[10], val[10];
  long nAddr = 0, nVal = 0;
  FILE *fp;

  if (argc != 4)
  {
    usage();
    return 0;
  }
  strcpy(filename, argv[1]);

  if ((fp = fopen(filename, "r+b")) == NULL)
  {
    printf("Bad filename supplied\n");
    return 0;
  }
```

Listing 1.2 (continued)

```
  strcpy(address, argv[2]);
  strcpy(val, argv[3]);

  sscanf(address, "%lx", &nAddr);
  sscanf(val, "%lx", &nVal);

  if (nVal < 0 || nVal > 255)
  {
    usage();
    printf("Error: Supply a val between 0 and 255\n");
  }

  fseek(fp, 0, SEEK_END);
  if (ftell(fp) < nAddr)
  {
    printf("The address supplied is beyond the end of this file.\n");
    printf("Please supply a value within the scope of this file.\n");
    fclose(fp);
    return 0;
  }

  fseek(fp, nAddr + 1, SEEK_SET);
  fwrite(&nVal, 1, 1, fp);
  fclose(fp);

  printf("File modified successfully.\n");
  return 0;
}
```

Multiresolution Bitmap (.MRB)FileFormat

My second published article was on the .SHG and .MRB file formats. The original work was in the February 1993 issue of *Windows Developer's Journal*. In the original article, I handled both formats together, because both file formats are very similar. In fact, the formats are virtually identical, except that each has aspects the other one doesn't have.

More clearly, a SHED file can have hotspots, but an .MRB file can't. An .MRB file can have multiple bitmaps, but a SHED file can't. However, you can combine them. If you create several SHED files for monitors of different resolutions, you can combine them into one .MRB file using the Multiresolution Bitmap Compiler (MRBC).

A magazine article is usually very limited in length. Because I wanted to cover both formats, and because the formats were so similar, it made sense to cover them in the same article. However, because of space limitations, I couldn't reveal as much about the formats as I would have liked to. A book, on the other hand, usually doesn't suffer from the same limitations. To help better separate the issues, we felt it would be best to treat .MRB and .SHG files separately.

Because of the similarity, however, I had to choose a single naming convention for the structures. I chose to use the .SHG naming convention, mainly because my original work was on the .SHG file format.

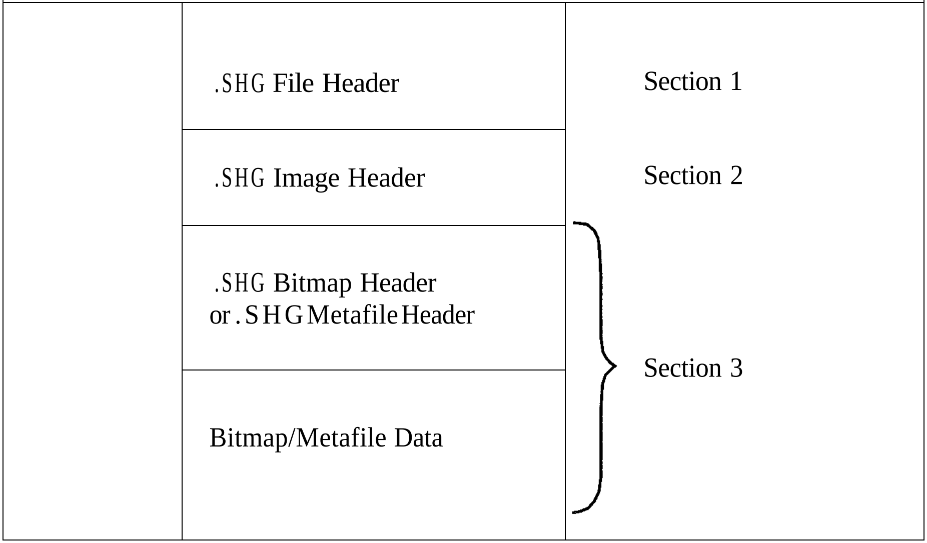
.MRB Format

Figure 2.1 shows the layout of a .MRB file. Notice that I'm using the .SHG notation for the data structures. .MRB files are laid out in three basic sections. Section 1 is the SHG file header. Every .MRB file has only one. Section 2 is the image header. Basically, it tells you whether the image is a bitmap or a metafile. Section 3 combines the SHG Bitmap or SHG Metafile header and the bitmap or metafile data. The SHG Bitmap and SHG Metafile headers should not be confused with bitmap headers or metafile headers. The structures are distinctly different.

You're probably thinking, "What's this metafile business? Who puts metafiles through the MRB compiler?" Good questions. Ever try it? The MRB compiler verifies that the file is a metafile, compresses it, and stores it as a .MRB file. I can't really give an explanation as to why the MRB compiler would support metafiles, especially since they're resolution independent, but there you have it. Now, for those of you familiar with SHED, you know why metafiles are supported by the SHG file format. SHG files can have bitmaps or metafiles. Whichever you import, MRBC and SHED keep them in their natural (bitmap or metafile), but slightly altered, form. You'll understand when you get to section 3. I suppose that metafiles are supported by the MRB compiler simply because they have to be supported by SHED.

With a .MRB file, sections 2 and 3 are repeated for each image. So if a .MRB file has three bitmaps in it, there will be three copies of sections 2 and 3.

Figure 2.1 .MRB layout.



SHGFILEHEADER

Each .MRB file begins with a SHGFILEHEADER (Table 2.1) structure. This structure has a type, or magic number field, a count of the number of images in the file, and offsets to the image header structure for each image. So if there are three images in the .MRB file, there will be three offsets to image headers. Notice that there are two magic numbers. The magic number lets you know who created the .MRB/.SHG file and which form of compression is used in the .MRB/.SHG file. The 0x706C magic number indicates the file was created with MRBC or SHED. I'll discuss this later in this chapter.

For now though, I need to bring up a topic I haven't talked about yet. The purpose of .MRB and .SHG files, obviously, is to include them in WinHelp. What isn't apparent is that every bitmap or metafile included in WinHelp is actually converted to the .MRB/.SHG format. WinHelp adds another layer of compression, LZ77 compression. WinHelp checks to see if the LZ77 compression is actually going to reduce the size of the image. If it does, then WinHelp will use the LZ77 algorithm, and the magic number for the .MRB/.SHG image will be 0x506C. I'll discuss this again later in this chapter, and the WinHelp aspects will be discussed more in Chapter 4.

SHGIMAGEHEADER

Each individual image in the .MRB file has a SHGIMAGEHEADER record (Table 2.2). The SHGIMAGEHEADER tells you three things about the image. It tells you whether the image is a metafile or a bitmap (IT_WMF or IT_BMP respectively), whether the data in the image is compressed, and the resolution in dots per inch (for metafiles, DPI is unused and is set to 0x10).

Two values are currently supported for the siImageType field:

```
#define  IT_BMP      0x06
#define  IT_WMF      0x08
```

Table 2.1 SHGFILEHEADER record.

Field Name	*Data Type*	*Comments*
sfType[2]	char	lp (0x706C) or lP (0x506C)
sfNumObjects	int	Number of images in file
sfObjectOff[]	DWORD	Array of offsets to images

Four values are supported for the siCompression field:

```
#define IC_NONE    0x00
#define IC_RLE     0x01
#define IC_LZ77    0x02
#define IC_BOTH    0x03
```

I'll discuss the compression algorithms later in this chapter, but notice that the last compression option, IC_BOTH indicates that the bitmap or metafile was first compressed with the RLE compression algorithm, then the results of that compression were compressed further with the LZ77 algorithm.

You'll notice that the value for dots per inch (siDPI field) is multiplied by 2. It's also listed as a BYTE or WORD, although in most cases it will only appear as a BYTE in the file. This is something you'll see over and over in other structures, apparently to save some space. How would you be saving space? Well, it's a really bizarre concept that doesn't make much sense, but it works like this: If the value is odd, then you double the size of the field. To read in the siDPI field, you'd read only a single byte instead of a word. If the value is odd, 0x21 for example, then you'd read a second byte and divide the total word value by two, discarding the remainder. This seems to be some sort of attempt to save a few bytes and seems to me to be a lot more trouble than it's really worth.

I wrote four short routines, WriteWordVal(), WriteDWordVal(), ReadWordVa1(), and ReadDWordVal() (Listing 2.1). These routines read and write the fields properly, which is a lot of work, because instead of reading or writing the structure as a whole, you have to handle it field by field.

Table 2.2 SHGIMAGEHEADER record.

Field Name	*Data Type*	*Comments*
siImageType	BYTE	0x06 = BMP 0x08 = WMF
siCompression	BYTE	0x00 = No compression 0x01 = RLE compression 0x02 = LZ77 compression
siDPI	BYTE or WORD	Dots per inch x 2 (0x10 for metafiles)

SHGBITMAPHEADER

The SHGBITMAPHEADER **structure (Table 2.3) follows the** SHGIMAGEHEADER **structure if the image is a bitmap** (IT_BMP). **Notice how most of the fields are multiplied by two, so most of the** WORDs **will be read in only as** BYTEs, **and most of the** DWORDs **will be** read in as WORDs.

Two fields, sbIsZero and sbTwoHund, **appear to be constant values.** sbunk1 **is simply an unknown field.**

Table 2.3 SHGBITMAPHEADER record.

Field Name	*Data Type*	*Comments*
sbIsZero	BYTE	Always 0x00
sbDPI	BYTE or WORD	Dots per inch x 2
sbTwoHnd	WORD	0x200
sbNumBits	BYTE or WORD	Bits per pixel x 2
sbWidth	WORD or DWORD	Width x 2
sbHeight	WORD or DWORD	Height x 2
sbNumQuads	WORD or DWORD	Number of RGBQUADS x 2
sbNumImp	WORD	Number of "important" RGBQUADS
sbCDataSize	WORD or DWORD	Size of bitmap data x 2
sbSizeHS	WORD or DWORD	Size of hotspot data area (used only by SHED)
sbunk1;	DWORD	Unknown
sbSizeImage	WORD or DWORD	(ImageHdr + BitmapHdr + sbCDataSize) x 2

Listing 2.1 WriteWordVal(), WriteDWordVal() ReadWordVal(), and ReadDWordVal().

```
/*********************************************************************
 *
 * PROGRAM: WriteWordVal(), WriteDWordVal(), ReadWordVal(), and ReadDWordVal()
 *
 * PURPOSE: Routines to read and write the fields properly -- field by field,
 * rather than reading or writing the structure as a whole.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 2, Multiresolution Bitmap (.MRB) File Format,
 * from Undocumented Windows File Formats, published by R&D Books,
 * an imprint of Miller Freeman, Inc.
 *
 *********************************************************************/
```

The sbDPI field should match the siDPI field in the SHGIMAGEHEADER. The sbNumBits field is the number of bits per pixel. sbWidth and sbHeight have the dimensions of the bitmap. sbNumQuads and sbNumImp are the number of RGBQUADS listed and the number of RGBQUADS required to render the image properly, respectively. The sbCmpSize field is the size of the compressed bitmap data. sbSizeImage is the size of the SHGIMAGEHEADER + the SHGBITMAPHEADER + the image data.

The sbSizeHS field is the size of the hotspot data area in BYTEs. This is used only by SHED and is 0 in an .MRB file. However, this field would be used in an .MRB file created from .SHG files. I'll discuss this information in more detail in the next chapter.

To give you an idea of how to read this structure, take a look at ReadBMHeader() (Listing 2.2). Notice how, instead of one simple fread() call, we have to resort to five fread()s and seven calls to ReadWordVal() or ReadDWordVal(), each of which will make one or two calls to fread(). This is certainly a lot of overhead to save a couple bytes here and there and a lot more work than it should be.

Listing 2.1 (continued)

```
void WriteWordVal(FILE *SHGFile, WORD val) {
    BYTE a;
    WORD b;

    b = val * 2;
    if (b > 255) fwrite(&b, sizeof(b), 1, SHGFile);
    else {
        a = (BYTE) b;
        fwrite(&a, sizeof(a), 1, SHGFile);
    }
}

void WriteDWordVal(FILE *SHGFile, DWORD val) {
    WORD a;
    DWORD b;

    b = val * 2;
    if (b > 65535) fwrite(&b, sizeof(b), 1, SHGFile);
    else {
        a = (WORD) b;
        fwrite(&a, sizeof(a), 1, SHGFile);
    }
}

WORD ReadWordVal(FILE *SHGFile) {
    BYTE a, b=0;
    fread(&a, sizeof(a), 1, SHGFile);
    if (a % 2) fread(&b, sizeof(b), 1, SHGFile);
    return (WORD) ((WORD)b*256 + a) / 2;
}

DWORD ReadDWordVal(FILE *SHGFile) {
    WORD a, b=0;    fread(&a, sizeof(a), 1, SHGFile);
    if (a % 2) fread(&b, sizeof(b), 1, SHGFile);
    return (DWORD) ((DWORD)b*65536 + a) / 2;
}
```

SHGMETAFILEHEADER

The SHGMETAFILEHEADER (Table2.4) follows the SHGIMAGEHEADER structure if the siImageType field is IT_WMF. This structure is essentially a scaled down version of the SHGBITMAPHEADER. All smXXXXX fields are the same as their sbXXXXX equivalents. The only difference is the smXWidth and smYHeight. These values are given in metafile units and are not multiplied by two.

Table 2.4 SHGMETAFILEHEADER record.

Field Name	*Data Type*	*Comments*
smXWidth	WORD	Width of image in metafile units
smYHeight	WORD	Height of image in metafile units
smUDataSize	WORD or DWORD	Size of metafile data x 2 (uncompressed)
smCDataSize	WORD or DWORD	Size of metafile data x 2 (compressed)
smSizeHS	WORD or DWORD	Size of hotspot data area x 2 (used only by SHED)
smunk1	DWORD	Unknown
smSizeImage	WORD or DWORD	(ImageHdr + WMFHdr + smCDataSize) x 2

Listing 2.2 ReadBMHeader().

```
/*********************************************************************
 *
 * PROGRAM: ReadBMHeader()
 *
 * PURPOSE: An example of how the SHGBITMAPHEADER structure works.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 2, Multiresolution Bitmap (.MRB) File Format,
 * from Undocumented Windows File Formats,
 * published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 *********************************************************************/
```

Bitmaps

Following the SHGBITMAPHEADER is a list of RGBQUADS. These provide the color values used by the bitmap. Immediately following the RGBQUADS is the actual bitmap data. If the compression flag is set in the SHGIMAGEHEADER, then this data will be compressed (but the RGBQuad information will not be compressed). This is a fairly simple RLE compression algorithm, which I'll describe shortly.

Metafiles

Metafiles, as I said earlier, don't go through quite as drastic a change as bitmaps. The structure for a metafile can be seen in Figure 2.2.

MRBC and SHED only accept placeable metafiles. Placeable metafiles are metafiles that are preceded by the METAFILEHEADER structure (Table 2.5). Microsoft documents this in the API references (Volume 4, Chapter 3 of *Microsoft Windows 3.1 Programmer's Reference)* and API help files, but it doesn't provide the structure in WINDOWS.H.

The key field is the value 0x9AC6CDD7L. This of course has a deep cosmic meaning that only Bill Gates is aware of, or maybe it's his phone number in hex.

The hmf and reserved fields are unused and must be set to 0. The bbox field has the bounding box rectangle for the image. The values are in metafile units. The inch field tells how many metafile units there are to an inch. This value is usually 576 or 1000. It should definitely be less than 1440.

Listing 2.2 (continued)

```
void ReadBMHeader(FILE *SHGFile, SHGBITMAPHEADER *SHGBM) {
    fread(&(SHGBM->sbIsZero), 1, 1, SHGFile);
    SHGBM->sbDPI = ReadWordVal(SHGFile);
    fread(&(SHGBM->sbTwoHund), 2, 1, SHGFile);
    SHGBM->sbNumBits = ReadWordVal(SHGFile);
    SHGBM->sbWidth = ReadDWordVal(SHGFile);
    SHGBM->sbHeight = ReadDWordVal(SHGFile);
    SHGBM->sbNumQuads = ReadDWordVal(SHGFile);
    fread(&(SHGBM->sbNumImp), 2, 1, SHGFile);
    SHGBM->sbCmpSize = ReadDWordVal(SHGFile);
    SHGBM->sbSizeHS = ReadDWordVal(SHGFile);
    fread(&(SHGBM->sbunk1), 4, 1, SHGFile);
    fread(&(SHGBM->sbSizeImage), 4, 1, SHGFile);
)
```

The checksum field is an XORed sum of the first 10 words of the METAFILEHEADER structure. There's a safety feature! In case you were reading a text file by mistake that begins with 0x9AC6CDD7L, this is where you can be sure it's really a metafile.

Figure 2.2 Placeable metafile layout.

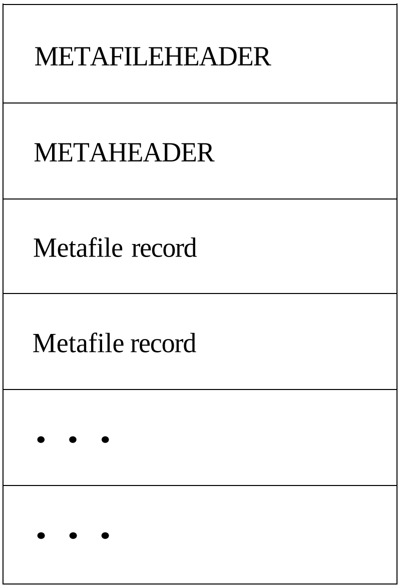


Table 2.5 METAFILEHEADER record.

Field Name	*Data Type*	*Comments*
key	DWORD	0x9AC6CDD7L
hmf	HANDLE	**unused; must be 0**
bbox	RECT	**Bounding rectangle for image**
inch	WORD	**Metafile units per inch**
reserved	DWORD	**Unused; must be 0**
checksum	WORD	**XORed sum of first 10 WORDs of structure.**

This is followed immediately by the METAHEADER record (Table 2.6). This header is described in the *Microsoft Windows 3.1 Programmer's Reference*, Volumes 3 and 4. The only field that should change in this record when you create a .MRB or .SHG is the mtSize field. MRBC or SHED will make two modifications to a metafile. First, it will discard the METAFILEHEADER record and add SetWindowOrg() and SetWindowExt() functions to the metafile. I'll discuss why this is done later. For now, though, you need to know that the mtSize field is changed because of this.

This is immediately followed by a string of metafile records (Table 2.7). The rdSize field contains the size of the metafile record, which varies depending on the number of parameters in rdParm. rdFunction can be any of the metafile-supported GDI functions.

When MRBC or SHED reads a metafile, it discards the METAFILEHEADER structure (but it is required to be in the original metafile). The information from the bbox field is used to add two metafile records to the beginning of the metafile itself. The first record is a 0x020B [SetWindowOrg()] function and the second is a 0x020C [SetWindowExt()]. These provide the dimensions of the metafile. There is an exception. If the metafile

Table 2.6 METAHEADER record.

Field Name	*Data Type*	*Comments*
mtType	UINT	Always 1 for .MRBs and .SHGs
mtHeaderSize	UINT	Size of this header in WORDs (9 WORDs)
mtVersion	UINT	0x300 if it contains DIBs, else 0x100
mtSize	DWORD	Size of metafile in WORDs
mtNoObjects	UINT	Maximum number of objects that exist in the metafile simultaneously
mtMaxRecord	DWORD	Size in WORDs of the largest record in the metafile
mtNoParameters	UINT	Reserved

Table 2.7 Typical metafile record.

Field Name	***Data Type***	***Comments***
rdSize	DWORD	**Size of metafile record**
rdFunction	WORD	**GDI function**
rdParm[]	WORD	**Parameters for GDI function**

already has a `SetWindowExt()` function in it, MRBC or SHED simply ignores the `METAFILEHEADER` structure altogether. If there is a `SetWindowExt()` but no `SetWindowOrg()`, `SetWindowOrg()` is assumed to be at `0,0`.

Other than this alteration, the metafile remains more or less intact. The only other change is that it is usually compressed.

.MRB Compression

MRBC and SHED use a simple RLE (Run Length Encoding) compression algorithm to compress data. The help compiler, when importing bitmaps and converting them to the `.MRB/.SHG` images, sometimes uses an LZ77 compression algorithm. I'll show the code as it would apply to these images, but Chapter 7 will have a much more in-depth discussion of the LZ77 algorithm in general, and Chapter 4 will talk more about the implementation specific to WinHelp. Because this only occurs with WinHelp, I will save most of the discussion about how bitmaps are handled in WinHelp for Chapter 4.

Compression is applied to both bitmaps and metafiles. With bitmaps, the compression begins on the data immediately following the RGBQuad list and continues through all of the bitmap data. With metafiles, the compression begins immediately after the `SHGMETAFILEHEADER` and continues through the entire metafile.

RLE compression basically works as follows: When you get a series, or "Run", of bytes that are identical, say 20 zeros, instead of keeping those 20 zeros, you put in a flag, a 20, and a zero. So instead of writing 20 bytes of zeros, you're writing three pieces of information: A flag that indicates compression, the character that's repeated, and the number of times that character is repeated. That's the theory; the implementations vary. In this case, there are seven simple steps to the decompression algorithm:

1. If not at the end of the data, read 1 byte into `count`, else quit.
2. If bit 8 of `count` is set, `goto` step 5.
3. Copy next input byte `count` times to the output file.
4. Goto step 1.
5. Subtract 0x80 from `count` (unset bit 8).
6. Copy next `count` bytes from the input file to the output file.
7. Go to step 2.

Listing 2.3 shows the code for the routine `doRLE()`, which decompresses data from an input file to an output file. `doRLE()` has three parameters: the input file, the output file, and the number of bytes to decompress in the input file. It then returns the size of the data after it's expanded.

Listing 2.3 doRLE().

```
/**********************************************************************
 *
 * PROGRAM: doRLE()
 *
 * PURPOSE: Decompress data from an input file to an output file.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 2, Multiresolution Bitmap (.MRB) File Format,
 * from Undocumented Windows File Formats,
 * published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 **********************************************************************/

DWORD doRLE(FILE *InFile, FILE *OutFile, DWORD NumBytes) {
    DWORD i=1, imagesize=0;
    BYTE j, count, data;
    do {
        fread(&count, sizeof(count), 1, InFile);
        i++;
        /* If the 8th bit is set, lower 7 bits has # bytes to copy */
        if (count & 0x80) {
            count -=0x80;
            while (count--) {
                fread(&data, sizeof(data), 1, InFile);
                i++;
                fwrite(&data, sizeof(data), 1, OutFile);
            }
        }
        /* If 8th bit not set, it's an RLE count */
        else {
            count = count & 0x7F;
            fread(&data, sizeof(data), 1, InFile);
            i++;
            /* Write out uncompressed */
            for (j = 1; j <= count; j++, imagesize++)
                fwrite(&data, sizeof(data), 1, OutFile);
        }
    }
    while(i < NumBytes);
    return imagesize;
}
```

The LZ77 algorithm is quite a bit more complex. Again, we'll discuss it more in-depth in Chapter 7, but for now, the code here shows how we handle it. Listing 2.4 shows the `doLZ77V3()` routine. This is the version of LZ77 used by WinHelp. (Two other versions, much like this routine, are supported by `COMPRESS.EXE` and `LZEXPAND.DLL` See Chapter 7 for more details.)

Where Do I Go from Here?

Clearly, the .MRB file format isn't used quite as much anymore. Mainly because almost everyone has Super VGA monitors. However, for those that are disassembling old help files (see Chapter 4), you can remove the .MRB files and extract the bitmaps from them. And because the SHED file format is still being used extensively, you'll need this knowledge to deal with SHED files.

Listing 2.4 doLZ77V3().

```
/*********************************************************************
 *
 * PROGRAM: doLZ77V3()
 *
 * PURPOSE: The version of LZ77 used by WinHelp
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 2, Multiresolution Bitmap (.MRB) File Format,
 * from Undocumented Windows File Formats,
 * published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 *********************************************************************/

#define WINSIZE 4096
#define LENGTH(x) ((((x) & 0x0F)) + 3)
#define OFFSET(x1, x2) (((((x2&0xF0)>>4)*0x0100)+x1+0x0001)&0x0FFF)
#define FAKE2REAL_POS(x) ((x) & (WINSIZE - 1))
#define BITSET(byte, bit) (((byte) & (1<<bit)) > 0)

void doLZ77V3(FILE *InFile, FILE *OutFile, DWORD NumBytes) {

    unsigned char BitMap, byte1, byte2;
    int Length, counter, NumToRead;
    long Offset, CurrPos=0;
```

Listing 2.4 (continued)

```
    /* Window must initially be filled with spaces */
    for (counter = 0; counter < WINSIZE; counter ++)
        Window[counter] = ' ';
    /* Go through until we're done */
    while (CurrPos < NumBytes) {
        /* Get BitMap and data following it */
        BitMap = fgetc(InFile);
        if (feof(InFile)) return;
        NumToRead = BytesToRead(BitMap);
        /* Go through and decode data */
        for (counter = 0; counter < 8; counter++) {
            /* It's compressed, so decode it and copy the data */
            if (!BITSET(BitMap, counter)) {
                byte1 = fgetc(InFile);
                if (feof(InFile)) return;
                byte2 = fgetc(infile);
                Length = LENGTH(byte2);
                Offset = OFFSET(byte1, byte2);
                /* Copy data from 'window' */
                while (Length) {
                    byte1 = Window[FAKE2REAL_POS(Offset)];
                    Window[FAKE2REAL_POS(CurrPos)] = byte1;
                    fputc(byte1, OutFile);
                    CurrPos++;
                    Offset++;
                    Length--;
                }
            }/* if */
            /* It's not compressed, so copy the bytes */
            else {
                byte1 = fgetc(InFile);
                Window[FAKE2REAL_POS(CurrPos)] = byte1;
                fputc(byte1, OutFile);
                CurrPos++;
            }
            if (feof(InFile)) return;
        }/* for */
    }/* while */
}
```

Chapter 3

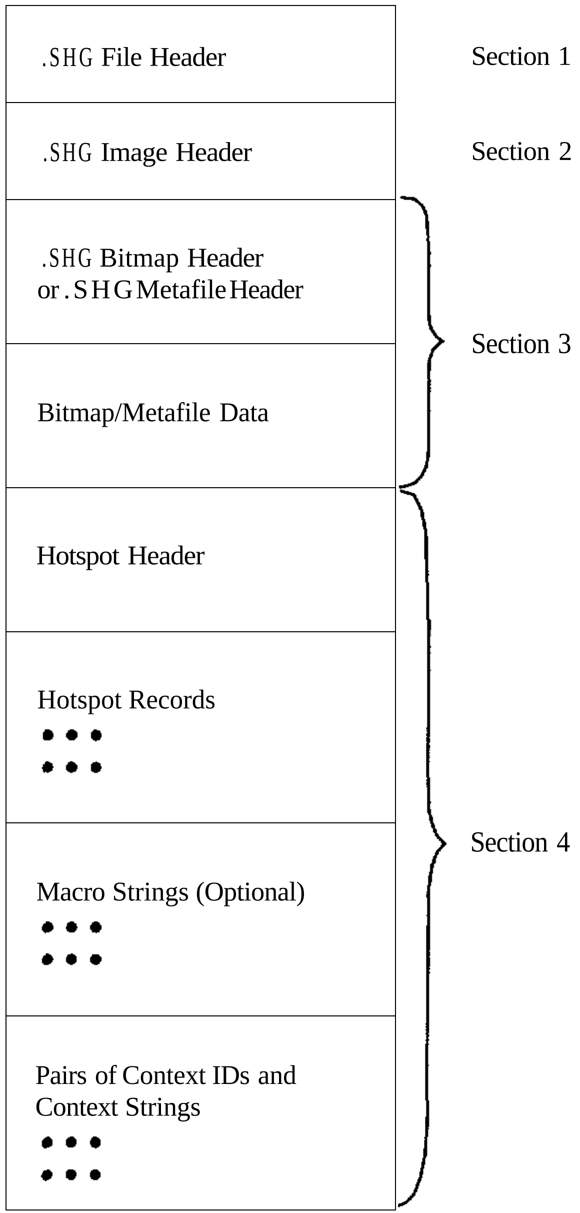
Segmented Hypergraphic (.SHG) FileFormat

Chapter 2 discussed the Multiresolution Bitmap (.MRB) file format. The Segmented Hypergraphic (.SHG) file format is almost identical. It provides a few key differences and some additional structures not normally found in a .MRB file. I say not normally, because the Multiresolution Bitmap Compiler (MRBC) is capable of taking .SHG files as input and retaining all hotspot data. Figure 3.1 shows a normal .SHG file. In a .MRB file with multiple .SHG files (hereafter referred to as a multi-image .SHG file), sections 3 and 4 are repeated instead of section 3 only.

The primary difference between the .SHG file and the .MRB file is the support for hotspots. Hotspots are rectangular areas you can define in a .SHG to produce an event in WinHelp. Three events are supported by WinHelp 3.1, including a topic jump, a topic popup, and macro execution.

SHED doesn't support multiple images within a .SHG file, but WinHelp does. As far as WinHelp is concerned, a multi-image .SHG file is the same as a .MRB with hotspots. My guess is that WinHelp doesn't really distinguish between .SHG files and .MRB files. MRBC and SHED do distinguish between them. MRBC will take a .SHG file as input, but it will truncate an input .MRB file to one image. SHED too will truncate a .MRB file to one image.

Figure 3.1 .SHG file layout.

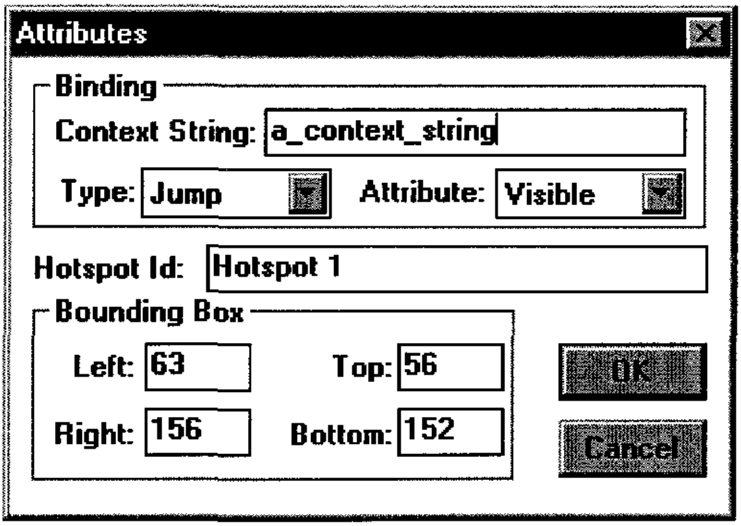


Hotspots

As stated earlier, the difference between .MRB and .SHG files is hotspots. Hotspots are kept in the last section (section 4 and see Figure 3.2) of a .SHG file. They too are broken into four general sections: hotspot header, hotspot records, macro strings, and pairs of context IDs and context strings.

The HOTSPOTHEADER (Table 3.1) has three fields. The hhVersion field is the version of hotspot records you are dealing with. The hhNumHS field tells you how many hotspots are defined in this .SHG file. hhContextOffset has the offset to the list of context strings and context IDs relative to the end of the array of HOTSPOTRECORDs (see below).

Figure 3.2 Hotspot Attributes dialog box.


Attributes

Binding

Context String: a_context_string

Type: Jump

Attribute: Visible

Hotspot Id: Hotspot 1

Bounding Box

Left: 63

Top: 56

Right: 156

Bottom: 152

Table 3.1 HOTSPOTHEADER record.

Field Name	*Data Type*	*Comments*
hhVersion	BYTE	Always 0x01
hhNumHS	WORD	Number of hotspots
hhContextOffset	DWORD	Offset to context strings and context IDs

The HOTSPOTHEADER is followed by an array of HOTSPOTRECORDs (Table 3.2), one for each hotspot. Figure 3.2 shows the hotspot Attributes dialog box from SHED. These values are reflected in the HOTSPOTRECORD.

Valid hrType values are:

- 0x0042 = visible popup
- 0x0043 = visible jump
- 0x00C8 = visible macro
- 0x04E6 = invisible popup
- 0x04E7 = invisible jump
- 0x04CC = invisible macro

hrBox contains the bounding rectangle of the hotspot. The values contain the left, top, right, and bottom sides of the rectangle relative to the upper left corner of the image in pixels.

The hrMacOffset field contains the offset to the macro string, if this hotspot is a macro. If it is not a macro, this value can be ignored. As shown in Figure 3.1, the list of macros immediately follows the array of HOTSPOTRECORDs. The offsets in hrMacOffsets are relative to the start of the macro list, so the offset to the first macro would be 0. Each macro is a null-terminated string, so if the first macro was Next(), for example, this string would be null-terminated and the second macro, if there was one, would start at an offset of 6.

Immediately following the macro list is an array of context IDs (called hotspot IDs by SHED, Figure 3.2) and context strings. Because the context string is also the macro string for macros, macros are essentially listed twice, once in the macro string list following the HOTSPOTRECORDs and again in this list. Each context ID and context string is a null-terminated string.

We've included the source code for SHGDUMP, an MS-DOS program that will read an .SHG or .MRB file. This program is meant only to illustrate reading hotspots from such files, and could be used as a starting point for a more elaborate program, such as a graphics editor.

Table 3.2 HOTSPOTRECORD record.

Field Name	*Data Type*	*Comments*
hrType	WORD	**Hotspot type**
hrZero	BYTE	**Always 0**
hrBox	RECT	**Bounding box of hotspot**
hrMacOffset	DWORD	**Offset to macro data**

Where Do I Go from Here?

.SHG files are used quite a bit in WinHelp files. Utilities for working with .SHG files could easily be created using the information in this chapter. For example, a program to modify the tab order of the hotspots in an .SHG file could be quickly built using the source code in this chapter. You could also extract the bitmaps from an existing .SHG file so you can modify them with your favorite graphics editor. You could even write a converter to create image maps and graphics for your HTML documents, which is one step of an .HLP to HTML converter.

Listing 3.1   SHEDEDIT.H.

```
/*************************************************************************
 *
 * PROGRAM: SHEDEDIT.H
 *
 * PURPOSE: Header file for SHGDUMP.C.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 3, Segmented Hypergraphic (.SHG) File Format,
 * from Undocumented Windows File Formats, published by R&D Books,
 * an imprint of Miller Freeman, Inc.
 *
 *************************************************************************/

/* Structures for .SHG files */

/* SHG File Header */
typedef struct tagSHGFILEHEADER
{
  char  sfType[2];    /* Must be 'lp' or 0x706C     */
  WORD  sfNumObjects; /* Number of objects in file  */
  DWORD *sfObjectOff; /* Offsets to objects in file */
}
SHGFILEHEADER;

/* SHG Image Header */
typedef struct tagSHGIMAGEHEADER
{
  BYTE  siImageType;      /* 0x06=.BMP  0x08=.WMF                 */
  BYTE  siCompression;    /* 0x00 = None, 0x01 = RLE, 0x02= LZ77 */
  BYTE  siDPI;            /* Dots Per Inch x 2 (0x10 for .wmf)   */
}
SHGIMAGEHEADER;
```

Listing 3.1 (continued)

```
/* Defines for image type and compression type */
#define IT_BMP   0x06
#define IT_WMF   0x08
#define IC_NONE  0x00
#define IC_RLE   0x01
#define IC_LZ77  0x02
#define IC_BOTH  0x03

/* SHG Bitmap Header */
typedef struct tagSHGBITMAPHEADER
{
  BYTE  sbIsZero;       /* Always 0x00                  */
  BYTE  sbDPI;          /* Dots Per Inch x 2            */
  WORD  sbTwoHund;      /* 0x0200                       */
  WORD  sbNumBits;      /* Number bits per pixel x 2    */
  DWORD sbWidth;        /* Width x 2                    */
  DWORD sbHeight;       /* Height x 2                   */
  DWORD sbNumQuads;     /* Number RGB Quads x 2         */
  WORD  sbNumImp;       /* Number of 'important' RGB Qds */
  DWORD sbCmpSize;      /* Size of Compressed BMP x 2   */
  DWORD sbSizeHS;       /* Size of Hotspot Data area x 2 */
  DWORD sbunk1;
  DWORD sbSizeImage;    /* size ImageHdr+BmpHdr+ImageDat */
}
SHGBITMAPHEADER;

/* SHG Metafile Header */
typedef struct tagSHGMETAFILEHEADER
{
  WORD  smXWidth;    /* Width of image in metafile units  */
  WORD  smYHeight;   /* Height of image in metafile units */
  DWORD smUncSize;   /* Size of uncompressed metafile     */
  DWORD smCmpSize;   /* Size of compressed metafile       */
  DWORD smSizeHS;    /* Size of hot spot data area x 2    */
  DWORD smUnk1;
  DWORD smSizeImage; /* Size ImageHdr+wmfHdr+ImageDat     */
}
SHGMETAFILEHEADER;
```

Listing 3.1 (continued)

```
/* Documented in "Microsoft Windows 3.1 Programmer's
   Referenece Volume 4, Resources." Not in WINDOWS.H.
   This is the header for a Placeable Metafile       */
typedef struct tagMETAFILEHEADER
{
  DWORD  key;
  HANDLE hmf;      /* Must be 0 */
  RECT   bbox;
  WORD   inch;
  DWORD  reserved; /* Must be 0 */
  WORD   checksum;
}
METAFILEHEADER;

/* Hot Spot Header */
typedef struct tagHOTSPOTHEADER
{
  BYTE  hhOne;            /* Always 0x01                  */
  WORD  hhNumHS;          /* Number of Hot Spots          */
  DWORD hhContextOffset; /* Offset to Cntxt Strings & IDs */
}
HOTSPOTHEADER;

/* Hot Spot Record */
typedef struct tagHOTSPOTRECORD
{
  WORD  hrType; /* Hot Spot Type. See below */
  BYTE  hrZero; /* Always 0                 */
  WORD  hrLeft; /* Bounding box of Hot Spot */
  WORD  hrTop;
  WORD  hrRight;
  WORD  hrBottom;
  DWORD hrMacOffset; /* Offset to macro for Hot Spot */
}
HOTSPOTRECORD;

#define HS_INVISJUMP  0x04E7
#define HS_INVISPOPUP 0x04E6
#define HS_INVISMACRO 0x04CC
#define HS_VISJUMP    0x00E3
#define HS_VISPOPUP   0x00E2
#define HS_VISMACRO   0x00C8
```

Listing 3.2 SHGDUMP.C.

```
/*******************************************************************
 *
 * PROGRAM: SHGDUMP.C
 *
 * PURPOSE: Dump the hotspot information for a SHG file
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 3, Segmented Hypergraphic (.SHG) File Format,
 * from Undocumented Windows File Formats, published by R&D Books,
 * an imprint of Miller Freeman, Inc.
 *
 *******************************************************************/


#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "shededit.h"

/* Flags an incorrect value for known fields */
#define CheckUVal(a,b) \
       {if (a != b) {  \
       printf("UINT: Not a 0x%04X?\n\n 0x%04x\n",b, a); exit(1); } }

#define CheckBVal(a,b) \
       {if (a != b) {  \
       printf("BYTE: Not a 0x%02X?\n\n 0x%02x\n",b, a); exit(1); } }

#define ReadString(f, s) { char *p = (char *)(s); \
                  while ((*p++ = fgetc(f)) != 0) ; *p = 0; }


/* Reads in a BYTE. If it's odd, it reads in a WORD. Returns 1/2 value */
WORD ReadWordVal(FILE *SHGFile)
{
  BYTE a, b;

  b = 0;
  fread(&a, sizeof(a), 1, SHGFile);
  if (a % 2) fread(&b, sizeof(b), 1, SHGFile);
  return (WORD) ((WORD)b*256 + a) /2;
}


/* Reads in a WORD. If it's odd, it reads in a DWORD. Returns 1/2 value */
DWORD ReadDWordVal(FILE *SHGFile)
{
  WORD a, b;

  b = 0;
  fread(&a, sizeof(a), 1, SHGFile);
  if (a % 2) fread(&b, sizeof(b), 1, SHGFile);
  return (DWORD) ((DWORD)b*65536 + a) /2;
}
```

Listing 3.2 (continued)

```
/* Dumps HotSpot data from .SHG file */
void DumpHotSpots(FILE *SHGFile)
{
  HOTSPOTHEADER HSHead;
  HOTSPOTRECORD HSRec;
  WORD i, NumMacs = 0;
  char AString[128];
  long fileLoc;

  fread(&HSHead, sizeof(HSHead), 1, SHGFile);

  for(i = 1; i <= HSHead.hhNumHS; i++)
  {
    fileLoc = ftell(SHGFile);
    fread(&HSRec, sizeof(HSRec), 1, SHGFile);
    printf("Hot Spot %u (offset %ld) - ", i, fileLoc);
    switch(HSRec.hrType)
    {
      case HS_INVISJUMP:  printf("Invisible Jump\n");  break;
      case HS_VISJUMP:    printf("Visible Jump\n");    break;
      case HS_INVISPOPUP: printf("Invisible Popup\n"); break;
      case HS_VISPOPUP:   printf("Visible Popup\n");   break;

      case HS_INVISMACRO:
        printf("Invisible Macro\n");
        NumMacs++;
        break;

      case HS_VISMACRO:
        printf("Visible Macro\n");
        NumMacs++;
        break;

      default:
        printf("Invalid Record Type\n");
        return;
    }
  }

  /* Print out the list of macros */
  for (i = 1; i <= NumMacs; i++)
  {
    ReadString(SHGFile, AString);
    printf("Macro %u> %s\n", i, AString);
  }

  printf("\n");

  /* Print out the list of hotspot IDs */
  for(i = 1; i <= HSHead.hhNumHS; i++)
  {
    ReadString(SHGFile, AString);
    printf("Hotspot ID #%u> %s\n", i, AString);
    ReadString(SHGFile, AString);
    printf("Context String-> %s\n\n", AString);
  }
}
```

Listing 3.2 (continued)

```
/* Reads in a bitmap header. Doubles the
   size of certain fields when necessary. */
void ReadBMHeader(FILE *SHGFile, SHGBITMAPHEADER *SHGBM)
{
  /* Read first three fields unmodified */
  fread(&(SHGBM->sbIsZero), 1, 1, SHGFile);
  SHGBM->sbDPI = ReadWordVal(SHGFile);
  fread(&(SHGBM->sbTwoHund), 2, 1, SHGFile);
  SHGBM->sbNumBits = ReadWordVal(SHGFile);
  SHGBM->sbWidth = ReadDWordVal(SHGFile);
  SHGBM->sbHeight = ReadDWordVal(SHGFile);
  SHGBM->sbNumQuads = ReadDWordVal(SHGFile);
  fread(&(SHGBM->sbNumImp), 2, 1, SHGFile);
  SHGBM->sbCmpSize = ReadDWordVal(SHGFile);
  SHGBM->sbSizeHS = ReadDWordVal(SHGFile);
  fread(&(SHGBM->sbunk1), 4, 1, SHGFile);
  fread(&(SHGBM->sbSizeImage), 4, 1, SHGFile);
}


/* Dump bitmap data in .SHG file */
void BitMapDump(FILE *SHGFile)
{
  RGBQUAD         AnRGBQuad;
  DWORD           i=0L;
  SHGBITMAPHEADER SHGBM;

  ReadBMHeader(SHGFile, &SHGBM);

  /* Read past bitmap quad info */
  for(i = 0L; i < SHGBM.sbNumQuads; i++)
  {
    fread(&AnRGBQuad, sizeof(AnRGBQuad), 1, SHGFile);
  }

  /* Jump past the image to the hotspot data */
  fseek(SHGFile, SHGBM.sbCmpSize, SEEK_CUR);

  if (SHGBM.sbSizeHS)
    DumpHotSpots(SHGFile);
  else
    printf("No Hot Spot data for this Bitmap.\n");
}
```

Listing 3.2 (continued)

```
/* Reads in a metafile header. Doubles the
   size of certain fields when necessary. */
void ReadWMHeader(FILE *SHGFile, SHGMETAFILEHEADER *SHGWM)
{
  /* Read first two fields unmodified */
  fread(&(SHGWM->smXWidth), 4, 1, SHGFile);
  SHGWM->smUncSize = ReadDWordVal(SHGFile);
  SHGWM->smCmpSize = ReadDWordVal(SHGFile);
  SHGWM->smSizeHS  = ReadDWordVal(SHGFile);
  fread(&(SHGWM->smUnk1), 4, 1, SHGFile);
  SHGWM->smSizeImage = ReadDWordVal(SHGFile);
}


/* Dump the .WMF data from the .SHG file */
void WMFDump(FILE *SHGFile, BOOL bCompUsed)
{
  SHGMETAFILEHEADER SHGWM;

  ReadWMHeader(SHGFile, &SHGWM);

  /* Jump to the hot spot information (how far we jump */
  /* is based on whether the image is compressed) */
  if (bCompUsed)
    fseek(SHGFile, SHGWM.smCmpSize, SEEK_CUR);
  else
    fseek(SHGFile, SHGWM.smUncSize, SEEK_CUR);

  if (SHGWM.smSizeHS)
    DumpHotSpots(SHGFile);
  else
    printf("No Hot Spot Data for this metafile.\n");
}


/* Dumps the .SHG file */
void SHGDump(FILE *SHGFile)
{
  SHGFILEHEADER SHGHead;
  SHGIMAGEHEADER SHGImage;
  WORD i;
  BOOL bCompUsed;

  /* Read in first 4 bytes */
  fread(&SHGHead, 4, 1, SHGFile);
  printf("Number of images = %d\n", SHGHead.sfNumObjects);
  SHGHead.sfObjectOff=malloc(4*SHGHead.sfNumObjects);
  fread(SHGHead.sfObjectOff, 4, SHGHead.sfNumObjects, SHGFile);
```

Listing 3.2 (continued)

```
  /* Make sure it is an .SHG file */
  if (strncmp(SHGHead.sfType, "lp", 2))
  {
    printf("Invalid .SHG or .MRB file! \n\nType: %c%c\n", SHGHead.sfType[0],
           SHGHead.sfType[1]);
    exit(1);
  }

  for (i = 0; i < SHGHead.sfNumObjects; i++)
  {
    /* Jump to Image Header and read it in */
    fseek(SHGFile, SHGHead.sfObjectOff[i], SEEK_SET);
    fread(&SHGImage, sizeof(SHGImage), 1, SHGFile);

    if (SHGImage.siImageType == IT_BMP)
      printf("\nFile is a BITMAP using ");
    else
      printf("\nFile is a METAFILE using ");

    if (SHGImage.siCompression == IC_NONE)
      printf("no compression.\n");
    else if (SHGImage.siCompression == IC_RLE)
      printf("RLE compression.\n");
    else if (SHGImage.siCompression == IC_LZ77)
      printf("LZ77 compression.\n");
    else if (SHGImage.siCompression == IC_BOTH)
      printf("RLE and LZ77 compression.\n");
    else
      printf("unknown compression.\n");

    bCompUsed = (SHGImage.siCompression == IC_NONE) ? FALSE : TRUE;

    if (SHGImage.siImageType == IT_BMP)
      BitMapDump(SHGFile);
    else if (SHGImage.siImageType == IT_WMF)
      WMFDump(SHGFile, bCompUsed);
    else {
      printf("Unknown sfImageType value.\n");
      exit(1);
    }
  }

  free(SHGHead.sfObjectOff);
}


/* Show usage for SHGDUMP */
void Usage(void)
{
  printf("Usage:\n");
  printf(" SHGDUMP shgfile[.shg] \n");
  printf("   shgfile  - Name of .SHG file\n");
}
```

Listing 3.2 (continued)

```
/* main routine */
int main(int argc, char *argv[])
{
  char inputFile[20];
  FILE *SHGFile;

  /* Check if the program was invoked correctly */
  if (argc < 2) {
    Usage();
    return EXIT_FAILURE;
  }

  /* Save the input filename */
  strcpy(inputFile, argv[1]);

  /* If no extension in the input filename, assume .shg */
  if (0 == strchr(inputFile, '.'))
    strcat(inputFile, ".SHG");

  /* Check that the input file exists */
  if ((SHGFile = fopen(inputFile, "rb")) == NULL)
  {
    printf("%s does not exist!", inputFile);
    return EXIT_FAILURE;
  }

  /* Dump the hotspot information */
  SHGDump(SHGFile);

  /* Close the file and exit */
  fclose(SHGFile);
  return EXIT_SUCCESS;
}
```

Chapter 4

Windows Help File Format

Since this information was first published in *Dr. Dobb's Journal,* many mistakes have been corrected and a lot of information has been added. In addition to many internal HFS files that were intentionally omitted due to space considerations, the |TOPIC file is described in much greater detail and is now complete.

Because of all of these changes, some structures have been modified or renamed. Although my articles in *Dr. Dobb's Journal* were a good start, the information in this book is much more accurate and up-to-date.

It should be clear that this description applies only to WinHelp 3.1 and WinHelp 4.0. WinHelp 3.0 was significantly different and is, for all intents and purposes, a dead product. Although a lot of the information here will apply to WinHelp 3.0, some key areas differ, including the layout of the internal |TOPIC file, which is where the actual topic text and layout information is kept.

In this chapter, I'll lay out the different parts of the WinHelp .HLP file and then provide a dump program that lets you view internal WinHelp files.

Overview

WinHelp, on the surface, may not seem all that complex, and unless you've actually developed WinHelp .HLP files, it wouldn't occur to you how incredibly complex they can get. This format has been able to easily handle incredibly large and complex files like the MSDN-CD, Cinemania, and so on which get as large as 300Mb-400Mb. Obviously Microsoft had some forethought in developing the format in the sense that they knew it should be able to handle very large files. On the other hand, there are many instances where the structures and fields in structures don't make a lot of sense. After talking with various people about the format (though no one at Microsoft who would know), it appears that the WinHelp file format was probably developed by a single person who, in the process of developing it, made ad hoc modifications to handle new features. Because of this, WinHelp .HLP files can be very complex and messy. I would hazard a guess that this is the main reason Microsoft never released the file format.

The basic structure of .HLP files is that of multiple files. A .HLP file has an internal structure called the Help File System (HFS). The HFS is like a single directory in DOS and contains, simply, a list of filenames and pointers to where those files are in relation to the beginning of the help file. Each of these files, in one way or another contributes something to the help file, such as keywords, context strings, font information, and so on. All of these are then used together to render the help file on-screen. It's not that all of these files are terribly complex, but that the combination of them all, and using them all together, is very complex. As I discuss the different parts of the .HLP file format, I'll try to give insights into how Microsoft uses these files and ways you can use them to enhance WinHelp.

WinHelp B-Trees

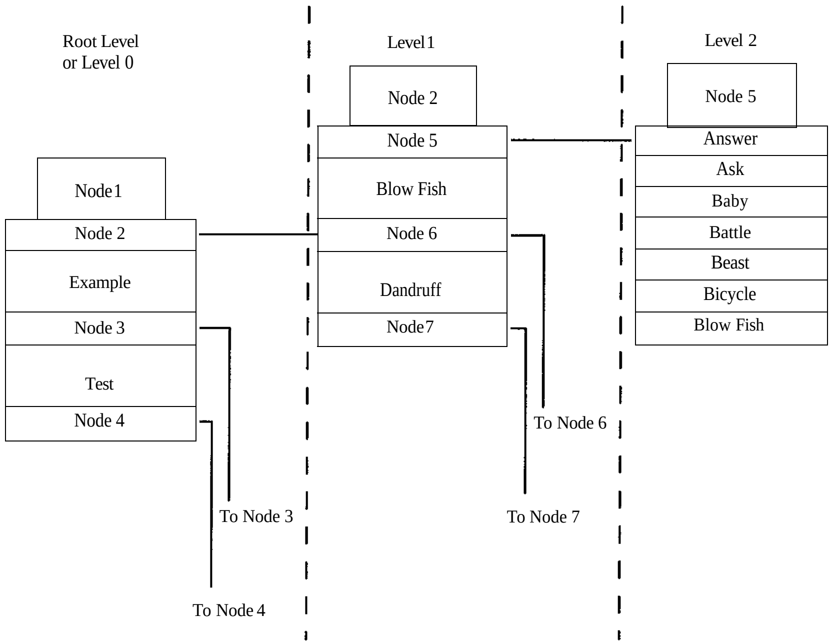
A WinHelp .HLP file is a combination of many files. These files are kept internally in what is called the Help File System (discussed later). The HFS and some of the data files kept internally in WinHelp files are organized into b-trees. B-trees may be familiar to those of you who didn't sleep through your data structures classes in school. Until I started working on the WinHelp file, I didn't really know much about b-trees, I think I slept in class that day. To refresh my memory, I went back to some old data structures books that use phrases like "branching factor", and they calculate disk accesses with logarithms. Not exactly my cup of tea, so I'm going to try to explain them in English.

A b-tree is a structure made up of nodes or pages. There are two types of nodes, index nodes and leaf nodes (Figure 4.1). Index nodes contain a list of "keys" and links to other nodes. All of the keys are in alphabetical order. In Figure 4.1, Level-0 and Level-1 nodes are all index nodes. Say you're looking for the word "Beast". To find it, search Node 1 first. You don't find it there, but "Beast" alphabetically comes before "Example", our first key, so you know to go to Node 2 because there is a link pointer to Node 2 before "Example". From Node 2 you see that, again, "Beast" comes before "Blow Fish", so you go on to Node 5. Node 5 is a leaf node. All nodes on the highest level, in this case, Level 2, are leaf nodes. Leaf nodes contain the data you're searching for. From Node 5, simply perform a linear search to find the word you're looking for.

The b-tree format is very efficient in WinHelp. The reason is this: If a WinHelp file has 3,000 internal files, it will need about 15 to 18, 1Kb nodes to store the list of files. If you have to search this file in a linear search, you'd average about 7 node-sized disk reads. On the other hand, if the file list is in a b-tree format, 3,000 files could be kept in 15 to 18 nodes of a two-level b-tree, meaning you'd have to read exactly two node-sized pages from disk before you had the node with your data. From there, a simple linear search of the node would provide the filename quickly. Although b-trees save little time for small files (100Kb or less), for larger files, it can provide a tremendous boost in performance. Microsoft was obviously looking ahead to the days of 300Mb help files.

These large numbers can come up in other places too, such as the Topic Titles b-tree, KeyWord b-trees, and so on. These are all kept in internal files and can grow quite large with large .HLP files.

Figure 4.1 B-tree structure made up of index nodes and leaf nodes.



Help File Header

Each .HLP file begins with a HELPHEADER (Table 4.1). This header simply has a magic number that identifies this file as a WinHelp .HLP file, a pointer to the WHIFS (discussed below), and a FileSize field. There is also a reserved field, which should always be set to -1. The FileSize is really only useful as a sanity check and should match the size of the file displayed by DOS. I suppose it could be useful if you wanted to read the entire file into memory, but there's really no reason to do that.

The HFSLoc field contains the offset to the beginning of the Help File System inside the .HLP file.

The Help File System (HFS)

WinHelp files are based on a structure called the HFS, or Help File System. In my article in *Dr. Dobb's Journal*, I referred to this as the WHIFS (WinHelp Internal File System). It turns out that Microsoft has provided some documentation on this (not much, though), so I thought it would be less confusing (and two characters shorter) to adopt their naming convention.

The HFS is a directory, much like a DOS directory, that contains a list of filenames and offsets to those files within the help file. All of these "files" are actually inside the WinHelp file. When the help compiler generates a .HLP file, it builds temporary files, in the form of true DOS files, that contain various types of information. When it has finished building all of these temporary files, it then combines them into a single .HLP file and generates the HFS, which has pointers to these different internal files.

The help compiler also allows you to import "baggage" files into your WinHelp file. This essentially brings a DOS file into a WinHelp file and provides a pointer in the HFS to this file. WinHelp exports several functions for dealing with baggage files. These functions can also be used to access other HFS files.

Table 4.1 HELPHEADER record.

Field Name	*Data Type*	*Comments*
MagicNumber	DWORD	0x00035F3F
HFSOff	long	Offset to HFS
Reserved	long	-1
FileSize	long	Size of entire .HLP file

The following is a list of the internal files you are going to run into with a short description of each one. Notice that most of the internal files generated by WinHelp begin with a "|" (pipe) character. All filenames in the HFS are case sensitive.

|CONTEXT Contains a list of hash values, generated from context words, and offsets into the |CTXOMAP file.

|CTXOMAP Lists all the topics from the [MAP] section of the Help Project File (.HPJ) with a Map number (from the .HPJ file), and an offset to the actual topic data.

|FONT Contains a list of fonts and font descriptors. These are used to display text in the proper fonts within the topics.

|KWBTREE, |KWDATA, |KWMAP (as well as |AWBTREE and |AWDATA, discussed later) These three files provide access to the keyword list and the topics associated with the keywords. Using the Multikey option will get you another set of files. For example, using the Multikey option with the letter "L" will add |LWBTREE, |LWDATA, and |LWMAP.

|Phrases Contains a list of phrases that WinHelp uses to provide extra compression of topic text. This list may also be compressed with LZ77 compression.

|PhrImage and |PhrIndex These are used by Hall compression. As discussed later, we were unable to decipher the formats used here.

|SYSTEM This contains various pieces of information about the help file, including the date the file was generated, the version of the compiler, the type of compression, etc. It also contains a lot of information that was listed in the Help Project File (.HPJ), such as copyright notice, secondary window information, etc.

|TOPIC This is the biggest and most complex of all the internal files. This file contains all of the actual text from the topics, including formatting information.

|TTLBTREE Contains a list of topic titles with their associated offsets into the |TOPIC file.

|bmx These are bitmap files referred to by the topics. "x" is a sequential whole number beginning at zero. If you have three bitmaps, they'd be referred to as |bm0, |bm1, and |bm2. As a side note, in version 3.0 help files, these filenames are the same except the "|" (pipe) character did not precede the name.

Baggage files Baggage files retain their case-sensitive filenames and extensions exactly as they were specified in the .HPJ file (any path information is discarded). If in the .HPJ you refer to a file as C:\MYPATH\FiLeNaMe.ExT, it will be stored in the help file with the same case, leaving you with an internal file called FiLeNaMe.ExT.

The first 9 bytes of every file in a WinHelp file is the HFSFILEHEADER record (Table 4.2). The HFS itself is considered a file (although it doesn't list itself in the HFS directory), so even the HFS has an HFSFILEHEADER record. This record contains three fields. The FilePlusHeader field is the size of the file plus the 9 bytes of the header. The second field, FileSize, is the size of the file without the header. Why are both values included? Beats me. The first is always 9 bytes larger than the second. I suppose it was to allow for the possibility of having a different HFSFILEHEADER record, although I haven't come across one yet.

Table 4.2 HFSFILEHEADER record.

Field Name	*Data Type*	*Comments*
FilePlusHeader	long	Size of HFS file in bytes + 9-byte header
FileSize	long	Size of file without header
FileType	char	1-byte file type

Table 4.3 HFSBTREEHEADER record.

Field Name	*Data Type*	*Comment*
Signature	WORD	Signature for header, always 0x293B
Unknown1	char	Always 0x02
FileType	char	Same as in FILEHEADER record
PageSize	int	Size of the b-tree pages
SortOrder[16]	char	Describes sort order
FirstLeaf	int	First leaf page number
NSplits	int	Number of splits in b-tree
RootPage	int	Page number of root page
FirstFree	int	First free page
TotalPages	int	Total number of pages in tree
NLevels	int	Number of levels in tree
TotalHFSEntries	DWORD	Total number of entries in HFS b-tree

The last field is FileType, which takes one of two values: FT_NORMAL (0x00), which is any normal file, and FT_HFS (0x04), which is used in the HFSFILEHEADER record for the HFS.

```
#define  FT_NORMAL      0x00
#define  FT_HFS         0x04
```

The first 9 bytes of the HFS, therefore, will be an HFSFILEHEADER record. The FileSize and FilePlusHeader fields will tell you how large the entire HFS is. The FileType field should always be FT_HFS. This is the only time I'll really describe the HFSFILEHEADER record. From now on, when I discuss the first record of a file, I will mean the first record following the HFSFILEHEADER record. For example, if I'm talking about the |TTLBTREE file, I will say that the first record is the BTREEHEADER record. It is assumed that the file header record has already been read.

As mentioned earlier, the HFS is organized into a b-tree. So, the first record in the HFS (following the HFSFILEHEADER record, of course), is the BTREEHEADER record (Table 4.3).

The FileType byte is the same as the byte in the HFSFILEHEADER record, and they should always match. (As you will see, this sort of redundant information pops up in WinHelp quite often.)

The next field, PageSize, tells how large the individual pages of the b-tree are. For the HFS, this always appears to be 1Kb for help files, but it's probably best to use this field and be able to handle different-sized b-tree pages. In fact, as you'll see in Chapter 5, Annotation and Bookmark files use a different page size. For those of you that go on to write your own help compiler, I would suggest trying to come up with an algorithm that optimizes the HFS b-tree page size, not only for speed, but for size. This is one area where a few kilobytes of file space can be saved, especially with smaller help files.

The PageSize field is followed by 17 characters that I call SortOrder here. This is just a guess, but it appears that different language versions of the help compiler produce different values for this field. Sometimes only part of the field is used. I can only assume that it, somehow, describes the sorting order for different languages. I have not been able to figure out how it is used.

The code in Listing 4.1 traverses the HFS b-tree to find an HFS filename. This code is from HLPDUMP2.C, which comes on the companion diskette. Since HLPDUMP2 isn't really concerned with speed or efficiency, actual traversal of the b-tree wasn't necessary. I felt it was important to show how it's actually done, however, so I added this function for that reason.

The first section, encompassed by the if (HFSHeader.TotalPages > 1), is where you search the index pages and follow the keys down to the leaf page. The way this works is simple. Read the first index page, or root page (provided in the b-tree header record described later). Search through the list of keys on this page to find out which page the next key is on. Continue this process until you're down to the leaf

Listing 4.1 B-tree traversal.

```
/***********************************************************************
 *
 * PROGRAM: HLPDUMP2.C
 *
 * PURPOSE: Traverses the HFS b-tree to find an HFS filename.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 4, Windows Help File Format, from Undocumented Windows
 * File Formats, published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 ***********************************************************************/

char FindFile(FILE *HelpFile, char* filename, long* offset)
{
 HFSFILEHEADER     HFSFileHeader;
 BTREEHEADER       HFSHeader;
 BTREEINDEXHEADER* HFSIndexHeader;
 BTREELEAFHEADER* HFSLeafHeader;
 long              HFSStart;
 int*              pNextPage;
 char*             buffer;
 char*             currPtr;
 int               nKeys, nFiles;
 char              found = 0;
 int               currLevel = 1;

 /* Go to the HFS and read the header. */
 fseek(HelpFile, HelpHeader.HFSLoc, SEEK_SET);
 LoadHeader(HelpFile);
 fread(&HFSHeader, sizeof(HFSHeader), 1, HelpFile);

 /* Allocate space for read buffer */
 buffer = malloc(HFSHeader.PageSize);

 HFSIndexHeader = (BTREEINDEXHEADER*) buffer;

 HFSStart = ftell(HelpFile);

 /* Advance to root page */
 fseek(HelpFile, (HFSHeader.RootPage * HFSHeader.PageSize) + HFSStart,
         SEEK_SET);

       /* If there's only one page, then it must be a leaf */
 if (HFSHeader.TotalPages > 1)
 {
    /* Traverse b-tree looking for the key for the leaf page */
    while (!found)
    {
      /* Read in the page */
      fread(buffer, HFSHeader.PageSize, 1, HelpFile);
      currPtr = buffer + sizeof(BTREEINDEXHEADER);
      pNextPage = (int *) currPtr;
      currPtr += sizeof(int);
```

pages, and do a simple sequential search to find the string or value you're searching for. (This code is found in the second half of the function, after the comment "Loop through all files on this page".) If it's not found on this leaf page, it's not in the tree. If it is found, then whatever data is associated with it will follow. For example, in the leaf pages of the HFS b-tree, each key is a string with an HFS filename. Each string is followed by an offset to where that file is located in the help file.

Although this code is specific to the HFS b-tree, similar code could be used to traverse the `KWBTREE`, `TTLBTREE`, and other b-tree structures in a `.HLP` file.

Listing 4.1 (continued)

```
      /* Go through all keys in the page */
      for (nKeys = 0; nKeys < HFSIndexHeader->NEntries; nKeys++)
      {
        /* If filename is less than key, this is our page. */
        if (strcmp(filename, currPtr) < 0)
        {
          break;
        }
        else
        {
          /* Advance to the next page# */
          while (*currPtr)
            currPtr++;
          currPtr++;
          pNextPage = (int *) currPtr;
          currPtr += sizeof(int);
        }
      }

      /* Advance to next page */
      fseek(HelpFile,
            (*pNextPage * HFSHeader.PageSize) + HFSStart, SEEK_SET);

      /* If this is the last index page */
      /* then pNextPage points to a     */
      /* leaf page                      */
      if (currLevel == HFSHeader.nLevels - 1)
      {
        found = 1;
      }
      currLevel++;
    }
  }

  fread(buffer, HFSHeader.PageSize, 1, HelpFile);
  HFSLeafHeader = (BTREELEAFHEADER*) buffer;
  currPtr = buffer + sizeof(BTREELEAFHEADER);
```

A Note on Object-Oriented Programming

The C language is used for code in this book. We feel C is still the best known language and is better as a demonstration tool, at least at this point. We both normally use C++ for our work and that's why I feel it's important to point out a few things for those who might want to implement code in C++ for dealing with help files. One place where C++, or any object-oriented language, would be really helpful is dealing with b-trees. The problem with coding for WinHelp b-trees in C is that there's no easy way to have a single piece of code deal with the different data types held in Winhelp b-trees. On the other hand, in C++, if you have a b-tree abstract base class, you could share much of the functionality of traversing, or even building, a b-tree in one base class. All functionality dependent on the individual files could be broken down into the individual derived classes for those files.

Listing 4.1 (continued)

```
  found = 0;

  /* Loop through all files in this page */
  for (nFiles = 0; nFiles < HFSLeafHeader->NEntries; nFiles++)
  {
     if (strcmp(filename, currPtr))
     {
       /* Advance to the file offset */
       while (*currPtr)
         currPtr++;

       /* Move past the null and file offset */
       /* to next file                        */
       currPtr += 5;
     }
     else
     {
       /* Save the offset of the file */
       while (*currPtr)
         currPtr++;
       currPtr++;

       *offset = (long) *((long*) currPtr);
       found = 1;
       break;
     }
  }

  /* TRUE if file was found, FALSE if it wasn't */
  return found;
}
```

Another place that C++ would come in very handy is supporting both the Windows 3.1 and Windows 95 version of the help file. If certain things are handled differently, you can easily override that specific behavior much easier in C++ than in C.

.HLP File Organization

Before we start getting into the nuts and bolts of `.HLP` files, we wanted to give a brief overview of the organization of help files and how they work. The center point of all help files is the `|TOPIC` file. This is where the text for the actual help topics is kept. When WinHelp reads a topic, it must then refer to at least one other file, and possibly others, to produce the text. When WinHelp reads data for a topic, it must first figure out which font is being used. A reference to the font number is in the `|TOPIC` file. From there WinHelp goes to the `|FONT` file to get the information on the font. If the text is compressed, WinHelp must go to the `|Phrases` file to extract phrases to insert into the topic text.

These are just some of the interdependencies of WinHelp's internal files. When developing software for WinHelp, you need to think about these things beforehand. If you're planning on extracting topic text, for example, it's a good idea to keep the entire `|Phrases` file in memory so you don't have to extract the text from the files every time you locate a phrase replacement. You'll also want to keep the `|FONT` file in memory, if you're using fonts, to keep disk activity to a minimum.

Other interdependencies include the `|KWBTREE`, `|KWDATA`, and `|KWMAP` files. All of these files work together to perform one function — keyword lookup.

Getting a handle on these interdependencies is crucial to understanding how WinHelp works as a whole. As I discuss the different files, I'll discuss how they interact with other files. In some places I'll point out how WinHelp performs tasks that aren't obvious from a simple look at the file formats. Armed with this knowledge, you should be able to do everything from writing your own WinHelp viewer to writing your own WinHelp compiler.

WinHelp Compression

The help compiler (`HC.EXE`) for Windows 3.0 provided a method of compression called "phrase replacement" compression, during which, while scanning through text from the help file, the compiler put together a list of phrases. As it encountered duplicates of these phrases, it built a table of the most common ones. In the last pass of the compile, it then removed these phrases from the actual text and inserted a reference number in its place. This reference number pointed to a phrase in the phrase table which then could be inserted whenever the topic text was displayed.

This compression was activated by adding the command COMPRESSION=TRUE in the [CONFIG] section of the Help Project File (.HPJ).

When Windows 3.1 came out, a new WinHelp and help compiler were released. One of the improvements was an additional level of compression. This was activated by either COMPRESSION = TRUE or COMPRESSION = HIGH (the new command COMPRESSION = MEDIUM replaced the old COMPRESSION = TRUE). This new level of compression added an LZ77 compression algorithm (called Zeck compression), which is identical to the compression used by COMPRESS.EXE (see Chapter 6), although the actual implementation of the algorithm is slightly different. This compression was implemented in two places — the |TOPIC file and the |Phrases file. The compression of the |Phrases file starts after the PHRASEHDR (discussed later) and encompasses the rest of the |Phrases file. For the |TOPIC file, the compression is done in increments of 2Kb blocks. This is necessary to allow one to get to topics without having to decompress every preceding topic. At the most, a preceding 2Kb would need to be decompressed to get to the beginning of a topic.

Because I've already discussed the LZ77 compression used by COMPRESS.EXE, I will simply mention the areas in which the compression is different. The changes are rather subtle and the code changes for the decompression are fairly moderate. Specifically, in the LZ77 implementation used by COMPRESS.EXE, compression codes and uncompressed data are thought of as "terms". For every 8 "terms", a flag BYTE precedes those terms. Each bit in the flag BYTE tells you if the corresponding "term" is a BYTE of uncompressed data or a 2-BYTE compression code. In COMPRESS.EXE, a set bit (or a 1) means that the term is uncompressed data, whereas a cleared bit (or a 0) indicates a compression code. In WinHelp, this same format is used; however, the meanings of set bits and cleared bits is reversed.

Table 4.4 SYSTEMHEADER record.

Field Name	*Data Type*	*Comment*
Magic	BYTE	0x6C
Version	BYTE	0x03
Revisionf	BYTE	0x0F, 0x15, 0x21
Always0	BYTE	Always 0
Always1	WORD	Always 0x0001
GenDate	DWORD	Time/date stamp help file created
Flags	WORD	See the discussion of Flags in the section "|SYSTEM"

With the release of WinHelp 4.0, a further level of compression was added — Hall Compression. Sadly, we have been unable to come up with the exact format of the Hall compression. It seems to operate completely differently from the |Phrases and LZ77 algorithms that we have managed to reverse engineer.

|SYSTEM

The |SYSTEM file is probably the single most important source of general information within a .HLP file. The |SYSTEM file contains a lot of information kept in the Help Project file and if you want to decompile a .HLP file, this is where you get the information for that file.

Following the HFSFGLEHEADER record (which is at the beginning of all HFS files, remember?) is the SYSTEMHEADER record (Table 4.4). The SYSTEMHEADER record contains the version of WinHelp needed to use the .HLP file (in a rather vague numbering scheme), the date the file was generated, and a Flags WORD.

If Flags = 0x04, then the help file implements Zeck compression, which leads to the question, how does one know which compression algorithm is used if it isn't Zeck? Fairly simply. If a |Phrases file exists, then Phrase compression is used. If |PhrImage and |PhrIndex files exist, then Hall compression is used. And finally, if this flag is set to 0x04, then Zeck compression is used.

Following the SYSTEMHEADER records is a list of SYSTEMREC records (Table4.5). These contain the juicy information. The SYSTEMREC structure is very simple.

The RecordType field identifies the type of information in the SYSTEMREC record. This can be a macro, copyright information, icon data, etc. The valid values are listed in the following code fragment:

Table 4.5 SYSTEMREC record.

Field Name	*Data Type*	*Comment*
RecordType	WORD	Type of data in record (see the discussion of RecordType in the section "|SYSTEM".)
DataSize	WORD	Size of Rdata
RData	void *	Record data

```
#define HPJ_TITLE                0x0001
#define HPJ_COPYRIGHT            0x0002
#define HPJ_CONTENTS             0x0003
#define HPJ_MACRO                0x0004
#define HPJ_ICON                 0x0005
#define HPJ_SECWINDOW            0x0006
#define HPJ_CITATION             0x0008
#define HPJ_CONTENTS_FILE        0x000A
```

The DataSize field is the number of bytes to read into RData.

RData is defined as a void * because it can contain anything from the text of a macro to the data of the icon associated with the help file.

HPJ_TITLE, HPJ_COPYRIGHT, HPJ_CONTENT, and HPJ_CITATION simply contain a string (not null-terminated) for the TITLE=, COPYRIGHT=, CONTENT=, and CITATION= lines of the .HPJ file. The only odd ball in the group is HPJ_COPYRIGHT, which seems to appear in all help file compiles, regardless of whether or not a COPYRIGHT= line is in the .HPJ. In the case where there is no COPYRIGHT= line, the DataSize field is 1 and RData is simply a single null byte (0x00).

HPJ_MACRO is also just text. It contains the text of each macro call listed in the .HPJ. You'll notice that the macros are kept in the same format as they are in the .HPJ, meaning that if you use "RR" instead of RegisterRoutine, the HPJ_MACRO record will contain RR. These are then read and parsed at run time by WinHelp.

Table 4.6 SECWINDOW record.

Field Name	*Data Type*	*Comment*
Flag	WORD	Valid fields
Type[10]	char	Type of secondary window
Name[9]	char	Name of secondary window
Caption[15]	char	Caption for secondary window
X	int	Starting x-coordinate
Y	int	Starting y-coordinate
Width	int	Width of secondary window
Height	int	Height of secondary window
Maximize	WORD	Maximize flag
SR_RGB	RGBQUAD	Scrolling region background color
NSR_RGB	RGBQUAD	Nonscrolling region background color

HPJ_ICON is the actual data of the icon for the help file (generated from an ICON= statement in the .HPJ). This format is exactly the same as the standard ICON format. You can find a description of this format in the SDK documentation.

HPJ_SECWINDOW is slightly more complicated than the others, because it contains a SECWINDOW structure (Table 4.6).

The Flag WORD contains a flag that basically describes which fields of the SECWINDOW record are valid. Because a secondary window definition in the .HPJ includes many optional fields, some of these may be invalid. The following values describe the valid fields:

```
#define WSYSFLAG_TYPE                          0x0001
#define WSYSFLAG_NAME                          0x0002
#define WSYSFLAG_CAPTION                       0x0004
#define WSYSFLAG_X                             0x0008
#define WSYSFLAG_Y                             0x0010
#define WSYSFLAG_WIDTH                         0x0020
#define WSYSFLAG_HEIGHT                        0x0040
#define WSYSFLAG_MAXIMIZE                      0x0080
#define WSYSFLAG_SRRGB                         0x0100
#define WSYSFLAG_NSRRGB                        0x0200
#define WSYSFLAG_ONTOP                         0x0400
```

The Type field contains the null-terminated word "Secondary". Presumably this was to allow for different classes of secondary windows; however, I have only seen this one used.

The Name field contains the name of the window as it is referred to in jumps. For example, mywindow>mytopic would show mytopic in the mywindow secondary window.

Caption contains the text of the window title bar.

X, Y, Width, and Height contain the location and dimensions of the window (very cryptic, isn't it?).

The Maximize field is either a 0 or 1. A 0x0000 indicates that the window is set to the dimensions specified in X, Y, Width, and Height (or whatever defaults WinHelp uses if these aren't specified). A 0x0001 tells WinHelp to maximize the secondary window and to disregard the dimensions for initially showing the window. (If the user hits the Restore button after WinHelp has displayed the window maximized, it will return to specified dimensions.)

SR_RGB and NSR_RGB contain the default RGB values for the background of the scrolling region and nonscrolling region, respectively.

|Phrases

The |Phrases file (remember, HFS filenames are case sensitive) is used as part of the compression in WinHelp. When you use the COMPRESSION=HIGH or COMPRESSION=MEDIUM statement in your .HPJ, the help compiler generates a |Phrases file. This file contains a list of the most common "phrases" in a help file.

A phrase is actually any series of characters. For example, "Their help file" could be a phrase, but so could ". Their he". Then, in the topic text, instead of actually storing the text, a pointer to the phrase is given allowing WinHelp to use 2 bytes instead of however many bytes it takes to store the phrase. This provides a significant space savings in larger help files. Because help files tend to be topic specific, many words and phrases tend to be reused. For example, in the help file for the solitaire game that comes with Windows, the word "card" is used repeatedly. If you were to replace every occurrence of "card" with a 2-byte code, you'd save 2 bytes for every occurrence. Usually, though, the phrases are longer than four letters, so the space savings can be tremendous.

There are two possible layouts of the |Phrases file, depending on the level of compression used. For COMPRESSION=MEDIUM, the PHRASEHDR record doesn't contain the PhrasesSize field. For COMPRESSION=HIGH, the phrases file is compressed with an LZ77 algorithm. The reason for this is that the |Phrases file can grow quite large with long phrases and a large number of phrases (as many as 1,024). The actual implementation of the LZ77 algorithm was discussed earlier in this chapter. The compression begins immediately following the PHRASEHDR record and continues to the end of the file.

The PhrasesSize field in the PHRASEHDR record contains the size of the |Phrases file (minus the size of the PHRASEHDR record) after it has been decompressed. The main purpose is that it tells you how much space you'll need to allocate to hold all the phrases after decompression.

Once loaded (or decompressed and loaded), the |Phrases file consists of two sections: Offsets and Phrases. The Offsets section is a list of offsets (of WORD data type) to the beginning of phrases in the Phrases section. For example, if there are 10 phrases, there will be 11 offsets, one for the beginning of each phrase and one for the end of the last phrase. The first phrase will begin immediately after the offsets, that is, 22 bytes after the first offset. To find the length of the first phrase, subtract the first offset (22) from the second offset, which points to the beginning of the second phrase.

|KWBTREE, |KWDATA, and |KWMAP

As mentioned earlier, in ".HLP File Organization", there are a lot of interdependencies in WinHelp. That is the case with these three files, and that is why they are grouped into one topic. |KWBTREE is a simple b-tree of keywords with a pointer to a list of topic offsets in |KWDATA. This is necessary because a single keyword can be associated with more than one topic. The |KWMAP file is used to provide quick access back into the |KWBTREE file based on a keyword number.

In WinHelp 4.0, two other files were introduced: |AWBTREE and |AWDATA. These files mimic the |KWBTREE and |KWDATA files, except they work with the new A-type keywords in WinHelp 4.0 instead of the regular keywords. This, however, is along the same lines of the Multikey option in WinHelp. WinHelp has always allowed for keywords based on different letters by adding the line MULTIKEY=x to the help project file, where x is a letter from A to Z. In each of these cases, a new set of keyword files are created. For example, with MULTIKEY=V, you will have the files |VWBTREE, and |VWDATA. Also notice that there is no equivalent of |KWMAP. The reason is that |KWMAP appears to be used specifically for the keyword search facility in WinHelp, which does not work for non-"K" keyword lists. This will become clearer later on as I discuss the |KWMAP file and how it helps the WinHelp search engine.

As I said, the |KWBTREE file is another simple b-tree. It contains a list of KWBTREEREC structures (Table 4.7) in the leaf pages. The Keyword, of course, is the key in the index pages. I have defined Keyword as char[80] for simplicity. It is a variable-length string that is null-terminated and should be read in that way, but it is limited to 80 characters. The Count field contains the number of occurrences of the keyword in the |TOPIC file and the number of topic offsets you'll find in the |KWDATA file. KWDataOffset, obviously, is the offset to the list of topic offsets in the |KWDATA file.

The |KWDATA file is very simple. It has a list of topic offsets (DWORDs), which are referenced from the |KWBTREE file. For example, if you traverse the |KWBTREE file for the keyword "Flower", you could find six occurrences. The KWDataOffset field tells you that the first offset is located in |KWBTREE at 24h. From there, you would go to byte 24h in |KWDATA and the next six long data types would be the topic offsets for the occurrences of the word "Flower". When WinHelp displays the keyword lists, it provides a list of topic titles associated with the keywords. How does it pull that off? Well, it's actually simple, but it's another example of the interdependencies of the internal files. WinHelp reads through the keyword topic offsets and then goes to the |TTLBTREE file (topic titles) and matches the keyword offsets with topic title offsets, giving it the information it needs to display the topics with the keywords.

Table 4.7 KWBTREEREC record.

Field Name	*Data Type*	*Comments*
Keyword	**char[80]**	**Keyword**
count	**int**	**No. of keyword occurrences**
KWDataOffset	**long**	**Offset into |KWDATA file**

The |KWMAP file is used as a shortcut method for avoiding traversal of the |KWBTREE file. WinHelp probably uses this file for the following situation: You select the Search button from WinHelp. WinHelp goes through the |KWBTREE file and reads the entire list of keywords. When you go through this list, you pick a keyword to retrieve a list of associated topics. At this point, instead of retraversing the b-tree to find the proper KWBTREEREC, WinHelp takes the index number of the keyword and then goes to the |KWMAP file. The |KWMAP file has a long data type that gives the number of KWMAPREC records (Table 4.8) in the file. This is immediately followed by a list of KWMAPREC structures. The first field, FirstRec, contains the index number of the first keyword on a given leaf page. This is followed by PageNum, which has the page number this keyword is located on. This allows WinHelp to find the proper b-tree leaf page, just by knowing the number of the keyword.

|TTLBTREE

|TTLBTREE contains a list of the titles for all the topics in a .HLP file along with an offset to the topics the titles are associated with. As the name implies, this list is kept in the form of a b-tree. As with the |KWBTREE file, this b-tree uses a 2Kb page size and uses the same BTREENODEHEADER and BTREEINDEXHEADER records. The key used in the index pages is the topic title itself.

The data on the leaf pages consists of a topic offset, followed by a null-terminated string containing the topic title. If you look through a |TTLBTREE, you'll notice a lot of offsets without any actual titles. The reason for this is that not all topics necessarily have a title. When this is the case, there will be no title in |TTLBTREE, but the offset to the topic will appear.

|FONT

The |FONT file is where .HLP files keep all of their information about fonts used in the topic text (big surprise!). .HLP files actually maintain very specific font information, not just the name and point size. The reason is that when WinHelp encounters a font that isn't available on the system, with very specific information, it can find a much closer match than it could by name alone. This is important in maintaining the consistency of viewed text.

Table 4.8 KWMAPREC record.

Field Name	Data Type	Comments
FirstRec	long	Index number of first keyword
PageNum	int	Page number of leaf with keyword

.HLP files keep two lists of fonts. One is a simple list of font names followed by the font descriptor table. The font descriptor table provides information about point size (in half point increments), color in the scrolling region, color in the nonscrolling region, font family, and attributes (such as bold, italics, etc.).

The layout of the |FONT file is quite simple. It begins with the FONTHEADER (Table 4.9).

The DescriptorsOffset has the distance to the beginning of the descriptor table relative to the end of the FONTHEADER record. In between the FONTHEADER and the descriptor list is the list of font names. This list is simply fixed-length, null-terminated font names. For WinHelp 3.1, font names are 20 characters, and in WinHelp 4.0, font names are 32 characters (a quick check of the system record will, of course, tell you which version you're working with). Because the font names are null-terminated, one character must be the NULL, allowing 19 or 31 characters per font name.

The font list is immediately followed by an array of FONTDESCRIPTOR records (Table 4.10), which should not be confused with LOGFONT structures, which are completely different (so there shouldn't be any confusion anyway, just making sure). The Attributes field is the bitwise ORed sum of font attributes such as bold, italic, etc. The HalfPoints field is the size of the text in half points. The FontFamily field tells WinHelp the general variety of font. This is useful in determining close matches if the existing font is not available. FontName is an index into the font list that preceded the font descriptors. This is followed by two RGBQUADs, one for the color of the font when it is in the scrolling region and one for when the font is displayed in the nonscrolling region. The idea behind this is to prevent repeating the same font descriptor simply because the font is in the nonscrolling region.

Table 4.9 FONTHEADER record.

Field Name	*Data Type*	*Comments*
NumFonts	WORD	Number of fonts in font list
NumDescriptors	WORD	Number of font descriptors
DefDescriptor	WORD	Default font descriptor
DescriptorsOffset	WORD	Offset to descriptor list

```
/* Font Attribute Values */
#define FONT_NORM       0x00      /* Normal          */
#define FONT_BOLD       0x01      /* Bold            */
#define FONT_ITAL       0x02      /* Italics         */
#define FONT_UNDR       0x04      /* Underline       */
#define FONT_STRK       0x08      /* Strike Through  */
#define FONT_DBUN       0x10      /* Dbl Underline   */
#define FONT_SMCP       0x20      /* Small Caps      */

/*Font Family Values */
#define FAM_MODERN    0x01
#define FAM_ROMAN       0x02
#define FAM_SWISS       0x03
#define FAM_TECH        0x03
#define FAM_NIL         0x03
#define FAM_SCRIPT      0x04
#define FAM_DECOR       0x05
```

Font descriptors are created for every variation of a font in the help file. For example, if you have 16-point Helvetica Bold in a title and follow that with 12-point Helvetica in the text, you'd have two font descriptors. If you bold a word in the 12-point Helvetica text, that would create a third font descriptor. If you italicize a word, you would then have a fourth font descriptor, and so on. As you can see, it's easy to accumulate a lot of font descriptors.

Table 4.10 FONTDESCRIPTOR record.

Field Name	*Data Type*	*Comments*
Attributes	BYTE	Font attributes (See the discussion of font descriptors in the section "|FONT".)
HalfPoints	BYTE	Point size x 2
FontFamily	BYTE	Font family (see the discussion of font families in the section "|Font".)
FontName	BYTE	Font name (refers to font list no.)
Unknown	BYTE	Unknown but always seems to be 0
SRRGB	RGBTRIPLE	Scrolling region color
NSRRGB	RGBTRIPLE	Nonscrolling region color

|CTXOMAP

|CTXOMAP (Table 4.11) is the simplest of the WinHelp internal files. In the Help Project (.HPJ) file, you can create a list, under the [MAP] section, of context strings (created in the help file with a context string footnote) and assign unique identification numbers to these topics. These IDs, in turn, can be used from the WinHelp() API call to display a topic.

The first WORD of the |CTXOMAP file is a count of the number of CTXOMAPREC records to follow. The CTXOMAPREC has two fields. The MapID field is the map ID assigned in the .HPJ. The second field is the offset of the topic.

|CONTEXT

Like many of the other files, |CONTEXT is a b-tree structure. The leaf nodes consist of a list of hash values and topic offsets. The key in the index nodes is the hash value. The hash values are generated from a list of keywords and context strings. The purpose appears to be to allow a user to type a keyword or context string and to quickly locate that with a minimum amount of space. In other words, if the key was the actual keyword or context string instead of a hash table, the space required for the text would take up too much space. The hash values are calculated using a hashing algorithm (Listing 4.2) that Ron Burk was able to reverse-engineer. Following is a sample program that calculates a hash value, given a string. As with all hash functions, there is no way to determine the string, given a hash value.

The actual data in the leaf pages of the |CONTEXT file is simply one hash value followed by a topic offset. The hash value is a DWORD. Although it's not really important, it is known that the hash values are treated as signed long integers in WinHelp, because they are sorted in that fashion.

|TOPIC

The |TOPIC file format is, by far, the most complex of all the HFS files. The |TOPIC file, as its name would imply, contains all of the information for individual topics. It contains paragraph formatting options, paragraph text, pointers to phrases in |Phrases (if it exists), and so on.

Table 4.11 CTXOMAP record.

Field Name	*Data Type*	*Comments*
MapID	long	Map ID from HPJ
TopicOffset	long	Offset to topic in |TOPIC file

To add one small layer of additional complexity, when the help file is compiled with COMPRESS=HIGH, the |TOPIC file is compressed with the LZ77 compression.

The main complexity of the |TOPIC file involves two things: topic offsets and multiple layers. I'll discuss the topic offsets later. The multiple layers can get a little

Listing 4.2 WinHelp hashing function.

```
/*********************************************************************
 *
 * PROGRAM: MakeHash.C
 *
 * PURPOSE: Calculates and outputs the hash value of a string. These hash
 * values are used in the |CONTEXT file of a WinHelp .HLP file.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 4, Windows Help File Format, from Undocumented Windows
 * File Formats, published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 *********************************************************************/

#include <stdio.h>

char MapTable[256];

/* Function prototypes */
void BuildMap(void);
long Hash (char *);

/*******************************************
 Builds character set map for hash function.
*******************************************/
void BuildMap() {
   char c;
   int counter;

   /* Map A-Z and a-z as 0-25. */
   for (counter = 'A', c = 17; counter <= 'Z'; counter++, c++)
      MapTable[counter] = MapTable[counter + 32] = c;

   for (counter = '1', c = 1; counter <= '9'; counter++, c++)
      MapTable[counter] = c;

   MapTable['0'] = 0x0A;
   MapTable['.'] = 0x0C;
   MapTable['_'] = 0x0D;
}
```

confusing. The |TOPIC file has two layers, really: the paragraph layer and the topic layer. The topic layer is embedded within the paragraph layer. So when you first start traversing the |TOPIC file, you do it one paragraph at a time. The paragraphs are connected via a doubly linked list. There are three different types of paragraph records: topic headers, paragraph data, and table data. The topic headers, in turn, create the topic layer. These topic headers create another doubly linked list of topic records. I'll discuss all of these later in greater depth.

Topic Offsets

Most of us are familiar with offsets. Offsets are used in many aspects of programming. Because of the complex nature of WinHelp, direct offsets to a location in the |TOPIC file are inadequate for many jobs. For example, when finding the exact location of a keyword, you can't simply say that it's 85 bytes into the |TOPIC file. Why not? One reason is compression. if you're looking for the keyword "Carthage", what do you do if it is part of a phrase being used in phrase replacement? You could replace all the phrases and then use a direct offset, right? That works fine if your help file is tiny, but what if you have a 2Mb |TOPIC file? To use a direct offset to a word near the end of the file, you'd have to replace all the phrases of the previous information in the |TOPIC file just to get the correct offset.

Listing 4.2 (continued)

```
/********************************************
   Hash function by Ron Burk
*********************************************/
long Hash (char *p) {
   long h = 0;
   while(*p) {
      char c = MapTable[*p++];
      h = h * 0x2B + c;
   }
   return h;
}

void main(int argc, char *argv[]) {
 long HashVal;
 BuildMap();
 HashVal = Hash(argv[1]);
 printf(" Hash value = %ld\n", HashVal);
}
```

To avoid this problem, offsets in WinHelp are broken into two pieces: a block number and a block offset. The entire |TOPIC file is then broken into 4Kb blocks. Unfortunately, it's a little more complex than this. Two types of these broken offsets are used: one called Extended Offsets and the other Character Offsets.

Extended Offset

```
3                          1                        
1                          3                        0
+-------------------------------------------------+
|       Block Number       |     Block Offset      |
+-------------------------------------------------+
```

Extended offsets are simply a block number and a block offset. Each offset is a DWORD with the upper 18 bits used as the block number and the lower 14 bits as the offset within that block. Extended offsets are used only within the |TOPIC file in TOPICBLOCKHEADER and TOPICHEADER (discussed later) records.

Character Offset

```
3                          1                     
1                          4                     0
+----------------------------------------------+
|         Block Number     |    Block Offset    |
+----------------------------------------------+
```

Character offsets are, obviously, very similar to extended offsets. The only difference is an additional bit for calculating the block offset and one less bit for calculating the block number. Why the difference? Beats me. I've always thought that either would do, and I still think that's the case, but who knows what's going through the heads at Microsoft.

Character offsets are used as references to the |TOPIC file by files external to |TOPIC. Offsets in |TTLBTREE, |KWDATA, |CONTEXT, and so on use character offsets.

If you're only using 4Kb blocks, you might wonder why the block offsets for extended offsets (see the following paragraph) are 14 bits (0 to 16Kb) and character offsets are 15 bits (0 to 32Kb). This allows for the compression of the 4Kb blocks. If a 4Kb block is decompressed to 5 or 6Kb, then you need more than 12 bits (0 to 4Kb) to find the offset within the block.

On top of the block number/block offset split of character offsets, there is one additional difference between character offsets and extended offsets. Where extended offsets always point to a direct offset within a block, character offsets point to a specific character within the text. What you essentially have to do, once you've loaded the 4Kb block and decompressed it, if necessary, is to go through and count all of the characters of displayable text to get the character offset. For example, a character offset with a block number of 1 and a block offset of 124 would point to the first block and the 124th displayable character within the block. Fortunately, to make this easier, each paragraph has a character count. So if the first paragraph has 110 characters, and the second paragraph has 165 characters, your character offset would point to the 14th character in the second paragraph.

On the other hand, if you had an extended offset with a block number of 1 and a block offset of 124, you would go directly to the 124th byte in the first 4Kb block (after decompression and/or phrase replacement, if necessary).

Later in this chapter I'll show examples of how to find a topic using extended offsets and character offsets.

Compression in |TOPIC

Compression in the |TOPIC file is handled in two ways. The first is that common phrases are removed and placed in the |Phrases file. In place of the actual phrases, references are placed in the topic text to point to these phrases. After this, the actual topic text is then compressed with the LZ77 (Zeck) algorithm. This happens in 4Kb blocks as described in the following text.

TOPICBLOCKHEADER

As I said earlier, the |TOPIC file is broken into 4Kb blocks. Each of these blocks has a TOPICBLOCKHEADER record (Table 4.12).

The LastParagraph field is an extended offset to the last PARAGRAPH record (see the section "TOPICHEADER") of the previous 4Kb topic block. Topic data is an extended offset to the first paragraph record of this block. LastTopicHeader is an extended offset to the beginning of the last TOPICHEADER record (see below) in this block.

Table 4.12* TOPICBLOCKHEADER *record.

Field Name	*Data Type*	*Comments*
LastParagraph	**long**	**Last paragraph in this block**
TopicData	**long**	**First paragraph in this block**
LastTopicHeader	**long**	**Last topic in this block**

PARAGRAPH

The name PARAGRAPH is a bit misleading for this structure. It doesn't necessarily mean an actual paragraph, though, in most cases, it is, and it seems like the most logical name for the structure for that reason. PARAGRAPH records (Table 4.13) are where the actual "meat" of the |TOPIC file is stored. PARAGRAPH records contain the text of the topic, hotspot markers, font markers, etc.

The key piece of information is the RecordType field. There are three different record types: topic headers (0x02), text records (0x20), and table records (0x23). Then the actual data is kept in the LinkData1 and LinkData2 fields. For topic headers, a TOPICHEADER record (see later) is stored in the LinkData2 field. For text records, the LinkData1 field contains formatting information (fonts, hotspot markers, etc.) for the paragraph, and the LinkData2 field contains the text, phrase pointers, and so on, for the paragraph.

TOPICHEADER (0x02 Paragraph Records)

TOPICHEADER records (Table 4.14) are kept in the LinkData2 field of PARAGRAPH records with a RecordType of 0x02. TopicHeader records precede every topic.

Text Records (0x20 Paragraph Records)

PARAGRAPH records of type 0x20 contain actual text and other displayable information for the topic. PARAGRAPH records are one of the more complex aspects of WinHelp, because there are quite a few paragraph features, and if you plan on displaying help text, you have to make use of all the formatting information.

Table 4.13 PARAGRAPH record.

Field Name	*Data Type*	*Comments*
BlockSize	long	Size of this record + link data 1 & 2
DataLen2	long	Length of LinkData2
PrevPara	long	Offset of previous paragraph
NextPara	long	Offset of next paragraph
DataLen1	long	Length of LinkData1 + 11 bytes
RecordType	BYTE	Type of paragraph record
LinkData1	char*	Data set one for this paragraph
LinkData2	char*	Data set two for this paragraph

Two types of paragraph breaks are used by the help compiler: \par and \pard. \pard creates a new PARAGRAPH record, whereas \par breaks are considered part of a single paragraph. This is important for several reasons, \pard is meant to create a new paragraph and start new defaults for the paragraph. This allows you to break a paragraph with \par and keep all the previous formatting information. To manage this, it's all kept as part of the same PARAGRAPH record within WinHelp.

Understand that, in terms of the PARAGRAPH record, I don't necessarily mean a physical paragraph of text, but text with like formatting information. Every time the formatting information changes, a new PARAGRAPH record is created.

Type 0x20 PARAGRAPH records have two data links, DataLink1 and DataLink2. DataLink1 primarily contains paragraph formatting information and begins with a FORMATHEADER record (Table 4.15). The FormatSize and DataSize fields are doubled values. Simply read a byte. If the lsb in these fields is set, then a second byte is read as the high byte to create a WORD value. Then this total is divided by two. This is like many fields in the .SHG/.MRB file formats. As with those, you can use the ReadWordVal() functions provided in Chapter 2 to read these fields.

Table 4.14 TOPICHEADER record.

Field Name	***Data Type***	***Comments***
BlockSize	**long**	**Size of block**
BrowseBck	**TOPICOFFSET**	**Previous topic in browse sequence**
BrowseFor	**TOPICOFFSET**	**Next topic in browse sequence**
TopicNum	**DWORD**	**Topic number**
NonScroll	**TOPICOFFSET**	**Start of nonscrolling region**
Scroll	**TOPICOFFSET**	**Start of scrolling region**
NextTopic	**TOPICOFFSET**	**Start of next topic**

Table 4.15 FORMATHEADER record.

Field Name	*Data Type*	*Comment*
FormatSize	BYTE or WORD	2x the no. of bytes of formatting information
Flags	BYTE	Flag byte (unknown values)
DataSize	BYTE or WORD	2x the no. of bytes of text

Another slightly stupid thing about the FORMATHEADER is that this information is duplicated in the PARAGRAPH record's DataLen1 and DataLen2 fields. The only real difference is that the size given in FormatSize doesn't include the FORMATHEADER record, whereas the DataLen1 value does.

Following the FORMATHEADER record is a single NULL byte.

The next section is a list of paragraph attribute strings. The list is terminated by a NULL (0x00) byte. It starts with a DWORD paragraph set-up attribute. The following values have the following meanings:

0x00020000	Space before
0x00040000	Space after
0x00080000	Line spacing before
0x00100000	Left margin indent
0x00200000	Right margin indent
0x00400000	First line indent
0x01000000	Paragraph border
0x02000000	Tab setting information
0x04000000	Right justify
0x08000000	Center justify
0x10000000	Don't wrap lines in paragraph

This value may be followed by a 3-byte paragraph border setting, if there is a border. The first byte is 0x01, indicating a border setting. This is followed by the border description byte. The third byte is always 0x51. The following values are valid border descriptions. They can be ORed together.

0x80	Dotted border
0x40	Double border
0x20	Thick border
0x10	Right border
0x08	Bottom border
0x04	Left border
0x02	Top border
0x01	Boxed border

After the paragraph set-up attribute and, if it exists, the border setting, a NULL byte indicates the end of the paragraph set-up.

This is followed by a string of bytes that consist of format codes and parameters for the format codes. The LinkData2 field contains the text for the paragraph. Within the text will be NULL (0x00) bytes. For each null byte there is a formatting code in the LinkData1 field. So obviously, these fields need to be handled together. Each time you run into a NULL byte in the LinkData2 field, you need to pull in the appropriate formatting code from the LinkData1 field.

Figure 4.2 shows a dump of a PARAGRAPH record. The area labeled PARAGRAPH Rec is actually all of the fields except LinkData1 and LinkData2, whereas FORMATHEADER Rec is part of the LinkData1 field.

Formatting codes are variable length. Basically, there's a 1-byte code, and depending on what that code is, a variable number of parameters follow it. For example, the format code 0x80 specifies a font change. It is followed by one word that tells you the font descriptor for the font. The following is a list of codes and their meanings. Below each code is a list of parameters for that code.

0x80: Font change. Specifies that a new font starts here.
WORD. Font descriptor for font to insert

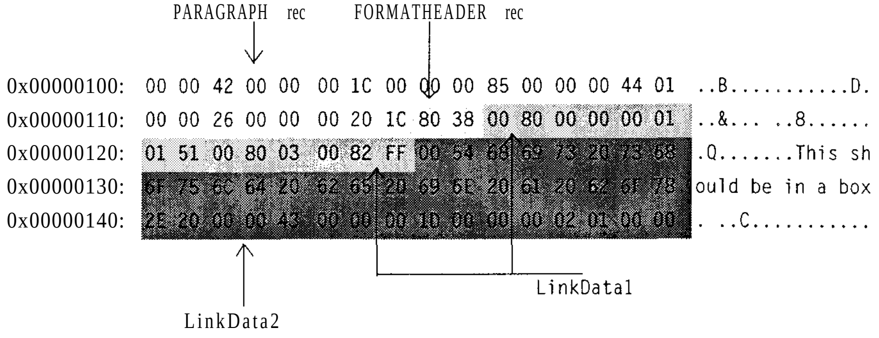
0x81: Newline. Caused by the \line RTF command.
No parameters

0x82: New paragraph. Caused by the \par RTF command, but not the \pard command, which starts a new PARAGRAPH record.
No parameters

0x83: Tab. Caused by the \tab RTF command.
No parameters

0x86: Bitmap current. Caused by the \bmc RTF command. For \bmc, \bml, and \bmr, there is a second case of each. If you use \bmcwd, \bmlwd, or \bmrwd, the actual bitmap data follows instead of a reference to the |bmxxx file. So in each case where the second parameter is 0x92 instead of 0x08, the \bmxwd version of the function is used, and the final word, the bitmap number, is not provided. Instead, the actual bitmap or metafile is included. See the following section, "Bitmaps and Metafiles", for a description of the format.

Figure 4.2 PARAGRAPH Record Dump



```
0x00000100:  00 00 42 00 00  00 1C 00 00 00 85 00 00 00 44 01  ..B...........D.
0x00000110:  00 00 26 00 00  00 20 1C 80 38 00 80 00 00 00 01  ..&... ..8......
0x00000120:  01 51 00 80 03  00 82 FF 00 54 68 69 73 20 73 68  .Q.......This sh
0x00000130:  6F 75 6C 64 20  62 65 20 69 6E 20 61 20 62 6F 78  ould be in a box
0x00000140:  2E 20 00 00 43  00 00 00 10 00 00 00 02 01 00 00  . ..C...........
```

BYTE. 0x22
BYTE. 0x08 or 0x92 (\bmcwd)
BYTE. 0x80 or 0x83 (\bmcwd)
BYTE. 0x02
WORD. 0x0000 or 0x0001 (\bmcwd)
WORD. Bitmap number (HSF file = |bmxxx, \bmc only)

0x87: Bitmap left. Caused by the \bml RTF command.
BYTE. 0x22
BYTE. 0x08 or 0x92 (\bmlwd)
BYTE. 0x80 or 0x83 (\bmlwd)
BYTE. 0x02
WORD. 0x0000 or 0x0001 (\bmlwd)
WORD. Bitmap number (HSF file = |bmxxx, \bml only)

0x88: Bitmap right. Caused by the \bmr RTF command.
BYTE. 0x22
BYTE. 0x08 or 0x92 (\bmrwd)
BYTE. 0x80 or 0x83 (\bmrwd)
BYTE. 0x02
WORD. 0x0000 or 0x0001 (\bmrwd)
WORD. Bitmap number (HSF file = |bmxxx, \bmr only)

0x89: End of hotspot. Follows a hotspot code.

0xE2: Popup hotspot. Hotspot caused by \ul RTF command
DWORD. Context hash value for topic title (use |CONTEXT to find topic)

0xE3: Jump hotspot. Hotspot caused by \uldb RTF command
DWORD. Context hash value for topic title

0xFF: End attributes. Always the last byte of LinkData1.
No parameters

Bitmaps and Metafiles

When you insert bitmaps and metafiles by reference (meaning you use the RTF bml, bmc, or bmr command to insert the bitmap), the bitmap or metafile is stored within the HFS under the name |bmX (where X is a sequential number starting at 0 or 1). Embedded bitmaps and metafiles are also stored in separate HFS files and treated as if they were inserted by reference with a \bmc command. In addition to this change, the actual format of the bitmap or metafile is changed. In fact, it's changed to a .SHG/.MRB file. (See chapters 2 and 3 for information about the .MRB and .SHG file formats.)

Essentially, it gets converted to a .MRB file with only one image. The help compiler does a few things that MRBC and SHED never do, though.

When the help compiler converts the bitmap, it tests two different compression methods on the data: RLE and LZ77 (the WinHelp version). Whichever is more efficient, no compression, RLE, or LZ77, is the format the image uses. SHED and MRBC never use LZ77 compression, only RLE. In addition, if the help compiler uses LZ77 compression, it changes the first 2 bytes of the image to "lP" instead of the standard "lp" used by SHED and MRBC.

The images are given names like |bm0, |bm1, and so on, which causes one of the biggest annoyances for reversing a help file back into RTF source: the original bitmap filenames are lost.

Conclusion

That pretty much wraps up the WinHelp file format. To be sure, this is obviously not the complete format. There are unknown fields and entirely unknown files, such as the |VIOLA file. Most embarrassing is the .GID file, which contains the HFS file called |Pete. This was clearly left for me as a joke, and unfortunately the joke is on me, because I have yet to figure out the contents of the file. I believe that, to some degree, a bit of a challenge was placed specifically for me by giving these files names that have no real meanings. (I know the head of WinHelp development. It's like a game to him.)

Speaking of the .GID file, you may have noticed that it has gone totally unaddressed. Some of you may not even know what the .GID file is. The .GID file is a hidden file that WinHelp creates every time you open a new .HLP file. It is created in the same directory as the help file. Most of the information in this file is a binary version of your .CNT file. Because there is already a text version of the .CNT file (the .CNT file itself, obviously), it has never seemed particularly important to get into the nuts and bolts of the .GID file. Among other things, it includes additional information on the last position of the .HLP file on the screen.

In addition, you'll notice a |KWBTREE and a |KWMAP file in the .GID file. Whereas the |KWMAP file is the same format as |KWMAP in a help file, the format for the |KWBTREE file is different in a .GID file than in a help file. In the .CNT file, you can associate multiple help files with a single help file. This |KWBTREE file provides a place where |KWBTREE files can be merged together from all the .HLP files. The difference between the |KWBTREE file of a regular help file and that of the .GID file is in the additional information that tells which help file the keywords reference.

Where Do I Go from Here?

There's really so much you can do with this information, it's hard to know where to begin. I've seen .HLP to .DOC converters, .HLP to .RTF converters, and a group over at Sun Microsystems even wrote a program to read .HLP files under UNIX. I believe the most useful tool someone could create with this information now, however, would be a .HLP to HTML program. This might not be too difficult if you have the source for your .HLP file, but what if you want to take a .HLP file that you didn't create, and create an HTML document from that?

Listing 4.3 WINHELP.H.

```
/*********************************************************************
 *
 * PROGRAM: WINHELP.H
 *
 * PURPOSE: Sample header file for an .HLP file.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 4, Windows Help File Format, from Undocumented Windows
 * File Formats, published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 *********************************************************************/

/* Force byte aligned packing of data structures */
#pragma pack(1)

/*
   The following are defined as based on the Windows 3.1
   Programmer's Reference.
*/
#ifndef _INC_WINDOWS
typedef unsigned char   BYTE;
typedef unsigned short  WORD;
typedef unsigned long   DWORD;

typedef struct tagRGBTRIPLE
{
    BYTE    rgbBlue;
    BYTE    rgbGreen;
    BYTE    rgbRed;
} RGBTRIPLE;

typedef DWORD COLORREF;

#define RGB(r,g,b) ((COLORREF) \

(((BYTE)(r)|((WORD)(g)<<8))|(((DWORD)(BYTE)(b))<<16)))

#endif

/* Define the size of a Topic Block 4k - sizeof(TOPICLINK) */
#define TopicBlockSize   4096L
```

Listing 4.3 (continued)

```
/*************************************************************
   WinHelp Common Structures
*************************************************************/

/* Help file Header record */
typedef struct tagHELPHEADER {
    DWORD   MagicNumber;      /* 0x00035F3F                 */
    long    HFSLoc;           /* Pointer to WHIFS header    */
    long    Negative1;
    long    FileSize;         /* Size of entire .HLP File   */
} HELPHEADER;

#define HF_MAGIC  0x00035F3F

/* File Header for WHIFS files */
typedef struct tagHFSFILEHEADER {
    long    FilePlusHeader;  /* File size including this header */
    long    FileSize;        /* File size not including header  */
    char    FileType;        /*                                 */
} HFSFILEHEADER;

/* File types used by HFS                         */
/* FT_NORMAL is a regular file                    */
/* FT_HFS is the HFS directory file               */
/* FT_UNK Found in MSDNCD4.MVB |TTLBTREE file     */

#define FT_NORMAL 0x00
#define FT_HFS    0x04


/*************************************************************
   B-Tree related structures
*************************************************************/

/* Keyword & TTL BTREE Headers - Slightly different than HFS B-tree Header.
   Both HFS and Keyword B-Trees use same leaf and index node headers.    */

typedef struct tagBTREEHEADER {
  WORD  Signature;         /* 0x293B                                   */
  char  Unknown1;          /* 0x02 always                              */
  char  FileTypeFlag;      /* Same as FILEHEADER FileTypeFlag field */
  short PageSize;          /* Size of tree pages                       */
  char  SortOrder[16];     /* Used for internationalization            */
  short FirstLeaf;         /* Probably First Leaf page!!!              */
  short NSplits;           /* # of page splits Btree has suffered      */
  short RootPage;          /* page #of root page                       */
  short Reserved2;
  short TotalPages;        /* total # of 2Kb pages in Btree */
  short nLevels;           /* # of levels in this Btree     */
  DWORD TotalBtreeEntries;
} BTREEHEADER;
```

Listing 4.3 (continued)

```
/* Modified B-Tree Leaf Header */
typedef struct tagBTREELEAFHEADER {
    WORD    Signature;      /* Signature word           */
    short   NEntries;       /* Number of entries        */
    short   PreviousPage;   /* Index of Previous Page   */
    short   NextPage;       /* Index of Next Page       */
} BTREELEAFHEADER;

/* Modified B-Tree Index node header */
typedef struct tagBTREEINDEXHEADER {
    WORD    Signature;      /* Signature byte           */
    short   NEntries;       /* Number of entries in node */
} BTREEINDEXHEADER;

/************************************************************
   |Phrases header
************************************************************/

/* Phrases header. In uncompressed, last field doesn't exist */
typedef struct tagPHRASEHEADER    {
    short   NumPhrases;     /* Number of phrases in table                      */
    WORD    OneHundred;     /* 0x0100                                          */
    long    PhrasesSize;    /* Amount of space uncompressed phrases requires. */
} PHRASEHEADER;


/************************************************************
   |FONT File structures
************************************************************/

/* Header for |FONT file */
typedef struct tagFONTHEADER {
    WORD    NumFonts;           /* Number of fonts in Font List */
    WORD    NumDescriptors;     /* Number of font descriptors   */
    WORD    DefDescriptor;      /* Default font descriptor      */
    WORD    DescriptorsOffset;  /* Offset to descriptor list    */
} FONTHEADER;

typedef struct tagFONTDESCRIPTOR {
    BYTE    Attributes;         /* Font Attributes See values below   */
    BYTE    HalfPoints;         /* PointSize * 2                      */
    BYTE    FontFamily;         /* Font Family. See values below      */
    BYTE    FontName;           /* Number of font in Font List        */
    BYTE    Unknown;            /* Unknown                            */
    RGBTRIPLE SRRGB;            /* RGB values of foreground           */
    RGBTRIPLE NSRRGB;           /* background RGB Values (?? Not sure */
} FONTDESCRIPTOR;
```

Listing 4.3 (continued)

```
/* Font Attributes */
#define FONT_NORM    0x00        /* Normal         */
#define FONT_BOLD    0x01        /* Bold           */
#define FONT_ITAL    0x02        /* Italics        */
#define FONT_UNDR    0x04        /* Underline      */
#define FONT_STRK    0x08        /* Strike Through */
#define FONT_DBUN    0x10        /* Dbl Underline  */
#define FONT_SMCP    0x20        /* Small Caps     */

/* Font Families */
#define FAM_MODERN   0x01
#define FAM_ROMAN    0x02
#define FAM_SWISS    0x03
#define FAM_TECH     0x03
#define FAM_NIL      0x03
#define FAM_SCRIPT   0x04
#define FAM_DECOR    0x05

/************************************************************
   |SYSTEM file structures
************************************************************/

/* Header for |SYSTEM file */
typedef struct tagSYSTEMHEADER {
    BYTE    Magic;      /* 0x6C                 */
    BYTE    Version;    /* Version #            */
    BYTE    Revision;   /* Revision code        */
    BYTE    Always0;    /* Unknown              */
    WORD    Always1;    /* Always 0x0001        */
    DWORD   GenDate;    /* Date/Time that the help file was generated    */
    WORD    Flags;      /* Values seen: 0x0000 0x0004, 0x0008, 0x000A    */
} SYSTEMHEADER;

/* Magic number of SYSTEM record */
#define SYS_MAGIC     0x6C

/* Flags for |SYSTEM header Flags field below */
#define NO_COMPRESSION          0x0000
#define COMPRESSION_HIGH        0x0004

/* Help Compiler 3.1 System record. Multiple records possible */
typedef struct tagSYSTEMREC {
    WORD    RecordType;    /* Type of Data in record      */
    WORD    DataSize;      /* Size of RData               */
    char    *RData;        /* Raw data (Icon, title, etc) */
} SYSTEMREC;
```

Listing 4.3 (continued)

```
/* Types for SYSTEMREC RecordType below */
#define HPJ_TITLE        0x0001      /* Title from .HPJ file          */
#define HPJ_COPYRIGHT    0x0002      /* Copyright notice from .HPJ file */
#define HPJ_CONTENTS     0x0003      /* Contents=??? from .HPJ        */
#define MACRO_DATA       0x0004      /* SData = 4 nulls if no macros  */
#define ICON_DATA        0x0005
#define HPJ_SECWINDOWS   0x0006      /* Secondary window info in .HPJ  */
#define HPJ_CITATION     0x0008      /* CITATION= under [OPTIONS]      */


/* Secondary Window Record following type 0x0006 System Record */

typedef struct tagSECWINDOW {
    WORD    Flags;           /* Flags (See Below)       */
    BYTE    Type[10];        /* Type of window          */
    BYTE    Name[9];         /* Window name             */
    BYTE    Caption[51];     /* Caption for window      */
    WORD    X;               /* X coordinate to start at */
    WORD    Y;               /* Y coordinate to start at */
    WORD    Width;           /* Width to create for     */
    WORD    Height;          /* Height to create for    */
    WORD    Maximize;        /* Maximize flag           */
    BYTE    Rgb[3];
    BYTE    Unknown1;
    BYTE    RgbNsr[3];       /* RGB for non scrollable region */
    BYTE    Unknown2;
} SECWINDOW;

/** Values for Flags **/

#define WSYSFLAG_TYPE       0x0001  /* Type is valid          */
#define WSYSFLAG_NAME       0x0002  /* Name is valid          */
#define WSYSFLAG_CAPTION    0x0004  /* Ccaption is valid      */
#define WSYSFLAG_X          0x0008  /* X    is valid          */
#define WSYSFLAG_Y          0x0010  /* Y    is valid          */
#define WSYSFLAG_WIDTH      0x0020  /* Width    is valid      */
#define WSYSFLAG_HEIGHT     0x0040  /* Height   is valid      */
#define WSYSFLAG_MAXIMIZE   0x0080  /* Maximize is valid      */
#define WSYSFLAG_RGB        0x0100  /* Rgb  is valid          */
#define WSYSFLAG_RGBNSR     0x0200  /* RgbNsr   is valid      */
#define WSYSFLAG_TOP        0x0400  /* On top was set in HPJ file */
```

Listing 4.3 (continued)

```
/**********************************************************
   Keyword file structures
**********************************************************/

/* Keyword Map Record */
typedef struct tagKWMAPREC {
    long    FirstRec;       /* Index number of first keyword on leaf page    */
    WORD    PageNum;        /* Page number that keywords are associated with */
} KWMAPREC;

/* Record for the |KWBTREE file */
typedef struct tagKWBTREEREC {
    char     Keyword[80];     /* Variable Length Keyword      */
    short    Count;           /* Count of Keywords occurances */
    long     KWDataOffset;    /* Offset into |KWDATA file     */
} KWBTREEREC;


/**********************************************************
   |TOPIC file structures
**********************************************************/

/* |TOPIC Block header - Header for a block of topic data. If
   uncompressed, there's only one of these at the beginning of the
   file. If the help file is compressed, then these occur in 4k
   increments. (e.g. 0x0000, 0x1000, 0x2000, 0x3000, 0x4000, etc. ) */

typedef long  TOPICOFFSET;

typedef struct tagTOPICBLOCKHEADER {
    long    LastTopicLink;   /* Offset of last topic link in previous block   */
    long    TopicData;       /* Offset of topic data start                    */
    long    LastTopicHeader; /* Offset of last topic header in previous block */
} TOPICBLOCKHEADER;

/* Linked list record for |TOPIC file */
typedef struct tagPARAGRAPH {
    long      BlockSize;   /* Size of this link + Data          */
    long      DataLen2;    /* Length of LinkData2               */
    long      PrevBlock;   /* Relative to first byte of |TOPIC */
    long      NextBlock;   /* Relative to first byte of |TOPIC */
    long      DataLen1;    /* Len(LinkData1 + 11(hdr size))     */
    BYTE      RecordType;  /* See below                         */
    BYTE      *LinkData1;  /* Data associated with this link    */
    BYTE      *LinkData2;  /* Second set of data                */
} PARAGRAPH;
```

Listing 4.3 (continued)

```
/* Known record types for topic link */
#define TL_TOPICHDR    0x02  /* Topic header information */
#define TL_DISPLAY     0x20  /* Displayable information  */
#define TL_TABLE       0x23  /* WinHelp Table            */

/* Topic header. Starts inside LinkData of a type 0x02 record */
typedef struct tagTOPICHEADER {
    long           BlockSize; /* Size of topic, including internal topic links  */
    TOPICOFFSET    BrowseBck; /* Topic offset for prev topic in Browse sequence */
    TOPICOFFSET    BrowseFor; /* Topic offset for next topic in Browse sequence */
    DWORD          TopicNum;  /* Topic Number(?)                                */
    long           NonScroll; /* Start of Non-Scroll Region                     */
    long           Scroll;    /* Start of Scrolling Region of text.             */
    TOPICOFFSET    NextTopic; /* Start of next Type 0x02 record                 */
} TOPICHEADER;

/************************************************************
   Structures for other system files
************************************************************/

/* Header for |TOMAP file */
typedef struct tagTOMAPHEADER {
    long    IndexTopic;   /* Index topic for help file */
    long    Reserved[15];
    short   ToMapLen;     /* Number of topic pointers  */
    long    *TopicPtr;    /* Pointer to all the topics */
} TOMAPHEADER;

/* Record from |CTXOMAP file. Created from the [MAP] section of .HPJ file */
typedef struct tagCTXOMAPREC {
    long     MapID;
    long     TopicOffset;
} CTXOMAPREC;


/* Record from |CONTEXT file */
typedef struct tagCONTEXTREC {
    long     HashValue;       /* Hash value of a phrase     */
    long     TopicOffset;     /* Topic offset of the phrase */
} CONTEXTREC;
```

Listing 4.4 *HLPDUMP2.H.*

```
/***********************************************************************
 *
 * PROGRAM: HLPDUMP2.H
 *
 * PURPOSE: Header file for HLPDUMP2.C
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 4, Windows Help File Format, from Undocumented Windows
 * File Formats, published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 ***********************************************************************/

/************************************************************
  Global Variables
************************************************************/

int          ReadHFSFile;
int          ForceHex;

/* While it appears HFS filenames don't exceed 8          */
/* it's safer to assume that the limit is higher.         */
char         HFSFileToRead[255];
char         SysLoaded;
HELPHEADER   HelpHeader;
SYSTEMHEADER SysInfo;
int*         PhrOffsets;
char*        Phrases;
int          NumPhrases;

/************************************************************
  Function Prototypes
************************************************************/

long LoadHeader(FILE *);
long Decompress(FILE *, long, char *);
void HexDumpData(FILE *, long);
void HexDumpFile(FILE *, long);
void SystemDump(FILE *, long);
void FontDump(FILE *, long);
void ContextDump(FILE *, long);
void PrintPhrase(long);
void KWBTreeDump(FILE *, long);
void KWDataDump(FILE *, long);
void KWMapDump(FILE *, long);
void TTLDump(FILE *, long);
void PhrasesDump(FILE *, long);
int  SysLoad(FILE *, long);
void PhrasesLoad(FILE *, long);
char FindFile(FILE *, char*, long*);
void DumpFile(FILE *);
void ListFiles(FILE *);
void HelpDump(FILE *);
void Usage();
int  main(int, char**);
```

Listing 4.5 *HLPDUMP2.C.*

```
/**********************************************************************
 *
 * PROGRAM: HLPDUMP2.C
 *
 * PURPOSE: A dump program that lets you view internal WinHelp files.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 4, Windows Help File Format, from Undocumented Windows
 * File Formats, published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 **********************************************************************/

#define MEM_DEBUG 1

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>
#include <ctype.h>
#include <limits.h>
#include "winhelp.h"
#include "hlpdump2.h"

#define HLP_DEBUG 1

#define CHECK_SIGNATURE(is, shouldbe) \
         {if (is != shouldbe) { \
            printf("Signature should be %x, but is %x\n"); \
            return;} }


/* Tells us if a particular bit is set or not */
#define BITSET(bitmap, bit) ((bitmap & (1 << bit)) ? 1 : 0 )

/* Sum of set bits in a byte + 8 */
#define BYTESTOREAD(bitmap) \
        ( BITSET(bitmap, 0) + BITSET(bitmap, 1) + \
          BITSET(bitmap, 2) + BITSET(bitmap, 3) + \
          BITSET(bitmap, 4) + BITSET(bitmap, 5) + \
          BITSET(bitmap, 6) + BITSET(bitmap, 7) + 8 )
```

Listing 4.5 (continued)

```
/**************************************************
  Loads the HFSFileHeader
**************************************************/

long LoadHeader(FILE *HelpFile)
{
  HFSFILEHEADER fileHeader;

  fread(&fileHeader, sizeof(HFSFILEHEADER), 1, HelpFile);

#ifdef HLP_DEBUG
  printf("DEBUG -> LoadHeader()\n");
  printf("File plus Header: %ld\n", fileHeader.FilePlusHeader);
  printf("File size: %ld\n", fileHeader.FileSize);
  printf("File type: 0x%02x\n\n", fileHeader.FileType);
#endif

  return fileHeader.FileSize;
}


/************************************************
  Decompresses the data using Microsoft's LZ77
  derivative (called Zeck Compression)
************************************************/

long Decompress(FILE *HelpFile, long CompSize, char *Buffer)
{

long InBytes = 0;        /* How many bytes read in                        */
long OutBytes = 0;       /* How many bytes written out                    */
BYTE BitMap, Set[16];    /* Bitmap and bytes associated with it           */
long NumToRead;          /* Number of bytes to read for next group        */
long counter, Index;     /* Going through next 8-16 codes or chars        */
long Length, Distance;   /* Code length and distance back in 'window' */
char *CurrPos;           /* Where we are at any given moment              */
char *CodePtr;           /* Pointer to back-up in LZ77 'window'           */


  CurrPos = Buffer;

  /* Go through until we're done */
  while (InBytes < CompSize) {

    /* Get BitMap and data following it */
    BitMap = (BYTE) fgetc(HelpFile);
    NumToRead = BYTESTOREAD(BitMap);
```

Listing 4.5 (continued)

```
    /* If we're trying to read more than we've
       got left, only read what we have left.  */
    NumToRead = (CompSize - InBytes) < NumToRead ? CompSize-InBytes : NumToRead;

    fread(Set, 1, (int) NumToRead, HelpFile);
    InBytes += NumToRead + 1;

    /* Go through and decode data */
    for (counter = 0, Index = 0; counter < 8; counter++) {

      /* It's a code, so decode it and copy the data */
      if (BITSET(BitMap, counter)) {
        Length = ((Set[Index+1] & 0xF0) >> 4) + 3;
        Distance = ((Set[Index+1] & 0x0F) << 8) + Set[Index] + 1;
        CodePtr = CurrPos - Distance;

        /* Copy data from 'window' */
        while (Length) {
          *CurrPos++ = *CodePtr++;
          OutBytes++;
          Length--;
        }
        Index += 2;
      }
      else {
        *CurrPos++ = Set[Index++];
        OutBytes++;
      }
    } /* for */
  } /* while  */
  return OutBytes;
}


/***************************************************
  Performs a Hex/ASCII dump of an HFS file.
***************************************************/

void HexDumpData(FILE *HelpFile, long fileSize)
{
  char          Buffer[16];
  long          counter;
  long          BytesToPrint, Index;

  printf("Offset                      Hex Values                              Ascii\n");

printf("---------------------------------------------------------------------\n");

  for (counter = 0; counter < fileSize; counter+=16) {

    printf("0x%08lX: ", counter);

    /* If this is the last line, how many bytes are in it? */
    BytesToPrint = ((fileSize - counter) > 16) ? 16 : (fileSize - counter);
    fread(Buffer, BytesToPrint, 1, HelpFile);
```

Listing 4.5 (continued)

```
    /* Dump Hex */
    for (Index=0; Index < BytesToPrint; Index++)
      printf("%02X ", (BYTE) Buffer[Index]);

    /* If last line, fill in blanks */
    for (Index=0; Index < 16-BytesToPrint; Index++)
      printf("   ");

    /* Dump Ascii */
    for (Index=0; Index < BytesToPrint; Index++)
      putchar( isprint( Buffer[Index] ) ? Buffer[Index] : '.' );

    putchar('\n');
  }

  free(Buffer);
}


/**************************************************
  Performs a Hex/ASCII dump of an HFS file.
**************************************************/

void HexDumpFile(FILE *HelpFile, long FileStart)
{
  long           fileSize;

  fseek(HelpFile, FileStart, SEEK_SET);

  fileSize = LoadHeader(HelpFile);

  printf("File Size: 0x%08lX\n\n", fileSize);

  HexDumpData(HelpFile, fileSize);
}

/**************************************************
  Dumps the |SYSTEM info
**************************************************/

void SystemDump(FILE *HelpFile, long FileStart)
{

  char            HelpFileTitle[33];
  SYSTEMREC       SystemRec;
  long            CurrentLocation;
  struct tm       *TimeRec;
  SECWINDOW       *SWin;     /* Secondary Window record */
  long            fileSize;
  SYSTEMHEADER    SysHeader;

  fseek(HelpFile, FileStart, SEEK_SET);

  fileSize = LoadHeader(HelpFile);

  fread(&SysHeader, sizeof(SysHeader), 1, HelpFile);
  printf("|SYSTEM Dump\n\n\n");
```

Listing 4.5 (continued)

```
/* Figure out Version and Revision */
if (SysHeader.Revision == 0x15) printf("HC.EXE  3.10 Help Compiler used\n");
else if (SysHeader.Revision == 0x21) printf("HCW.EXE. 4.00 or MVC.EXE\n");

printf("\nVersion: %d\nRevision: %d\n", SysHeader.Version, SysHeader.Revision);
printf("Flag: 0x%04X  - ",SysHeader.Flags);

/* Determine compression, if any. */
if (SysHeader.Flags == NO_COMPRESSION) printf("No compression\n");
else if (SysHeader.Flags & COMPRESSION_HIGH) printf("High Compression\n");
else printf("Unknown Compression: 0x%02x\n", SysHeader.Flags);

TimeRec=localtime(&SysHeader.GenDate);
printf("Help File Generated: %s", asctime(TimeRec));

/* If 3.0 get title */
CurrentLocation=12;
if (SysHeader.Revision == 0x0F) {
   fgets(HelpFileTitle, 33, HelpFile);
   printf("Help File Title: %s\n", HelpFileTitle);
}

/* Else, get 3.1 System records */
else {
  while (CurrentLocation < fileSize) {

    /* Read in system record and SystemRec data */
    fread(&SystemRec, 4, 1, HelpFile);
    SystemRec.RData = malloc(SystemRec.DataSize);
    if (SystemRec.RData == NULL)
    {
      printf("Allocation of SystemRec.RData failed.");
      return;
    }
    fread(SystemRec.RData, SystemRec.DataSize, 1, HelpFile);
    CurrentLocation=CurrentLocation+4+SystemRec.DataSize;

    switch(SystemRec.RecordType)
    {
      case 0x0001:  printf("Help File Title: %s\n", SystemRec.RData);
        break;

      case 0x0002:  printf("Copyright Notice: %s\n", SystemRec.RData);
        break;

      case 0x0003:  printf("Contents ID: 0x%04X\n", (long) *SystemRec.RData);
        break;
```

Listing 4.5 (continued)

```
        case 0x0004:  printf("Macro Data: %s\n",SystemRec.RData);
          break;

        case 0x0005:  printf("Icon in System record\n");
          break;

        case 0x0006:  printf("\nSecondary window:\n");
                      SWin = (SECWINDOW *)SystemRec.RData;
                      printf("Flag: %d\n", SWin->Flags);
                      if (SWin->Flags & WSYSFLAG_TYPE)
                        printf("Type: %s\n", SWin->Type);
                      if (SWin->Flags & WSYSFLAG_NAME)
                        printf("Name: %s\n", SWin->Name);
                      if (SWin->Flags & WSYSFLAG_CAPTION)
                        printf("Caption: %s\n", SWin->Caption);
                      if (SWin->Flags & WSYSFLAG_X)
                        printf("X: %d\n", SWin->X);
                      if (SWin->Flags & WSYSFLAG_Y)
                        printf("Y: %d\n", SWin->Y);
                      if (SWin->Flags & WSYSFLAG_WIDTH)
                        printf("Width: %d\n", SWin->Width);
                      if (SWin->Flags & WSYSFLAG_HEIGHT)
                        printf("Height: %d\n", SWin->Height);
                      if (SWin->Flags & WSYSFLAG_MAXIMIZE)
                        printf("Maximize Flag: %d\n", SWin->Maximize);
                      if (SWin->Flags & WSYSFLAG_RGB)
                        printf("RGB Foreground Colors Set\n");
                      if (SWin->Flags & WSYSFLAG_RGBNSR)
                        printf("RGB For Non-Scrollable Region Set\n");
                      if (SWin->Flags & WSYSFLAG_TOP)
                        printf("Secondary Window is always On Top\n");
          break;

        case 0x0008:  printf("Citation: %s\n", SystemRec.RData);
          break;

        case 0x000A:  printf("\nContents File: %s\n\n",SystemRec.RData);
          break;

        default:      printf("\nUnknown record type: 0x%04X\n",SystemRec.RecordType);
                      /* Back-up and hex-dump the data */
                      fseek(HelpFile, ftell(HelpFile) - SystemRec.DataSize, SEEK_SET);
                      HexDumpData(HelpFile, SystemRec.DataSize);
      } /* switch */

      free(SystemRec.RData);

    } /* while */
  } /* else */
} /* SysDump */
```

Listing 4.5 (continued)

```
/*************************************************
  Dumps the |FONT file
**************************************************/

void FontDump(FILE *HelpFile, long FileStart)
{
  FONTHEADER      FontHdr;
  FONTDESCRIPTOR  FontDesc;
  char            AFont[32];
  long            FontStart, CurrLoc;
  long            fileSize;
  long            counter;
  long            NameLen;

  /* Go to the FONT file and get the headers */
  fseek(HelpFile, FileStart, SEEK_SET);
  fileSize = LoadHeader(HelpFile);

  fread(&FontHdr, sizeof(FontHdr), 1, HelpFile);

  printf("|FONTS\n\n Number Fonts: %d\n",FontHdr.NumFonts);
  printf("Font #  -  Font Name\n");


  /* Font names are 20 chars prior to Winhelp 4.0 */
  /* In WinHelp 4.0, they are 32 characters.      */
  if (SysInfo.Revision == 0x15)
  {
    NameLen = 20;
  }
  else
  {
    NameLen = 32;
  }

  /* Keep track of start of fonts */
  FontStart = ftell(HelpFile);
  for (counter = 0; counter < (long) FontHdr.NumFonts; counter++)
  {
    fread(AFont, NameLen, 1, HelpFile);
    printf(" %3d    -  %s\n", counter, AFont);
  }

  /* Go to Font Descriptors. Don't actually need this, because we're
     there, but wanted to show how to get there using the offset.    */
  fseek(HelpFile, FontStart + (long)(FontHdr.DescriptorsOffset) - sizeof(FontHdr),
        SEEK_SET);
  printf("\nNum Font Descriptors: %d\n", FontHdr.NumDescriptors);
  printf("Default Descriptor: %d\n\n", FontHdr.DefDescriptor);
  printf("Attributes: n=none  b=bold  i=ital  u=undr  s=strkout")
  printf("            d=dblundr  C=smallcaps\n\n");
  printf("Font Name                            PointSize  Family")
  printf("   FG RGB      BG RGB   Attr\n");
```

Listing 4.5 (continued)

```
printf("--------------------------------------------------------------------------\n");

  for (counter = 0; counter < (long) FontHdr.NumDescriptors; counter++) {
    fread(&FontDesc, sizeof(FontDesc), 1, HelpFile);
    CurrLoc = ftell(HelpFile);
    fseek(HelpFile, FontStart + (NameLen * FontDesc.FontName), SEEK_SET);
    fread(AFont, NameLen, 1, HelpFile);
    fseek(HelpFile, CurrLoc, SEEK_SET);

    /* write out info on Font descriptor */
    printf("%-32s    %4.1f    ", AFont, (float)(FontDesc.HalfPoints / 2));
    switch (FontDesc.FontFamily) {
      case FAM_MODERN: printf("Modern");
                       break;

      case FAM_ROMAN:  printf("Roman ");
                       break;

      case FAM_SWISS:  printf("Swiss ");
                       break;

      case FAM_SCRIPT: printf("Script");
                       break;

      case FAM_DECOR:  printf("Decor ");
                       break;

      default:         printf("0X%02X ", FontDesc.FontFamily);
                       break;
    } /* Switch */
    printf(" 0X%08lX  ",RGB(FontDesc.SRRGB.rgbRed,
                            FontDesc.SRRGB.rgbGreen,
                            FontDesc.SRRGB.rgbBlue));
    printf("0X%08lX    ",RGB(FontDesc.NSRRGB.rgbRed,
                             FontDesc.NSRRGB.rgbGreen,
                             FontDesc.NSRRGB.rgbBlue));

    if (FontDesc.Attributes == 0) putchar('n');
    if (FontDesc.Attributes & FONT_BOLD) putchar('b');
    if (FontDesc.Attributes & FONT_ITAL) putchar('i');
    if (FontDesc.Attributes & FONT_UNDR) putchar('u');
    if (FontDesc.Attributes & FONT_STRK) putchar('s');
    if (FontDesc.Attributes & FONT_DBUN) putchar('d');
    if (FontDesc.Attributes & FONT_SMCP) putchar('C');
    printf("\n");

#ifdef HLP_DEBUG
     printf("Unknown = %d\n", FontDesc.Unknown);
#endif
   }
}
```

Listing 4.5 (continued)

```
/***************************************************
  Dumps the |CONTEXT file
****************************************************/

void ContextDump(FILE *HelpFile, long FileStart)
{
  long             count;
  long             CurrPage, FirstPageLoc;
  long             TopicOffset, HashValue;
  BTREEHEADER      BTreeHdr;
  BTREELEAFHEADER CurrNode;

  /* Go to the TTLBTREE file and get the headers */
  fseek(HelpFile, FileStart, SEEK_SET);
  LoadHeader(HelpFile);
  fread(&BTreeHdr, sizeof(BTreeHdr), 1, HelpFile);

  /* Save the current location */
  FirstPageLoc = ftell(HelpFile);

  fseek(HelpFile, FirstPageLoc+(BTreeHdr.RootPage * BTreeHdr.PageSize), SEEK_SET);

  printf("# Context Hash Values in |CONTEXT %lu\n\n", BTreeHdr.TotalBtreeEntries);
  CurrPage = BTreeHdr.FirstLeaf;

  do
  {
    fseek(HelpFile, FirstPageLoc+(CurrPage * BTreeHdr.PageSize), SEEK_SET);
    fread(&CurrNode, 8, 1, HelpFile);
    for(count = 1; count <= CurrNode.NEntries; count++)
    {
      fread(&HashValue, sizeof(HashValue), 1, HelpFile);
      fread(&TopicOffset, sizeof(TopicOffset), 1, HelpFile);
      printf("Topic Offset:0x%08lX  Hash Value: 0x%08lX\n", TopicOffset, HashValue);
    }
    CurrPage = CurrNode.NextPage;
  } while(CurrPage != -1);
}


/***************************************************
  Prints a Phrase from the Phrase table.
****************************************************/
void PrintPhrase(long PhraseNum)
{
  short *Offsets;
  char  *p;

  p = Phrases+Offsets[PhraseNum];
  while (p < Phrases + Offsets[PhraseNum + 1])
    putchar(*p++);
}
```

Listing 4.5 (continued)

```
/***************************************************
  Dumps the Keyword B-Tree
***************************************************/

void KWBTreeDump(FILE *HelpFile, long FileStart)
{
  char            Keyword[80], c;
  long            count, Index;
  long            CurrPage, FirstPageLoc;
  long            KeywordOffset;
  long            KeywordCount;
  BTREEHEADER     BTreeHdr;
  BTREELEAFHEADER CurrNode;

  /* Go to the KWBTREE file and get the headers */
  fseek(HelpFile, FileStart, SEEK_SET);
  LoadHeader(HelpFile);
  fread(&BTreeHdr, sizeof(BTreeHdr), 1, HelpFile);

  /* Save the current location */
  FirstPageLoc = ftell(HelpFile);

  fseek(HelpFile, FirstPageLoc+(BTreeHdr.RootPage * BTreeHdr.PageSize), SEEK_SET);

  printf("# Keywords - %lu\n\n", BTreeHdr.TotalBtreeEntries);
  CurrPage = BTreeHdr.FirstLeaf;

  do
  {
    fseek(HelpFile, FirstPageLoc+(CurrPage * BTreeHdr.PageSize), SEEK_SET);
    fread(&CurrNode, 8, 1, HelpFile);
    for(count = 1; count <= CurrNode.NEntries; count++)
    {
      Index = 0;
      while(c = (char) fgetc(HelpFile))
        Keyword[Index++] = c;

      Keyword[Index] = 0;
      fread(&KeywordCount, sizeof(KeywordCount), 1, HelpFile);
      fread(&KeywordOffset, sizeof(KeywordOffset), 1, HelpFile);

      printf("Offset: 0x%08lX  Count: %d  Keyword: %s\n",
             KeywordOffset,
             KeywordCount,
             Keyword);
    }
    CurrPage = CurrNode.NextPage;
  } while(CurrPage != -1);
}
```

Listing 4.5 (continued)

```
/**************************************************
  Dumps the Keyword Data file
**************************************************/

void KWDataDump(FILE *HelpFile, long FileStart)
{
  long fileSize;
  long nIndex;
  long Offset;

  /* Go to the KWDATA file and get the headers */
  fseek(HelpFile, FileStart, SEEK_SET);
  fileSize = LoadHeader(HelpFile);

  printf("Dumping Keyword Data File\n\n");
  /* Go through all keyword offsets (fileSize / 4) */
  for (nIndex = 0; nIndex < (fileSize / sizeof(Offset)); nIndex++)
  {
    fread(&Offset, sizeof(Offset), 1, HelpFile);
    printf("Index: %d   Offset: 0x%08lX\n", nIndex, Offset);
  }
}


/**************************************************
  Dumps the Keyword Map file
**************************************************/

void KWMapDump(FILE *HelpFile, long FileStart)
{
  long      fileSize;
  long      nIndex;
  WORD      nKWMaps;
  KWMAPREC  kwMap;

  /* Go to the KWMAP file and get the headers */
  fseek(HelpFile, FileStart, SEEK_SET);
  fileSize = LoadHeader(HelpFile);

  printf("Dumping Keyword Map\n\n");

  fread(&nKWMaps, sizeof(nKWMaps), 1, HelpFile);

  /* Go through all keyword offsets (fileSize / 4) */
  for (nIndex = 0; nIndex < (int) nKWMaps; nIndex++)
  {
    fread(&kwMap, sizeof(KWMAPREC), 1, HelpFile);
    printf("Index: %d   First Keyword 0x%08lX    Leaf Page#: %05u\n",
           nIndex, kwMap.FirstRec, kwMap.PageNum);
  }
}
```

Listing 4.5 (continued)

```
/***************************************************
  Dumps the Topic Titles B-Tree
***************************************************/

void TTLDump(FILE *HelpFile, long FileStart)
{
  char            Title[80], c;
  long            count, Index;
  long            CurrPage, FirstPageLoc, TopicOffset;
  BTREEHEADER     BTreeHdr;
  BTREELEAFHEADER CurrNode;

  /* Go to the TTLBTREE file and get the headers */
  fseek(HelpFile, FileStart, SEEK_SET);
  LoadHeader(HelpFile);
  fread(&BTreeHdr, sizeof(BTreeHdr), 1, HelpFile);

  /* Save the current location */
  FirstPageLoc = ftell(HelpFile);

  fseek(HelpFile, FirstPageLoc+(BTreeHdr.RootPage * BTreeHdr.PageSize),
        SEEK_SET);

  printf("# Titles in |TTLBTREE %lu\n\n",BTreeHdr.TotalBtreeEntries);
  CurrPage = BTreeHdr.FirstLeaf;

  do
  {
    fseek(HelpFile, FirstPageLoc+(CurrPage * BTreeHdr.PageSize), SEEK_SET);
    fread(&CurrNode, 8, 1, HelpFile);
    for(count = 1; count <= CurrNode.NEntries; count++)
    {
      fread(&TopicOffset, sizeof(TopicOffset), 1, HelpFile);
      Index = 0;
      while(c = (char) fgetc(HelpFile))
        Title[Index++] = c;

      Title[Index] = 0;

      printf("Topic Offset:0x%08lX  Title: %s\n", TopicOffset, Title);
    }
    CurrPage = CurrNode.NextPage;

  } while(CurrPage != -1);
}
```

Listing 4.5 (continued)

```
/**************************************************
  Dumps the |Phrases file
**************************************************/

void PhrasesDump(FILE *HelpFile, long FileStart)
{
  long  nOuterIndex, nInnerIndex;
  WORD  start, len;

  PhrasesLoad(HelpFile, FileStart);

  printf("Phrase#     Phrase\n");
  for (nOuterIndex = 0; nOuterIndex < NumPhrases; nOuterIndex++)
  {
    start = (WORD) PhrOffsets[nOuterIndex] ;
    len = PhrOffsets[nOuterIndex + 1] - start;
    printf(" %5d     ", nOuterIndex + 1);
    for (nInnerIndex = 0; nInnerIndex < (int) len; nInnerIndex++)
    {
      printf("%c", (Phrases[start + nInnerIndex]));
    }
    printf("\n");
  }
}


/**************************************************
  Loads the |SYSTEM info
**************************************************/

int SysLoad(FILE *HelpFile, long FileStart)
{
  fseek(HelpFile, FileStart, SEEK_SET);

  LoadHeader(HelpFile);

  fread(&SysInfo, sizeof(SYSTEMHEADER), 1, HelpFile);

#ifdef HLP_DEBUG
  printf("DEBUG -> SysLoad()\n");
  printf("Magic: 0x%02x\n", SysInfo.Magic);
  printf("Version: 0x%02x\n", SysInfo.Version);
  printf("Revision: 0x%02x\n", SysInfo.Revision);
  printf("Flags: 0x%04x\n\n", SysInfo.Flags);
#endif

  if (SysInfo.Magic != SYS_MAGIC)
  {
    return 0;
  }

  return 1;
}
```

Listing 4.5 (continued)

```
/**************************************************
  Loads the compression phrases
**************************************************/

void PhrasesLoad(FILE *HelpFile, long FileStart)
{
  PHRASEHEADER  phraseHeader;
  long          FileSize;
  long          DeCompSize;

  fseek(HelpFile, FileStart, SEEK_SET);
  FileSize = LoadHeader(HelpFile);

  fread(&phraseHeader, sizeof(phraseHeader), 1, HelpFile);

  if (SysInfo.Flags != NO_COMPRESSION)
  {
    if ((PhrOffsets = malloc(phraseHeader.PhrasesSize +
                       (phraseHeader.NumPhrases + 1) * 2 + 10)) == NULL)
    {
      printf("Unable to allocate space for Phrases.\n");
      return;
    }

    /* Assign Phrases to  where the comrpessed phrases are */
    Phrases = (char*) PhrOffsets + fread(PhrOffsets,
                                         2,
                                         phraseHeader.NumPhrases + 1,
                                         HelpFile);

    DeCompSize = Decompress(HelpFile, FileSize - (sizeof(phraseHeader) +
                           2 * (phraseHeader.NumPhrases + 1)), Phrases);
    if (DeCompSize != (long) phraseHeader.PhrasesSize)
    {
      printf("Warning, Phrases did not decompress to the proper size.\n");
    }
  }
  else
  {
    if ((PhrOffsets = malloc(phraseHeader.PhrasesSize +
                       (phraseHeader.NumPhrases + 1) * 2 + 10)) == NULL)
    {
      printf("Unable to allocate space for Phrases.\n");
      return;
    }
    /* Back up four bytes if phrases aren't compressed */
    /* because PhrasesSize field doesn't exist.        */
    fseek(HelpFile, -4, SEEK_CUR);
    fread(PhrOffsets, FileSize - 4, 1, HelpFile);
  }
  /* Reset Phrases to be equal to PhrOffsets */
  Phrases = (char *) PhrOffsets;
  NumPhrases = phraseHeader.NumPhrases;
}
```

Listing 4.5 (continued)

```
/**************************************************
  Finds an HFS File by traversing the HFS b-tree
  Note: This is the only place I actually traverse
  the b-tree instead of cycling through the leaf
  pages. I put this in specifically to show how to
  traverse the b-tree, since speed isn't a real
  concern for HelpDump.
***************************************************/

char FindFile(FILE *HelpFile, char* filename, long* offset)
{
  BTREEHEADER        HFSHeader;
  BTREEINDEXHEADER*  HFSIndexHeader;
  BTREELEAFHEADER*   HFSLeafHeader;
  long               HFSStart;
  short*             pNextPage;
  char*              buffer;
  char*              currPtr;
  long               nKeys, nFiles;
  char               found = 0;
  long               currLevel = 1;

  /* Go to the HFS and read the header. */
  fseek(HelpFile, HelpHeader.HFSLoc, SEEK_SET);
  LoadHeader(HelpFile);
  fread(&HFSHeader, sizeof(HFSHeader), 1, HelpFile);

  /* Allocate space for read buffer */
  buffer = malloc(HFSHeader.PageSize);
  if (buffer == NULL)
  {
    printf("Unable to allocate space for buffer.\n");
    return found;
  }
  HFSIndexHeader = (BTREEINDEXHEADER*) buffer;

  HFSStart = ftell(HelpFile);

  /* Advance to root page */
  fseek(HelpFile, (HFSHeader.RootPage * HFSHeader.PageSize) + HFSStart, SEEK_SET);

  /* If there's only one page, then it must be a leaf */
  if (HFSHeader.TotalPages > 1)
  {
    /* Traverse b-tree looking for the key for the leaf page */
    while (!found)
    {
      /* Read in the page */
      fread(buffer, HFSHeader.PageSize, 1, HelpFile);
      currPtr = buffer + sizeof(BTREEINDEXHEADER);
      pNextPage = (int *) currPtr;

      currPtr += sizeof(int);
```

Listing 4.5 (continued)

```
      /* Go through all keys in the page */
      for (nKeys = 0; nKeys < HFSIndexHeader->NEntries; nKeys++)
      {
        /* If filename is less than key, this is our page. */
        if (strcmp(filename, currPtr) < 0)
        {
          break;
        }
        else
        {
          /* Advance to the next page# */
          while (*currPtr)
            currPtr++;
          currPtr++;
          pNextPage = (int *) currPtr;
          currPtr += sizeof(int);
        }
      }

      /* Advance to next page */
      fseek(HelpFile,
            (*pNextPage * HFSHeader.PageSize) + HFSStart,
            SEEK_SET);

      /* If this is the last index page */
      /* then pNextPage points to a     */
      /* leaf page                      */
      if (currLevel == HFSHeader.nLevels - 1)
      {
        found = 1;
      }
      currLevel++;
    }
  }

  fread(buffer, HFSHeader.PageSize, 1, HelpFile);
  HFSLeafHeader = (BTREELEAFHEADER*) buffer;
  currPtr = buffer + sizeof(BTREELEAFHEADER);


  found = 0;

  /* Loop through all files in this page */
  for (nFiles = 0; nFiles < HFSLeafHeader->NEntries; nFiles++)
  {
    if (strcmp(filename, currPtr))
    {
      /* Advance to the file offset */
      while (*currPtr)
        currPtr++;

      /* Move past the null and file offset */
      /* to next file                       */
      currPtr += 5;
    }
```

Listing 4.5 (continued)

```
    else
    {
      /* Save the offset of the file */
      while (*currPtr)
        currPtr++;
      currPtr++;

     *offset = (long) *((long*) currPtr);
      found = 1;
      break;
    }
  }

  /* TRUE if file was found, FALSE if it wasn't */
  return found;
}


/**************************************************
  DumpFile
**************************************************/

void DumpFile(FILE *HelpFile)
{
  long fileOffset;
  char fileName[255];

  /* For many files we need to know information about the system,            */
  /* so we'll load the system data info here  if it's available             */
  SysLoaded = 0;
  strcpy(fileName, "|SYSTEM");
  if (FindFile(HelpFile, fileName, &fileOffset))
  {
     SysLoaded = (char) SysLoad(HelpFile, fileOffset);
  }

  /* If it's not a version 3 help file  then we can't handle it.             */
  if (SysInfo.Version != 0x03)
  {
    printf("Warning: Not a version 3 help file. Version is %d.\n", SysInfo.Version);
  }

  /* If it's version 3, but not revision 0x15 or 0x21, we also can't handle it */
  if (SysInfo.Revision != 0x15 && SysInfo.Revision != 0x21)
  {
    printf("Revision not 0x15 or 0x21. Revision is 0x%02x.\n", SysInfo.Revision);
  }

  /* If we're reading the |TOPIC file, then we need to */
  /* pre-load the phrases from the |Phrases file       */
  if (!strcmp(HFSFileToRead, "|TOPIC") || !strcmp(HFSFileToRead, "TOPIC"))
  {
    strcpy(fileName, "|Phrases");
    if (FindFile(HelpFile, fileName, &fileOffset))
    {
      PhrasesLoad(HelpFile, fileOffset);
    }
  }
```

Listing 4.5 (continued)

```
  strcpy(fileName, HFSFileToRead);
  if (!FindFile(HelpFile, fileName, &fileOffset))
  {
    /* Append a "|" character to the beginning  */
    /* of the filename and try to find the file */
    strcpy(fileName, "|");
    strcat(fileName, HFSFileToRead);
    if (!FindFile(HelpFile, fileName, &fileOffset))
    {
      /* File not found */
      printf("Error: Unable to find HFS file %s or %s\n", HFSFileToRead, fileName);
      return;
    }
  }

  if (!ForceHex)
  {
    if (!strcmp(fileName, "|SYSTEM"))
      SystemDump(HelpFile, fileOffset);
    else if (!strcmp(fileName, "|TTLBTREE"))
      TTLDump(HelpFile, fileOffset);
    else if (!strcmp(fileName, "|CONTEXT"))
      ContextDump(HelpFile, fileOffset);
    else if (!strcmp(fileName, "|FONT"))
      FontDump(HelpFile, fileOffset);
    else if (!strcmp(fileName, "|KWBTREE"))
      KWBTreeDump(HelpFile, fileOffset);
    else if (!strcmp(fileName, "|AWBTREE"))
      KWBTreeDump(HelpFile, fileOffset);
    else if (!strcmp(fileName, "|KWDATA"))
      KWDataDump(HelpFile, fileOffset);
    else if (!strcmp(fileName, "|AWDATA"))
      KWDataDump(HelpFile, fileOffset);
    else if (!strcmp(fileName, "|KWMAP"))
      KWMapDump(HelpFile, fileOffset);
    else if (!strcmp(fileName, "|Phrases"))
      PhrasesDump(HelpFile, fileOffset);
    else
      HexDumpFile(HelpFile, fileOffset);
  }
  else
  {
    HexDumpFile(HelpFile, fileOffset);
  }
}

/***************************************************
  List out all the HFS files in a .HLP file
***************************************************/

void ListFiles(FILE *HelpFile)
{
  BTREEHEADER       HFSHeader;
  BTREELEAFHEADER*  HFSLeafHeader;
  long              HFSStart;
  char*             buffer;
  char*             currPtr;
  long              nIndex, nFiles;
  long*             offset;
```

Listing 4.5 (continued)

```
  /* Go to the HFS and read the header. */
  fseek(HelpFile, HelpHeader.HFSLoc, SEEK_SET);
  LoadHeader(HelpFile);
  fread(&HFSHeader, sizeof(HFSHeader), 1, HelpFile);

  CHECK_SIGNATURE(HFSHeader.Signature, 0x293B);

#ifdef HLP_DEBUG
  printf("DEBUG -> ListFiles()\n");
  printf("B-Tree Page Size %d\n", HFSHeader.PageSize);
  printf("B-Tree First Leaf %d\n", HFSHeader.FirstLeaf);
  printf("B-Tree Num. Levels %d\n", HFSHeader.nLevels);
  printf("B-Tree Total Pages %d\n", HFSHeader.TotalPages);
  printf("B-Tree Total # Entries %ld\n", HFSHeader.TotalBtreeEntries);
#endif
  /* Start of the HFS b-tree pages */
  HFSStart = ftell(HelpFile);

  /* Go to the first leaf page of the HFS */
  fseek(HelpFile, HFSHeader.FirstLeaf * HFSHeader.PageSize, SEEK_CUR);

  /* Allocate space for read buffer and read first page */
  buffer = malloc(HFSHeader.PageSize);
  if (buffer == NULL)
  {
    printf("Unable to allocate space for buffer.\n");
    return;
  }
  HFSLeafHeader = (BTREELEAFHEADER*) buffer;

  printf("\nHFS Filename                 Offset\n");
  printf("----------------------------------\n");

  /* Loop through all HFS leaf pages */
  for (nIndex = 0; nIndex < HFSHeader.TotalPages; nIndex++)
  {
    fread(buffer, HFSHeader.PageSize, 1, HelpFile);

    currPtr = buffer + sizeof(BTREELEAFHEADER);

    /* Loop through all files in this page */
    for (nFiles = 0; nFiles < HFSLeafHeader->NEntries; nFiles++)
    {

      /* Print filename */
      printf("%-30s", currPtr);

      /* Advance to next filename */
      while (*currPtr)
        currPtr++;
      currPtr++;

            offset = (long*) currPtr;
      /* print offset to file */
      printf("0x%08lX\n", *offset);

      /* Move past the file offset to next file */
      currPtr += 4;
    }
  }
}
```

Listing 4.5 (continued)

```
/**************************************************
  Check to make sure it's a help file. Then
  either dump the HFS directory or dump an HFS
  file.
**************************************************/

void HelpDump(FILE *HelpFile)
{
  fread(&HelpHeader, sizeof(HelpHeader), 1, HelpFile);
  if (HelpHeader.MagicNumber != HF_MAGIC)
  {
    printf("Fatal Error:\n");
    printf("  Not a valid WinHelp file!\n");
    return;
  }

  if (ReadHFSFile)
     DumpFile(HelpFile);
  else
     ListFiles(HelpFile);

}


/**************************************************
  Show usage
**************************************************/

void Usage()
{
  printf("HLPDUMP2  (version 2.0 of Help Dump)\n");
  printf("By Pete Davis and Mike Wallace              Copyright 1997\n\n");
  printf("Usage: HLPDUMP2 helpfile[.hlp] [hfsfilename] [/H]\n\n");
  printf("where:\n");
  printf("  helpfile    - name of .HLP/.GID/.ANN/.BMK file to open\n");
  printf("  hfsfilename - name of HFS file to read\n");
  printf("  /H          - force a hex dump\n\n");
  printf("note: Do not include the pipe '|' character in the hfsfilename.\n");
}


/**************************************************
 Entry point to HLPDUMP2
**************************************************/

int main(int argc, char *argv[])
{
  char filename[_MAX_PATH];
  FILE *HelpFile;

  if (argc < 2)
  {
    Usage();
    return EXIT_FAILURE;
  }
  ReadHFSFile = 0;
  if (argc >= 3)
  {
    strcpy(HFSFileToRead, argv[2]);
    ReadHFSFile = 1;
  }
```

Listing 4.5 (continued)

```
  /* Are we forcing a hex dump? */
  ForceHex = 0;
  if (argc == 4)
  {

  if (stricmp(argv[3], "/H"))
    {
      printf("Error: Argument 3 unrecognized\n");
      return EXIT_FAILURE;
    }
    ForceHex = 1;

  }
  strcpy(filename, argv[1]);
  strupr(filename);
  if (!strchr(filename, '.'))
    strcat(filename, ".HLP");

  if ((HelpFile = fopen(filename, "rb")) == NULL)
  {
    printf("%s does not exist!", filename);
    return EXIT_FAILURE;
  }

  printf("Dumping %s\n\n", filename);
  HelpDump(HelpFile);
  fclose(HelpFile);

  return EXIT_SUCCESS;

}
```

Chapter 5

Annotation (.ANN) and Bookmark (.BMK) File Formats

WinHelp provides Annotation and Bookmark files that users can create while viewing a help file. Like many of the WinHelp files, Annotation and Bookmark files are stored with a Help File System (HFS) described in Chapter 4.

Annotation Files

Annotation (`.ANN`) files are the simpler of the two file formats. As I said earlier, `.ANN` files are based on the HFS, so I'll be speaking in terms of the internal files and ignore the HFS aspect altogether. Each .ANN file automatically contains two files; `@LINK` and `@VERSION`. In addition to these two files is one file for each annotation.

The `@VERSION` file is static and identical in all `.ANN` files that I've seen. It simply contains the following string of 6 bytes: 0x08 0x62 0x6D 0x66 0x01 0x00 (or 8 bytes in WinHelp 4.0: 0x08 0x62 0x6D 0x66 0x01 0x00 0x00 0x00). The meaning behind this has me stumped. 0x62 0x6D 0x66 spell out bmf (BookMark File?), but other than that, we could not come up with a meaning.

The @LINK file is slightly more complex, though still fairly simple. The file begins with a single WORD that contains a count of the annotations in the .ANN file. It is then followed by a record for each annotation (Table 5.1).

As I said, it's fairly simple. The rest of the .ANN file contains a single HFS file for each annotation. These files are named based on the topic offsets of the annotation. For the sake of simplicity, assume you have an annotation for a topic located at offset 0x100. 0x100 in decimal is 256, of course, so the name of the HFS file within the .ANN file for that annotation would be 256!0. Each file basically consists of the decimal equivalent of the offset followed by an exclamation point and a zero. The contents of each of these files is simply the text of the annotation itself.

That's it; that's all there is to an Annotation file.

Bookmark Files

Bookmark files, on the other hand, are actually a little more complex. Again, they are based on the Help File System (HFS). There is only one WinHelp Bookmark file. It is shared by all help files, and is named, appropriately, WINHELP.BMK. All bookmarks for all help files are kept in the WINHELP.BMK file.

One HFS file per help file has bookmarks. That is to say, that when a user creates the first bookmark in a help file, an HFS file is added to the WINHELP.BMK file. This HFS file will contain all bookmarks ever made with this help file. The name of the HFS file in the Bookmark file is based on two things: the name and the generation date of the help file. The generation date can be found in the |SYSTEM HFS file found in the help file. See Chapter 4 for more information on this. The HFS file in the Bookmark file, therefore, contains the name (without the extension) of the help file, and eight characters that represent the hex DWORD value of the generation date.

Assume on December 9th, 1994, at exactly 11:09:30am, you generated the help file TEST.HLP. The generation date in hex would be 2EE8AB6A. If a user were to create a bookmark with this file, the name of the HFS file to contain the bookmark would be TEST2ee8ab6a. Notice that the help file name is all capitals, whereas the hex digits that are letters are all lowercase. This is important because HFS filenames are case sensitive. But things are still fairly simple at this point.

Table 5.1 ANNLINKRECORD record.

Field Name	*Data Type*	*Comments*
TopOffset	DWORD	**Offset to topic associated with the annotation**
Reserved1	DWORD	**Probably reserved for future use**
Reserved2	DWORD	**Probably reserved for future use**

The Bookmark file itself has a bookmark header (Table 5.2).

This is followed by bookmark entries for each bookmark (Table 5.3).

I'm not sure why, but all offsets of bookmarks are associated with the beginning of the scrolling region of the topic instead of the beginning of the topic itself. I don't see why it would matter either way, though.

Where Do I Go from Here?

Surprisingly, quite a bit could be done with Bookmark and Annotation files that Microsoft has neglected to do. First of all, there is currently no utility for upgrading a Bookmark or Annotation file when a help file gets updated. Instead, your annotations and bookmarks just get lost completely when a new version of the help file comes out. That makes them almost entirely useless when you have a help file that is updated on a periodic basis.

Another utility that would be very useful is an expanded annotation editor. Why not have one that supports embedded graphics and maybe RTF text? It would be easy enough to take an existing Annotation file and add new files to it to hold your particular data types. For example, if your annotation's HFS filename is 1242!0, make another one called 1242!x that holds data that your annotation editor program reads. It could

Table 5.2 BOOKMARKHEADER record.

Field Name	*Data Type*	*Comment*
Magic	int	Always 0x0015
GenDate	DWORD	Same date that is part of the HFS filename
NumBookmarks	WORD	Number of bookmarks in this HFS file
FileLen	WORD	Contains the length of this HFS file in bytes

Table 5.3 BOOKMARKENTRY record.

Field Name	*Data Type*	*Comment*
TopOffset	DWORD	Offset to scrolling region of topic this bookmark is associated with
Reserved	DWORD	Reserved for future use
BkmrkText[]	char	Null-terminated string of bookmark text

use both files and keep straight text in the 1242!0 file (so that WinHelp's built-in annotation code could read it), and you could keep your special data in 1242!x. Simply create a DLL that attaches to WinHelp. (See the Bibliography for an article I wrote for *PC Magazine* with Jim Michel on how to force WinHelp to load your DLL.) This DLL could monitor topic jumps and the annotation menu item (via window subclassing) and know when to show annotations. (This is, in fact, a utility Jim Michel and I were thinking of writing quite some time ago, but never got around to.)

Chapter 6

Compression Algorithm and File Formats

The Microsoft compression algorithm is implemented in COMPRESS.EXE, EXPAND.EXE, and the LZEXPAND.DLL library. The most common use of these routines is for application install programs. Compressing the files on a set of installation disks results in the need for fewer disks. When a user is installing a Microsoft program, EXPAND.EXE is used to uncompress the files, writing the output to the hard drive. LZEXPAND.DLL is a collection of routines that allows a Windows program to expand files compressed with COMPRESS.EXE. This chapter covers version 2.0 of COMPRESS.EXE and EXPAND.EXE.

The Algorithm

The algorithm used by Microsoft is based on the LZ77 algorithm developed by Abraham Lempel and Jacob Ziv in a paper published in 1977. LZ77 is called a dictionary-based, sliding window algorithm. "Dictionary-based" means the compression is accomplished by replacing repeating character strings with a pointer to the first occurrence of the string, where the pointer is a pair of numbers indicating the offset and the length of the repeating string. For example, the string "abcdefabcd" would be compressed into "abcdef[0,4]", where "0" is the offset of the original string ("abcd") and

"4" is the length. This approach raises the following question: How many bytes should be reserved for storing the offset and length parameters? Assuming a maximum file size of 4Gb, you would have to use 4 bytes for the offset and 4 bytes for the length, for a total of 8 bytes, each time a repeating occurrence of a string is found. In practice, this would not produce a good compression ratio. If the average length of repeating strings in the input file is fewer than 8 bytes, the compressed file will be larger than the original file. This is considered bad compression.

This observation led to the concept of "sliding window" compression. Instead of searching the entire file (up to the character currently being read), use only the last *n* characters, where *n* is an integer denoting the size of the window. This is how it works: an array of the last *n* characters (from the input file) is maintained in memory (the "window"). When the next character in the input file is read, search the window for strings starting with the same character. After this step is done, the current character is added to the window, which means the character read *n* bytes ago is discarded. Only the previous *n* characters are in memory, so a window continuously slides through the input file. This approach has the advantage of decreasing the number of bytes required to store the `[offset, length]` pair of integers that denote a repeating string. The smaller the window, the fewer bytes needed to store the necessary information. The size of the window varies with the implementation. It should be small enough to reduce the size of the `[offset, length]` pair, but large enough that there is a high probability that a previous occurrence of the current string will be found in the window. It is important to find the longest match you can in order to improve the compression ratio.

Microsoft's Implementation

Microsoft uses a window of 4,096 bytes in their scheme. Such a window size would need 12 bits for an offset ($2^{12} = 4{,}096$). A compression code of 2 bytes (to minimize the number of bytes needed to code the offset and length) would leave 4 bits for the length. This means the maximum length of a repeating string that could be encoded in the compressed file is 15 bytes, but this can be improved. Because 2 bytes are needed for a compression code, repeating strings of fewer than 3 bytes can be stored uncompressed, because nothing would be gained by using a 2-byte compression code in place of a 2-byte string or, even worse, a 1-byte character. So a length of either zero, one, or two would never be found in the `length` field. You can take advantage of this by subtracting three from the length of the string when encoding it in the compressed file, thus allowing for a maximum length of 18 bytes. When the decompression routine encounters a compression code (`[offset, length]`), it adds three to obtain the actual length.

For reasons known only to Microsoft, the offset in a compression code is biased by 16 bytes. Before an offset is encoded in the compressed file, 16 is subtracted. The least significant bits of the offset are stored in the first byte of the 2-byte code; the most significant bits are stored in the upper 4 bits of the second byte. The length is stored in the lower 4 bits of the second byte. As an example, a compression code of offset = 36 and length = 15 would be processed as follows:

1. Subtract three from the length and 16 from the offset;
2. Offset = 20, so the 12 bits are 0000 0001 0100;
3. Length = 12, so the 4 bits are 1100;
4. Byte 1 is 0001 0100 (the lower 8 bits of the offset);
5. Byte 2 is 0000 1100 (remainder of the offset, length).

The Details

A problem with implementing this algorithm is distinguishing between uncompressed data and a 2-byte compression code. `COMPRESS.EXE` uses the following approach: data is stored in blocks of eight terms, where each term is either 1 byte of uncompressed data, or a 2-byte compression code. Each block is preceded by a 1-byte header, where each bit in the header is set to "1" if the corresponding term is uncompressed data, or "0" if it is a compression code (hence the eight-term size of the block). For example, a header byte of 0xC7 translates into 11000111 binary, which is read as follows: the first 3 bits are set, so the first three terms are single, uncompressed bytes and should be treated as literals; the following 3 bits are clear, so each of the next three terms are 2-byte compression codes; finally, the last two terms are literals. The data block immediately following the header, incidentally, would be 11 bytes (5 bytes of literals and 6 bytes of compression codes).

The following macro determines the compression code:

```
#define COMP_CODE(len, off) ((((len-3) & 0x0F ) « 8) + \\
      (((off - 0x10) & 0x0F00) « 4) + ((off - 0x10) & 0x00FF))
```

The following macros extract the length and offset from the compression code:

```
#define LENGTH(x) ((((x) & 0x0F)) + 3)
#define OFFSET(x1, x2) ((((((x2 & 0xF0) » 4) * 0x0100) + x1) & \\
                        0x0FFF)+ 0x0010)
```

Now for a concrete example. Say you have a file that only contains four words, each separated by a space. The words, in order, are "Plenty", "Plentiful", "Plenteous", and "lentic". When compressed with Microsoft's COMPRESS.EXE, the output file looks like Figure 6.1.

The first ten characters aren't related to the compression, so you can ignore them for the purposes of this discussion (see Table 6.1 for details on the header structure). The next field (a long integer) is the size of the data when decompressed: 0x21 bytes, or 33 characters. The data follows. The first block header is 0xBF, which is 10111111 binary. This means the first six terms in the block are 1-byte literals and can be written directly to the output file. The seventh term is a 2-byte compression code (0xEF 0xF3), and the eighth term is a 1-byte literal.

To fill in the string referenced by the seventh term in the first block, you have to decipher 0xEF 0xF3. The length of the replacement string is found in the lower 4 bits of 0xF3, which is three. Add three to this to obtain the true length, for a total of six. The offset (into the window) of the string is the upper 4 bits of 0xF3 (0x0F) and all of 0xEF, which is 0x0FEF. Add 16 to this, for a total of 0x0FFF, which is 4,095 in decimal. So the string starts with the last character in the window (index number 4,095) and is 6 bytes long. Are you wondering how the string could start with the 4,095th letter after only reading six characters? This occurs because the window must first be initialized with 0x20 (spaces). Because the words in the input file are separated by

Figure 6.1 *Sample output of* COMPRESS.EXE.

```
Offset       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
0x00000000: 53 5A 44 44 88 F0 27 33 41 00 21 00 00 00 BF 50  SZDD^.'3A.!....P
0x00000010: 6C 65 6E 74 79 EF F3 69 F7 66 75 6C EF F3 65 6F  lenty..i.ful..eo
0x00000020: 75 73 05 20 F8 F2 63                             us. ..c
```

Table 6.1 ***COMPHEADER record.***

Field Name	*Data Type*	*Comments*
Magic1	long	0x44445A53
Magic2	long	0x3327F088
Is41	char	0x41
FileFix	char	Last character in uncompressed file's name
DecompSize	long	Size of the uncompressed file

spaces and the first five characters of the first two words are the same, COMPRESS.EXE was able to start the replacement string with the last character in the window, a space. So far, by processing the first seven terms in the block, you have "Plenty Plent". The remainder of the file is processed in the same manner.

The Header

Each compressed file begins with a COMPHEADER (Table 6.1) structure. This structure has two magic number fields, a single-character constant, the last character of the name of the uncompressed file, and the size of the file when decompressed. The first three fields are always the same. The fourth field is used when the "-r" option is passed to COMPRESS.EXE, instructing it to store the last character of the uncompressed file's name, to be used when it is uncompressed.

Compressing

One of the programs included with this chapter is COMP.EXE. This DOS program will compress a file using Microsoft's implementation, so it can be decompressed using EXPAND.EXE. This section will detail how COMP.EXE works.

Once COMP.EXE has written the header described above, it starts compressing the data. Each character read is inserted into an array of 4,096 bytes. The array is initially filled with 0x20's (spaces). Once this array has been filled (i.e., 4,096 characters have been read from the input file), COMP.EXE returns to the beginning of the array and continues filling it in from there. As described in "Details" previously, data is written in blocks, where each block starts with a single character describing the remainder of the block. The remainder of the block is data from the input file (in compressed or uncompressed format). At this point, COMP.EXE begins reading the input file. When a new character is read, COMP.EXE searches the array containing the previous 4,095 characters. Each time a matching character is found, COMP.EXE continues to read to determine how many characters match. Once this step is completed, COMP.EXE moves through the window again, finding the next match, and the process is repeated. This method will produce the longest matching string in the previous 4,095 characters.

If no match is found or if the match is fewer than three characters, the data is written to the output file uncompressed. Also, the corresponding bit in the header character for the block is set. If a match of three or more characters is found, the offset of the matching string in the 4,096-byte window and the length of the match are used to create a 2-byte code. The code is written to the output file, and the corresponding bit in the block header is cleared. This process is repeated until the entire input file has been read.

Decompressing

Another program included with this chapter is DECOMP.EXE, which will decompress a file produced by either COMPRESS.EXE or COMP.EXE.

The DECOMP.EXE program decompresses a file by doing the opposite job of COMP.EXE. After it skips the header, it reads the 2-byte compression code that begins each block. If a bit is set, the corresponding character is written to the output file and inserted into the sliding window (array of 4,096 bytes). If a bit is not set, DECOMP.EXE gets the offset and size of the string from the corresponding 2-byte compression code in the block. Next it will retrieve that string from the window and write it to the output file.

If the sum of the offset and length (in a 2-byte compression code) is greater than the size of the window, it means the string wraps around from the end to the beginning of the array. After the byte at the end of the array is read, DECOMP.EXE jumps to the first byte in the array and continues reading.

The source code for COMP.EXE and DECOMP.EXE are shown in Listings 6.1 and 6.2 respectively. Each program has been tested extensively, and both seem to produce output compatible with the corresponding program from Microsoft, so any file compressed with either compression program can be decompressed using either decompression program. It is interesting to note that our version of the compression program will always achieve at least the same amount of compression as Microsoft's, but if the input file is larger than a few hundred bytes, COMP.EXE achieves a better compression ratio. It appears that Microsoft's implementation of the algorithm will only allow for a maximum length of 16 bytes when compressing a string, whereas our version allows for 18 bytes.

Where Do I Go from Here?

The code in Listing 6.1 could be used for a variety of purposes. Anyone interested in writing a quicker compressor or decompressor than those provided by LZEXPAND.DLL should study the code accompanying this chapter. If you're curious to see how one company implemented the LZ77 algorithm and want to attempt to improve the overall compression ratio, this code will serve as a good starting point.

Listing 6.1 COMP.C — Compression program compatible with EXPAND.EXE.

```
/*********************************************************************
 *
 * PROGRAM: COMP.C
 *
 * PURPOSE: Compress a file using something like Microsoft's derivative on LZ77
 * (i.e., it can be uncompressed using Microsoft's EXPAND.EXE).
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 6, Compression Algorithm and File Formats, from Undocumented Windows
 * File Formats, published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 *********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "decomp.h"

#define WINSIZE 4096
#define MAXLEN   18

#define COMP_CODE(x,y) ((((x-3) & 0x0F) << 8) + (((y - 0x10) & \
                       0x0F00) << 4) + ((y - 0x10) & 0x00FF))

#define LOBYTE(x) ((unsigned char)(x))
#define HIBYTE(x) ((unsigned char)(((unsigned short)(x) >> 8) & 0xFF))

#define DROP_INDEX(x) (x == 0) ? (WINSIZE - 1) : (x - 1)
#define ADD_INDEX(x) ((x + 1) == WINSIZE) ? 0 : (x + 1)

/* This is our Compression Window */
unsigned char Window[WINSIZE];

/**************************************************
 Set bit number "bit" in byte "byte"
**************************************************/
void BitSet(int bit, char *byte)
{

   short result = 1;

   /* make sure bit range is 0,..,7 */
   if (bit < 0) bit = 0;
   else if (bit > 7) bit = 7;

   while (bit--)
      result *= 2;
   *byte = result | *byte;

} /* BitSet - end */
```

Listing 6.1 (continued)

```
/************************************************
 InBetween - detects if lower <= target <= higher
************************************************/
int InBetween(int lower, int higher, int target)
{

 if(higher < lower)
     higher += WINSIZE;
 if((lower <= target) && (target <= higher))
     return 1;
 else
     return 0;

} /* InBetween - end */


/************************************************
 Force FlagByte and DataBytes to print out
************************************************/
void WriteFlagByte(FILE *outfile)
{

   int index = 0;

   if(FlagCount > 0) {
      DataBytes[DataCount] = '\0';
      fputc(FlagByte, outfile);
      for (; index < DataCount ; ++index)
         fprintf(outfile, "%c", DataBytes[index]);
      DataCount = FlagCount = 0;
      FlagByte = '\0';
      for (index = 0; index < 17; ++index)
         DataBytes[index] = ' ';
   }

} /* WriteFlagByte - end */


/************************************************
 Check if FlagByte is full and should be printed out
************************************************/
void CheckFlagByte(FILE *outfile)
{

   ++DataCount;
   if(++FlagCount == 8)
     WriteFlagByte( outfile );

} /* CheckFlagByte - end */
```

Listing 6.1 (continued)

```
/*************************************************
 Saves an uncompressed data byte
*************************************************/
void GetNextChar( int *CurrPos, FILE *infile)
{

   unsigned char ch;

   fread(&ch, sizeof(char), 1, infile);
   if (!feof(infile))
      Window[*CurrPos = ADD_INDEX(*CurrPos)] = ch;

} /* GetNextChar - end */


/*************************************************
 Unreads the last character read
*************************************************/
void UnreadChar(unsigned char ch, FILE *infile, int *CurrPos,
                unsigned char ch2)
{

   ungetc( ch, infile);
   Window[*CurrPos] = ch2;
   *CurrPos = DROP_INDEX(*CurrPos);

} /* UnreadChar - end */


/*************************************************
 Saves an uncompressed data byte
*************************************************/
void SaveUncompByte(unsigned char ch, FILE *outfile)
{

   BitSet(FlagCount, &FlagByte);
   DataBytes[DataCount] = ch;
   CheckFlagByte( outfile );

} /* SaveUncompByte - end */
```

Listing 6.1 (continued)

```
/**********************************************
 Compresses the data using Microsoft's LZ77
 derivative (Zeck).
**********************************************/
void Compress(FILE *infile, FILE *outfile)
{

   int count=0, shifter=0, CurrPos=0;
   int SavePos=0, iCompCode = 0, offset = 0;
   int newPos = 0;
   unsigned char ch;
   int bestcount = 0, bestoffset = 0;
   char oldchars[3];

   FlagCount = 0;
   DataCount = 0;
   FlagByte = '\0';

   for (count = 0; count < WINSIZE; count ++)
      Window[count] = ' ';

   rewind( infile );

   /* Go through input file until you're done */
   fread(&ch, sizeof(char), 1, infile);
   Window[CurrPos] = ch;
   while (!feof(infile)) {

      /* if less than 3 chars from end, just write out remainder */
      if((InfileSize - ftell(infile)) < 2) {
         SaveUncompByte( Window[CurrPos], outfile);
         GetNextChar( &CurrPos, infile);
         continue;
      }

      /* Find previous occurrence of character in window */
      for (count = 1, shifter = DROP_INDEX(CurrPos);
         (Window[shifter] != Window[CurrPos]) && (count < WINSIZE);
         ++count, shifter = DROP_INDEX(shifter)) {}

      /* check if char is unique so far in input file */
      if(count == WINSIZE) {
         SaveUncompByte( Window[CurrPos], outfile);
         GetNextChar( &CurrPos, infile);
         continue;
      }
```

Listing 6.1 (continued)

```
      else {

         /* find out how many characters match */
         SavePos = CurrPos;
         oldchars[2] = oldchars[0] = Window[ADD_INDEX(CurrPos)];
         GetNextChar( &CurrPos, infile);
         for(count = 1, offset=shifter, shifter = ADD_INDEX(shifter);
            (!feof(infile)) && (Window[shifter] == Window[CurrPos]) &&
            (count < MAXLEN);) {
            ++count;
            if(count == 2)
               oldchars[1] = Window[ADD_INDEX(CurrPos)];
            oldchars[2] = Window[ADD_INDEX(CurrPos)];
            GetNextChar( &CurrPos, infile);
            shifter = ADD_INDEX(shifter);
         }

         /* Since this is the first match, save it as the best so far */
         bestcount = count;
         bestoffset = offset;

         if(((Window[shifter] != Window[CurrPos]) || (count == MAXLEN)) &&
              (!feof(infile)))
            UnreadChar( Window[CurrPos], infile, &CurrPos, oldchars[2]);

         /* Now find the best match for the string in the window */
         shifter = DROP_INDEX( offset );
         while((shifter != CurrPos) && (bestcount < MAXLEN) &&
            (!InBetween( SavePos, CurrPos, shifter))) {

            for( ; (shifter != CurrPos) &&
                 (Window[shifter] != Window[SavePos]);
               shifter = DROP_INDEX(shifter)) {}
            if(shifter == CurrPos)
               continue;
            for(count = 0, offset = shifter, newPos = SavePos;
                 (!feof(infile)) && (Window[shifter] == Window[newPos]) &&
                 (count < MAXLEN); ++count, newPos = ADD_INDEX(newPos)) {
               if (count >= (bestcount - 1)) {
                  if(count == 1)
                     oldchars[1] = Window[ADD_INDEX(CurrPos)];
                  oldchars[2] = Window[ADD_INDEX(CurrPos)];
                  GetNextChar( &CurrPos, infile );
               }
               shifter = ADD_INDEX(shifter);
            }
```

Listing 6.1 (continued)

```
            if(((count >= MAXLEN) || ((Window[shifter] != Window[newPos]) &&
                 (count >= bestcount))) && (!feof(infile)))
               UnreadChar( Window[CurrPos], infile, &CurrPos, oldchars[2]);

            if(count > bestcount) {
               bestcount = count;
               bestoffset = offset;
            }
            shifter = DROP_INDEX( offset );

         } /* while(shifter != CurrPos) */

         if(!feof(infile))
            GetNextChar( &CurrPos, infile );

         count = bestcount;
         offset = bestoffset;

         /* if count < 3, then not enough chars to compress */
         if (count < 3) {
            SaveUncompByte( Window[SavePos], outfile);
            fseek(infile, ftell(infile) - count, 0);
            Window[SavePos = ADD_INDEX(SavePos)] = oldchars[0];
            CurrPos = DROP_INDEX(CurrPos);
            if (count == 2) {
               Window[ADD_INDEX(SavePos)] = oldchars[1];
               CurrPos = DROP_INDEX(CurrPos);
            }

         }
         else {
            iCompCode = COMP_CODE(count, offset);
            DataBytes[DataCount]   = LOBYTE(iCompCode);
            DataBytes[++DataCount] = HIBYTE(iCompCode);
            CheckFlagByte( outfile );

            if (!feof(infile))
                UnreadChar( Window[CurrPos], infile, &CurrPos, oldchars[2]);
         }

         if((!feof(infile)) && (count <= MAXLEN))
            GetNextChar( &CurrPos, infile);

      }

   } /* while - end */

   WriteFlagByte( outfile );

} /* Compress - end */
```

Listing 6.1 (continued)

```
/*************************************************
  Write the header that exists at the beginning of
  every file compressed using MS's Zeck compression
*************************************************/
void WriteHeader(FILE *infile, FILE *outfile)
{

   COMPHEADER CompHeader;

   CompHeader.Magic1 = MAGIC1;
   CompHeader.Magic2 = MAGIC2;
   CompHeader.Is41 = 0x41;
   CompHeader.FileFix = '\0';  /* This stores the original last */
                               /* char. of the input filename   */

   fseek( infile, 0L, 2);
   CompHeader.DecompSize = InfileSize = ftell(infile);
   rewind( infile );

   rewind( outfile);

   fwrite(&CompHeader, sizeof(CompHeader), 1, outfile);

   Compress(infile, outfile);

} /* WriteHeader - end */


/*************************************************
  Show usage.
*************************************************/
void Usage( void )
{

   printf("Usage:\n");
   printf(" COMP file1.ext file2.ext\n\n");
   printf("   file1.ext - Name of uncompressed file\n");
   printf("   file2.ext - Name of compressed file\n\n");

} /* Usage - end */


/*************************************************
  Open the input and output files, and call routine
  to compress the data.
*************************************************/
int main(int argc, char *argv[])
{

   char filename[128];
   FILE *infile, *outfile;
```

Listing 6.1 (continued)

```
    if (argc != 3) {
       Usage();
       return EXIT_FAILURE;
    }

    strcpy(filename, argv[1]);
    if ((infile = fopen(filename, "rb")) == NULL) {
       printf("%s does not exist\n", filename);
       return(EXIT_FAILURE);
    }

    strcpy(filename, argv[2]);
    if ((outfile=fopen(filename, "wb+")) == NULL) {
       printf("Error opening destination file\n");
       return(EXIT_FAILURE);
    }

    WriteHeader(infile, outfile);
    fclose(infile);
    fclose(outfile);

    return(EXIT_SUCCESS);

} /* main - end */

/* comp.c - end */
```

Listing 6.2 DECOMP.C *— Decompression program.*

```
/*********************************************************************
 *
 * PROGRAM: DECOMP.C
 *
 * PURPOSE: Decompresses a file compressed with Microsoft's COMPRESS.EXE utility.
 * Functionaly equivalent to EXPAND.EXE
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 6, Compression Algorithm and File Formats, from Undocumented Windows
 * File Formats, published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 *********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "decomp.h"

#define WINSIZE  4096
#define LENGTH(x) ((((x) & 0x0F)) + 3)
#define OFFSET(x1, x2) ((((((x2 & 0xF0) >> 4) *
               0x0100) + x1) & 0x0FFF) + 0x0010)
#define FAKE2REAL_POS(x) ((x) & (WINSIZE - 1))
#define BITSET(byte, bit)  (((byte) & (1<<bit)) > 0)

/* This is our Compression Window */
char Window[WINSIZE];

/**************************************************
  Decides how many bytes to read, depending on the
  number of bits set in the Bitmap
**************************************************/

int BytesToRead(unsigned char BitMap)
{
  int TempSum, counter, c;

  TempSum = 8;
  for (counter = 0; counter < 8; counter ++)
  {
    c = BITSET(BitMap, counter);
    TempSum += (1 - BITSET(BitMap, counter));
  }

  return TempSum;
}
```

Listing 6.2 (continued)

```
/************************************************************
  Decompresses the data using Microsoft's LZ77 derivative.
************************************************************/

void Decompress(FILE *infile, FILE *outfile, long CompSize)
{
  unsigned char BitMap, byte1, byte2;
  int Length, counter, NumToRead;
  long Offset, CurrPos=0;

  for (counter = 0; counter < WINSIZE; counter ++)
    Window[counter] = ' ';

  /* Go through until we're done */
  while (CurrPos < CompSize)
  {
    /* Get BitMap and data following it */
    BitMap = fgetc(infile);
    if (feof(infile)) return;
    NumToRead = BytesToRead(BitMap);

    /* Go through and decode data */
    for (counter = 0; counter < 8; counter++)
    {
      /* It's a code, so decode it and copy the data */
      if (!BITSET(BitMap, counter))
      {
        byte1 = fgetc(infile);
        if (feof(infile)) return;
        byte2 = fgetc(infile);
        Length = LENGTH(byte2);
        Offset = OFFSET(byte1, byte2);

        /* Copy data from 'window' */
        while (Length)
        {
          byte1 = Window[FAKE2REAL_POS(Offset)];
          Window[FAKE2REAL_POS(CurrPos)] = byte1;
          fputc(byte1, outfile);
          CurrPos++;
          Offset++;
          Length--;
        }
      }/* if */
      else
      {
        byte1 = fgetc(infile);
        Window[FAKE2REAL_POS(CurrPos)] = byte1;
        fputc(byte1, outfile);
        CurrPos++;
      }

      if (feof(infile)) return;

    } /* for */

  }/* while  */
}
```

Listing 6.2 (continued)

```
/**************************************************
  Read the file header
**************************************************/

void ReadHeader(FILE *infile, FILE *outfile)
{
  COMPHEADER CompHeader;
  long       CompSize;

  fseek(infile, 0, SEEK_END);
  CompSize = ftell(infile);
  fseek(infile, 0, SEEK_SET);
  fread(&CompHeader, sizeof(CompHeader), 1, infile);
  if ((CompHeader.Magic1 != MAGIC1) || (CompHeader.Magic2 != MAGIC2))   {
    printf("Fatal Error:\n");
    printf("  Not a valid Compressed file file!\n");
    return;
  }

  Decompress(infile, outfile, CompHeader.DecompSize);
}

/**************************************************
  Show usage.
**************************************************/

void Usage()
{
  printf("Usage:\n");
  printf(" DECOMP file1.ext file2.ext\n\n");
  printf("    file1.ext - Name of compressed file\n");
  printf("    file2.ext - Name of decompressed file\n\n");
}
```

Listing 6.2 (continued)

```
/*************************************************
  Open the file and dump it.
*************************************************/

int main(int argc, char *argv[])
{
  char filename[128];
  FILE *infile, *outfile;

  if (argc < 3)
  {
    Usage();
    return EXIT_FAILURE;
  }

  strcpy(filename, argv[1]);
  if ((infile = fopen(filename, "rb")) == NULL)
  {
    printf("%s does not exist!", filename);
    return EXIT_FAILURE;
  }

  strcpy(filename, argv[2]);
  if ((outfile=fopen(filename, "wb")) == NULL)
  {
    printf("Error opening destination file!");
    return EXIT_FAILURE;
  }

  ReadHeader(infile, outfile);
  fclose(infile);
  fclose(outfile);

  return EXIT_SUCCESS;
}
```

Chapter 7

Resource (.RES) File Format

In this chapter, I'll take a look a look at the format of .RES files. Our thanks go to Alex Fedorov and Dmitry Rogatkin for their work on this topic originally published in Andrew Schulman's "Undocumented Corner," *Dr. Dobb's Journal,* August 1993. We'd also like to thank Jonathan Erickson, *DDJ* editor, for allowing us to use the information in that article. After describing the file format, I'll present a program for decompiling .RES files into .RC files. This chapter references the resource compiler (RC.EXE) supplied with the Windows 3.1 SDK.

An .RC file contains information on the resources used by a Windows executable, such as bitmaps, buttons, and dialog boxes. These files must be compiled by Microsoft's resource compiler (RC.EXE) before they can be added to the executable. The output of this compiler is a .RES file. The format of resources in the executable has been documented by Microsoft, but never the format of the .RES file. If you have a copy of the *Microsoft Windows 3.1 Programmer's Reference,* Volume 4 *(Resources),* you already have this documentation. Chapter 7 in that reference is titled "Resource Formats Within Executable Files", and covers the format of each resource type. Although this information was helpful in writing a program described later in this chapter, it doesn't mention the .RES file format.

As it turns out, each type of resource is stored in a .RES file in the same format as it would appear in a Windows executable. The difference is the existence of a descriptive header before each resource. A .RES file is nothing more than a collection of pairs of resources and their respective headers. You can confirm this by writing a simple

.RC file and compiling it with RC.EXE. Take a look at the .RES file; strings from the .RC file are practically jumping off the page. What we'll do here is fill in the blanks and describe the format of the header.

The Format

Resource headers are type independent; that is, regardless of the type of resource, the header always has the same format. The first field in the header is the name or type of the resource. If the first character is 0xFF, it is immediately followed by a number (a WORD) which maps to a specific resource type (either predefined or defined by the user). A quick look in WINDOWS.H and VER.H (the header file required to include version information in a resource file) produces the list of predefined resource types in Table 7.1.

Any other number after 0xFF indicates a user-defined resource.

If the first character is not 0xFF, it is a null-terminated string naming the resource type (as defined by the user). This is different than a name given to an instance of a resource. If you're unfamiliar with defining your own type of resource, read *Programmer's Reference,* Volume 4, pp. 212-213.

Table 7.1 Numeric values for resource types.

Resource Type	*Identification Number (from WINDOWS.H)*
Cursor	RT_CURSOR (1)
Bitmap	RT_BITMAP (2)
Icon	RT_ICON (3)
Menu	RT_MENU (4)
Dialog box	RT_DIALOG (5)
String table	RT_STRING (6)
Font directory	RT_FONTDIR (7)
Font	RT_FONT (8)
Accelerator	RT_ACCELERATOR (9)
RCDATA (user defined)	RT_RCDATA (10)
Group cursor	RT_GROUP_CURSOR (12)
Group icon	RT_GROUP_ICON (14)
Name table (obsolete with v3.1)	15
Version information	16

The next field in the header is a number or name identifying an instance of a resource. Similar to the format of the resource type field, if the first byte is 0xFF, it is followed by a numeric value (also a WORD). Otherwise, the field is a null-terminated string naming the resource.

This is followed by a WORD value storing memory flags. A memory flag describes how the resource should be loaded, discarded, and moved around in memory. Because each flag (e.g., MOVEABLE, DISCARDABLE) is a WORD and doesn't overlap (at the bit level) with the other flags, they can be ORed together to produce a single WORD value. This ORed value of the individual memory flags is what appears in this field. These values are summarized in Table 7.2.

If the resource in question is a cursor or an icon, the table changes slightly. The value for "Discardable" is 0x20, and "Pure" appears to have no meaning in relation to these two types of resources.

The next (and final) field in the header is a DWORD containing the total length of the resource data, not including the header. This is followed by the resource data, whose format is documented in *Programmer's Reference*, Volume 4.

Look at the following example. Suppose you come across a resource header in a .RES file that consists of the following characters (in hexadecimal):

```
FF 05 00 46 4F 4F 42 41 52 00 30 10 9A 00 00 00
```

Because the first character is 0xFF, this resource type is identified by the number immediately following it, which, in this case, is 0x05. Using Table7.1, you know this is the header for a dialog box. The next field is the name of the dialog box, and since it does not start with 0xFF, it is a null-terminated string. Converting the hex string 46 4F 4F 42 41 52 to alphabetic letters, you get "FOOBAR". Next is the memory flag for this dialog, which is 0x1030. Using Table 7.2, you know this dialog box is discardable, pure, moveable, and loaded on call (this last aspect can be deduced because 0x1030 does not contain 0x40). Finally, the length of the dialog box resource data is 0x0000009A.

Table 7.2 Memory flags ORed values.

Value	*Meaning*
0x1000	Discardable
0x40	Preload (otherwise, load on call)
0x20	Pure
0x10	Moveable (otherwise, fixed)

One shortcoming of this file format is the absence of a signature at the start of the .RES file. This prevents any utility that reads a .RES file as input from knowing if the input file is indeed a .RES file. When we were testing the program that decompiles .RES files into .RC files (RES2RC), sometimes we accidentally gave it an .RC file as input. The results were unpredictable, but demonstrated how sensitive utility programs must be to non-.RES files. When Microsoft wanted to distinguish a 32-bit .RES file from a 16-bit .RES file, they evaded this shortcoming by starting all 32-bit .RES files with an entry illegal in a 16-bit .RES file: 0. Because this value is not 0xFF, it must be a character string naming the resource type, but because it begins with zero, the length of the string must be zero, which is illegal. This is documented in the RESFMT.TXT file on Microsoft's Win32 CD-ROM.

The Program

This chapter describes a DOS program called RES2RC (Listing 7.1, see page 140) which decompiles an .RES file into an .RC file. It requires two arguments: an input filename (a .RES file) and an output filename. Because each resource has its own format, we had to write code to handle each type of resource. As you may guess this made for a lot of code — roughly 3,200 lines of C. This section will detail how the program works.

Before processing any resources in the input file, the program will verify that it is not a Win32 resource file by checking the first byte. If that test succeeds, the program rewinds the input file and enters a small loop to read the first byte of each resource header. If it is 0xFF, it is a resource listed in Table 7.1, and must be processed accordingly; otherwise, it is a user-defined resource, in which case the program saves the data to a uniquely named external file of the format UR###.USR. An entry is added to the output file referencing this file.

Resources with a header starting with 0xFF are those predefined by Windows or the user and include resources typically used in a Windows program (e.g., dialog boxes, cursors). The bulk of the code was written to handle these types of resources for which the program reads the number (integer) following the 0xFF, and compares it against the list of types defined in WINDOWS.H and VER.H (the latter exclusively for a version resource). Once the program determines the type, it jumps to the code written for that resource.

Nearly everything described so far is handled in under 200 lines of code. The remainder of the code is needed to handle each resource type. The rest of this section will describe the format of each of these types. I will also point out any discrepancies between the format and Microsoft's documentation.

All of the tables in the remainder of this chapter were either defined in VER H or WINDOWS.H or described in the *Microsoft Windows 3.1 Programmer's Reference.*

Cursors and Group Cursors

Each cursor file (.CUR) included in an .RC file starts with a header, called CURSORHEADER (Table 7.3). The only interesting field in the header is cdCount, which is the number of cursors in the .CUR file. The header is followed (in the .RES file) by a CURSORDIRENTRY structure for each cursor (Table 7.4). The number of these structures in the file will be the same as cdCount. After the array of these structures, the .CUR file contains the data for each of its cursors. It is important to note that the CURSORDIRENTRY structure in the .RES file is different than the structure of the same name defined in the *Microsoft Windows 3.1 Programmer's Reference*, Volume 4, Chapter 1; rather, it matches the format described in Chapter 7 of the same Reference. The structure used in Table 7.4 is the one used in RES2RC.

In the .RES file, the data is arranged somewhat differently. The data for each of the cursors (excluding the headers) appears first. In the .RES file, each cursor is listed separately and successively. These will be followed by a group cursor resource, which contains the data described previously.

When RES2RC encounters cursor data in the .RES file, it does a bit of jumping about, so it's important to describe what it is doing. RES2RC skips over the cursor resources when they first appear, since the header has to be written first, and that doesn't appear until later. The next resource to appear should be a group cursor. At this point, RES2RC will generate a uniquely named filename of the form CU###.CUR,

Table 7.3 CURSORHEADER record.

Field Name	*Data Type*	*Comments*
cdReserved	WORD	Must be zero
cdType	WORD	Resource type; must be 2
cdCount	WORD	Number of cursors in the file

Table 7.4 CURSORDIRENTRY record.

Field Name	*Data Type*	*Comments*
wWidth	WORD	Width, in pixels
wHeight	WORD	Height, in pixels
wPlanes	WORD	Number of color planes; set to 1
wBitCount	WORD	Number of bits in cursor; set to 1
dwBytesInRes	DWORD	Size of resource in bytes
wImageOffset	WORD	Image number

which will hold the cursor data, including headers. It will then add an entry to the output file for the cursor data, using this filename. Next, it reads the header, which starts the group cursor resource and writes it to the data file. It then makes two passes through the CURSORDIRENTRY fields. The first time, it reads each occurrence of the structure so it can write the data to the data file. However, it is stored in the data file in a slightly different format, which is described in Table 7.5.

Next, RES2RC runs through the array again, this time using the wImageOffset field to find the cursor data in the .RES file. The wImageOffset field isn't the offset into the .RES file of the image; each image in the .RES file is numbered (e.g., 1, 2, 3) in the first field in its respective header, and this is the number RES2RC searches to find a match against the wImageOffset field. Once it finds a match, it writes the data for that particular image to the data file.

The Microsoft documentation on cursors (in the .RC file) says the default is LOADONCALL, MOVEABLE, and DISCARDABLE. This does not appear to be the case. It seems DISCARDABLE must be specified explicitly for the resource compiler to mark it as such.

Table 7.5 CURSORRESENTRY record.

Field Name	*Data Type*	*Comments*
bWidth	BYTE	Width, in pixels
bHeight	BYTE	Height, in pixels
bColorCount	BYTE	Must be zero
bReserved	BYTE	Must be zero
wXHotSpot	WORD	X-coordinate of hotspot, in pixels
wYHotSpot	WORD	Y-coordinate of hotspot, in pixels
dwBytesInRes	DWORD	Size of the resource, in bytes
dwImageOffset	DWORD	Offset to image

Table 7.6 BITMAPFILEHEADER record.

Field Name	*Data Type*	*Comments*
bfType	UINT	0x4D42
bfSize	DWORD	Size of the bitmap
bfReserved1	UINT	Zero
bfReserved2	UINT	Zero
bfOffBits	DWORD	0x76

Bitmaps

Bitmap files (.BMP) are composed of two distinct parts: header and data. The header is defined in WINDOWS.H and described in Table 7.6.

Most of the fields in the header are constants; the exception is bfSize, which is computed as the sum of the size of the header and the size of the data. In the .RES file, only the data is included because the header can be computed. When RES2RC finds a bitmap resource, it generates a unique filename of the form BM###.BMP and writes a reference to this filename to the output file. It then calculates bfSize, writes the header, and then writes the data in the bitmap resource (in the .RES file).

Icons and Group Icons

Each icon file (.ICO) included in an .RC file starts with a header, called ICONHEADER (Table 7.7). The only interesting field in the header is idCount, which is the number of icons in the .ICO file. The header is followed (in the .RES file) by an ICONDIR-ENTRY structure (Table 7.8) for each icon (the number of these structures in the file will be the same as idCount). After the array of these structures, the .ICO file contains the data for each of its icons.

In the .RES file, the data is arranged somewhat differently. The data for each of the icons (excluding the headers) appears first. In the .RES file, each icon in the .ICO file

Table 7.7 ICONHEADER record.

Field Name	*Data Type*	*Comments*
idReserved	WORD	Must be zero
idType	WORD	Resource type; set to 1
idCount	WORD	Number of entries in directory

Table 7.8 ICONDIRENTRY record.

Field Name	*Data Type*	*Comments*
bWidth	BYTE	Width, in pixels (16, 32, or 64)
bHeight	BYTE	Height, in pixels (16, 32, or 64)
bColorCount	BYTE	Number of colors in icon (2, 8, or 16)
bReserved	BYTE	Must be zero
wPlanes	WORD	Number of color planes
wBitCount	WORD	Number of bits in the icon bitmap
dwBytesInRes	DWORD	Size of the resource, in bytes
wImageOffset	WORD	Image number

has a separate resource, each of which is listed successively. These will be followed by a group icon resource, which contains the data described previously.

When RES2RC encounters icon data in the .RES file, it does a bit of jumping about, so it's important that I describe what it is doing. RES2RC skips over the icon resources when they first appear, because the header has to be written first, and it doesn't appear until later. The next resource to appear should be a group icon. At this point, RES2RC generates a uniquely named filename of the form IC###.ICO, which holds the icon data, including headers. It then adds an entry to the output file for the icon data, using this filename. Next, it reads the header that starts the group icon resource and writes it to the data file. It then makes two passes through the ICONDIRENTRY fields. The first time, it reads each occurrence of the structure so it can write the data to the data file. However, it is stored in the data file in a slightly different format, which is described in Table 7.9.

Next, RES2RC runs through the array again, this time using the wImageOffset field to find the icon data in the .RES file. The wImageOffset field isn't the offset into the .RES file of the image; each image in the .RES file is numbered (e.g., 1, 2, 3) in the first field in its respective header, and this is the number RES2RC searches to find a match against the wImageOffset field. Once it finds a match, it writes the data for that particular image to the data file.

The Microsoft documentation on icons (in the .RC file) says the default is LOADONCALL, MOVEABLE, and DISCARDABLE. This does not appear to be the case. It seems DISCARDABLE must be specified explicitly for the resource compiler to mark it as such.

Table 7.9 ICONRESENTRY record.

Field Name	*Data Type*	*Comments*
bWidth	BYTE	Width, in pixels (16, 32, or 64)
bHeight	BYTE	Height, in pixels (16, 32, or 64)
bColorCount	BYTE	No. of colors (2, 8, or 16)
bReserved	BYTE	Must be zero
wPlanes	WORD	Number of color planes
wBitCount	WORD	Number of bits in icon bitmap
dwBytesInRes	DWORD	Size of the resource, in bytes
dwImageOffset	DWORD	Offset to start of icon data

Table 7.10 MenuHeader record.

Field Name	*Data Type*	*Comments*
wVersion	WORD	For Windows 3.0 and later, zero
wReserved	WORD	Must be zero

Menus

Menu resources begin with a header, followed by the data for the menu. The format of the header is described in Table 7.10.

The remainder of the header contains data describing the menu. The data is really a series of small blocks describing each menu item. The first field in the block is a WORD, containing flags that describe how the menu item is displayed. The possible values for this field depend on whether the item is a popup or normal menu item, but they have several values in common. Valid flag values for both types are MF_GRAYED, MF_DISABLED, MF_CHECKED, MF_MENUBARBREAK, MF_MENUBREAK, and MF_END (any combination). In addition to these possible values, popup items must have a flag value of MF_POPUP. Another possible value for normal menu items is the undocumented MF_HELP attribute. If the menu item has the MF_POPUP attribute, it is immediately followed by a null-terminated string containing the text of the menu item. Otherwise, it is followed by a WORD containing the menu ID and the text of the item.

The code in RES2RC for processing menu resources is fairly straightforward. For each menu item, it checks the possible values for the flag field and prints the corresponding attribute. Whenever it encounters a popup menu item, it starts a new submenu. When an item has an attribute of MF_END, it prints a "}" to end the block. This continues until it has processed the entire resource.

Dialog Boxes

Dialog box resources are stored in a .RES file beginning with a dialog box header, which is then followed by the data for each control in the dialog box. The header is described in Table 7.11.

Table 7.11 DIALOGHEADER record.

Field Name	*Data Type*	*Comments*
lStyle	DWORD	Style of dialog box
bNumberofItems	BYTE	Number of controls
x	WORD	X-coordinate of dialog box
y	WORD	Y-coordinate of dialog box
width	WORD	Width of dialog box
height	WORD	Height of dialog box
szMenuName	char[]	Name of any associated menu
szClassName	char[]	Name of class of dialog box
szCaption	char[]	Caption of dialog box
wPointSize	WORD	Only if lStyle has DS_SETFONT
szFaceName	char[]	Only if lStyle has DS_SETFONT

The last two fields only exist if the lStyle field has a value of DS_SETFONT. This structure is declared in RESTYPES.H, but in a slightly different format. Everything after the height field is dropped, because of the way variable strings are processed. RES2RC does not store those strings in memory; it will read the string one character at a time and immediately write it to the output file, repeating this process until it hits the null character; this is done partly because most strings in a compiled resource file have no length restrictions.

After reading the resource header for the dialog box, RES2RC reads the DIALOG-HEADER structure and writes the appropriate strings to the output file.

The resource compiler stores information for each control in the format described in Table 7.12.

As declared in RESTYPES.H, this structure contains only the first six fields, since the remainder can consist of variable-length strings. If the class field contains 0x80, it is a predefined control type: button, edit control, static control, listbox, scroll bar, or combo box; otherwise, it is the first character in a string naming the resource type. RES2RC will process the CONTROLDATA structure differently for each control type. The bulk of this code consists of checking for each possible value of the lStyle field (which differs with the control type) and writing the appropriate string to the output file. For example, if the control is a combo box, the program will check lStyle for any of 14 possible values, such as CBS_HASSTRINGS and WS_TABSTOP.

If the class field does not contain 0x80, the program does not process the lStyle field, because it has no prior knowledge of the resource, such as how to interpret lStyle.

Table 7.12 CONTROLDATA record.

Field Name	*Data Type*	*Comments*
x	WORD	X-coordinate of control
y	WORD	Y-coordinate of control
width	WORD	Width of control
height	WORD	Height of control
id	WORD	Numeric ID of control
lStyle	DWORD	Style of control
class/szClass (union)	BYTE/char[]	Type/name of control
szText	char[]	Text in the control

String Tables

String tables allow a programmer to group strings used by a program into a single area in the resource file. The documentation says each table is composed of 16 strings. However, something it doesn't make clear is that the resource compiler will read any string table defined in a resource file and group them into sets of 16, regardless of how they were originally declared. Each string is stored in a compiled resource file as a separate resource, but they can't be treated as such. RES2RC maintains a record of the number of consecutive string table entries it has processed, starting a new table in the output file after every 16 strings. If fewer than 16 strings are specified in the resource file, the compiler fills out the table with zero-length strings. These zero-length strings are skipped by RES2RC. After the standard resource header, string resources contain two fields: length of the string and the string itself. Because the string can contain null characters, the length is required.

The code in RES2RC to handle string resources is very short — roughly 50 lines. You should keep in mind that a table produced by RES2RC may not appear as it was originally defined, but it is functionally the same. For example, if you define two string tables, each containing three strings, the compiler will merge them into one table of six strings. RES2RC will use this format when writing it to the output file. If you compile the RES2RC output, the resulting .RES file will be identical to the original .RES file you used as input to RES2RC.

Fonts and Font Directories

Fonts and font directories are closely related resources. For each font file (.FON) referenced in your .RC file, the resource compiler will add a font resource, containing all of the data in the original .FON file. For each unique font directory, the compiler will add a font directory resource. Related fonts are grouped into font directories; a directory contains a table of metrics for each of its fonts. Because all the information the resource compiler needs is contained within the .FON file, RES2RC skips all font directory resources it finds in the input file. When it finds a font resource, it will save the data to an external and uniquely named file of the format FO###.FON. A reference to this file is added to the output file. This .FON file will be identical to the original font file.

Accelerator Tables

Accelerator resources are very straightforward to process. All of the accelerators grouped into a single table in the resource file are grouped likewise in the .RES file as a single resource. The resource header contains the name of the table. The resource data itself contains a sequence of small structure(s) for each accelerator in the table. The format of this structure is described in Table 7.13.

The fFlags field describes whether the accelerator uses any combination of the Alt, Shift, and Ctrl keys, and whether the top-level menu item is highlighted when the key is pressed. Each of these values is unique (at the bit level), and are ORed together to produce the final value. The Microsoft documentation describes the possible values of fFlags, but they left one out: if it contains 0x01, the accelerator is a virtual key. This is transparent to the user, but is important to RES2RC. The wEvent field contains the key used in the accelerator, and wID is the numeric value passed to the window procedure when the accelerator is pressed.

The code in RES2RC that handles these tables isn't complicated at all; it should be easy to follow. The longest part of the code is the processing for the wEvent field; we attempted to make the output as easy to read as possible. If the field is a virtual key, a string describing the key is printed (e.g., "VK_TAB", "VK_HOME"). Otherwise, the character is ASCII and printed as a letter. If it is a control character, it is printed as a caret ("^") followed by the letter, all within quotes. After that, the wId field is printed, followed by fFlags processing. Each possible value is checked for, and if TRUE, the appropriate string is written to the output file.

RCDATA

An RCDATA resource allows a raw data resource to be included in a resource file, and ultimately, the executable. This type of resource can include binary data. RES2RC processes this type of resource by running through the data, one character at a time,

Table 7.13 AccelTableEntry record.

Field Name	*Data Type*	*Comments*
fFlags	BYTE	Shows Alt, Ctrl, Shift, highlighted, virtual
wEvent	WORD	Key used in the accelerator
wId	WORD	Value passed to window

and printing it to the output file. We attempted to make the data as readable as possible; the characters are grouped into lines of 60 characters each. If a character has an ASCII value between 32 and 126 (inclusive), it is printed as a character. Otherwise, it is printed in octal, with a leading backslash ("\"), so the resource compiler will interpret it as octal. As with the other resources, RES2RC knows the length of the data from the resource header, the last field of which is the length of the resource data.

Name Tables

Name tables were once used under Windows 3.0, but are now obsolete. If RES2RC comes across a name table, it will add a three-line comment to the output file saying one was found, but the table is otherwise ignored.

Version Information

The VersionInfo resource allows a programmer to specify information such as the version number and intended operating system for a program. This information is used by the File Installation library functions (VER.DLL). This resource is stored in a compiled resource file as a sequence of blocks. Each block has the same format and is described in Table 7.14. The abValue field is always aligned on 32-bit boundaries.

The data is stored starting with a root block, which contains the fixed information (specified immediately after VERSIONINFO in the .RC file), such as FILEVERSION and FILEOS. The documentation contains some discrepancies about the FILEOS field. It says two possible values for this field are VOS_WINDOWS16 and VOS_WINDOWS32. We couldn't find references to these in VER.H; furthermore, we found definitions for VOS_OS216, VOS_OS232, VOS_OS216_PM16, and VOS_OS232_PM32. We assume the first two are for 16-bit and 32-bit OS/2 programs, respectively; the last two are for 16-bit and 32-bit OS/2 Presentation Manager programs, respectively. All of the fields in the

Table 7.14 VERSIONINFO block header record.

Field Name	*Data Type*	*Comments*
cbBlock	WORD	Complete size of the block
cbValue	WORD	Size of the abValue field
szKey[]	char	Name of the block
abValue	BYTE	Either an array of WORDs or a string

root block are processed by RES2RC by reading the abValue field into the VS_FIXEDFILEINFO structure, which is defined in VER.H. The fields are described in Table 7.15. The program then checks each field against the list of possible values defined in VER.H, and writes the appropriate string to the output file.

In an .RC file, a VersionInfo block can have two types of information blocks: string and variable. String information blocks allow a programmer to specify different string information (e.g., "CompanyName", "InternalName") in different languages. For example, you could set up a block of this type of information, label the block as U.S. English — 7-bit ASCII, and Windows would use that information when installing onto the appropriate set-up. These blocks are stored in a .RES file using the same format as the root block, with an szKey field starting with "S", so RES2RC runs through the abValue field (for the next cbBlock number of bytes) and processes each nested block of resource data.

A similar algorithm is used for variable information blocks. When RES2RC encounters a block with an szKey field starting with "V", the block is treated as a variable block. These are different than string blocks, in that variable blocks cannot be nested, and each line defines the languages and character sets supported by the executable. Each variable block is immediately followed by one or more "Translation"

Table 7.15 VS_FIXEDFILEINFO record.

Field Name	*Data Type*	*Comments*
dwSignature	DWORD	0XFEEF04BD
dwStrucVersion	DWORD	Binary version number of this structure.
dwFileVersionMS	DWORD	High 32 bits of the file's version number
dwFileVersionLS	DWORD	Low 32 bits of the file's version number
dwProductVersionMS	DWORD	High 32 bits of the product's version number
dwProductVersionLS	DWORD	Low 32 bits of the product's version number
dwFileFlagsMask	DWORD	Specifies which bits in dwFileFlags are valid
dwFileFlags	DWORD	Describes various attributes of the file
dwFileOS	DWORD	Intended operating system of the file
dwFileType	DWORD	Type of file
dwFileSubtype	DWORD	Specifies the function of the file
dwFileDateMS	DWORD	High 32 bits of the file's time stamp
dwFileDateLS	DWORD	Low 32-bits of the file's time stamp.

blocks, because each line in a variable information block has the form 'Value "Translation", lang-ID, charset-ID'. RES2RC processes these blocks by reading each translation block and writing out the appropriate information, one block at a time.

One of the first things RES2RC does when it encounters a `VersionInfo` block is to check whether it is the first resource of this type found in the input file. If so, RES2RC will write the line "#include <ver.h>" to ensure that the version strings listed in `VER.H` (and used by RES2RC) will be defined if the user attempts to compile the output of RES2RC with the resource compiler.

User-Defined Data

If a resource header begins with `0xFF` followed by a number not found in Table 7.1, RES2RC considers it data defined by the user. It will save the data in an external and uniquely named file of the format `UR###.USR`. An entry is put in the output file containing a reference to this file. Some of the code for this case is shared by the case of a resource header starting with a character other than `0xFF` (which signifies a name for the resource, rather than numeric identification).

In order to use RES2RC, all you need is a `.RES` file (such as that produced by `RC.EXE`, Microsoft's resource compiler). RES2RC will convert it to an .RC file. RES2RC requires two arguments: input filename and output filename. It will write any bitmaps, cursors, icons, or user-defined data to external files, generating unique filenames when needed. The line "#include <windows.h>" is always written to the output file; if any version resources are defined, the line "#include <ver.h>" is also written. In order to compile the output from RES2RC into a `.RES` file (which should be identical to the original input to RES2RC), you will need to include the path for `WINDOWS.H`. This can be done with a command of the form `rc -r -i<path> <input file>.rc`. Substitute the directory name containing `WINDOWS.H` for `<path>`, and the name of the input file for `<input file>.rc`. For example, on my computer, I use `rc -r -i\msvc\include xyz.rc`. The `-r` means produce only a `.RES` file. The `-i` means to include the following directory in the search path for header files.

Where Do I Go from Here?

An adventurous soul could use this code to write a program that would extract the resources from an executable and produce the corresponding `.RC` file, sort of an "Exe2Rc". The resource data itself is stored in an `.EXE` file in the same format as in a `.RES` file. The latter includes a header before each resource. An executable uses a resource table to maintain a list of resources.

Listing 7.1 RES2RC.C— Converts a .RESfile to an .R C file.

```
/*********************************************************************
 *
 * PROGRAM: RES2RC.C
 *
 * PURPOSE: Converts a .RES file to an .RC file.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 7, Resource (.RES) File Format, from Undocumented Windows
 * File Formats, published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 *********************************************************************/

#include "restypes.h"

/***************************************************
   Read a byte from infile
***************************************************/
BYTE get_byte(FILE *infile) {

    BYTE ch;

    fread(&ch, sizeof(BYTE), 1, infile);
    return(ch);

} /* get_byte - end */


/***************************************************
   Read a word from infile
***************************************************/
WORD get_word(FILE *infile) {

    WORD num;

    fread(&num, sizeof(WORD), 1, infile);
    return(num);

} /* get_word - end */


/***************************************************
   Write out an integer to outfile
***************************************************/
void write_number(int num, FILE *outfile) {

    fprintf( outfile, "%d ", num);

} /* write_number - end */
```

Listing 7.1 (continued)

```
/*************************************************
   Write out a word (unsigned short integer) to outfile
*************************************************/
void write_word(WORD num, FILE *outfile) {
    fprintf( outfile, "%hu", num);

} /* write_word - end */


/*************************************************
   Write out a dword (unsigned long) to outfile
*************************************************/
void write_dword(DWORD num, FILE *outfile) {

    fprintf( outfile, "%lu", num);

} /* write_dword - end */


/*************************************************
   Write out a character to outfile
*************************************************/
void write_char(char ch, FILE *outfile) {

   /* if the character is null, write out the string "\0" */
   if (ch == '\0')
      fprintf( outfile, "\\0");
   else
      fprintf( outfile, "%c", ch);

} /* write_char - end */


/*************************************************
   Write out an unsigned character to outfile
*************************************************/
void write_byte(BYTE ch, FILE *outfile) {

    fputc( ch, outfile);

} /* write_byte - end */


/*************************************************
   Write out a string to outfile, then add a newline
*************************************************/
void writeline(char *string, FILE *outfile) {

   if(strlen(string))
       fputs( string, outfile);

   fputc( CR, outfile);
   fputc( NL, outfile);

} /* writeline - end */
```

Listing 7.1 (continued)

```
/************************************************
   Write out a string to outfile
*************************************************/
void writestring(char *string, FILE *outfile) {

   if(strlen(string))
       fputs( string, outfile);

} /* writestring - end */


/************************************************
   Increase the INDENT variable value by INDENT_SPACES
*************************************************/
void increase_indent( void ) {

    INDENT += INDENT_SPACES;

} /* increase_indent - end */


/************************************************
   Decrease the INDENT variable value by INDENT_SPACES
*************************************************/
void decrease_indent( void ) {

    if((INDENT -= INDENT_SPACES) < 0)
       INDENT = 0;

} /* decrease_indent - end */


/************************************************
   Write out INDENT spaces to outfile
*************************************************/
void write_indent(FILE *outfile) {

    int n;

    for(n = INDENT; (n); --n)
       write_char(' ', outfile);

} /* write_indent - end */
```

Listing 7.1 (continued)

```
/***************************************************
   If the last stringtable wasn't closed, do so now.
***************************************************/
void finish_off_stringtable(FILE *outfile) {

   if (StringCount)
      writeline("}", outfile);

} /* finish_off_stringtable - end */


/***************************************************
   Get name of custom resource
***************************************************/
void get_custom_type( BYTE ch, FILE *infile, FILE *outfile) {

   while(ch != 0x00) {
      write_char(ch, outfile);
      ch = get_byte(infile);
   }

} /* get_custom_type - end */


/***************************************************
   Get the resource name from the .res file (infile)
***************************************************/
DWORD write_item_text(FILE *infile, FILE *outfile) {

   BYTE ch;
   DWORD length;

   ch = get_byte( infile );
   if(ch == 0x00)       return(1);
   writestring("\"", outfile);
   length = 1;
   while(ch != 0x00) {
      ++length;
      write_char(ch, outfile);
      ch = get_byte( infile );
   }
   writestring("\"", outfile);
   return(length);

} /* write_item_text - end */
```

Listing 7.1 (continued)

```
/************************************************
   Returns the first new filename of the form
   <prefix>#.<suffix>
************************************************/
char *get_data_filename(char *prefix, char *suffix) {

   FILE *fp;
   WORD number;
   char fname[13];
   char fname_start[3];
   char fname_end[5];
   int name_length;

   /* form the prefix to the new filename */
   name_length = (strlen(prefix) > 2) ? 2 : strlen(prefix);
   strncpy(fname_start, prefix, name_length);
   fname_start[name_length] = '\0';

   /* form the suffix to the new filename */
   fname_end[0] = '.';
   name_length = (strlen(suffix) > 3) ? 3 : strlen(suffix);
   strncat( fname_end, suffix, name_length);
   fname_end[name_length + 1] = '\0';
   /* Keep forming new filenames until you get to one you can't open */
   number = 0;
   sprintf( fname, "%s%u%s", fname_start, number, fname_end);
   while((number < 1000) && ((fp = fopen( fname, "r")) != NULL)) {
      fclose(fp);
      ++number;
      sprintf( fname, "%s%u%s", fname_start, number, fname_end);
   }

   if(number == 1000)
      fname[0] = '\0';

   return(fname);

} /* get_data_filename - end */


/************************************************
   Write out reslen bytes of data to datafile
************************************************/
void write_data( FILE *infile, FILE *datafile, DWORD reslen ) {

   BYTE ch;

   ch = get_byte(infile);
   while(reslen) {
      fputc( ch, datafile);
      if(--reslen)
         ch = get_byte(infile);
   }

} /* write_data - end */
```

Listing 7.1 (continued)

```
/*****************************************************
   Retrieves the length of the resource from the .res file
*****************************************************/
DWORD get_resource_length(FILE *infile) {

   DWORD reslen;

   fread(&reslen, sizeof(DWORD), 1, infile);
   return(reslen);

} /* get_resource_length - end */


/*****************************************************
   Get the resource name from the .res file (infile)
*****************************************************/
void get_resource_name(FILE *infile) {

   int resname;
   BYTE ch;

   ch = get_byte(infile);
   if(ch == 0xFF) {
       fread(&resname, sizeof(int), 1, infile);
   }
   else {
      ch = get_byte(infile);
      while((!feof(infile)) && (ch != 0x00)) {
         ch = get_byte(infile);
      }
   }

} /* get_resource_name - end */


/*****************************************************
   Get the version name from the .res file (infile)
*****************************************************/
void get_version_name(FILE *infile) {

   int  count=1; /* record number of chars read */
   BYTE ch;

   ch = get_byte(infile);
   while((!feof(infile)) && (ch != 0x00)) {
      ++count;
      ch = get_byte(infile);   }
```

Listing 7.1 (continued)

```
    /* Ensure number of characters read is a multiple of 4.    */
    /* According to the MS documentation, this is the format.  */
    /* See "MS Windows 3.1 Programmer's Reference" Vol.4, p.99 */
    count = count % 4;
    count = (count > 0) ? (4 - count) : 0;
    while ((!feof(infile)) && (count > 0)) {
       ch = get_byte(infile);
       --count;
    }

} /* get_version_name - end */


/****************************************************
   Get the resource number from the .res file (infile)
****************************************************/
WORD get_resource_number(FILE *infile) {

   WORD resname = 0;
   BYTE ch;

   ch = get_byte(infile);
   if(ch == 0xFF) {
       resname = get_word( infile );
   }
   else {
      ch = get_byte(infile);
      while((!feof(infile)) && (ch != 0x00)) {
         ch = get_byte(infile);
      }
   }

   return(resname);

} /* get_resource_number - end */


/****************************************************
   Get the resource name from the .res file (infile)
****************************************************/
void write_resource_name(FILE *infile, FILE *outfile) {

   int resname;
   BYTE ch;

   writeline( "", outfile);
```

Listing 7.1 (continued)

```
   ch = get_byte(infile);
   if(ch == 0xFF) {       fread(&resname, sizeof(int), 1, infile);
       write_number(resname, outfile);
   }
   else {
      write_char(ch, outfile);
      ch = get_byte(infile);
      while((!feof(infile)) && (ch != 0x00)) {
         write_char(ch, outfile);
         ch = get_byte(infile);
      }
   }

} /* write_resource_name - end */


/****************************************************
   Get the version number from the .res file (infile)
****************************************************/
void write_version_number(FILE *infile, FILE *outfile) {

   int resname;
   BYTE ch;

   writeline( "", outfile);

   ch = get_byte(infile);
   if(ch == 0xFF) {
       fread(&resname, sizeof(int), 1, infile);
       if (resname == (int) VS_VERSION_INFO)
          writestring("VS_VERSION_INFO ", outfile);
       else
          write_number(resname, outfile);
   }
   else {
      write_char(ch, outfile);
      ch = get_byte(infile);      while((!feof(infile)) && (ch != 0x00)) {
         write_char(ch, outfile);
         ch = get_byte(infile);
      }
   }

} /* write_version_number - end */
```

Listing 7.1 (continued)

```
/************************************************
   Get the memory flags for the current resource.
************************************************/
void get_mem_flags(FILE *infile) {

   /* Save the flag in a global variable.  See the header file! */
   prev_mem_flag = get_word( infile );
   return;

} /* get_mem_flags - end */


/************************************************
   Write the memory flag values for the "special"
   resources - cursors and icons. They're different
   than the rest in that 0x20 means "discardable",
   not "pure".
************************************************/
void write_special_mem_flag_values( WORD memtype, FILE *outfile) {

   if(memtype & 0x40)
      writestring(" PRELOAD", outfile);
   else
      writestring(" LOADONCALL", outfile);

   if(memtype & 0x10)
      writestring(" MOVEABLE", outfile);
   else
      writestring(" FIXED", outfile);

   if(memtype & 0x20)
      writestring(" DISCARDABLE", outfile);

} /* write_special_mem_flag_values - end */


/************************************************
   Write the memory flag values for the current resource.
************************************************/
void write_mem_flag_values( WORD memtype, FILE *outfile) {

   if(memtype & 0x40)
      writestring(" PRELOAD", outfile);
   else
      writestring(" LOADONCALL", outfile);

   if(memtype & 0x10)
      writestring(" MOVEABLE", outfile);
   else
      writestring(" FIXED", outfile);
```

Listing 7.1 (continued)

```
/* This one isn't really used - "PURE" doesn't seem to make a difference.

   if(memtype & 0x20)
      writestring(" PURE", outfile);
*/

   if(memtype & 0x1000)
      writestring(" DISCARDABLE", outfile);

} /* write_mem_flag_values - end */


/***************************************************
   Get the memory flags for the current resource.
***************************************************/
void write_mem_flags(FILE *infile, FILE *outfile) {

   WORD memtype;

   memtype = get_word( infile );
   write_mem_flag_values( memtype, outfile );

} /* write_mem_flags - end */


/***************************************************
  Process the cursor resource
***************************************************/
void process_cursor( FILE *infile ) {

   DWORD reslen;
   /* Skip it for now.  You'll come back and get the information   */
   /* when you hit the group cursor resource - that has the header */
   /* that has to go at the start of the .cur file.                */

   get_resource_name(infile);
   get_mem_flags(infile);
   reslen = get_resource_length(infile);

   fseek(infile, reslen, 1);

} /* process_cursor - end */
```

Listing 7.1 (continued)

```
/**************************************************
  Process the bitmap resource
**************************************************/
void process_bitmap(int restype, FILE *infile, FILE *outfile) {

   DWORD reslen;
   BITMAPFILEHEADER bmfh;
   char  datafilename[13];
   FILE  *datafile;

   write_resource_name(infile, outfile);
   write_char(' ', outfile);
   writestring(res_array[restype], outfile);
   write_mem_flags(infile, outfile);
   write_char(' ', outfile);

   reslen = get_resource_length(infile);

   strcpy(datafilename, get_data_filename("BM", "BMP"));
   if(strlen(datafilename) == 0) {
      writeline("<Unable to open output file>", outfile);
      fseek( infile, reslen, 1);
      return;
   }

   writeline( datafilename, outfile );

   if((datafile = fopen(datafilename, "wb")) == NULL) {
      printf("Unable to open file %s\n", datafilename);
      fseek( infile, reslen, 1);
      return;
   }

   bmfh.bfType      = 0x4D42;
   bmfh.bfSize      = reslen + sizeof(BITMAPFILEHEADER);
   bmfh.bfReserved1 = 0;
   bmfh.bfReserved2 = 0;
   bmfh.bfOffBits   = 0x76L;

   fwrite(&bmfh, sizeof(BITMAPFILEHEADER), 1, datafile);

   write_data( infile, datafile, reslen );
   fclose( datafile );

} /* process_bitmap - end */
```

Listing 7.1 (continued)

```
/*************************************************
  Process the flags for a popup menu
*************************************************/
void process_popup_flags(DWORD menuitem, FILE *outfile) {

   if(menuitem & MF_GRAYED)
      writestring(", GRAYED", outfile);
   if(menuitem & MF_DISABLED)
      writestring(", INACTIVE", outfile);
   if(menuitem & MF_CHECKED)
      writestring(", CHECKED", outfile);
   if(menuitem & MF_MENUBARBREAK)
      writestring(", MENUBARBREAK", outfile);
   if(menuitem & MF_MENUBREAK)
      writestring(", MENUBREAK", outfile);
   if(menuitem & MF_END) {
   }

} /* process_popup_flags - end */


/*************************************************
  Process the flags for a normal menuitem
*************************************************/
void process_menuitem_flags(DWORD menuitem, FILE *outfile) {
   if(menuitem & MF_GRAYED)
      writestring(", GRAYED", outfile);
   if(menuitem & MF_DISABLED)
      writestring(", INACTIVE", outfile);
   if(menuitem & MF_CHECKED)
      writestring(", CHECKED", outfile);
   if(menuitem & MF_MENUBARBREAK)
      writestring(", MENUBARBREAK", outfile);
   if(menuitem & MF_MENUBREAK)
      writestring(", MENUBREAK", outfile);
   if(menuitem & MF_HELP)
      writestring(", HELP", outfile);
   if(menuitem & MF_END) {
      writeline("", outfile);
      decrease_indent();
      write_indent(outfile);
      writestring("}", outfile);
   }

} /* process_menuitem_flags - end */
```

Listing 7.1 (continued)

```
/***********************************************
  Process a normal menu resource
************************************************/
DWORD process_normal_menu(DWORD menuitem, FILE *infile, FILE *outfile) {

   WORD  menuid;
   DWORD bytesread;

   fread(&menuid, sizeof(menuid), 1, infile);
   bytesread = sizeof(menuid);

   increase_indent();
   write_indent(outfile);

   writestring("MENUITEM ", outfile);
   bytesread += write_item_text( infile, outfile);
   if((bytesread - 1) > sizeof(menuid)) {
      writestring(", ", outfile);
      write_word( menuid, outfile);
      process_menuitem_flags(menuitem, outfile);
   }
   else if((!menuid) && (!menuitem)) {
      /* *** remove the leading quote from above */
      writestring("SEPARATOR", outfile);
   }
   writeline("", outfile);

   decrease_indent();

   return(bytesread);

} /* process_normal_menu - end */


/***********************************************
  Process a popup menu resource
************************************************/
DWORD process_popup_menu(DWORD menuitem, FILE *infile, FILE *outfile) {

   DWORD bytesread;

   increase_indent();
   write_indent(outfile);

   writestring("POPUP ", outfile);
   bytesread = write_item_text( infile, outfile);
   process_popup_flags(menuitem, outfile);
   writeline(" {", outfile);
```

Listing 7.1 (continued)

```
   return(bytesread);

} /* process_popup_menu - end */


/***************************************************
  Process the menu resource
***************************************************/
void process_menu(int restype, FILE *infile, FILE *outfile) {

   DWORD reslen;
   DWORD bytesread;
   WORD  menuitem;
   WORD  popupmenuitem;
   struct MenuHeader menuhdr;

   write_resource_name(infile, outfile);
   write_char(' ', outfile);
   writestring(res_array[restype], outfile);
   write_mem_flags(infile, outfile);
   writeline(" {", outfile);
   reslen = get_resource_length(infile);

   fread(&menuhdr, sizeof(struct MenuHeader), 1, infile);
   reslen -= sizeof(struct MenuHeader);

   fread(&menuitem, sizeof(menuitem), 1, infile);
   reslen -= sizeof(menuitem);
   while(reslen) {

      if(menuitem & MF_POPUP) {
          popupmenuitem = menuitem;
          bytesread = process_popup_menu(  menuitem, infile, outfile);
      }
      else
          bytesread = process_normal_menu( menuitem, infile, outfile);

      reslen -= bytesread;

      if(reslen) {
         fread(&menuitem, sizeof(menuitem), 1, infile);
         reslen -= sizeof(menuitem);
      }
      else if(popupmenuitem & MF_END) /* check last popup was the end */
          writeline("}", outfile);

   }

} /* process_menu - end */
```

Listing 7.1 (continued)

```
/****************************************************
  Process the icon resource
****************************************************/
void process_icon( FILE *infile ) {

   DWORD reslen;

   /* Skip it for now.  You'll come back and get the information */
   /* when you hit the group icon resource - that has the header */
   /* that has to go at the start of the .ico file.              */

   get_resource_name(infile);
   get_mem_flags(infile);
   reslen = get_resource_length(infile);

   fseek(infile, reslen, 1);

} /* process_icon - end */


/****************************************************
  Write the dialog box size and shape information
****************************************************/
void write_dialog_sizes( DIALOGHEADER dlg_hdr, FILE *outfile) {

   write_char(' ', outfile);
   write_word( dlg_hdr.x, outfile );
   writestring(", ", outfile);
   write_word( dlg_hdr.y, outfile );
   writestring(", ", outfile);
   write_word( dlg_hdr.width, outfile );
   writestring(", ", outfile);
   write_word( dlg_hdr.height, outfile );
   writeline("", outfile);

} /* write_dialog_sizes - end */


/****************************************************
  Check if the style is true; if so, write it out
****************************************************/
void check_for_style( DWORD style, char *name, FILE *outfile ) {

   static WORD first_style = 1;
   static WORD line_size = 6;          /* length of 'STYLE ' */
```

Listing 7.1 (continued)

```
   if (style) {

      if (first_style) {
         writestring("STYLE ", outfile);
         first_style = 0;
      }
      else {
         writestring(" | ", outfile);
         line_size += 3;                /* add lenght of ' | ' */
      }

      /* this is an attempt to keep the line length reasonable */
      if ((line_size + strlen(name)) > 75) {
         writeline("", outfile);
         line_size = 6;
         writestring("      ", outfile);
      }

      writestring( name, outfile );
      line_size += strlen(name);

   } /* if (style) - end */

} /* check_for_style - end */


/****************************************************
  Write the dialog box style information
****************************************************/
void write_dialog_style( DIALOGHEADER dlg_hdr, FILE *outfile) {

   DWORD style;

   style = dlg_hdr.lStyle;

   check_for_style( style & WS_OVERLAPPED,   "WS_OVERLAPPED",   outfile );
   check_for_style( style & WS_POPUP,        "WS_POPUP",        outfile );
   check_for_style( style & WS_CHILD,        "WS_CHILD",        outfile );
   check_for_style( style & WS_CLIPSIBLINGS, "WS_CLIPSIBLINGS", outfile );
   check_for_style( style & WS_CLIPCHILDREN, "WS_CLIPCHILDREN", outfile );
   check_for_style( style & WS_VISIBLE,      "WS_VISIBLE",      outfile );
   check_for_style( style & WS_DISABLED,     "WS_DISABLED",     outfile );
   check_for_style( style & WS_MINIMIZE,     "WS_MINIMIZE",     outfile );
   check_for_style( style & WS_MAXIMIZE,     "WS_MAXIMIZE",     outfile );
```

Listing 7.1 (continued)

```
   check_for_style( style & WS_BORDER,      "WS_BORDER",      outfile );
   check_for_style( style & WS_DLGFRAME,    "WS_DLGFRAME",    outfile );
   check_for_style( style & WS_VSCROLL,     "WS_VSCROLL",     outfile );
   check_for_style( style & WS_HSCROLL,     "WS_HSCROLL",     outfile );
   check_for_style( style & WS_SYSMENU,     "WS_SYSMENU",     outfile );
   check_for_style( style & WS_THICKFRAME,  "WS_THICKFRAME",  outfile );
   check_for_style( style & WS_MINIMIZEBOX, "WS_MINIMIZEBOX", outfile );
   check_for_style( style & WS_MAXIMIZEBOX, "WS_MAXIMIZEBOX", outfile );

   check_for_style( style & WS_GROUP,          "WS_GROUP",          outfile );
   check_for_style( style & WS_TABSTOP,        "WS_TABSTOP",        outfile );
   check_for_style( style & WS_EX_TOPMOST,     "WS_EX_TOPMOST",     outfile );
   check_for_style( style & WS_EX_ACCEPTFILES, "WS_EX_ACCEPTFILES", outfile );

   check_for_style( style & WS_EX_NOPARENTNOTIFY, "WS_EX_NOPARENTNOTIFY",
                    outfile );

   check_for_style( style & DS_ABSALIGN,   "DS_ABSALIGN",   outfile );
   check_for_style( style & DS_SYSMODAL,   "DS_SYSMODAL",   outfile );
   check_for_style( style & DS_LOCALEDIT,  "DS_LOCALEDIT",  outfile );
   check_for_style( style & DS_SETFONT,    "DS_SETFONT",    outfile );
   check_for_style( style & DS_MODALFRAME, "DS_MODALFRAME", outfile );
   check_for_style( style & DS_NOIDLEMSG,  "DS_NOIDLEMSG",  outfile );

   writeline("", outfile);

} /* write_dialog_style - end */


/*****************************************************
  Write the dialog box menu name
*****************************************************/
void write_dialog_menu( FILE *infile, FILE *outfile) {

   BYTE ch;
   WORD menuid;

   /* Read the first character and check for non-zero start byte */
   ch = get_byte(infile);

   /* if first byte is 0x00, no menu name */
   if(ch != 0x00) {

      writestring("MENU ", outfile);
      if(ch == 0xFF) {
         /* menu id is a number */
         menuid = get_word( infile );
         write_word(menuid, outfile);
      }
```

Listing 7.1 (continued)

```
      else
         get_custom_type( ch, infile, outfile);

      writeline("", outfile);
   }

} /* write_dialog_menu - end */


/*************************************************
  Write the dialog box class name
*************************************************/
void write_dialog_class( FILE *infile, FILE *outfile) {

   BYTE ch;

   /* Read the first character and check for non-zero start byte */
   ch = get_byte(infile);

   /* get the resource type */
   if(ch != 0x00) {
      writestring("CLASS \"", outfile);
      get_custom_type( ch, infile, outfile);
      writeline("\"", outfile);
   }

} /* write_dialog_class - end */


/*************************************************
  Write the dialog box font information
*************************************************/
void write_dialog_font( FILE *infile, FILE *outfile) {

   WORD pointsize;

   writestring("FONT ", outfile);

   /* read and write the font point size */
   pointsize = get_word( infile );
   write_word( pointsize, outfile);
   writestring(", ", outfile);

   /* write the font name */
   (void) write_item_text( infile, outfile);

   writeline("", outfile);

} /* write_dialog_font - end */
```

Listing 7.1 (continued)

```
/***************************************************
  Write the dialog box caption
***************************************************/
void write_dialog_caption( FILE *infile, FILE *outfile) {

   BYTE ch;

   /* Read the first character and check for non-zero start byte */
   ch = get_byte(infile);

   /* get the resource type */
   if(ch != 0x00) {
      writestring("CAPTION \"", outfile);
      get_custom_type( ch, infile, outfile);
      writeline("\"", outfile);
   }

} /* write_dialog_caption - end */


/***************************************************
  Write the start of the "CONTROL" string
***************************************************/
void write_control_header( char *text, WORD id, FILE *outfile) {

   write_indent(outfile);

   writestring("CONTROL \"", outfile);
   writestring( text, outfile);
   writestring( "\", ", outfile);

   /* check if id is -1 */
   if (id != 0xFFFF)
      write_word( id, outfile);
   else
      writestring( "-1", outfile);

   writestring( ", ", outfile);

} /* write_control_header - end */
```

Listing 7.1 (continued)

```
/****************************************************
  Write the end of the "CONTROL" statement.
****************************************************/
void write_control_end( CONTROLDATA ctrl, FILE *outfile) {

   char string[50];

   /* write the size/dimensions of the object */
   write_indent(outfile);
   sprintf( string, "%hu, %hu, %hu, %hu", ctrl.x, ctrl.y,
            ctrl.width, ctrl.height);
   writeline( string, outfile);

} /* write_control_end - end */


/****************************************************
  Write the style for the dialog box
****************************************************/
void check_for_dlg_style( DWORD style, WORD *first_style, WORD *line_len,
                          char *style_name, FILE *outfile ) {

   if (style) {

      /* if first_style = 2, write out a leading comma */
      if (*first_style == 2) {
         writestring(", ", outfile);
         *first_style = 1;
      }

      if (*first_style == 1) {

         /* this is the first style, so just write out the style name */
         writestring( style_name, outfile);
         *first_style = 0;
         *line_len += strlen(style_name);

      }
      else {

         /* this is after first style, so write '|' for concat */
         writestring(" | ", outfile);
         *line_len += (3 + strlen(style_name));

         /* try to keep each line to a reasonable length */
         if (*line_len >= 75) {
            writeline("", outfile);
            write_indent(outfile);
            *line_len = strlen(style_name);
         }

         /* write out the name of the style */
         writestring(style_name, outfile);

      } /* if (*first_style) / else - end */

   } /* if (style) - end */

} /* check_for_dlg_style - end */
```

Listing 7.1 (continued)

```
/*************************************************
  Process the button control in the dialog box
*************************************************/
void process_control_button( DWORD style, FILE *outfile ) {

   WORD first_style = 1;
   WORD line_len = 0;

   writestring( "\"button\", ", outfile);


   check_for_dlg_style( BS_LEFTTEXT & style, &first_style, &line_len,
                        "BS_LEFTTEXT", outfile );
   check_for_dlg_style( WS_TABSTOP & style, &first_style, &line_len,
                        "WS_TABSTOP", outfile );
   check_for_dlg_style( WS_GROUP & style, &first_style, &line_len,
                        "WS_GROUP", outfile );
   check_for_dlg_style( WS_DISABLED & style, &first_style, &line_len,
                        "WS_DISABLED", outfile );

   if(first_style == 0)
      writestring(" | ", outfile);

   /* checked for non-exclusive properties, now clear out high bits */
   style &= 0xF;

   if (style == BS_DEFPUSHBUTTON)
      writestring("BS_DEFPUSHBUTTON", outfile);
   else if (style == BS_CHECKBOX)
      writestring("BS_CHECKBOX", outfile);
   else if (style == BS_AUTOCHECKBOX)
      writestring("BS_AUTOCHECKBOX", outfile);
   else if (style == BS_RADIOBUTTON)
      writestring("BS_RADIOBUTTON", outfile);
   else if (style == BS_3STATE)
      writestring("BS_3STATE", outfile);
   else if (style == BS_AUTO3STATE)
      writestring("BS_AUTO3STATE", outfile);
   else if (style == BS_GROUPBOX)
      writestring("BS_GROUPBOX", outfile);
   else if (style == BS_USERBUTTON)
      writestring("BS_USERBUTTON", outfile);
   else if (style == BS_AUTORADIOBUTTON)
      writestring("BS_AUTORADIOBUTTON", outfile);
   else if (style == BS_OWNERDRAW)
      writestring("BS_OWNERDRAW", outfile);
   else
      writestring("BS_PUSHBUTTON", outfile);

   writeline(",", outfile);

} /* process_control_button - end */
```

Listing 7.1 (continued)

```
/**************************************************
  Process the edit control in the dialog box
**************************************************/
void process_control_edit( CONTROLDATA ctrl, FILE *outfile ) {

   WORD first_style = 2;
   char string[100];
   WORD line_len = 0;
   DWORD style;

   style = ctrl.lStyle;

   write_indent(outfile);
   sprintf( string, "EDITTEXT %hu, %hu, %hu, %hu, %hu", ctrl.id, ctrl.x,
            ctrl.y, ctrl.width, ctrl.height);
   line_len = strlen(string);
   writestring( string, outfile);

   check_for_dlg_style( ES_LEFT & style, &first_style, &line_len,
                        "ES_LEFT", outfile );
   check_for_dlg_style( ES_CENTER & style, &first_style, &line_len,
                        "ES_CENTER", outfile );
   check_for_dlg_style( ES_RIGHT & style, &first_style, &line_len,
                        "ES_RIGHT", outfile );
   check_for_dlg_style( ES_LINE & style, &first_style, &line_len,
                        "ES_MULTILINE", outfile );
   check_for_dlg_style( ES_UPPERCASE & style, &first_style, &line_len,
                        "ES_UPPERCASE", outfile );
   check_for_dlg_style( ES_LOWERCASE & style, &first_style, &line_len,
                        "ES_LOWERCASE", outfile );
   check_for_dlg_style( ES_PASSWORD & style, &first_style, &line_len,
                        "ES_PASSWORD", outfile );
   check_for_dlg_style( ES_AUTOVSCROLL & style, &first_style, &line_len,
                        "ES_AUTOVSCROLL", outfile );
   check_for_dlg_style( ES_AUTOHSCROLL & style, &first_style, &line_len,
                        "ES_AUTOHSCROLL", outfile );
   check_for_dlg_style( ES_NOHIDESEL & style, &first_style, &line_len,
                        "ES_NOHIDESEL", outfile );
   check_for_dlg_style( ES_OEMCONVERT & style, &first_style, &line_len,
                        "ES_OEMCONVERT", outfile );
   check_for_dlg_style( ES_READONLY & style, &first_style, &line_len,
                        "ES_READONLY", outfile );
   check_for_dlg_style( ES_WANTRETURN & style, &first_style, &line_len,
                        "ES_WANTRETURN", outfile );
```

Listing 7.1 (continued)

```
   check_for_dlg_style( WS_TABSTOP & style, &first_style, &line_len,
                        "WS_TABSTOP", outfile );
   check_for_dlg_style( WS_GROUP & style, &first_style, &line_len,
                        "WS_GROUP", outfile );
   check_for_dlg_style( WS_VSCROLL & style, &first_style, &line_len,
                        "WS_VSCROLL", outfile );
   check_for_dlg_style( WS_HSCROLL & style, &first_style, &line_len,
                        "WS_HSCROLL", outfile );
   check_for_dlg_style( WS_DISABLED & style, &first_style, &line_len,
                        "WS_DISABLED", outfile );

   writeline("", outfile);

} /* process_control_edit - end */


/*****************************************************
  Process the static control in the dialog box
*****************************************************/
void process_control_static( DWORD style, FILE *outfile ) {

   WORD first_style = 1;
   WORD line_len = 0;

   writestring( "\"static\", ", outfile);

   check_for_dlg_style( SS_NOPREFIX & style, &first_style, &line_len,
                        "SS_NOPREFIX", outfile );
   check_for_dlg_style( WS_GROUP & style, &first_style, &line_len,
                        "WS_GROUP", outfile );
   check_for_dlg_style( WS_TABSTOP & style, &first_style, &line_len,
                        "WS_TABSTOP", outfile );

   if(first_style == 0)
      writestring(" | ", outfile);

   style &= 0xF;
```

Listing 7.1 (continued)

```
   if ( style == SS_CENTER )
      writestring( "SS_CENTER", outfile);
   else if ( style == SS_RIGHT )
      writestring( "SS_RIGHT", outfile);
   else if ( style == SS_ICON )
      writestring( "SS_ICON", outfile);
   else if ( style == SS_BLACKRECT )
      writestring( "SS_BLACKRECT", outfile);
   else if ( style == SS_GRAYRECT )
      writestring( "SS_GRAYRECT", outfile);
   else if ( style == SS_WHITERECT )
      writestring( "SS_WHITERECT", outfile);
   else if ( style == SS_BLACKFRAME )
      writestring( "SS_BLACKFRAME", outfile);
   else if ( style == SS_GRAYFRAME )
      writestring( "SS_GRAYFRAME", outfile);
   else if ( style == SS_WHITEFRAME )
      writestring( "SS_WHITEFRAME", outfile);
   else if ( style == SS_SIMPLE )
      writestring( "SS_SIMPLE", outfile);
   else if ( style == SS_LEFTNOWORDWRAP )
      writestring( "SS_LEFTNOWORDWRAP", outfile);
   else
      writestring( "SS_LEFT", outfile);

   writeline(",", outfile);

} /* process_control_static - end */


/****************************************************
  Process the list box control in the dialog box
****************************************************/
void process_control_listbox( CONTROLDATA ctrl, FILE *outfile ) {

   WORD first_style = 2;
   char string[100];
   WORD line_len = 0;
   DWORD style;

   style = ctrl.lStyle;

   write_indent(outfile);
   sprintf( string, "LISTBOX %hu, %hu, %hu, %hu, %hu", ctrl.id, ctrl.x,
            ctrl.y, ctrl.width, ctrl.height);
   line_len = strlen(string);
   writestring( string, outfile);
```

Listing 7.1 (continued)

```
   check_for_dlg_style( LBS_NOTIFY & style, &first_style, &line_len,
                        "LBS_NOTIFY", outfile );
   check_for_dlg_style( LBS_SORT & style, &first_style, &line_len,
                        "LBS_SORT", outfile );
   check_for_dlg_style( LBS_NOREDRAW & style, &first_style, &line_len,
                        "LBS_NOREDRAW", outfile );
   check_for_dlg_style( LBS_MULTIPLESEL & style, &first_style, &line_len,
                        "LBS_MULTIPLESEL", outfile );
   check_for_dlg_style( LBS_OWNERDRAWFIXED & style, &first_style, &line_len,
                        "LBS_OWNERDRAWFIXED", outfile );
   check_for_dlg_style( LBS_OWNERDRAWVARIABLE & style, &first_style,
                        &line_len, "LBS_OWNERDRAWVARIABLE", outfile );
   check_for_dlg_style( LBS_HASSTRINGS & style, &first_style, &line_len,
                        "LBS_HASSTRINGS", outfile );
   check_for_dlg_style( LBS_USETABSTOPS & style, &first_style, &line_len,
                        "LBS_USETABSTOPS", outfile );
   check_for_dlg_style( LBS_NOINTEGRALHEIGHT & style, &first_style,
                        &line_len, "LBS_NOINTEGRALHEIGHT", outfile );
   check_for_dlg_style( LBS_MULTICOLUMN & style, &first_style, &line_len,
                        "LBS_MULTICOLUMN", outfile );
   check_for_dlg_style( LBS_WANTKEYBOARDINPUT & style, &first_style,
                        &line_len, "LBS_WANTKEYBOARDINPUT", outfile );
   check_for_dlg_style( LBS_EXTENDEDSEL & style, &first_style, &line_len,
                        "LBS_EXTENDEDSEL", outfile );
   check_for_dlg_style( LBS_DISABLENOSCROLL & style, &first_style,
                        &line_len, "LBS_DISABLENOSCROLL", outfile );

   check_for_dlg_style( WS_BORDER & style, &first_style, &line_len,
                        "WS_BORDER", outfile );
   check_for_dlg_style( WS_VSCROLL & style, &first_style, &line_len,
                        "WS_VSCROLL", outfile );

   writeline("", outfile);

} /* process_control_listbox - end */


/**************************************************
  Process the scroll bar control in the dialog box
**************************************************/
void process_control_scrollbar( CONTROLDATA ctrl, FILE *outfile ) {

   WORD first_style = 2;
   char string[100];
   WORD line_len = 0;
   DWORD style;
```

Listing 7.1 (continued)

```
   style = ctrl.lStyle;

   write_indent(outfile);
   sprintf( string, "SCROLLBAR %hu, %hu, %hu, %hu, %hu", ctrl.id, ctrl.x,
            ctrl.y, ctrl.width, ctrl.height);
   line_len = strlen(string);
   writestring( string, outfile);

   check_for_dlg_style( SBS_HORZ & style, &first_style, &line_len,
                        "SBS_HORZ", outfile );
   check_for_dlg_style( SBS_VERT & style, &first_style, &line_len,
                        "SBS_VERT", outfile );
   check_for_dlg_style( SBS_TOPALIGN & style, &first_style, &line_len,
                        "SBS_TOPALIGN", outfile );
   check_for_dlg_style( SBS_BOTTOMALIGN & style, &first_style, &line_len,
                        "SBS_BOTTOMALIGN", outfile );
   check_for_dlg_style( SBS_SIZEBOX & style, &first_style, &line_len,
                        "SBS_SIZEBOX", outfile );

   check_for_dlg_style( WS_TABSTOP & style, &first_style, &line_len,
                        "WS_TABSTOP", outfile );
   check_for_dlg_style( WS_GROUP & style, &first_style, &line_len,
                        "WS_GROUP", outfile );
   check_for_dlg_style( WS_DISABLED & style, &first_style, &line_len,
                        "WS_DISABLED", outfile );

   writeline("", outfile);

} /* process_control_scrollbar - end */


/*****************************************************
  Process the combo box control in the dialog box
*****************************************************/
void process_control_combobox( CONTROLDATA ctrl, FILE *outfile) {

   WORD first_style = 2;
   WORD line_len = 0;
   char string[100];
   DWORD style;

   style = ctrl.lStyle;

   write_indent(outfile);
   sprintf( string, "COMBOBOX %hu, %hu, %hu, %hu, %hu", ctrl.id, ctrl.x,
            ctrl.y, ctrl.width, ctrl.height);
   line_len = strlen(string);
   writestring( string, outfile);
```

Listing 7.1 (continued)

```
   check_for_dlg_style( CBS_SIMPLE & style, &first_style, &line_len,
                         "CBS_SIMPLE", outfile );
   check_for_dlg_style( CBS_DROPDOWN & style, &first_style, &line_len,
                         "CBS_DROPDOWN", outfile );
   check_for_dlg_style( CBS_OWNERDRAWFIXED & style, &first_style, &line_len,
                         "CBS_OWNERDRAWFIXED", outfile );
   check_for_dlg_style( CBS_OWNERDRAWVARIABLE & style, &first_style,
                         &line_len, "CBS_OWNERDRAWVARIABLE", outfile );
   check_for_dlg_style( CBS_AUTOHSCROLL & style, &first_style, &line_len,
                         "CBS_AUTOHSCROLL", outfile );
   check_for_dlg_style( CBS_OEMCONVERT & style, &first_style, &line_len,
                         "CBS_OEMCONVERT", outfile );
   check_for_dlg_style( CBS_SORT & style, &first_style, &line_len,
                         "CBS_SORT", outfile );
   check_for_dlg_style( CBS_HASSTRINGS & style, &first_style, &line_len,
                         "CBS_HASSTRINGS", outfile );
   check_for_dlg_style( CBS_NOINTEGRALHEIGHT & style, &first_style,
                         &line_len, "CBS_NOINTEGRALHEIGHT", outfile );
   check_for_dlg_style( CBS_DISABLENOSCROLL & style, &first_style,
                         &line_len, "CBS_DISABLENOSCROLL", outfile );

   check_for_dlg_style( WS_TABSTOP & style, &first_style, &line_len,
                         "WS_TABSTOP", outfile );
   check_for_dlg_style( WS_GROUP & style, &first_style, &line_len,
                         "WS_GROUP", outfile );
   check_for_dlg_style( WS_VSCROLL & style, &first_style, &line_len,
                         "WS_VSCROLL", outfile );
   check_for_dlg_style( WS_DISABLED & style, &first_style, &line_len,
                         "WS_DISABLED", outfile );

   writeline("", outfile);

} /* process_control_combobox - end */


/**************************************************
  Process each control in the dialog box
**************************************************/
void process_control( FILE *infile, FILE *outfile ) {

   CONTROLDATA ctrl;
   BYTE class_id;
   BYTE ch;
   char ctrl_text[260];
   char ctrl_class[260];
   WORD index = 0;
```

Listing 7.1 (continued)

```
   fread( &ctrl, sizeof(CONTROLDATA), 1, infile);

   /* read the class type (if 0x8?) or the string (otherwise) */
   ch = get_byte(infile);
   if (ch & 0x80) {
      class_id = ch;
   }
   else {
      class_id = 0x00;
      while (ch != 0x00) {
         if (index < 260) {
            ctrl_class[index] = ch;
            ++index;
         }
         ch = get_byte(infile);
      }
   }
   ctrl_class[index] = '\0';

   /* read the text field */
   ch = get_byte(infile);
   index = 0;
   while (ch != 0x00) {
      if (index < 260) {
         ctrl_text[index] = ch;
         ++index;
      }
      ch = get_byte(infile);
   }
   ctrl_text[index] = '\0';

   /* read the extra 0x00 */
   ch = get_byte(infile);
   if (ch != 0x00) {
      writestring("Error ** ch ->", outfile);
      write_char( ch, outfile);
      writeline("< - should be 00", outfile);
   }

   increase_indent();

   if (class_id & 0x80) {

      switch (class_id) {

         /* Control is a button */
         case 0x80:
            write_control_header( ctrl_text, ctrl.id, outfile);
            process_control_button( ctrl.lStyle, outfile );
            write_control_end( ctrl, outfile);
            break;
```

Listing 7.1 (continued)

```
         /* Control is an edit widget */
         case 0x81:
            process_control_edit( ctrl, outfile );
            break;

         /* Control is a static widget */
         case 0x82:
            write_control_header( ctrl_text, ctrl.id, outfile);
            process_control_static( ctrl.lStyle, outfile );
            write_control_end( ctrl, outfile);
            break;

         /* Control is a listbox */
         case 0x83:
            process_control_listbox( ctrl, outfile );
            break;

         /* Control is a scrollbar */
         case 0x84:
            process_control_scrollbar( ctrl, outfile );
            break;

         /* Control is a combobox */
         case 0x85:
            process_control_combobox( ctrl, outfile );
            break;

         default:
            break;  /* Unknown type, so skip */
      }

   }
   else {
      ; /* The resource type is unknown, so skip */
   }

   decrease_indent();

} /* process_control - end */
```

Listing 7.1 (continued)

```
/**************************************************
  Process the dialog resource
**************************************************/
void process_dialog(int restype, FILE *infile, FILE *outfile) {

   DIALOGHEADER dlg_hdr;

   /* write the generic dialog box info */
   write_resource_name(infile, outfile);
   write_char(' ', outfile);
   writestring(res_array[restype], outfile);
   write_mem_flags(infile, outfile);

   /* get the dialog box resource length, and the dialog box header */
   (void) get_resource_length(infile);
   fread( &dlg_hdr, sizeof(DIALOGHEADER), 1, infile);

   write_dialog_sizes( dlg_hdr, outfile);
   write_dialog_style( dlg_hdr, outfile);
   write_dialog_menu(  infile,  outfile);
   write_dialog_class( infile,  outfile);
   write_dialog_caption( infile, outfile);

   if (dlg_hdr.lStyle & DS_SETFONT)
      write_dialog_font( infile, outfile);

   writeline("BEGIN", outfile);

   while(dlg_hdr.bNumberOfItems) {

      process_control( infile, outfile );

      dlg_hdr.bNumberOfItems -= 1;
   }


   writeline("END", outfile);

} /* process_dialog - end */
```

Listing 7.1 (continued)

```
/*************************************************
  Process the string resource
*************************************************/
void process_string(int restype, FILE *infile, FILE *outfile) {

   WORD  sID;
   int   index;
   BYTE  ch;
   BYTE  strlen;

   sID = get_resource_number(infile);
   if (StringCount == 0) {
      writeline( "", outfile);
      writestring(res_array[restype], outfile);
      write_mem_flags(infile, outfile);
      writeline(" {", outfile);
   }
   else {
      get_mem_flags(infile);
   }

   ++StringCount;

   increase_indent();

   (void) get_resource_length(infile);

   for(index = 0; index < 16; ++index) {
      if((strlen = get_byte(infile)) != 0x00) {
         write_indent(outfile);
         sID = index + ((sID - 1) * 16);
         write_word( sID, outfile);
         writestring(", \"", outfile);
         while(strlen--) {
            ch = get_byte(infile);
            write_char(ch, outfile);
         }
         writeline("\"", outfile);
      }
   }

   decrease_indent();

   if(StringCount >= 16) {
      writeline("}", outfile);
      StringCount = 0;
   }

} /* process_string - end */
```

Listing 7.1 (continued)

```
/**************************************************
  Process the fontdir resource
**************************************************/
void process_fontdir( FILE *infile ) {

   DWORD reslen;

   get_resource_name(infile);
   get_mem_flags(infile);
   reslen = get_resource_length(infile);

   fseek(infile, reslen, 1);

} /* process_fontdir - end */


/**************************************************
  Process the font resource
**************************************************/
void process_font(int restype, FILE *infile, FILE *outfile) {

   DWORD reslen;
   char  datafilename[13];
   FILE  *datafile;

   write_resource_name(infile, outfile);
   write_char(' ', outfile);
   writestring(res_array[restype], outfile);
   write_mem_flags(infile, outfile);
   write_char(' ', outfile);

   reslen = get_resource_length(infile);

   strcpy(datafilename, get_data_filename("FO", "FON"));
   if(strlen(datafilename) == 0) {
      writeline("<Unable to open output file>", outfile);
      fseek( infile, reslen, 1);
      return;
   }

   writeline( datafilename, outfile );

   if((datafile = fopen(datafilename, "wb")) == NULL) {
      printf("Unable to open file %s\n", datafilename);
      fseek( infile, reslen, 1);
      return;
   }

   write_data( infile, datafile, reslen );
   fclose( datafile );

} /* process_font - end */
```

Listing 7.1 (continued)

```
/**************************************************
  Write the virtual character for the current accelerator
**************************************************/
void write_virtual_accel_event( WORD wEvent, FILE *outfile) {

   switch(wEvent) {

      case VK_LBUTTON:
         writestring( "VK_LBUTTON", outfile);
         break;

      case VK_RBUTTON:
         writestring( "VK_RBUTTON", outfile);
         break;

      case VK_CANCEL:
         writestring( "VK_CANCEL", outfile);
         break;

      case VK_MBUTTON:
         writestring( "VK_MBUTTON", outfile);
         break;

      case VK_BACK:
         writestring( "VK_BACK", outfile);
         break;

      case VK_TAB:
         writestring( "VK_TAB", outfile);
         break;

      case VK_CLEAR:
         writestring( "VK_CLEAR", outfile);
         break;

      case VK_RETURN:
         writestring( "VK_RETURN", outfile);
         break;

      case VK_SHIFT:
         writestring( "VK_SHIFT", outfile);
         break;

      case VK_CONTROL:
         writestring( "VK_CONTROL", outfile);
         break;
```

Listing 7.1 (continued)

```
      case VK_MENU:
         writestring( "VK_MENU", outfile);
         break;

      case VK_PAUSE:
         writestring( "VK_PAUSE", outfile);
         break;

      case VK_CAPITAL:
         writestring( "VK_CAPITAL", outfile);
         break;

      case VK_ESCAPE:
         writestring( "VK_ESCAPE", outfile);
         break;

      case VK_SPACE:
         writestring( "VK_SPACE", outfile);
         break;

      case VK_PRIOR:
         writestring( "VK_PRIOR", outfile);
         break;

      case VK_NEXT:
         writestring( "VK_NEXT", outfile);
         break;

      case VK_END:
         writestring( "VK_END", outfile);
         break;

      case VK_HOME:
         writestring( "VK_HOME", outfile);
         break;

      case VK_LEFT:
         writestring( "VK_LEFT", outfile);
         break;

      case VK_UP:
         writestring( "VK_UP", outfile);
         break;

      case VK_RIGHT:
         writestring( "VK_RIGHT", outfile);
         break;
```

Listing 7.1 (continued)

```
      case VK_DOWN:
         writestring( "VK_DOWN", outfile);
         break;

      case VK_SELECT:
         writestring( "VK_SELECT", outfile);
         break;

      case VK_PRINT:
         writestring( "VK_PRINT", outfile);
         break;

      case VK_EXECUTE:
         writestring( "VK_EXECUTE", outfile);
         break;

      case VK_SNAPSHOT:
         writestring( "VK_SNAPSHOT", outfile);
         break;

      case VK_INSERT:
         writestring( "VK_INSERT", outfile);
         break;

      case VK_DELETE:
         writestring( "VK_DELETE", outfile);
         break;

      case VK_HELP:
         writestring( "VK_HELP", outfile);
         break;

      case VK_NUMPAD0:
         writestring( "VK_NUMPAD0", outfile);
         break;

      case VK_NUMPAD1:
         writestring( "VK_NUMPAD1", outfile);
         break;

      case VK_NUMPAD2:
         writestring( "VK_NUMPAD2", outfile);
         break;
```

Listing 7.1 (continued)

```
    case VK_NUMPAD3:
       writestring( "VK_NUMPAD3", outfile);
       break;

    case VK_NUMPAD4:
       writestring( "VK_NUMPAD4", outfile);
       break;

    case VK_NUMPAD5:
       writestring( "VK_NUMPAD5", outfile);
       break;

    case VK_NUMPAD6:
       writestring( "VK_NUMPAD6", outfile);
       break;

    case VK_NUMPAD7:
       writestring( "VK_NUMPAD7", outfile);
       break;

    case VK_NUMPAD8:
       writestring( "VK_NUMPAD8", outfile);
       break;

    case VK_NUMPAD9:
       writestring( "VK_NUMPAD9", outfile);
       break;

    case VK_MULTIPLY:
       writestring( "VK_MULTIPLY", outfile);
       break;

    case VK_ADD:
       writestring( "VK_ADD", outfile);
       break;

    case VK_SEPARATOR:
       writestring( "VK_SEPARATOR", outfile);
       break;

    case VK_SUBTRACT:
       writestring( "VK_SUBTRACT", outfile);
       break;
```

Listing 7.1 (continued)

```
    case VK_DECIMAL:
       writestring( "VK_DECIMAL", outfile);
       break;

    case VK_DIVIDE:
       writestring( "VK_DIVIDE", outfile);
       break;

    case VK_F1:
       writestring( "VK_F1", outfile);
       break;

    case VK_F2:
       writestring( "VK_F2", outfile);
       break;

    case VK_F3:
       writestring( "VK_F3", outfile);
       break;

    case VK_F4:
       writestring( "VK_F4", outfile);
       break;

    case VK_F5:
       writestring( "VK_F5", outfile);
       break;

    case VK_F6:
       writestring( "VK_F6", outfile);
       break;

    case VK_F7:
       writestring( "VK_F7", outfile);
       break;

    case VK_F8:
       writestring( "VK_F8", outfile);
       break;

    case VK_F9:
       writestring( "VK_F9", outfile);
       break;
```

Listing 7.1 (continued)

```
      case VK_F10:
         writestring( "VK_F10", outfile);
         break;

      case VK_F11:
         writestring( "VK_F11", outfile);
         break;

      case VK_F12:
         writestring( "VK_F12", outfile);
         break;

      case VK_F13:
         writestring( "VK_F13", outfile);
         break;

      case VK_F14:
         writestring( "VK_F14", outfile);
         break;

      case VK_F15:
         writestring( "VK_F15", outfile);
         break;

      case VK_F16:
         writestring( "VK_F16", outfile);
         break;

      case VK_F17:
         writestring( "VK_F17", outfile);
         break;

      case VK_F18:
         writestring( "VK_F18", outfile);
         break;

      case VK_F19:
         writestring( "VK_F19", outfile);
         break;

      case VK_F20:
         writestring( "VK_F20", outfile);
         break;
```

Listing 7.1 (continued)

```
      case VK_F21:
         writestring( "VK_F21", outfile);
         break;

      case VK_F22:
         writestring( "VK_F22", outfile);
         break;

      case VK_F23:
         writestring( "VK_F23", outfile);
         break;

      case VK_F24:
         writestring( "VK_F24", outfile);
         break;

      case VK_NUMLOCK:
         writestring( "VK_NUMLOCK", outfile);
         break;

      case VK_SCROLL:
         writestring( "VK_SCROLL", outfile);
         break;

      default:
         write_word( wEvent, outfile);
         break;

    } /* switch(wEvent) - end */

} /* write_virtual_accel_event - end */


/****************************************************
  Write the character for the current accelerator
****************************************************/
int write_accel_event( WORD wEvent, FILE *outfile) {

   if(((wEvent >= 65) && (wEvent <= 90)) ||
      ((wEvent >= 97) && (wEvent <= 122))) {
      write_char('"', outfile);
      write_char((BYTE) wEvent, outfile);
      write_char('"', outfile);
      return(0);
   }
```

Listing 7.1 (continued)

```
   if((wEvent >= 1) && (wEvent <= 26)) {
      write_char('"', outfile);
      write_char('^', outfile);
      write_char((BYTE) (wEvent + 64), outfile);
      write_char('"', outfile);
      return(0);
   }

   write_word( wEvent, outfile);
   return(1);

} /* write_accel_event - end */


/***************************************************
  Write the flags for the current accelerator
***************************************************/
void write_accel_flags( BYTE flags, FILE *outfile, int ascii_value_used) {

   if(flags & 0x02)
      writestring(", NOINVERT", outfile);
   if(flags & 0x04)
      writestring(", SHIFT", outfile);
   if(flags & 0x08)
      writestring(", CONTROL", outfile);
   if(flags & 0x10)
      writestring(", ALT", outfile);
   if(flags & 0x01)
      writestring(", VIRTKEY", outfile);
   else if (ascii_value_used)
      writestring(", ASCII", outfile);
   writeline("", outfile);

} /* write_accel_flags - end */


/***************************************************
  Process the accelerator resource
***************************************************/
void process_accelerator(int restype, FILE *infile, FILE *outfile) {

   DWORD  reslen;
   struct AccelTableEntry accelhdr;
   int    ascii_value_used;
```

Listing 7.1 (continued)

```
   write_resource_name(infile, outfile);
   write_char(' ', outfile);
   writestring(res_array[restype], outfile);
   get_mem_flags(infile);
   writeline(" {", outfile);

   increase_indent();

   reslen = get_resource_length(infile);

   while(reslen) {

      write_indent(outfile);

      fread(&accelhdr, sizeof(struct AccelTableEntry), 1, infile);
      reslen -= sizeof(struct AccelTableEntry);

      if(accelhdr.fFlags & 0x01)
         write_virtual_accel_event( accelhdr.wEvent, outfile);
      else
         ascii_value_used = write_accel_event( accelhdr.wEvent, outfile);

      writestring(", ", outfile);
      write_word(accelhdr.wId, outfile);
      write_accel_flags( accelhdr.fFlags, outfile, ascii_value_used);

   }

   decrease_indent();

   writeline("}", outfile);

} /* process_accelerator - end */


/***************************************************
  Process the rcdata resource
***************************************************/
void process_rcdata(int restype, FILE *infile, FILE *outfile) {

   DWORD reslen;
   int ch;
   unsigned short ch_count;
```

Listing 7.1 (continued)

```
   write_resource_name(infile, outfile);
   write_char(' ', outfile);
   writestring(res_array[restype], outfile);
   write_mem_flags(infile, outfile);
   writeline(" {", outfile);

   reslen = get_resource_length(infile);
   increase_indent();

   /* copy the data out to the .rc file */
   ch_count = 0;
   while (reslen--) {
      if (ch_count == 0) {
         write_indent(outfile);
         write_char('"', outfile);
      }
      ch = fgetc( infile );
      if ((ch >= 32) && (ch <= 126)) {
         fputc( ch, outfile);
         ++ch_count;
      }
      else {
         write_char('\\', outfile);
         fprintf( outfile, "%o", ch);
         ch_count += 4;
      }
      if (ch_count >= 60) {
         writeline("\"", outfile);
         ch_count = 0;
      }
   }

   /* if last string wasn't terminated with end quotes, do so now */
   if (ch_count > 0)
      writeline("\"", outfile);

   decrease_indent();

   writeline("}", outfile);

} /* process_rcdata - end */
```

Listing 7.1 (continued)

```
/*************************************************
  Search infile for image #image_num of type image_type
*************************************************/
int search_for_image( WORD image_num, FILE *infile, int image_type ) {

   BYTE ch;
   long reslen;
   int  restype;
   WORD dest_image_num;

   fseek(infile, 0, 0);

   ch = get_byte(infile);
   while(!feof(infile)) {

      /* get the resource type */
      if(ch == 0xFF) {

         fread(&restype, sizeof(int), 1, infile);

         /* If it's not the right image type, skip it */
         if (restype != image_type) {

            get_resource_name(infile);
            get_mem_flags(infile);
            reslen = get_resource_length(infile);

            fseek(infile, reslen, 1);
         }
         else {
            dest_image_num = get_resource_number(infile);
            get_mem_flags(infile);
            reslen = get_resource_length(infile);

            if (dest_image_num == image_num)
               return(1);
            else
               fseek(infile, reslen, 1);
         }

      }
```

Listing 7.1 (continued)

```
      else {

         /* read the name of the resource */
         while((ch = get_byte(infile)) != 0x00) {}

         get_resource_name(infile);
         get_mem_flags(infile);
         reslen = get_resource_length(infile);

         fseek(infile, reslen, 1);
      }

      ch = get_byte(infile);

   } /* while(not eof(infile)) - end */

   return(0);

} /* search_for_image - end */


/*************************************************
  Write the Cursor Directory entry to the cursor file
*************************************************/
void write_cursor_direntry( CURSORDIRENTRY cursorentry, FILE *datafile,
                            FILE *infile, DWORD cursor_size) {

   CURSORRESENTRY cursorres;
   long filepos;
   WORD hotspot;

   cursorres.bWidth      = (BYTE) cursorentry.wWidth;
   cursorres.bHeight     = (BYTE) (cursorentry.wHeight - cursorentry.wWidth);
   cursorres.bColorCount = 0;
   cursorres.bReserved   = 0;

   /* Initialize to 0's */
   cursorres.wXHotSpot = 0;
   cursorres.wYHotSpot = 0;

   /* Save the current position of the input file */
   filepos = ftell(infile);

   /* Search for cursor resource number #wImageOffset to get the hotspots */
   if (search_for_image( cursorentry.wImageOffset, infile, CURSOR_TYPE)) {
      hotspot = get_word( infile );
      cursorres.wXHotSpot = hotspot;
```

Listing 7.1 (continued)

```
      hotspot = get_word( infile );
      cursorres.wYHotSpot = hotspot;
   }

   /* return to the prior position in the input file */
   fseek(infile, filepos, 0);

   /* subtract size of 2 WORD values - X & Y Hot Spot - they occur */
   /* at the beginning of the cursor resource data, but really     */
   /* belong in the header, so they get subtracted from the length */
   cursorres.dwBytesInRes  = cursorentry.dwBytesInRes - (2 * sizeof(WORD));

   cursorres.dwImageOffset = cursor_size;

   fwrite( &cursorres, sizeof(CURSORRESENTRY), 1, datafile);

} /* write_cursor_direntry - end */


/*****************************************************
  Process the group cursor resource
*****************************************************/
void process_group_cursor(FILE *infile, FILE *outfile) {

   DWORD reslen;
   char  datafilename[13];
   FILE  *datafile;
   long  CurrPos;
   DWORD cursor_size;

   CURSORHEADER   cursorinfo;
   CURSORDIRENTRY cursorentry;
   WORD count;

   write_resource_name(infile, outfile);
   writestring(" CURSOR", outfile);

   /* Now write out the memory flags for the previous resource */
   /* (cursor), since that has the correct value; this must be */
   /* done before the succeeding call to get_mem_flags, since  */
   /* that will change the value of prev_mem_flag.             */
   write_special_mem_flag_values( prev_mem_flag, outfile);

   /* Now skip over the memory flag for this resource */
   get_mem_flags(infile);
```

Listing 7.1 (continued)

```
write_char(' ', outfile);

reslen = get_resource_length(infile);

/* Determine a unique .cur filename in the current directory */
strcpy(datafilename, get_data_filename("CU", "CUR"));
if(strlen(datafilename) == 0) {
   writeline("<Unable to open output file>", outfile);
   fseek( infile, reslen, 1);
   return;
}

/* Write the name of the new cursor file to the .rc file */
writeline( datafilename, outfile );

/* open .cur output file */
if((datafile = fopen(datafilename, "wb")) == NULL) {
   printf("Unable to open file %s\n", datafilename);
   fseek( infile, reslen, 1);
   return;
}

fread( &cursorinfo, sizeof(CURSORHEADER), 1, infile);
fwrite( &cursorinfo, sizeof(CURSORHEADER), 1, datafile);

cursor_size = sizeof(CURSORHEADER) +
              (sizeof(CURSORRESENTRY) * cursorinfo.cdCount);
CurrPos = ftell(infile);

/* Loop through each Cursor entry in the Group Cursor resource */
count = cursorinfo.cdCount;
while (count--) {

   /* Read the header for this cursor, and save to the output file */
   fread( &cursorentry, sizeof(CURSORDIRENTRY), 1, infile);

   write_cursor_direntry( cursorentry, datafile, infile, cursor_size);
   cursor_size += cursorentry.dwBytesInRes;

}

fseek(infile, CurrPos, 0);

/* Loop through each Cursor entry in the Group Cursor resource */
count = cursorinfo.cdCount;
while (count--) {
```

Listing 7.1 (continued)

```
      fread( &cursorentry, sizeof(CURSORDIRENTRY), 1, infile);

      CurrPos += sizeof(CURSORDIRENTRY);

      /* Search for cursor resource number #wImageOffset */
      if (search_for_image( cursorentry.wImageOffset, infile, CURSOR_TYPE)) {

         /* skip the 2 WORDS for XHotSpot and YHotSpot */
         fseek(infile, 4, 1);

         /* subtract the size of 2 WORDs at the start: hotspot data */
         write_data( infile, datafile,
                     (cursorentry.dwBytesInRes - (2 * sizeof(WORD))));
      }

      fseek(infile, CurrPos, 0);

   }

   fclose( datafile );

} /* process_group_cursor - end */


/***************************************************
  Write the Icon Directory entry to the icon file
***************************************************/
void write_icon_direntry( ICONDIRENTRY iconentry, FILE *datafile,
                          DWORD icon_size) {

   ICONRESENTRY iconres;

   iconres.bWidth       = iconentry.bWidth;
   iconres.bHeight      = iconentry.bHeight;
   iconres.bColorCount  = iconentry.bColorCount;
   iconres.bReserved    = iconentry.bReserved;

/* The Planes and BitCount values stored in the .ico file
   seem to be ignored.

   iconres.wPlanes      = iconentry.wPlanes;
   iconres.wBitCount    = iconentry.wBitCount;

*/
```

Listing 7.1 (continued)

```
   iconres.wPlanes       = 0;
   iconres.wBitCount     = 0;

   iconres.dwBytesInRes  = iconentry.dwBytesInRes;
   iconres.dwImageOffset = icon_size;

   fwrite( &iconres, sizeof(ICONRESENTRY), 1, datafile);

} /* write_icon_direntry - end */


/****************************************************
  Process the group icon resource
*****************************************************/
void process_group_icon(FILE *infile, FILE *outfile) {

   DWORD reslen;
   char  datafilename[13];
   FILE  *datafile;
   long  CurrPos;
   DWORD icon_size;

   ICONHEADER   iconinfo;
   ICONDIRENTRY iconentry;
   WORD count;

   write_resource_name(infile, outfile);
   writestring(" ICON", outfile);

   /* Now write out the memory flags for the previous resource */
   /* (icon), since that has the correct value; this must be   */
   /* done before the succeeding call to get_mem_flags, since  */
   /* that will change the value of prev_mem_flag.             */
   write_special_mem_flag_values( prev_mem_flag, outfile);

   /* Now skip over the memory flag for this resource */
   get_mem_flags(infile);

   write_char(' ', outfile);

   reslen = get_resource_length(infile);

   /* Determine a unique .ico filename in the current directory */
   strcpy(datafilename, get_data_filename("IC", "ICO"));
   if(strlen(datafilename) == 0) {
      writeline("<Unable to open output file>", outfile);
      fseek( infile, reslen, 1);
      return;
   }
```

Listing 7.1 (continued)

```
   /* Write the name of the new icon file to the .rc file */
   writeline( datafilename, outfile );

   /* Open the output file containing the icon resource data */
   if((datafile = fopen(datafilename, "wb")) == NULL) {
      printf("Unable to open file %s\n", datafilename);
      fseek( infile, reslen, 1);
      return;
   }

   fread( &iconinfo, sizeof(ICONHEADER), 1, infile);
   fwrite( &iconinfo, sizeof(ICONHEADER), 1, datafile);

   icon_size = sizeof(ICONHEADER) +
               (sizeof(ICONRESENTRY) * iconinfo.idCount);
   CurrPos = ftell(infile);

   /* Loop through each Icon entry in the Group Icon resource */
   count = iconinfo.idCount;
   while (count--) {

      fread( &iconentry, sizeof(ICONDIRENTRY), 1, infile);

      write_icon_direntry( iconentry, datafile, icon_size);
      icon_size += iconentry.dwBytesInRes;

   }

   fseek(infile, CurrPos, 0);

   /* Loop through each Icon entry in the Group Icon resource */
   count = iconinfo.idCount;
   while (count--) {

      fread( &iconentry, sizeof(ICONDIRENTRY), 1, infile);

      CurrPos += sizeof(ICONDIRENTRY);

      /* Search for icon resource number #wImageOffset */
      if (search_for_image( iconentry.wImageOffset, infile, ICON_TYPE)) {
         write_data( infile, datafile, iconentry.dwBytesInRes );
      }

      fseek(infile, CurrPos, 0);

   }

   fclose( datafile );

} /* process_group_icon - end */
```

Listing 7.1 (continued)

```
/**************************************************
  Save a user-defined resource data to a file.
**************************************************/
void save_user_resource( FILE *infile, FILE *outfile) {

   DWORD reslen;
   char  datafilename[13];
   FILE  *datafile;

   reslen = get_resource_length(infile);

   strcpy(datafilename, get_data_filename("UR", "USR"));
   if(strlen(datafilename) == 0) {
      writeline("<Unable to open output file>", outfile);
      fseek( infile, reslen, 1);
      return;
   }

   writeline( datafilename, outfile );

   if((datafile = fopen(datafilename, "wb")) == NULL) {
      printf("Unable to open file %s\n", datafilename);
      fseek( infile, reslen, 1);
      return;
   }

   write_data(infile, datafile, reslen);
   fclose( datafile );

} /* save_user_resource - end */


/**************************************************
  Process a user-defined resource (by number).
**************************************************/
void process_user_resource_num(int restype, FILE *infile, FILE *outfile) {

   write_resource_name(infile, outfile);
   write_char(' ', outfile);
   write_number( restype, outfile);
   write_mem_flags(infile, outfile);
   write_char(' ', outfile);

   save_user_resource( infile, outfile );

} /* process_user_resource_num - end */
```

Listing 7.1 (continued)

```
/*************************************************
  Read the name table, but ignore it.
*************************************************/
void process_name_table( FILE *infile, FILE *outfile) {

   DWORD reslen;

   writeline( "", outfile);
   writeline( "//", outfile);
   writeline( "// Name table found, but ignored.", outfile );
   writeline( "//", outfile);

   get_resource_name(infile);
   get_mem_flags(infile);

   reslen = get_resource_length(infile);
   fseek(infile, reslen, 1);

} /* process_name_table - end */


/*************************************************
  Checks if ver.h has been included; if not, does so
*************************************************/
void check_if_ver_header_included( FILE *outfile ) {

   if (VersionUsed == 0) {

      /* update VersionUsed so <ver.h> can't be included twice */
      VersionUsed = 1;

      writeline( "", outfile);
      writeline( "#include <ver.h>", outfile);

   }

} /* check_if_ver_header_included - end */


/*************************************************
  Process version fileflags
*************************************************/
void write_version_fileflags( DWORD style, FILE *outfile ) {

   WORD first_style = 1;
   WORD line_len = 0;

   writestring( " FILEFLAGS ", outfile);
```

Listing 7.1 (continued)

```
   if (style == 0L)
      writeline( "0x0L", outfile);
   else {
      check_for_dlg_style( VS_FF_DEBUG & style, &first_style,
                           &line_len, "VS_FF_DEBUG", outfile);
      check_for_dlg_style( VS_FF_INFOINFERRED & style, &first_style,
                           &line_len, "VS_FF_INFOINFERRED", outfile);
      check_for_dlg_style( VS_FF_PATCHED & style, &first_style,
                           &line_len, "VS_FF_PATCHED", outfile);
      check_for_dlg_style( VS_FF_PRERELEASE & style, &first_style,
                           &line_len, "VS_FF_PRERELEASE", outfile);
      check_for_dlg_style( VS_FF_PRIVATEBUILD & style, &first_style,
                           &line_len, "VS_FF_PRIVATEBUILD", outfile);
      check_for_dlg_style( VS_FF_SPECIALBUILD & style, &first_style,
                           &line_len, "VS_FF_SPECIALBUILD", outfile);

      writeline( "", outfile);
   }

} /* write_version_fileflags - end */


/***************************************************
  Process the version driver subtype
***************************************************/
void process_version_driver_subtype( DWORD driver_subtype, FILE *outfile ) {

   switch (driver_subtype) {

      case VFT2_UNKNOWN:
         writeline("VFT2_UNKNOWN", outfile);
         break;

      case VFT2_DRV_COMM:
         writeline("VFT2_DRV_COMM", outfile);
         break;

      case VFT2_DRV_PRINTER:
         writeline("VFT2_DRV_PRINTER", outfile);
         break;

      case VFT2_DRV_KEYBOARD:
         writeline("VFT2_DRV_KEYBOARD", outfile);
         break;
```

Listing 7.1 (continued)

```
      case VFT2_DRV_LANGUAGE:
         writeline("VFT2_DRV_LANGUAGE", outfile);
         break;

      case VFT2_DRV_DISPLAY:
         writeline("VFT2_DRV_DISPLAY", outfile);
         break;

      case VFT2_DRV_MOUSE:
         writeline("VFT2_DRV_MOUSE", outfile);
         break;

      case VFT2_DRV_NETWORK:
         writeline("VFT2_DRV_NETWORK", outfile);
         break;

      case VFT2_DRV_SYSTEM:
         writeline("VFT2_DRV_SYSTEM", outfile);
         break;

      case VFT2_DRV_INSTALLABLE:
         writeline("VFT2_DRV_INSTALLABLE", outfile);
         break;

      case VFT2_DRV_SOUND:
         writeline("VFT2_DRV_SOUND", outfile);
         break;

      default:
         fprintf( outfile, "%lu", driver_subtype );
         writeline("", outfile);
         break;

   } /* switch (driver_subtype) - end */

} /* process_version_driver_subtype - end */


/**************************************************
  Process the version font subtype
**************************************************/
void process_version_font_subtype( DWORD font_subtype, FILE *outfile ) {

   switch (font_subtype) {
```

Listing 7.1 (continued)

```
      case VFT2_UNKNOWN:
         writeline("VFT2_UNKNOWN", outfile);
         break;

      case VFT2_FONT_RASTER:
         writeline("VFT2_FONT_RASTER", outfile);
         break;

      case VFT2_FONT_VECTOR:
         writeline("VFT2_FONT_VECTOR", outfile);
         break;

      case VFT2_FONT_TRUETYPE:
         writeline("VFT2_FONT_TRUETYPE", outfile);
         break;

      default:
         fprintf( outfile, "%lu", font_subtype );
         writeline("", outfile);
         break;

   } /* switch (font_subtype) - end */

} /* process_version_font_subtype - end */


/****************************************************
  Process version information root block
****************************************************/
void process_version_root_block( FILE *infile, FILE *outfile) {

   VS_FIXEDFILEINFO infostruct;

   /* Process each field of the root block */
   fread( &infostruct, sizeof(VS_FIXEDFILEINFO), 1, infile);

   fprintf( outfile, " FILEVERSION %u,%u,%u,%u",
               HIWORD(infostruct.dwFileVersionMS),
               LOWORD(infostruct.dwFileVersionMS),
               HIWORD(infostruct.dwFileVersionLS),
               LOWORD(infostruct.dwFileVersionLS));
   writeline( "", outfile);
```

Listing 7.1 (continued)

```
   fprintf( outfile, " PRODUCTVERSION %u,%u,%u,%u",
            HIWORD(infostruct.dwProductVersionMS),
            LOWORD(infostruct.dwProductVersionMS),
            HIWORD(infostruct.dwProductVersionLS),
            LOWORD(infostruct.dwProductVersionLS));
   writeline( "", outfile);

   write_version_fileflags( infostruct.dwFileFlags, outfile );

   if(infostruct.dwFileFlagsMask == VS_FFI_FILEFLAGSMASK)
      fprintf( outfile, " FILEFLAGSMASK VS_FFI_FILEFLAGSMASK");
   else
      fprintf( outfile, " FILEFLAGSMASK %lu", infostruct.dwFileFlagsMask);
   writeline( "", outfile);

   writestring( " FILEOS ", outfile);
   switch (infostruct.dwFileOS) {

      case VOS_UNKNOWN:
         writeline("VOS_UNKNOWN", outfile);
         break;

      case VOS_DOS:
         writeline("VOS_DOS", outfile);
         break;

      case VOS_OS216:
         writeline("VOS_OS216", outfile);
         break;

      case VOS_OS232:
         writeline("VOS_OS232", outfile);
         break;

      case VOS_NT:
         writeline("VOS_NT", outfile);
         break;

      case VOS_DOS_WINDOWS16:
         writeline("VOS_DOS_WINDOWS16", outfile);
         break;

      case VOS_DOS_WINDOWS32:
         writeline("VOS_DOS_WINDOWS32", outfile);
         break;
```

Listing 7.1 (continued)

```
      case VOS_OS216_PM16:
         writeline("VOS_OS216_PM16", outfile);
         break;

      case VOS_OS232_PM32:
         writeline("VOS_OS232_PM32", outfile);
         break;

      case VOS_NT_WINDOWS32:
         writeline("VOS_NT_WINDOWS32", outfile);
         break;

      default:
         fprintf( outfile, "%lu", infostruct.dwFileOS );
         writeline("", outfile);
         break;

   } /* switch (FileOS) - end */

   writestring( " FILETYPE ", outfile);
   switch (infostruct.dwFileType) {

      case VFT_UNKNOWN:
         writeline("VFT_UNKNOWN", outfile);
         break;

      case VFT_APP:
         writeline("VFT_APP", outfile);
         break;

      case VFT_DLL:
         writeline("VFT_DLL", outfile);
         break;

      case VFT_DRV:
         writeline("VFT_DRV", outfile);
         break;

      case VFT_FONT:
         writeline("VFT_FONT", outfile);
         break;

      case VFT_VXD:
         writeline("VFT_VXD", outfile);
         break;
```

Listing 7.1 (continued)

```
      case VFT_STATIC_LIB:
         writeline("VFT_STATIC_LIB", outfile);
         break;

      default:
         fprintf( outfile, "%lu", infostruct.dwFileType );
         writeline("", outfile);
         break;

   } /* switch (FileType) - end */

   writestring( " FILESUBTYPE ", outfile);
   switch (infostruct.dwFileType) {

      case VFT_DRV:
         process_version_driver_subtype( infostruct.dwFileSubtype, outfile);
         break;

      case VFT_FONT:
         process_version_font_subtype( infostruct.dwFileSubtype, outfile);
         break;

      default:
         fprintf( outfile, "%lu", infostruct.dwFileSubtype );
         writeline("", outfile);
         break;

   } /* switch (FileSubtype) - end */


} /* process_version_root_block - end */


/*************************************************
  Write out the name of the current block, and return
   whether it was string, variable or other type of
   block.
*************************************************/
int write_block_name( FILE *infile, FILE *outfile ) {

   int  count=1;      /* record number of chars read */
   int  blocktype=0;  /* record which type of block it is */
   BYTE ch;
```

listing 7.1 (continued)

```
   ch = get_byte(infile);
   if (ch == 'S')
      blocktype = STRINGBLOCK;
   else if (ch == 'V')
      blocktype = VARBLOCK;
   else
      blocktype = OTHERBLOCK;

   while((!feof(infile)) && (ch != 0x00)) {
      ++count;
      write_char( ch, outfile);
      ch = get_byte(infile);
   }

   /* Ensure number of characters read is a multiple of 4.    */
   /* According to the MS documentation, this is the format.  */
   /* See "MS Windows 3.1 Programmer's Reference" Vol.4, p.99 */
   count = count % 4;
   count = (count > 0) ? (4 - count) : 0;
   while ((!feof(infile)) && (count > 0)) {
      ch = get_byte(infile);
      --count;
   }

   return(blocktype);

} /* write_block_name - end */


/***************************************************
  Write out the name of the current block, and return
   the total number of characters in the field.
***************************************************/
WORD write_ver_field_name( FILE *infile, FILE *outfile ) {

   WORD count=1;      /* record number of chars read */
   WORD fieldsize=1;  /* record number of chars read */
   BYTE ch;

   ch = get_byte(infile);

   while((!feof(infile)) && (ch != 0x00)) {
      ++count;
      write_char( ch, outfile);
      ch = get_byte(infile);
   }
   fieldsize = count;
```

Listing 7.1 (continued)

```
   /* Ensure number of characters read is a multiple of 4.    */
   /* According to the MS documentation, this is the format.  */
   /* See "MS Windows 3.1 Programmer's Reference" Vol.4, p.99 */
   count = count % 4;
   count = (count > 0) ? (4 - count) : 0;
   while ((!feof(infile)) && (count > 0)) {
      ++fieldsize;
      ch = get_byte(infile);
      --count;
   }

   return(fieldsize);

} /* write_ver_field_name - end */


/*****************************************************
  Read "wordsize" # of bytes and write them to the
   output file; these bytes are the second part of
   the  'VALUE "---", "---"' line in a version info
   string block.  Return the number of bytes needed
   to align the string on a 32-bit boundary.
*****************************************************/
WORD write_ver_field_name_size_n( FILE *infile, FILE *outfile,
                                  WORD wordsize ) {

   WORD count=0;      /* record number of chars read */
   WORD diff=0;
   BYTE ch;

   while((!feof(infile)) && (count < wordsize)) {
      ++count;
      ch = get_byte(infile);
      write_char( ch, outfile);
   }

   /* Ensure number of characters read is a multiple of 4.    */
   /* According to the MS documentation, this is the format.  */
   /* See "MS Windows 3.1 Programmer's Reference" Vol.4, p.99 */
   count = count % 4;
   count = (count > 0) ? (4 - count) : 0;
   diff = count;
   while ((!feof(infile)) && (count > 0)) {
      ch = get_byte(infile);
      --count;
   }

   return(diff);

} /* write_ver_field_name_size_n - end */
```

Listing 7.1 (continued)

```
/****************************************************
  Process a version string block
****************************************************/
void process_version_string_block( WORD blocksize, FILE *infile,
                                   FILE *outfile ) {

   WORD stringblocksize = 0;
   WORD subblocksize = 0;
   WORD subdatasize  = 0;
   WORD fieldsize = 0;
   WORD diff = 0;

   WORD totalblocksize = 0;  /* size so far of the entire block */
   WORD totalsubsize   = 0;  /* size so far of a subblock */

   /* add up: len(StringFileInfo\0) + sizeof(blocksize) +  */
   /* sizeof(datasize) (values already read in)         */
   totalblocksize = 16 + (2 * sizeof(WORD));

   while (totalblocksize < blocksize) {

      /* get the name of this entire string block */
      stringblocksize = get_word( infile );
      (void)get_word( infile );
      totalblocksize += stringblocksize;

      write_indent( outfile );
      writestring( "BLOCK \"", outfile);
      fieldsize = write_ver_field_name( infile, outfile );
      writeline( "\"", outfile);

      write_indent( outfile );
      writeline( "BEGIN", outfile);

      increase_indent();

      totalsubsize = fieldsize + (2 * sizeof(WORD));
      while (totalsubsize < stringblocksize) {

         subblocksize = get_word(infile);
         subdatasize  = get_word(infile);

         /* increment the counter by the size of the current block */
         totalsubsize += subblocksize;

         write_indent( outfile );
         writestring( "VALUE \"", outfile);
         (void)write_ver_field_name( infile, outfile );
         writestring( "\", \"", outfile);
```

Listing 7.1 (continued)

```
         diff = write_ver_field_name_size_n( infile, outfile, subdatasize );

         totalsubsize += diff;
         totalblocksize += diff;

         writeline( "\"", outfile);

      } /* while (totalsubsize < stringblocksize) - end */

      decrease_indent();
      write_indent( outfile );
      writeline( "END", outfile);

   } /* while (totalblocksize < blocksize) - end */

} /* process_version_string_block - end */


/***************************************************
  Process a version variable block
***************************************************/
void process_version_var_block( WORD blocksize, FILE *infile,
                                FILE *outfile ) {

   WORD subblocksize = 0;
   WORD subdatasize  = 0;
   WORD totalsize    = 0;

   WORD langid    = 0;
   WORD charsetid = 0;

   /* add up: len(VarFileInfo\0) + sizeof(blocksize) +  */
   /* sizeof(datasize) (values already read in)         */
   totalsize = 12 + (2 * sizeof(WORD));

   while (totalsize < blocksize) {

      subblocksize = get_word( infile );
      subdatasize  = get_word( infile );

      write_indent( outfile );
      writestring( "VALUE \"", outfile);
      (void)write_block_name( infile, outfile );
      writestring( "\", ", outfile);

      /* increment the byte counter by the size of the current block */
      totalsize += subblocksize;
```

Listing 7.1 (continued)

```
      while (subdatasize) {

         langid = get_word( infile );
         charsetid = get_word( infile );

         fprintf( outfile, "0x%X, %d", langid, charsetid);

         /* take off the size of the 2 variables read above */
         subdatasize -= (2 * sizeof(WORD));

         /* if another entry after this one, write out a comma */
         if (subdatasize) {
            writeline( ",", outfile);
            write_indent( outfile );
         }
         else
            writeline( "", outfile);

      } /* while (subdatasize) - end */

   } /* while (totalsize < blocksize) - end */

} /* process_version_var_block - end */


/****************************************************
  Process an unknown type of version block
****************************************************/
void process_version_other_block( WORD blocksize, long currpos,
                                  FILE *infile, FILE *outfile ) {

   long newPos = 0L;

   write_indent( outfile );
   writeline("// Unknown block type - skipping", outfile);

   newPos = currpos + ((long) blocksize);
   fseek( infile, newPos, 0);

} /* process_version_other_block - end */
```

Listing 7.1 (continued)

```
/************************************************
  Process a version block, which can be either a
   StringFileInfo or VarFileInfo block.
************************************************/
void process_version_block( FILE *infile, FILE *outfile ) {

   WORD blocksize = 0;   /* size of the next (complete) block */

   long currpos = 0L;    /* use in case the block type is unknown */

   int  block_type = 0;  /* mark block as StringFileInfo or VarStringInfo */

   INDENT = 0;           /* reset the amount of indentation */

   currpos = ftell(infile);

   blocksize = get_word(infile);
   (void) get_word(infile);     /* skip over the size of current dataset */

   increase_indent();
   write_indent( outfile );

   writestring( "BLOCK \"", outfile);
   block_type = write_block_name( infile, outfile );
   writeline( "\"", outfile);

   write_indent( outfile );
   writeline( "BEGIN", outfile);

   increase_indent();

   /* call the appropriate procedure for block_type */
   switch (block_type) {

      case STRINGBLOCK:
         process_version_string_block( blocksize, infile, outfile );
         break;

      case VARBLOCK:
         process_version_var_block( blocksize, infile, outfile );
         break;

      case OTHERBLOCK:
      default:
         process_version_other_block( blocksize, currpos, infile, outfile );
         break;

   } /* switch (block_type) - end */
```

Listing 7.1 (continued)

```
   decrease_indent();
   write_indent( outfile );

   writeline( "END", outfile);

   decrease_indent();

} /* process_version_block - end */


/****************************************************
  Process version information
****************************************************/
void process_version_info( FILE *infile, FILE *outfile) {

   DWORD reslen = 0L;      /* total size of version info     */
   DWORD endpos = 0L;      /* ftell() of end of version info */
   DWORD currpos = 0L;     /* current position in input file */

   /* see if "#include <ver.h>" has already been written */
   check_if_ver_header_included( outfile );

   /* start writing out the version information */
   write_version_number(infile, outfile);
   writeline( "VERSIONINFO", outfile);

   /* version info doesn't seem to use memory flags, so skip them */
   get_mem_flags(infile);

   reslen = get_resource_length(infile);
   currpos = ftell(infile);
   endpos = currpos + reslen;

   /* get the name and size of the root block - you can ignore */
   /* this because you know what it's going to be.            */
   (void)get_word(infile); /* cbBlock */
   (void)get_word(infile); /* cbValue */

   get_version_name(infile); /* szKey[] */

   process_version_root_block(infile, outfile);

   writeline( "BEGIN", outfile);

   /* now go through all of the remaining blocks */
   currpos = ftell(infile);
   while (currpos < endpos) {
```

Listing 7.1 (continued)

```
      process_version_block( infile, outfile );
      currpos = ftell(infile);

   }

   writeline( "END", outfile);

} /* process_version_info - end */


/***************************************************
  Call the appropriate function for each resource type
***************************************************/
void process_resource_by_number(FILE *infile, FILE *outfile) {

   int restype = 0;

   INDENT = 0;
   fread(&restype, sizeof(int), 1, infile);

   /* If you're in the middle of a string table, and the new resource */
   /* isn't a string table, finish it off */
   if ((restype != STRING_TYPE) && (StringCount)) {
      writeline("}", outfile);
      StringCount = 0;
   }

   switch(restype) {
      case RT_CURSOR:
         process_cursor( infile );
         break;

      case RT_BITMAP:
         process_bitmap(restype, infile, outfile);
         break;

      case RT_ICON:
         process_icon( infile );
         break;

      case RT_MENU:
         process_menu(restype, infile, outfile);
         break;

      case RT_DIALOG:
         process_dialog(restype, infile, outfile);
         break;
```

Listing 7.1 (continued)

```
      case RT_STRING:
         process_string(restype, infile, outfile);
         break;

      case RT_FONTDIR:
         process_fontdir( infile );
         break;

      case RT_FONT:
         process_font(restype, infile, outfile);
         break;

      case RT_ACCELERATOR:
         process_accelerator(restype, infile, outfile);
         break;

      case RT_RCDATA:
         process_rcdata(restype, infile, outfile);
         break;

      case RT_GROUP_CURSOR:
         process_group_cursor(infile, outfile);
         break;

      case RT_GROUP_ICON:
         process_group_icon(infile, outfile);
         break;

      /* name tables aren't used in Win3.1, so no predefined "RT_????" */
      case 15:
         process_name_table(infile, outfile);
         break;

      /* there doesn't seem to be an RT_????? for version info */
      case 16:
         process_version_info( infile, outfile);
         break;

      default:
         process_user_resource_num(restype, infile, outfile);
         break;

   }

} /* process_resource_by_number - end */
```

Listing 7.1 (continued)

```
/***************************************************
   Process user-defined resource (by name)
***************************************************/
void process_resource_by_name( BYTE ch, FILE *infile, FILE *outfile) {

   long  typeid_pos;  /* file position of type id */
   long  nameid_pos;  /* file position of name id */

   typeid_pos = ftell(infile);

   /* Skip the resource name */
   fseek( infile, -1, 1);
   get_resource_name(infile);

   /* Get the name of the resource itself */
   write_resource_name(infile, outfile);
   nameid_pos = ftell(infile);

   write_char(' ', outfile);

   /* now go back to the resource name and print it out */
   fseek( infile, typeid_pos, 0);
   get_custom_type( ch, infile, outfile );

   fseek( infile, nameid_pos, 0);
   write_mem_flags(infile, outfile);
   write_char(' ', outfile);

   save_user_resource( infile, outfile );

} /* process_resource_by_name - end */


/***************************************************
   Check the parameters passed on program invokation
***************************************************/
void check_usage(int argc) {

   if (argc != 3) {
      printf("Usage: listrec <.res filename> <.rc output filename>\n");
      exit(1);
   }

} /* check_usage - end */
```

Listing 7.1 (continued)

```
/***************************************************
   Write the header to the output file.
***************************************************/
void write_header( char *infname, char *outfname, FILE *outfile ) {

   writeline( "//", outfile);
   writestring( "// ", outfile);
   writestring( outfname, outfile);
   writestring( " - resource file decompiled from ", outfile);
   writeline( infname, outfile);
   writeline( "//", outfile);
   writeline( "#include <windows.h>", outfile);

} /* write_header - end */


/***************************************************
   Check if input file is a Win32 resource file
***************************************************/
void check_for_win32_res( FILE *infile ) {

   char ch;

   ch = get_byte(infile);
   if(ch == 0x00) {
      printf("Input file is a Win32 .res file.  Stopping.\n");
      exit(1);
   }
   rewind(infile);

} /* check_for_win32_res - end */


/***************************************************
   Read .res file and process each resource.
***************************************************/
int main(int argc, char *argv[]) {

   FILE *infile;
   FILE *outfile;
   BYTE ch;

   check_usage(argc);

   if((infile = fopen(argv[1], "rb")) == NULL) {
      printf("Error: Unable to open input file.\nStopping.\n");
      exit(1);   }
```

Listing 7.1 (continued)

```
   check_for_win32_res( infile );

   if((outfile = fopen(argv[2], "wb")) == NULL) {
      printf("Error: Unable to open output file.\nStopping.\n");
      exit(1);
   }

   write_header( argv[1], argv[2], outfile );

   StringCount   = 0;

   ch = get_byte(infile);

   while(!feof(infile)) {

      /* get the resource type */
      if(ch == 0xFF)
         process_resource_by_number(infile, outfile);
      else
         process_resource_by_name(ch, infile, outfile);

      ch = get_byte(infile);

   } /* while(not eof(infile)) - end */
   finish_off_stringtable(outfile);

   fclose(infile);

   return(0);

} /* main - end */

/* Res2Rc.c - end */
```

Chapter 8

PIF File Format

In this chapter, I'll take a look a look at the format of .PIF files. Our thanks go to Mike Maurice for his original work on this topic, published in Andrew Schulman's "Undocumented Corner", *Dr. Dobb's Journal*, July 1993. We'd also like to thank Jonathan Erickson, *DDJ* editor, for allowing us to use the information in that article. After describing the file format, I'll present a DOS program that retrieves data, which can be set through Microsoft's PIF Editor (included with Windows 3.1), from a PIF file.

As operating systems go, DOS is old. In the rapidly evolving world of computers, most software starts to show its age within a few years. Software that's still in use after more than 12 years is almost unheard of, but you can't ignore it — DOS is still around. Forget the version numbers; if you want your software to maintain backward compatibility (and DOS does), you soon realize that all you can really do is tack on bells and whistles. Same face, more makeup. DOS should have died years ago, but traces of it can even be found in Windows 95, and speaking of Windows...

When Microsoft introduced Windows, they realized it needed the ability to run DOS applications; otherwise, they would have lost a huge software base. So they introduced Program Information Files (PIFs), which allow a user to run DOS executables from within Windows. Microsoft includes a program called PIF Editor with each and every copy of Windows, just so you can run DOS and Windows programs alike all day long without leaving your comfortable GUI. PIF files contain information such as how much memory to use, background priority, and what type of graphics mode to use. The need for PIF files is rooted in the difference between DOS and the Windows operating environment. Windows executables contain a lot more information than their DOS counterparts. DOS executables start off with a relatively small header (0x3C bytes) that contains all the information it needs, such as the file size and signature word, before it runs the program. But this header is too small for Windows. The header for each executable is designed so that if the word value at 0x18 is 0x40 or greater, the word value at 0x3C is an offset to a Windows header. This second header includes such information as the segment table and resource table. In order for Windows to run DOS programs, it needs more information than is provided in the DOS header. This information is provided through a PIF file. Because there are some differences between Standard Mode and Enhanced Mode in Windows, PIF files allow you to specify different values for the two modes for some fields (but not all). For more detailed information on DOS and Windows executables, see Chapter 6, "Executable-File Header Format" of the *Microsoft Windows 3.1 Programmer's Reference,* Volume 4, *Resource.*

The Format

You can think of PIF files as series of blocks, where each block contains five pieces of information. The first is a 16-byte string, containing the title of the block. This is followed by three `WORD`s: offset to the next block and the offset and size of the current block. This is followed by the data record for the block. The only exception to this structure is the first block in the file, in which the data record comes first.

The known acceptable values for the title of a block are "MICROSOFT PIFEX", "WINDOWS 286 3.0", "WINDOWS 386 3.0" and "WINDOWS NT 3.1". The first block in the PIF file is always labeled "MICROSOFT PIFEX", and always occurs in the same place. This allows a program to verify that the file is a PIF. As mentioned previously, this block is slightly different than the rest, in that the data record comes first (at the beginning of the PIF). The size of this block is 0x171 (369) bytes. It contains all the information common to Standard Mode and Enhanced Mode in Windows, plus several fields specific to Standard Mode. If you run PIF Editor, you will notice several fields must be the same between the two modes, such as "Window Title". This is the type of information stored in the "MICROSOFT PIFEX" block.

The block labeled "WINDOWS 286 3.0" contains only information relevant to Standard Mode operation of Windows, such as whether the program directly modifies the keyboard. The size of the data record for this block is 6 bytes. Sometimes more than one block in the PIF file will have a similar title, but the first "W" will be zeroed out. These appear to be unused; there will be a block that does not have the "W" zeroed.

Another block type is labeled "WINDOWS 386 3.0", and is specific to Enhanced Mode in Windows. The size of this data record is 0x68 (104) bytes. If this or the 286 structure seems small, it is because both structures store much of their information as single bits. Both this structure and the 286 structure are documented in `PIFSTRUC.H.`

Windows NT has its own block, labeled "WINDOWS NT 3.1" (note the two spaces between NT and 3.1). The format of the data record for this block is currently unknown but appears to be 0x8C (140) bytes long.

Another type of block supported by PIF Editor is the "COMMENT" block. If you add the appropriate offsets and size, you can create a block containing comments about the PIF file.

Several new types of blocks have been found in Windows 95 .PIF files since the original article appeared in *Dr. Dobb's Journal.* These have appeared in various prereleases, so some may now be obsolete. These include "WINDOWS PIF.403" (with a length of 0x180 bytes), "WINDOWS PIF.402" (0x17A bytes), "WINDOWS ICO.001" (0x2A0 bytes) and "WINDOWS VMM 4.0" (0x1AC bytes). The formats of the data record for these blocks are unknown at this time.

The Program

We've written the DOS program PIFDUMP (Listing 8.1), which you can use to examine all of the fields in a PIF file. You can change the PIF fields with Microsoft's PIF Editor. PIFDUMP requires two arguments:

1. either -2 or -3, for information relevant to Standard Mode or Enhanced Mode, respectively (information common to both modes is always presented), and
2. the name of the input PIF file.

We tried to make the format of the output mimic the order of appearance of the data in PIF Editor as closely as possible, but this was strictly an arbitrary decision. For example, if you want all information relevant to Enhanced mode for a file called doom.pif, you would enter

```
PIFDUMP -3 doom.pif
```

The code is fairly straightforward. All of the information relevant to the format of the PIF file is contained in PIFSTRUC.H. You might notice several structures appear to be declared twice. As mentioned before, much of the data in a PIF file is encoded at the bit level. Some compilers complained when we passed these bits around, so we commented out the original declaration of these structures and added another structure (with the same name) that used BYTEs instead of bits. Also, when these structures were later referenced in PIFSTRUC.H as part of the declaration of a larger structure, we substituted the appropriate number of BYTEs. For example, a structure call HOTKEY contains hotkey information. If you look at the original declaration (in the comments), it uses 16 bits, or 2 bytes. Immediately after the comment in which the declaration appears, there is a declaration for a structure called HOTKEY that uses 16 bytes. The latter is used in PIFDUMP.C. The HOTKEY structure was originally apart of the DATA386 structure. It was replaced with 2 bytes to hold the data; in PIFDUMP.C, there is a function that uses those 2 bytes to fill in the HOTKEY structure. Similar functions exist for the other structures containing bit-level information.

Where Do I Go from Here?

An interesting use of the information in this chapter would be to write a Windows program for running DOS applications, but allowing the user to specify PIF settings. The program could then create and write a PIF file on-the-fly and then run the application in a manner reflecting the user's preferences.

Listing 8.1 PIFDUMP.C — Extracts information from a Windows PIF file.

```
/*********************************************************************
 *
 * PROGRAM: PIFDUMP.C
 *
 * PURPOSE: This program extracts information from an MS Windows
 *          PIF file (either 386 or 286 mode)
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 8, PIF File Format, from Undocumented Windows File Formats,
 * published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 *********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "windows.h"
#include "pifstruc.h"

#define SUCCESS       0
#define ERROR_FOUND   1
```

Listing 8.1 (continued)

```
/*****************************************************************\
 *                                                               *
 * Check how the program was called. Required parameters are    *
 * the mode of information to extract (-3 for 386, -2 for 286)   *
 * and the input filename.                                       *
 *                                                               *
\*****************************************************************/
int check_usage(int argc, char *argv[]) {

   int dump_type=0;
   if (argc == 3) {

      if (!strcmp(argv[1], "-2"))
         dump_type = 2;
      else if (!strcmp(argv[1], "-3"))
         dump_type = 3;

   } /* if (argc == 3) - end */

   if (dump_type == 0) {
      printf("Usage:  pifdump < -3 | -2 > <infile>\n");
      exit(0);
   }

   return(dump_type);

} /* check_usage - end */

/*****************************************************************\
 *                                                               *
 * Strip out trailing spaces from text_string.                   *
 *                                                               *
\*****************************************************************/
void trim(char text_string[], int size) {

   /* check for a non-positive size */
   if (size < 1) return;

   /* find last non-blank character, then set next char. to null */
   for (--size; (--size >= 0) && (text_string[size] != ' '); ) {}
   text_string[++size] = '\0';

} /* trim - end */

/*****************************************************************\
 *                                                               *
 * If test_flag is non-zero, print str1, else print str2.        *
 *                                                               *
\*****************************************************************/
void print_flag(WORD test_flag, char *str1, char *str2) {

   if (test_flag)
      puts(str1);
   else
      puts(str2);

} /* print_flag - end */
```

Listing 8.1 (continued)

```
/****************************************************************\
 *                                                              *
 * Convert the hotkey from the PIF file into a readable character.*
 *                                                              *
\****************************************************************/
void convert_hotkey( WORD hotkey, WORD num_flag) {

   switch(hotkey) {

      case 30: putchar('a');    break;
      case 48: putchar('b');    break;
      case 46: putchar('c');    break;
      case 32: putchar('d');    break;
      case 18: putchar('e');    break;
      case 33: putchar('f');    break;
      case 34: putchar('g');    break;
      case 35: putchar('h');    break;
      case 23: putchar('i');    break;
      case 36: putchar('j');    break;
      case 37: putchar('k');    break;
      case 38: putchar('l');    break;
      case 50: putchar('m');    break;
      case 49: putchar('n');    break;
      case 24: putchar('o');    break;
      case 25: putchar('p');    break;
      case 16: putchar('q');    break;
      case 19: putchar('r');    break;
      case 31: putchar('s');    break;
      case 20: putchar('t');    break;
      case 22: putchar('u');    break;
      case 47: putchar('v');    break;
      case 17: putchar('w');    break;
      case 45: putchar('x');    break;
      case 21: putchar('y');    break;
      case 44: putchar('z');    break;
      case 2: putchar('1');     break;
      case 3: putchar('2');     break;
      case 4: putchar('3');     break;
      case 5: putchar('4');     break;
      case 6: putchar('5');     break;
      case 7: putchar('6');     break;
      case 8: putchar('7');     break;
      case 9: putchar('8');     break;
      case 10: putchar('9');    break;
      case 11: putchar('0');    break;
```

Listing 8.1 (continued)

```
      case 27: putchar(']');    break;
      case 26: putchar('[');    break;
      case 39: putchar(';');    break;
      case 40: putchar('\'');   break;
      case 41: putchar('`');    break;
      case 51: putchar(',');    break;
      case 52: putchar('.');    break;
      case 53: print_flag(num_flag, "Num /", "/"); break;
      case 12: putchar('-');    break;
      case 13: putchar('=');    break;
      case 43: putchar('\\');   break;
      case 59: puts("F1");      break;
      case 60: puts("F2");      break;
      case 61: puts("F3");      break;
      case 62: puts("F4");      break;
      case 63: puts("F5");      break;
      case 64: puts("F6");      break;
      case 65: puts("F7");      break;
      case 66: puts("F8");      break;
      case 67: puts("F9");      break;
      case 68: puts("F10");     break;
      case 87: puts("F11");     break;
      case 88: puts("F12");     break;
      case 78: puts("Num +");   break;
      case 69: puts("NumLock"); break;
      case 76: puts("Num 5");   break;
      case 74: puts("Num -");   break;

      case 82: print_flag(num_flag, "Insert", "Num 0");  break;
      case 70: print_flag(num_flag, "Break", "Scroll Lock");  break;
      case 71: print_flag(num_flag, "Home", "Num 7");  break;
      case 72: print_flag(num_flag, "Up", "Num 8");  break;
      case 73: print_flag(num_flag, "Page Up", "Num 9");  break;
      case 75: print_flag(num_flag, "Left", "Num 4");  break;
      case 77: print_flag(num_flag, "Right", "Num 6");  break;
      case 79: print_flag(num_flag, "End", "Num 1");  break;
      case 80: print_flag(num_flag, "Down", "Num 2");  break;
      case 81: print_flag(num_flag, "Page Down", "Num 3");  break;
      case 83: print_flag(num_flag, "Delete", "Num Del");  break;

      default: printf("<Unknown: %d>", hotkey); break;
   }
   putchar('\n');

} /* convert_hotkey - end */
```

Listing 8.1 (continued)

```
/*****************************************************************\
 *                                                               *
 * Use the 8 bits in the 286-Flags BYTE to fill in the 8 bytes of *
 * the FLAGS286 structure.                                       *
 *                                                               *
\*****************************************************************/
void fill_flags286( BYTE flags286, FLAGS286 *f286_data) {

   /* The comment at the end of each line converts decimal to binary */
   f286_data->AltTab286    = (BYTE)(flags286 & 1);    /* 1   = 00000001 */
   f286_data->AltEsc286    = (BYTE)(flags286 & 2);    /* 2   = 00000010 */
   f286_data->AltPrtScr286 = (BYTE)(flags286 & 4);    /* 4   = 00000100 */
   f286_data->PrtScr286    = (BYTE)(flags286 & 8);    /* 8   = 00001000 */
   f286_data->CtrlEsc286   = (BYTE)(flags286 & 16);   /* 16  = 00010000 */
   f286_data->NoSaveScreen = (BYTE)(flags286 & 32);   /* 32  = 00100000 */
   f286_data->Unused10[0]  = (BYTE)(flags286 & 64);   /* 64  = 01000000 */
   f286_data->Unused10[1]  = (BYTE)(flags286 & 128);  /* 128 = 10000000 */

} /* fill_flags286 - end */


/*****************************************************************\
 *                                                               *
 * Use the 8 bits in the COM Ports BYTE to fill in the 8 BYTES of *
 * the COMPORT structure.                                        *
 *                                                               *
\*****************************************************************/
void fill_com_ports( BYTE comports, COMPORT *com_ports) {

   /* The comment at the end of each line converts decimal to binary */
   com_ports->Unused11[0]  = (BYTE)(comports & 1);    /* 1   = 00000001 */
   com_ports->Unused11[1]  = (BYTE)(comports & 2);    /* 2   = 00000010 */
   com_ports->Unused11[2]  = (BYTE)(comports & 4);    /* 4   = 00000100 */
   com_ports->Unused11[3]  = (BYTE)(comports & 8);    /* 8   = 00001000 */
   com_ports->Unused11[4]  = (BYTE)(comports & 16);   /* 16  = 00010000 */
   com_ports->Unused11[5]  = (BYTE)(comports & 32);   /* 32  = 00100000 */
   com_ports->Com3         = (BYTE)(comports & 64);   /* 64  = 01000000 */
   com_ports->Com4         = (BYTE)(comports & 128);  /* 128 = 10000000 */

} /* fill_com_ports - end */
```

Listing 8.1 (continued)

```
/****************************************************************\
 *                                                              *
 * Use the 16 bits in the video[2] bytes to fill in the 16 bytes *
 * of the VIDEO structure.                                      *
 *                                                              *
\****************************************************************/
void fill_video( BYTE video[2], VIDEO *video_data) {

   /* The comment at the end of each line converts decimal to binary */
   video_data->EmulateText  = (BYTE)(video[0] & 1);    /* 1   = 00000001 */
   video_data->MonitorText  = (BYTE)(video[0] & 2);    /* 2   = 00000010 */
   video_data->MonitorLoGr  = (BYTE)(video[0] & 4);    /* 4   = 00000100 */
   video_data->MonitorHiGr  = (BYTE)(video[0] & 8);    /* 8   = 00001000 */
   video_data->InitModeText = (BYTE)(video[0] & 16);   /* 16  = 00010000 */
   video_data->InitModeLoGr = (BYTE)(video[0] & 32);   /* 32  = 00100000 */
   video_data->InitModeHiGr = (BYTE)(video[0] & 64);   /* 64  = 01000000 */
   video_data->RetainVideo  = (BYTE)(video[0] & 128);  /* 128 = 10000000 */

   video_data->VideoUnused[0] = (BYTE)(video[1] & 1);    /* 1   = 00000001 */
   video_data->VideoUnused[1] = (BYTE)(video[1] & 2);    /* 2   = 00000010 */
   video_data->VideoUnused[2] = (BYTE)(video[1] & 4);    /* 4   = 00000100 */
   video_data->VideoUnused[3] = (BYTE)(video[1] & 8);    /* 8   = 00001000 */
   video_data->VideoUnused[4] = (BYTE)(video[1] & 16);   /* 16  = 00010000 */
   video_data->VideoUnused[5] = (BYTE)(video[1] & 32);   /* 32  = 00100000 */
   video_data->VideoUnused[6] = (BYTE)(video[1] & 64);   /* 64  = 01000000 */
   video_data->VideoUnused[7] = (BYTE)(video[1] & 128);  /* 128 = 10000000 */

} /* fill_video - end */


/****************************************************************\
 *                                                              *
 * Use the 16 bits in the hotkey[2] bytes to fill in the 16 bytes *
 * of the HOTKEY structure.                                     *
 *                                                              *
\****************************************************************/
void fill_hotkey( BYTE hotkey[2], HOTKEY *hotkey_data) {

   /* The comment at the end of each line converts decimal to binary */
   hotkey_data->HOT_KEYSHIFT = (BYTE)(hotkey[0] & 1);    /* 1   = 00000001 */
   hotkey_data->Unused4      = (BYTE)(hotkey[0] & 2);    /* 2   = 00000010 */
   hotkey_data->HOT_KEYCTRL  = (BYTE)(hotkey[0] & 4);    /* 4   = 00000100 */
   hotkey_data->HOT_KEYALT   = (BYTE)(hotkey[0] & 8);    /* 8   = 00001000 */
   hotkey_data->Unused5[0]   = (BYTE)(hotkey[0] & 16);   /* 16  = 00010000 */
   hotkey_data->Unused5[1]   = (BYTE)(hotkey[0] & 32);   /* 32  = 00100000 */
   hotkey_data->Unused5[2]   = (BYTE)(hotkey[0] & 64);   /* 64  = 01000000 */
   hotkey_data->Unused5[3]   = (BYTE)(hotkey[0] & 128);  /* 128 = 10000000 */
```

Listing 8.1 (continued)

```
    hotkey_data->Unused5[4]   = (BYTE)(hotkey[0] & 1);    /* 1   = 00000001 */
    hotkey_data->Unused5[5]   = (BYTE)(hotkey[0] & 2);    /* 2   = 00000010 */
    hotkey_data->Unused5[6]   = (BYTE)(hotkey[0] & 4);    /* 4   = 00000100 */
    hotkey_data->Unused5[7]   = (BYTE)(hotkey[0] & 8);    /* 8   = 00001000 */
    hotkey_data->Unused5[8]   = (BYTE)(hotkey[0] & 16);   /* 16  = 00010000 */
    hotkey_data->Unused5[9]   = (BYTE)(hotkey[0] & 32);   /* 32  = 00100000 */
    hotkey_data->Unused5[10]  = (BYTE)(hotkey[0] & 64);   /* 64  = 01000000 */
    hotkey_data->Unused5[11]  = (BYTE)(hotkey[0] & 128);  /* 128 = 10000000 */

} /* fill_hotkey - end */


/******************************************************************\
 *                                                                *
 * Use the 16 bits in the flags_XMS[2] bytes to fill in the 16    *
 * bytes of the FLAGSXMS structure.                               *
 *                                                                *
\******************************************************************/
void fill_flagsxms( BYTE fxms[2], FLAGSXMS *xms_data) {

    /* The comment at the end of each line converts decimal to binary */
    xms_data->XMS_Locked    = (BYTE)(fxms[0] & 1);    /* 1   = 00000001 */
    xms_data->Allow_FastPst = (BYTE)(fxms[0] & 2);    /* 2   = 00000010 */
    xms_data->Lock_App      = (BYTE)(fxms[0] & 4);    /* 4   = 00000100 */
    xms_data->Unused3[0]    = (BYTE)(fxms[0] & 8);    /* 8   = 00001000 */
    xms_data->Unused3[1]    = (BYTE)(fxms[0] & 16);   /* 16  = 00010000 */
    xms_data->Unused3[2]    = (BYTE)(fxms[0] & 32);   /* 32  = 00100000 */
    xms_data->Unused3[3]    = (BYTE)(fxms[0] & 64);   /* 64  = 01000000 */
    xms_data->Unused3[4]    = (BYTE)(fxms[0] & 128);  /* 128 = 10000000 */

    xms_data->Unused3[5]    = (BYTE)(fxms[0] & 1);    /* 1   = 00000001 */
    xms_data->Unused3[6]    = (BYTE)(fxms[0] & 2);    /* 2   = 00000010 */
    xms_data->Unused3[7]    = (BYTE)(fxms[0] & 4);    /* 4   = 00000100 */
    xms_data->Unused3[8]    = (BYTE)(fxms[0] & 8);    /* 8   = 00001000 */
    xms_data->Unused3[9]    = (BYTE)(fxms[0] & 16);   /* 16  = 00010000 */
    xms_data->Unused3[10]   = (BYTE)(fxms[0] & 32);   /* 32  = 00100000 */
    xms_data->Unused3[11]   = (BYTE)(fxms[0] & 64);   /* 64  = 01000000 */
    xms_data->Unused3[12]   = (BYTE)(fxms[0] & 128);  /* 128 = 10000000 */

} /* fill_flagsxms - end */
```

Listing 8.1 (continued)

```
/*****************************************************************\
 *                                                               *
 * Use the 16 bits in the 386-Flags[2] bytes to fill in the 16   *
 * bytes of the FLAGS386 structure.                              *
 *                                                               *
\*****************************************************************/
void fill_flags386( BYTE flags386[2], FLAGS386 *f386_data) {

   /* The comment at the end of each line converts decimal to binary */
   f386_data->AllowCloseAct = (BYTE)(flags386[0] & 1);   /* 1   = 00000001 */
   f386_data->BackgroundOn  = (BYTE)(flags386[0] & 2);   /* 2   = 00000010 */
   f386_data->ExclusiveOn   = (BYTE)(flags386[0] & 4);   /* 4   = 00000100 */
   f386_data->FullScreenYes = (BYTE)(flags386[0] & 8);   /* 8   = 00001000 */
   f386_data->Unused0       = (BYTE)(flags386[0] & 16);  /* 16  = 00010000 */
   f386_data->SK_AltTab     = (BYTE)(flags386[0] & 32);  /* 32  = 00100000 */
   f386_data->SK_AltEsc     = (BYTE)(flags386[0] & 64);  /* 64  = 01000000 */
   f386_data->SK_AltSpace   = (BYTE)(flags386[0] & 128); /* 128 = 10000000 */

   f386_data->SK_AltEnter   = (BYTE)(flags386[1] & 1);   /* 1   = 00000001 */
   f386_data->SK_AltPrtSc   = (BYTE)(flags386[1] & 2);   /* 2   = 00000010 */
   f386_data->SK_PrtSc      = (BYTE)(flags386[1] & 4);   /* 4   = 00000100 */
   f386_data->SK_CtrlEsc    = (BYTE)(flags386[1] & 8);   /* 8   = 00001000 */
   f386_data->Detect_Idle   = (BYTE)(flags386[1] & 16);  /* 16  = 00010000 */
   f386_data->UseHMA        = (BYTE)(flags386[1] & 32);  /* 32  = 00100000 */
   f386_data->Unused1       = (BYTE)(flags386[1] & 64);  /* 64  = 01000000 */
   f386_data->EMS_Locked    = (BYTE)(flags386[1] & 128); /* 128 = 10000000 */

} /* fill_flags386 - end */


/*****************************************************************\
 *                                                               *
 * Use the 8 bits in the PIF close_on_exit BYTE to fill in the   *
 * 8 BYTES of the CLOSEONEXIT structure.                         *
 *                                                               *
\*****************************************************************/
void fill_close_on_exit( BYTE close_on_exit, CLOSEONEXIT *coe_data) {

   /* The comment at the end of each line converts decimal to binary */
   coe_data->Unused0       = (BYTE)(close_on_exit & 1);  /* 1   = 00000001 */
   coe_data->Graph286      = (BYTE)(close_on_exit & 2);  /* 2   = 00000010 */
   coe_data->PreventSwitch = (BYTE)(close_on_exit & 4);  /* 4   = 00000100 */
   coe_data->NoScreenExch  = (BYTE)(close_on_exit & 8);  /* 8   = 00001000 */
   coe_data->Close_OnExit  = (BYTE)(close_on_exit & 16); /* 16  = 00010000 */
   coe_data->Unused1       = (BYTE)(close_on_exit & 32); /* 32  = 00100000 */
   coe_data->Com2          = (BYTE)(close_on_exit & 64); /* 64  = 01000000 */
   coe_data->Com1          = (BYTE)(close_on_exit & 128);/* 128 = 10000000 */

} /* fill_close_on_exit - end */
```

Listing 8.1 (continued)

```
/******************************************************************\
 *                                                                *
 * Search the linked list of records at the end of the PIF file   *
 * for the section with a title of "title".                       *
 *                                                                *
\******************************************************************/
long search_pif_file( char *title, FILE *infile) {

   SECTIONHDR sect_hdr;
   SECTIONNAME sect_name;

   long offset=0L;
   short match_found=0;
   short at_eof=0;

   /* Skip over "MICROSOFT PIFEX" header - it points to the first part */
   /* of the file (in its sect_hdr structure, next_section = 0x187,    */
   /* current_section = 0x0, and size_section = 0x171; next_section =  */
   /* 0x171 (size of section) + 0x10 (size of "MICROSOFT PIFEX\0") +   */
   /* 0x06 (size of section header).                                   */
   fread( &sect_name, sizeof(sect_name), 1, infile);
   fread( &sect_hdr,  sizeof(sect_hdr),  1, infile);

   if (strcmp(sect_name.name_string, "MICROSOFT PIFEX")) {
      printf("Invalid PIF file.  Stopping.\n");
      exit(1);
   }

   /* Now scan through remaining sections in the PIF file */
   while ((!match_found) && (!at_eof)) {

      if (feof(infile))
         at_eof = 1;
      else {

         /* Read in the name and header of the current section */
         fread( &sect_name, sizeof(sect_name), 1, infile);
         fread( &sect_hdr,  sizeof(sect_hdr),  1, infile);

         /* Check if section found, or at eof, or if can't fseek to the */
         /* next section (these 3 actions are mutually exclusive)       */
         if (!strcmp(title, sect_name.name_string))
            match_found = 1;
         else if (sect_hdr.next_section == 0xFFFF)
            at_eof = 1;
```

Listing 8.1 (continued)

```
         else {

            /* Convert a WORD to signed long */
            offset = 0x0000FFFF & sect_hdr.next_section;
            if (fseek(infile, offset, 0)) {
               printf("Unable to fseek to %ld.  Stopping.\n", offset);
               exit(1);
            }

         } /* if (section-not-found && not-at-end) - end */

      } /* if (not at end of file) - end */

   } /* while (match-not-found && not-at-end-of-file) - end */

   if (match_found)
      offset = sect_hdr.current_section;
   else
      offset = -1;

   return(offset);

} /* search_pif_file - end */


/*******************************************************************\
 *                                                                 *
 * Read the first 286 block from the PIF file.                     *
 *                                                                 *
\*******************************************************************/
void read_286_block( DATA286 *data, FILE *infile ) {

   long offset=0L;

   if ((offset = search_pif_file("WINDOWS 286 3.0", infile)) == -1) {
      printf("Error: 286 Section not found.  Stopping.\n");
      exit(1);
   }
   else {

      /* Read 286 section into memory */
      fseek(infile, offset, 0);
      fread( data, sizeof(DATA286), 1, infile);

   }

} /* read_286_block - end */
```

Listing 8.1 (continued)

```
/******************************************************************\
 *                                                                *
 * Read the first 386 block from the PIF file.                    *
 *                                                                *
\******************************************************************/
void read_386_block( DATA386 *data, FILE *infile ) {

   long offset=0L;

   if ((offset = search_pif_file("WINDOWS 386 3.0", infile)) == -1) {
      printf("Error: 386 Section not found.  Stopping.\n");
      exit(1);
   }
   else {

      /* Read 386 section into memory */
      fseek(infile, offset, 0);
      fread( data, sizeof(DATA386), 1, infile);

   }

} /* read_386_block - end */


/******************************************************************\
 *                                                                *
 * Process the PIF input file.  If the user passes in "-2",       *
 * print out "Standard Mode" dump; for "-3", print "Enhanced"     *
 * mode dump.                                                     *
 *                                                                *
\******************************************************************/
void main(int argc, char *argv[]) {

   int  dump_type=0;    /* 2=286 dump, 3=386 dump */

   FILE *infile;

   PIF      pif_header;
   DATA386  data386;
   DATA286  data286;

   CLOSEONEXIT close_onexit_data;
   FLAGS286    f286_data;
   FLAGS386    f386_data;
   COMPORT     com_ports;
   VIDEO       video_data;
   HOTKEY      hotkey_data;
   FLAGSXMS    xms_data;
```

Listing 8.1 (continued)

```
/* Determine (from command line) if user wants 286 or */
/* 386 info; set dump_type to 2 or 3, respectively.   */
dump_type = check_usage(argc, argv);

/* Try to open the file.  If it fails, exit. */
if ((infile = fopen(argv[2], "rb")) == NULL) {
   printf("Input file not found.  Stopping.\n");
   exit(0);
}

/* fopen() was successful, so read in the first header */
printf("Extracting %d86 information from %s.\n\n", dump_type, argv[2]);
fread( &pif_header, sizeof(pif_header), 1, infile);

/* Retrieve 286 or 386 (Standard or Enhanced) info from PIF file */
if (dump_type == 2) {
   read_286_block( &data286, infile);

   fill_flags286(  data286.flags_286, &f286_data);
   fill_com_ports( data286.com_ports, &com_ports);
}
else {
   read_386_block( &data386, infile);

   fill_flags386(  data386.flags_386, &f386_data);
   fill_video( data386.video, &video_data);
   fill_hotkey( data386.hot_key_state, &hotkey_data);
   fill_flagsxms( data386.flags_XMS, &xms_data);
}

/* You're done with the input file, so close it */
fclose(infile);

/* Initialize bit-replacement structures */
fill_close_on_exit( pif_header.close_on_exit, &close_onexit_data);

/* Remove trailing spaces from text fields */
trim(pif_header.prog_path,  sizeof(pif_header.prog_path));
trim(pif_header.title,      sizeof(pif_header.title));
trim(pif_header.def_dir,    sizeof(pif_header.def_dir));
trim(pif_header.prog_param, sizeof(pif_header.prog_param));

trim(pif_header.shared_prog_name, sizeof(pif_header.shared_prog_name));
trim(pif_header.shared_data_file, sizeof(pif_header.shared_data_file));
```

Listing 8.1 (continued)

```
    /* Print out the data common to both groups */
    printf("Program Filename   :  %s\n", pif_header.prog_path);
    printf("Window Title       :  %s\n", pif_header.title);

    /* print optional parameters based on dump_type */
    printf("Opt. Parameters    :  ");
    if (dump_type == 2)
       printf( "%s\n", pif_header.prog_param);
    else
       printf( "%s\n", data386.opt_params);

    printf("Startup Directory  :  %s\n", pif_header.def_dir);

    /* print out video data */
    if (dump_type == 2) {
       printf("Video Mode         :  ");
       if (close_onexit_data.Graph286)
          printf("Graphics/Mult. Text\n");
       else
          printf("Text\n");
    }
    else if (dump_type == 3) {
       printf("Video Memory       :");
       if (video_data.InitModeText)
          printf("  Text");
       if (video_data.InitModeLoGr)
          printf("  Low Graphics");
       if (video_data.InitModeHiGr)
          printf("  High Graphics");
       printf("\n");
    }

    if (dump_type == 3) {

       printf("Memory Requirements:  %dK Required\t %dK Desired\n",
             data386.mem_req, data386.mem_limit);
       printf("EMS Memory         :  %dK Required\t %dK Limit\n",
             data386.ems_min, data386.ems_max);
       printf("XMS Memory         :  %dK Required\t %dK Limit\n",
             data386.xms_min, data386.xms_max);

       if (f386_data.FullScreenYes)
          printf("Display Usage      :  Full Screen\n");
       else
          printf("Display Usage      :  Windowed\n");
          printf("Execution          :  ");
```

Listing 8.1 (continued)

```
      if (f386_data.BackgroundOn)
         printf("Background  ");
      else
         printf("Foreground  ");

      if (f386_data.ExclusiveOn)
         printf("Exclusive\n");
      else
         printf("Non-exclusive\n");

      if (close_onexit_data.Close_OnExit)
         printf("Close Window On Exit\n");
      else
         printf("Don't Close Window On Exit\n");

   }
   else if (dump_type == 2) {

      printf("Memory Requirements:  %dK Required\n", pif_header.min_mem);
      printf("XMS Memory           :  %dK Required\t %dK Limit\n\n",
            data286.xmsReq286, data286.xmsLimit286);

      printf("Directly Modifies:");
      if (close_onexit_data.Com1)
         printf("   Com1");
      if (close_onexit_data.Com2)
         printf("   Com2");
      if (com_ports.Com3)
         printf("   Com3");
      if (com_ports.Com4)
         printf("   Com4");
      if (pif_header.flags1 & 16)    /* Check the 5th bit for keyboard */
         printf("   Keyboard");
      printf("\n");

      if (close_onexit_data.NoScreenExch)
         printf("No Screen Exchange\n");
      else
         printf("Screen Exchange Allowed\n");

      if (close_onexit_data.PreventSwitch)
         printf("Program Switch Prevented\n");
      else
         printf("Program Switch Allowed\n");
```

Listing 8.1 (continued)

```
      if (close_onexit_data.Close_OnExit)
         printf("Close Window On Exit\n");
      else
         printf("Don't Close Window On Exit\n");

      if (f286_data.NoSaveScreen)
         printf("Save Screen is not Enabled\n");
      else
         printf("Save Screen is Enabled\n");

      printf("Reserve Shortcut Keys:");
      if (f286_data.AltTab286)
         printf("  Alt+Tab");
      if (f286_data.AltEsc286)
         printf("  Alt+Esc");
      if (f286_data.CtrlEsc286)
         printf("  Ctrl+Esc");
      if (f286_data.PrtScr286)
         printf("  PrtScr");
      if (f286_data.AltPrtScr286)
         printf("  Alt+PrtScr");
      printf("\n");
   }

   /* Print out the 386 Advanced screen data */
   if (dump_type == 3) {

      printf("\nAdvanced Options:\n\n");

      printf("Multitasking Options:\n");
      printf("Background Priority: %d\n", data386.back_pri);
      printf("Foreground Priority: %d\n", data386.for_pri);
      if (f386_data.Detect_Idle)
         printf("Detect Idle Time\n\n");
      else
         printf("Do not Detect Idle Time\n\n");

      printf("Memory Options:\n");
      if (f386_data.EMS_Locked)
         printf("EMS Memory Locked\n");
      else
         printf("EMS Memory Not Locked\n");

      if (xms_data.XMS_Locked)
         printf("XMS Memory Locked\n");
      else
         printf("XMS Memory Not Locked\n");
```

Listing 8.1 (continued)

```
     if (!f386_data.UseHMA)
        printf("Use High Memory\n");
     else
        printf("Do Not Use High Memory\n");

     if (xms_data.Lock_App)
        printf("Lock Application Memory\n");
     else
        printf("Do Not Lock Application Memory\n");

     printf("\nMonitor Ports:");
     if (!video_data.MonitorText)
        printf("   Text");
     if (!video_data.MonitorLoGr)
        printf("   Low Gr");
     if (!video_data.MonitorHiGr)
        printf("   High Gr");
     if (video_data.EmulateText)
        printf("   Emul. Text");
     if (video_data.RetainVideo)
        printf("   Retain Video Mem");
     printf("\n\n");

     if (xms_data.Allow_FastPst)
        printf("Allow Fast Paste\n");
     else
        printf("Don't Allow Fast Paste\n");

     if (f386_data.AllowCloseAct)
        printf("Allow Close When Active\n");
     else
        printf("Don't Allow Close When Active\n");
```

Listing 8.1 (continued)

```
      printf("\nReserved Shortcut Keys\n");
      if (f386_data.SK_AltTab)
         printf("   Alt+Tab");
      if (f386_data.SK_AltEsc)
         printf("   Alt+Esc");
      if (f386_data.SK_CtrlEsc)
         printf("   Ctrl+Esc");
      if (f386_data.SK_PrtSc)
         printf("   PrtScr");
      if (f386_data.SK_AltPrtSc)
         printf("   Alt+PrtScr");
      if (f386_data.SK_AltSpace)
         printf("   Alt+Space");
      if (f386_data.SK_AltEnter)
         printf("   Alt+Enter");

      printf("\n\nApplication Shortcut Key: ");
      if (data386.hot_key_flag == 0)
         printf("None\n");
      else {
         if (hotkey_data.HOT_KEYALT)
            printf("Alt+");
         if (hotkey_data.HOT_KEYCTRL)
            printf("Ctrl+");
         if (hotkey_data.HOT_KEYSHIFT)
            printf("Shift+");

         convert_hotkey(data386.hot_key_scan, data386.hk_numflag);
      }

   } /* if (dump_type = 3) - end */

} /* main - end */

/* pifdump.c - end */
```

Chapter 9

W3 and W4 File Formats

Overview

Unless you follow the literature on undocumented Windows features, you might not even know what a "W3" or "W4" file is. The W3 file is the file format used by `WIN386.EXE` in Windows 3.x and is actually quite simple. The W4 file is used by `VMM32.VXD` in Windows 95 and is a little more complex than the W3 file.

W3 and W4 files aren't really executable files, in the common sense of the term. They are more like a library. They contain a series of VxDs (LE files), and they're basically a way of packaging together a bunch of the core VxDs for Windows. Essentially, the W3 file contains a directory of all the LE files (by name), with offsets to their locations, and the length of each one.

Like NE, PE, and LE executables, the W3 and W4 have an MZ (MS-DOS compatible executable) stub program, with the W3 or W4 file immediately following it. I'm not really going to get into the details of the MZ stub program. All you really need is a very small piece of the header. Listing 9.1 shows the `MZHEADER` structure and a short routine, `SkipMZ()`, that allows you to seek to the "next" executable in the file. This same routine would work for LE, PE, and NE files, as well.

Listing 9.1 SkipMZ code sample — allows you to seek to the "next" executable in the file.

```
/*********************************************************************
 *
 * PROGRAM: SkipMZ
 *
 * PURPOSE: SkipMZ allows you to seek to the "next" executable in the file.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 9, W3 and W4 File Formats, from Undocumented Windows File Formats,
 * published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 *********************************************************************/

typedef struct tagMZHEADER
{
      char  MZMagic[2]; /* Should always be 'MZ'          */
      char  Stuff[58];  /* Stuff you don't care about     */
      long  OtherOff;   /* Offset to next executable      */
} MZHEADER;

BOOL SkipMZ(FILE *inFile)
{
      MZHEADER MZHeader;

      fread(&MZHeader, sizeof(MZHeader), 1, inFile);
      if (MZHeader.MZMagic[0] != 'M' || MZHeader.MZMagic[1] != 'Z')
      {
            printf("This is not an executable file!\n");
            return FALSE;
      }

      if (!MZHeader.OtherOff)
      {
            printf("This is a DOS executable.\n");
            return FALSE;
      }

      fseek(inFile, MZHeader.OtherOff, SEEK_SET);
      return TRUE;
}
```

The W3 File Format

After the MZ stub, you're basically just concerned with the W3 section of the file. The W3 file itself consists of a header, the VxD directory, and then the VxDs themselves. The header for the W3 file is shown in Table 9.1.

The W3 header is immediately followed by the list of VxDs. The list contains VxDRECORD structures (Table 9.2).

VxDName is simply the eight-character name of the VxD. If the name is fewer than eight characters, it's padded with spaces (0x20). There is no null-terminator. The starting location of the VxD is based on the beginning of the WIN386.EXE file, not the beginning of the W3HEADER record. The VxDHdrSize field provides the size of the VxD header in bytes. Actually, VxDHdrSize includes not only the LEHEADER structure, but everything in the Loader and Fixup sections of the LE file. So for our purposes, VxDHdrSize includes everything from the "LE" signature to the end of the import procedure name table. For more information on the LE file format, see Chapter 10.

How to Unmangle the VxDs

Unfortunately, VxDs within a W3 file are somewhat mangled, although the mangling is fairly minor and easy to rectify. First of all, the DataPages value of LEHEADER is changed to be relative to the beginning of the MZ header for the W3 file. Normally this is relative to the beginning of the MZ header of the LE file. Of course, in a W3 file, the

Table 9.1 W3HEADER record.

Field Name	Data Type	Comments
W3Magic	char[2]	Contains the characters "W3"
WinVer	WORD	Version of Windows (0x30A = Windows 3.1)
NumVxDs	WORD	Number of VxDs in the directory
Reserved	BYTE[10]	Basically filler

Table 9.2 VxDRECORD structure.

Field Name	Data Type	Comments
VxDName	char[8]	Name of VxD, padded at the end with blanks
VxDStart	long	Starting location of VxD in W3 file
VxDHdrSize	long	Size of LE header in VxD

stubs for VxDs have been stripped to save space. The changing of the `DataPages` value is weird though, because none of the other offset fields are changed at all.

The other mangling has to do with the nonresident name table. The nonresident name table is usually the last section of an LE file. In the case of W3 files, however, all nonresident name tables have been removed, so if you extract the LE files, you'll need to build one for it. This is a fairly simple process, however.

SUCKW3

Okay, the name sounds a little strange, but basically SUCKW3 (Listings 9.2 and 9.3) extracts VxDs from a W3 file. Running SUCKW3 with just the name of the W3 file gives you a listing of all the VxDs in the W3 file. Passing a second parameter, the name of a VxD, extracts the VxD into a separate .386 file. You need to make sure you have a `STUB.EXE` program in the same directory, because the LE file will require its own stub program.

Listing 9.2 SUCKW3.H.

```
/*********************************************************************
 *
 * PROGRAM: SUCKW3.H
 *
 * PURPOSE: Extracts VxDs from a W3 file.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 9, W3 and W4 File Formats, from Undocumented Windows File Formats,
 * published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 *********************************************************************/

typedef unsigned short WORD;
typedef unsigned char BYTE;
typedef unsigned long DWORD;
typedef unsigned char BOOL;

#define TRUE      1
#define FALSE     0

typedef struct tagMZHEADER
{
      char  MZMagic[2];
      char  Stuff[58];
      long  OtherOff;} MZHEADER;
```

Listing 9.2 (continued)

```
/* Header for W3 File */
typedef struct tagW3HEADER
{
     char  W3Magic[2];
     WORD  WinVer;
     WORD  NumVxDs;
     BYTE  Reserved[10];
} W3HEADER;

/* Listing for single VxD in W3 directory */
typedef struct tagVxDRECORD
{
     char   VxDName[8];
     long   VxDStart;
     long   VxDHdrSize;
} VxDRECORD;

/* LE Header structure */
typedef struct tagLEHEADER
{
      char  LEMagic[2];
      BYTE  ByteOrder;
      BYTE  WordOrder;
      DWORD FormatLevel;
      WORD  CPUType;
      WORD  OSType;
      DWORD ModuleVer;
      DWORD ModuleFlags;
      DWORD NumPages;
      DWORD EIPObjNum;
      DWORD EIP;
      DWORD ESPObjNum;
      DWORD ESP;
      DWORD PageSize;
      DWORD LastPageSize;
      DWORD FixupSize;
      DWORD FixupChecksum;
      DWORD LoaderSize;
      DWORD LoaderChecksum;
      DWORD ObjTblOffset;
      DWORD NumObjects;
      DWORD ObjPageTbl;
      DWORD ObjIterPage;
      DWORD ResourceTbl;
      DWORD NumResources;
      DWORD ResNameTable;
```

Listing 9.2 (continued)

```
        DWORD EntryTable;
        DWORD ModDirectTable;
        DWORD NumModDirect;
        DWORD FixUpPageTable;
        DWORD FixUpRecTable;
        DWORD ImportModTable;
        DWORD NumImports;
        DWORD ImportProcTable;
        DWORD PerPageChecksum;
        DWORD DataPages;
        DWORD NumPreloadPages;
        DWORD NonResTable;
        DWORD NonResSize;
        DWORD NonResChecksum;
        DWORD AutoDSObj;
        DWORD DebugInfoOff;
        DWORD DebugInfoLen;
        DWORD NumInstPreload;
        DWORD NumInstDemand;
        DWORD HeapSize;
} LEHEADER;
```

Listing 9.3 ***SUCKW3.C.***

```
/**********************************************************************
 *
 * PROGRAM: SUCKW3.C
 *
 * PURPOSE: Extracts VxDs from a W3 file.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 9, W3 and W4 File Formats, from Undocumented Windows File Formats,
 * published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 **********************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "suckw3.h"
```

The `PullVxD()` function in SUCKW3 performs all the patching of the `LEHEADER` structure and adds the nonresident name table. Again, for more information on the LE file format, see Chapter 10.

Listing 9.3 (continued)

```
// Pass by the MZ Header
BOOL SkipMZ(FILE *inFile)
{
      MZHEADER    MZHeader;
      fread(&MZHeader, sizeof(MZHeader), 1, inFile);
      if (MZHeader.MZMagic[0] != 'M' || MZHeader.MZMagic[1] != 'Z')
      {
            printf("This is not an executable file!\n");
            return FALSE;
      }

      if (!MZHeader.OtherOff)      {
            printf("This is a DOS Executable");
            return FALSE;
      }

      fseek(inFile, MZHeader.OtherOff, SEEK_SET);
      return TRUE;
}

// List VxDs in the W3 file
void ListW3File(FILE *W3File)
{
      long        W3Start;
      W3HEADER    W3Hdr;
      WORD        i;
      VxDRECORD   VxDRec;

      W3Start = ftell(W3File);

      fread(&W3Hdr, sizeof(W3Hdr), 1, W3File);
      if (W3Hdr.WinVer == 0x30A)
            printf("W3 File for Windows Version 3.1\n\n");
      else if (W3Hdr.WinVer == 0x400)
            printf("W3 File for Windows 95.\n\n");

      printf("%u VxDs in this W3 File.\n\n", W3Hdr.NumVxDs);

      printf("VxDName    VxDStart       VxDHdrLen\n");
      printf("------------------------------------\n");
      for (i=0; i<W3Hdr.NumVxDs; i++)
      {
            fread(&VxDRec, sizeof(VxDRec), 1, W3File);
            printf("%-10s 0x%08lX     0x%08lX\n",
                  VxDRec.VxDName, VxDRec.VxDStart, VxDRec.VxDHdrSize);
      }
}
```

Listing 9.3 (continued)

```
// Extract the VxD
void PullVxD(FILE *W3File, char *VxDName, long VxDStart)
{
      char        OutFile[12];
      FILE        *VxDFile;
      LEHEADER    LEHdr;
      long        Remaining;
      int         ToCopy;
      static char buffer[8192];

      fseek(W3File, VxDStart, SEEK_SET);

      strcpy(OutFile, VxDName);
      strcat(OutFile, ".386");
      if ((VxDFile = fopen(OutFile, "wb")) == NULL)
            printf("Unable to create file %s!\n", OutFile);

      fread(&LEHdr, sizeof(LEHdr), 1, W3File);
      Remaining = LEHdr.NonResTable;

      // Patch values for Non-Resident Name Table
      LEHdr.NonResSize = strlen(VxDName) * 2 + 6;

      // Patch Data Pages offset
      LEHdr.DataPages -= VxDStart;

      // Write the new LE Header
      fwrite(&LEHdr, sizeof(LEHdr), 1, VxDFile);
      Remaining -= sizeof(LEHdr);

      // Copy remaining information
      while (Remaining)
      {
            ToCopy = Remaining > 4096 ? 4096 : (int) Remaining;
            fread(buffer, ToCopy, 1, W3File);
            fwrite(buffer, ToCopy, 1, VxDFile);
            Remaining -= ToCopy;
      }

      // Patch Non-Resident Name Table itself
      buffer[0]=strlen(VxDName);
      memcpy(&buffer[1], VxDName, strlen(VxDName));
      buffer[strlen(VxDName) + 1] = 0;
      buffer[strlen(VxDName) + 2] = 0;
      buffer[strlen(VxDName) + 3] = buffer[0];
      ToCopy = strlen(VxDName) + 4;
      memcpy(&buffer[ToCopy], VxDName, strlen(VxDName));
      ToCopy += strlen(VxDName);
      buffer[ToCopy] = 0x01;      buffer[ToCopy + 1] = 0;
      ToCopy+=2;

      // Write the Non-Resident Name Table and close file.
      fwrite(buffer, ToCopy, 1, VxDFile);
      fclose(VxDFile);

}
```

The W4 File Format

The W4 file format is very similar to the W3 file format, except for one major difference: it uses compression. In fact, the compression used in the W4 file is exactly the same as the Double Space compression used in double-space drives. There's code reuse for you. In fact, in the first stories I heard of people working on the W4 file format, everyone was calling directly into the Double Space decompression routines to decompress the W4 files. A very ingenious method, but for our purposes, we needed to know the compression algorithm, which Clive Turvey was nice enough to provide. In fact, Clive Turvey, in addition to providing help with the W3 file format, provided

Listing 9.3 (continued)

```
// Find the VxD to "suck" out
void SuckVxD(FILE *W3File, char *VxDName)
{
      long        W3Start;
      W3HEADER    W3Hdr;
      WORD        i;
      VxDRECORD   VxDRec;

      W3Start = ftell(W3File);

      fread(&W3Hdr, sizeof(W3Hdr), 1, W3File);

      // Try to find the VxD
      for(i=0; i<W3Hdr.NumVxDs; i++)
      {
            fread(&VxDRec, sizeof(VxDRec), 1, W3File);
            if (!memcmp(VxDRec.VxDName, VxDName, strlen(VxDName)))
            {
                  printf("Extracting %s..\n", VxDName);
                  PullVxD(W3File, VxDName, VxDRec.VxDStart);
                  return;
            }
       }

       // Didn't find the VxD;
       printf("VxD %s not found in this W3 File.\n");
}

void Usage(void)
{
      printf("Usage: SUCKW3 W3Name [VxDName]\n\n");
      printf("W3Name   is the name of the W3 executable, probably\n");
      printf("         WIN386.EXE, VMM32.VXD, or VMM32.EXE\n");
      printf("VxDName  is, optionally, the name of the VxD to extract.\n\n");
      printf("Just providing the W3Name will give a directory\n");
      printf("of the contents of the W3 executable.\n");
}
```

everything we know about the W4 format. He was even nice enough to provide source code for decompressing a W4 file. Although we've rewritten this code from scratch, it is still similar to his and it wouldn't have been possible without his assistance.

The W4 compression algorithm is called a Lempel/Ziv/Welch compression algorithm, named after the three people who contributed to its design. The W4 file, when decompressed, actually contains a W3 file inside, so once you've decompressed the W4 file, you can traverse the W3 file structure described earlier.

The W4 file begins with the `W4HEADER` record (Table 9.3).

Listing 9.3 (continued)

```
int main(int argc, char *argv[])
{

      char  filename[256];
      char  VxDName[9];
      FILE  *W3File;

      if (argc < 2) {
            Usage();
            return EXIT_FAILURE;
      }

      strcpy(filename, argv[1]);

      if (argc == 3)
      {
            if (strlen(argv[2]) > 8)
            {
                  printf("Invalid VxDName. Must be 8 characters or less.\n");
                  return EXIT_FAILURE;
            }
            strcpy(VxDName, argv[2]);
      }
      if (!strchr(filename, '.'))
            strcat(filename, ".EXE");

      if ((W3File = fopen(filename, "rb")) == NULL)
      {
            printf("%s does not exist!\n", filename);
            return EXIT_FAILURE;
      }

      if (SkipMZ(W3File))
      {
            if (argc == 2)
            {
                  ListW3File(W3File);
            }
            if (argc == 3)
            {
                  SuckVxD(W3File, VxDName);
            }
      }
      fclose(W3File);
      return EXIT_SUCCESS;
}
```

W4Magic must be W4. Always0 is an unknown value, but appears to always be zero. ChunkSize is the size of "chunks" of data that need to be decompressed (discussed later). It's important to keep in mind that ChunkSize is the maximum size of the data, compressed or decompressed. This means that two buffers of ChunkSize bytes will be sufficient to hold one chunk of compressed data and one chunk of decompressed data. ChunkSize is followed by the number of "chunks" in the W4 file. DSMagic is simply the letters "DS" to indicate that this is DriveSpace compression. The Unknown field is just six NULL bytes, probably reserved for future use.

Following the W4HEADER record is a list of offsets to the "chunks"; there will be W4HEADER.NumChunks offsets (say that 10 times really fast). Each offset is a DWORD and is an offset to the beginning of an 8Kb chunk of compressed data. The offsets are relative to the beginning of the file (in other words, the beginning of the MZ file, not the W4 file). Each 8Kb chunk is just a block of compressed data that you'll decompress to recreate the W3 file.

The W4/Double Space Compression Algorithm

I am not an expert on compression algorithms. I understand the basics enough to reverse-engineer simple ones, like LZ77. This algorithm is quite a bit different in many ways than the LZ77 derivative used in WinHelp and COMPRESS.EXE (which I'll refer to as Zeck). I would recommend reading the chapter on COMPRESS.EXE first, however, because it will give you the basics.

This algorithm, although based in the Double Space algorithm, does not encompass the entire Double Space format. Due to time constraints, we were unable to tackle the entire Double Space algorithm for this book.

The W4 compression algorithm is also a Lempel-Ziv algorithm, but unlike the Zeck, this one is bit based. In other words, in the Zeck algorithm, all of the compressed data is held a BYTE at a time and everything is on BYTE boundaries. This algorithm has only a bit boundary, which can make it difficult to work with, but as you start to take it apart, I think you'll see that it's not as bad as it may appear.

Table 9.3 W4HEADER record.

Field Name	*Data Type*	*Comments*
W4Magic	char[2]	Contains the characters "W4"
Always0	WORD	Always zero
ChunkSize	WORD	Size of a "chunk"; should always be 8Kb
NumChunks	WORD	Number of "chunks"; must be less than 1Kb
DSMagic	char[2]	Contains characters "DS" for "Double Space"
Unknown	char[6]	These are all NULLs

Shannon-Fano Tables

As I mentioned before, I'm not an expert in compression, nor is it within the scope of this book to discuss LZW compression algorithms in general. So, instead of explaining what a Shannon-Fano table is, I will only go into how it specifically affects W4 files. At its most basic, the Shannon-Fano table provides codes that give the depth and count of repeated data in the compressed data.

The Shannon-Fano table for the W4 compression algorithm is shown in Figure 9.1.

Figure 9.1 Shannon-Fano table for W4 compression.

MSB................................LSB	Meaning
xxxxxxx01	1xxxxxxx - Uncompressed byte
xxxxxxx10	0xxxxxxx - Uncompressed byte
Depth	
00000000	Quit code
xxxxxx00	xxxxxx = 1 - 63
11111100	63
xxxxxxxx011	64 + xxxxxxxx = 64 - 319
xxxxxxxxxxxx111	320 + xxxxxxxxxxxx = 320-4414
111111111111111	(4415) = Check Buffer
Count	
1	2
010	3
110	4
xx100	5 + xx = 5 - 8
xxx1000	9 + xxx = 9 - 16
xxxx10000	17 + xxxx = 17 - 32
xxxxx100000	33 + xxxxx = 33 - 64
xxxxxx1000000	65 + xxxxxx = 65 - 128
xxxxxxx10000000	129 + xxxxxxx = 129 - 256
xxxxxxxx100000000	257 + xxxxxxxx = 257 - 512
000000000	Done

The idea of how the Shannon-Fano table works is quite simple. At this point, it's probably best just to examine the code in W4Decomp; in particular, `W4Decompress()` and `LoadMiniBuffer()`. Notice that `W4Decompress()` pulls only as many bits from `dwMiniBuffer` as it needs. After pulling a depth value, it calls `LoadMiniBuffer()` to shift in some new bits. Then it looks for the count value, again pulling only as many bits as it needs, and then calling `LoadMiniBuffer()` to fill up `dwMiniBuffer` again. The rest of the code should be very straight forward. This code is not optimized for speed. It has been written for clarity so that it's easy to understand. I'll leave the optimized version as an exercise for the reader. (I've always wanted to say that.)

Where Do I Go from Here?

I can see two major uses for this information. The first is to have a utility to extract the VxDs, like the one I wrote. Then, using a disassembler based on the information in Chapter 10, you could disassemble the VxDs in the W3 and W4 files (left as an exercise for the reader) for whatever purpose you may need.

The other use I can think of is to write your own utility to create and append to W3 and W4 files (again, left as an exercise for the reader), so that you can add your own VxDs to the W3 and W4 files.

Listing 9.4 W4DECOMP.H — Header file for W4DECOMP.C.

```
/***********************************************************************
 *
 * PROGRAM: W4DECOMP.H
 *
 * PURPOSE: Decompresses a W4 file into a W3 file.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 9, W3 and W4 File Formats, from Undocumented Windows File Formats,
 * published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 ***********************************************************************/

typedef unsigned char  BOOL;
typedef unsigned short WORD;
typedef unsigned char  BYTE;
typedef unsigned long  DWORD;

#define FALSE 0;
#define TRUE 1;

typedef struct tagMZHEADER
{
  int   Magic;
  char  Stuff[58];
  long  OtherOff;
} MZHEADER;

typedef struct tagW4HEADER {
  WORD  Magic;
  WORD  Unknown1;
  WORD  ChunkSize;
  WORD  ChunkCount;
  WORD  DS;
  WORD  Unknown2;
  WORD  Unknown3;
  WORD  Unknown4;
} W4HEADER;

#define MZMAGIC 0x5A4D
#define W4MAGIC 0x3457
```

Listing 9.5 *W4DECOMP.C* **— Decompresses a W4file into a W3 file.**

```
/*********************************************************************
 *
 * PROGRAM: W4DECOMP.C
 *
 * PURPOSE: Decompresses a W4 file into a W3 file.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 9, W3 and W4 File Formats, from Undocumented Windows File Formats,
 * published by R&D Books, an imprint of Miller Freeman, Inc.
 *
 *********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <string.h>
#include "w4decomp.h"

void LoadMiniBuffer(DWORD *pMiniBuffer,
                    BYTE **pSrcBuffer,
                    WORD *pBitsUsed,
                    WORD *pBitCount)
{
  while ((*pBitsUsed)--)
  {
    *pMiniBuffer >>= 1;
    if (--(*pBitCount) == 0)
    {
      *pMiniBuffer += (DWORD) **pSrcBuffer << 24l;
      *pSrcBuffer += 1;
      *pBitCount = 8;
    }
  }
}


WORD W4Decompress(BYTE *pSrcBuffer, BYTE *pDestBuffer, WORD nSize)
{
  DWORD dwMiniBuffer = 0;
  WORD  nCount, nDepth;     // Count and Depth of a compressed "string"
  WORD  nBitCount;          // How many bits are left before we read
                            // another BYTE into dwMiniBuffer
  WORD  nBitsUsed;          // Number of bits used by the last "code",
                            // a code being a count or depth value.
  WORD  nDestIndex;         // Index into pDestBuffer
  WORD  nIndex;
```

Listing 9.5 (continued)

```
  WORD  tmpSize = nSize;

  BYTE  *pTmpBuffer;

  pTmpBuffer = pSrcBuffer;

  // Load up dwMiniBuffer with first 4 bytes. We want it
  // to look like this:
  //
  // msb                dwMiniBuffer              lsb
  //  +----------+----------+----------+---------+
  //  | byte 3   | byte 2   | byte 1   | byte 0 |
  //  +----------+----------+----------+---------+
  //

  nDestIndex = 0;            // start at byte 0 of dest buffer

  for (nIndex = 0; nIndex <= 3; nIndex++)
    dwMiniBuffer = (dwMiniBuffer >> 8) + ((DWORD)*pSrcBuffer++ << 24);

  nBitCount = 8;  // We start with 8 bits left before reading another BYTE

  nDepth = 1;      // Just allows us into the following loop.

  // While nDepth != 0. In other words, if there's nothing left
  // to decompress, then we're done.
  while (nDepth)
  {

    // Is the next piece of data an uncompressed byte?
    if (((dwMiniBuffer & 0x0003l) == 0x0001l) ||
        ((dwMiniBuffer & 0x0003l) == 0x0002l))
    {

      if (--nSize == 0xFFFF)
      {
        printf("Error: Over-run of data\n");
        return 0;
      }

      pDestBuffer[nDestIndex++] = (BYTE)(((dwMiniBuffer & 0x01FCl) >> 2l) |
                                         ((dwMiniBuffer & 0x0001l) << 7l));
      nBitsUsed = 9;
    }
```

Listing 9.5 (continued)

```
else // Depth data is compressed
{
  // (0-63)
  if ((dwMiniBuffer & 0x0003l) == 0x0000l)
  {
    nDepth = (WORD)((dwMiniBuffer & 0x00FCl) >> 2);
    nBitsUsed = 8;
  }
  // (64-319)
  else if ((dwMiniBuffer & 0x0007l) == 0x0003l)
  {
    nDepth = (WORD)((dwMiniBuffer & 0x07F8l) >> 3) + 0x0040;
    nBitsUsed = 11;
  }
  // (320-4414)
  else if ((dwMiniBuffer & 0x0007l) == 0x0007l)
  {
    nDepth = (WORD)((dwMiniBuffer & 0x07FF8l) >> 3) + 0x0140;
    nBitsUsed = 15;
  }
  else
  {
    printf("Error, invalid depth data. \n");
    return 0;
  }

  // If depth isn't 0 and not a CheckBuffer,
  // load buffer, as needed.
  if ((nDepth) &&
      (nDepth != 0x113F)) // 0x113F == (4415 - 320)
  {
    LoadMiniBuffer(&dwMiniBuffer, &pSrcBuffer, &nBitsUsed, &nBitCount);

    // Get count
    if ((dwMiniBuffer & 0x0001l) == 0x0001l) // 2
    {
      nCount = 2;
      nBitsUsed = 1;
    }
    else if ((dwMiniBuffer & 0x0003l) == 0x0002l) // 3-4
    {
      nCount = (WORD)((dwMiniBuffer & 0x0004l) >> 2l) + 3;
      nBitsUsed = 3;
    }
```

Listing 9.5 (continued)

```
        else if ((dwMiniBuffer & 0x00007l) == 0x00004l) // 5-8
        {
          nCount = (WORD)((dwMiniBuffer & 0x00018l) >> 3l) + 5;
          nBitsUsed = 5;
        }
        else if ((dwMiniBuffer & 0x0000Fl) == 0x0008l) // 9-16
        {
          nCount = (WORD)((dwMiniBuffer & 0x00070l) >> 4l) + 9;
          nBitsUsed = 7;
        }
        else if ((dwMiniBuffer & 0x0001Fl) == 0x0010l) // 17-32
        {
          nCount = (WORD)((dwMiniBuffer & 0x001E0l) >> 5l) + 17;
          nBitsUsed = 9;
        }
        else if ((dwMiniBuffer & 0x0003Fl) == 0x00020l) // 33-64
        {
          nCount = (WORD)((dwMiniBuffer & 0x007C0l) >> 6l) + 33;
          nBitsUsed = 11;
        }
        else if ((dwMiniBuffer & 0x0007Fl) == 0x00040l) // 65-128
        {
          nCount = (WORD)((dwMiniBuffer & 0x01F80l) >> 7l) + 65;
          nBitsUsed = 13;
        }
        else if ((dwMiniBuffer & 0x000FFl) == 0x00080l) // 129-256
        {
          nCount = (WORD)((dwMiniBuffer & 0x07F00l) >> 8l) + 129;
          nBitsUsed = 15;
        }
        else if ((dwMiniBuffer & 0x001FFl) == 0x00100l) // 257-512
        {
          nCount = (WORD)((dwMiniBuffer & 0x01FE00l) >> 9l) + 257;
          nBitsUsed = 17;
        }
        else
        {
          // Bad data, but handle as if it were a quit condition
          printf("Bad count data, quiting at current point.\n");
          nDepth = 0;
          nCount = 0;
          nBitsUsed = 9;
        }
```

Listing 9.5 (continued)

```
        // Copy "nCount" bytes of data from "nDepth" bytes back.
        while (nCount--)
        {
          if (--nSize == 0xFFFF)
          {
            printf("Error: Over-run of data\n");
            return 0;
          }

          pDestBuffer[nDestIndex] = pDestBuffer[nDestIndex - nDepth];
          nDestIndex++;
        }
      }
      else
      {
        // If we get a Check Buffer and
        // size of the remaining data is 0,
        // then we're done.
        if ((nDepth == 0x113F) &&
            (nSize == 0x0000))
        {
          nDepth = 0;
        }
      }

    } // else

    LoadMiniBuffer(&dwMiniBuffer, &pSrcBuffer, &nBitsUsed, &nBitCount);

  } // while(nDepth)

  return nDestIndex;
}

int ExtractW3(FILE *W4File, char *filename)
{
  FILE     *W3File;
  MZHEADER mzHeader;
  W4HEADER w4Header;
  DWORD    *pChunkTable;
  DWORD    start, end;
  BYTE     *pSrcBuffer, *pDestBuffer;
  WORD     nChunkIndex;
  WORD     nDestSize;
  DWORD    i;
```

Listing 9.5 (continued)

```
if ((W3File = fopen("LIBRARY.W3", "wb")) == NULL)
{
  printf("Unable to open file LIBRARY.W3 for output\n");
  return 1;
}

fread(&mzHeader, sizeof(mzHeader), 1, W4File);
if (mzHeader.Magic != MZMAGIC)
{
  printf("Not an executable file.\n");
  return 1;
}

fseek(W4File, mzHeader.OtherOff, SEEK_SET);

fread(&w4Header, sizeof(w4Header), 1, W4File);

if (w4Header.Magic != W4MAGIC)
{
  printf("Not a W4 file.\n");
  fclose(W3File);
  return 1;
}

// Allocate space for chunk table
pChunkTable = malloc(w4Header.ChunkCount * 4);

if (pChunkTable == NULL)
{
  printf("Not enough memory to allocate chunk table.\n");
  return 1;
}

// Allocate space for source buffer
pSrcBuffer = malloc(w4Header.ChunkSize);

if (pSrcBuffer == NULL)
{
  printf("Not enough memory for source buffer.\n");
  return 1;
}

// Allocate space for destination buffer
pDestBuffer = malloc(w4Header.ChunkSize * 2);

if (pDestBuffer == NULL)
{
  printf("Not enough memory for destination buffer.\n");
  return 1;
}
```

Listing 9.5 (continued)

```
 // Read chunk table
fread(pChunkTable, w4Header.ChunkCount, 4, W4File);

// Pad W3File so that offsets in the list of VxDs
// will match up.
printf("Padding W3 File.\n");
fwrite(&mzHeader, sizeof(mzHeader), 1, W3File);

end = mzHeader.OtherOff - sizeof(mzHeader);
for (i = 0; i < end; i++)
{
  fputc(0, W3File);
}

for (nChunkIndex = 0; nChunkIndex < w4Header.ChunkCount; nChunkIndex++)
{
  start = pChunkTable[nChunkIndex];

  if (nChunkIndex == w4Header.ChunkCount)
  {
    end = fseek(W4File, 0l, SEEK_END);
  }
  else
  {
    end = pChunkTable[nChunkIndex + 1];
  }

  printf("Decompressing chunk %d   -   Compressed Size %d\n", nChunkIndex,
         (end - start));
  // Go to and read the current chunk
  // Note: This code assumes that the chunk size
  //       is not beyond the ability of fread.
  fseek(W4File, start, SEEK_SET);
  fread(pSrcBuffer, (WORD)(end - start), 1, W4File);

  // Fill destination buffer with clear marker
  memset(pDestBuffer, 0xE5, w4Header.ChunkSize);

  if ((WORD)(end - start) != w4Header.ChunkSize)
  {
    nDestSize = W4Decompress(pSrcBuffer, pDestBuffer, w4Header.ChunkSize);

    // This is an error condition.
    if (!nDestSize)
    {
      fclose(W3File);
      return 1;
    }

    fwrite(pDestBuffer, (WORD)nDestSize, 1, W3File);
  }
  else
  {
    fwrite(pSrcBuffer, (WORD)(end - start), 1, W3File);
  }
}
```

Listing 9.5 (continued)

```
  fclose(W3File);
  free(pChunkTable);
  free(pSrcBuffer);
  free(pDestBuffer);
}


void Usage(void)
{
  printf("Usage: W4DECOMP W4Name\n\n");
  printf("W4Name   is the name of the W4 executable, probably\n");
  printf("         VMM32.VXD\n");
}

int main(int argc, char *argv[])
{

  char  filename[256];
  FILE  *W4File;

  if (argc < 2) {
    Usage();
    return 1;
  }

  strcpy(filename, argv[1]);

  if (!strchr(filename, '.'))
    strcat(filename, ".EXE");

  if ((W4File = fopen(filename, "rb")) == NULL)
  {
    printf("%s does not exist!\n", filename);
    return 1;
  }

  ExtractW3(W4File, filename);

  fclose(W4File);
  return 0;
}
```

Chapter 10

LE File Format

Overview

Those of you who have been writing VxDs have probably been tied to Microsoft's tools for writing VxDs. Although I'm not really picky when it comes to assemblers and linkers, some people are. Beyond that, the information in Linear Executables (LE) can be useful. Microsoft has usually been pretty good about documenting executable file formats. They've provided information on the standard DOS executable (MZ), Windows 16-bit (NE), and Windows 32-bit (PE) file formats, among others. For some reason, Microsoft chose not to do the same with the LE format.

The LE format is actually based on, or at least very similar to, the LX file format used by OS/2 executables. In fact, all of the work in reverse-engineering the LE format was based on information available on the LX format. I have never written a linker or assembler, so I can't say that I'm absolutely positive about *all* of the information here. For models, I examined the MZ, NE, PE, and LX formats to give me as good an understanding of executable file formats as I could get. However, as I said, I haven't written an LE linker or an LE assembler, so it's possible that some of this information is not correct. As with all undocumented information, use it at your own risk.

The most useful tool that could probably be written with this material would be a linker (or possibly an assembler), but in the interest of time, space, and most importantly, my sanity, I've chosen to write an LE Dump utility modeled loosely on Andrew Schulman's EXEDUMP included with *Undocumented Windows* (see the Bibliography for more information).

LEDUMP simply goes through the linear executable and gives you some information about it, including global header information, relocations, and exports. This should be suitable to demonstrate how to get to the information.

Figure 10.1 Overview of LE file layout.

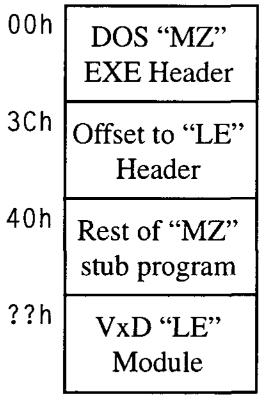
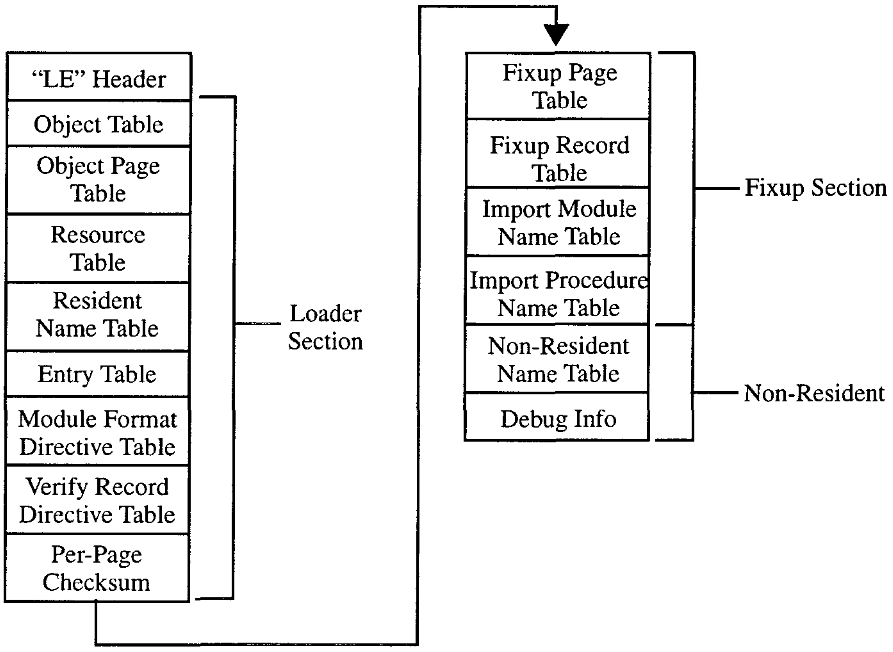


Figure 10.2 LE module layout.



General Layout

The general layout of an LE file is similar to the NE, PE, and LX file formats, in that the very first section is actually a stub MZ program that tells you that you can't run this program in DOS, or whatever MZ stub was attached. Following the stub program is the actual LE file. I won't go into a complete description of the MZ file format, because that isn't my purpose here, but I will give you enough information to maneuver around the MZ file and get to the LE file.

Figure 10.1 shows the basic layout of an LE file with the MZ stub. Offset 3Ch in the MZ file contains an offset to the LE file header.

The LE File itself is broken into several sections, as shown in Figure 10.2. Immediately following the header is the loader section. The loader section is everything that must be kept resident in memory while the program is running. This is followed by the fixup section. The fixup section contains everything required to resolve addresses within the code and to resolve dynamic links to other modules; although, as I'll discuss later, this isn't supported by Windows. The fixup section is followed by the non-resident section, which contains the export table and debugging information and does not need to be kept in memory while the VxD is running.

The simple code shown in Listing 10.1 can be used to jump to the LE header. This code reads a portion of the MZ header. You're really only interested in two fields: MZMagic, which lets you make sure this is an MZ executable file, and the offset to the LE header, which is located at offset 3Ch in the MZ header. Actually, this is just an offset to whatever executable might follow the MZ stub. NE executables, for example, are pointed to by the same offset.

Listing 10.1 SkipMZ code sample.

```
/***********************************************************************
 *
 * PROGRAM: SkipMZ
 *
 * PURPOSE: Read the MZ header and determine
 * whether the file has an LE header.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 10, LE File Format,
 * from Undocumented Windows File Formats, published by R&D Books,
 * an imprint of Miller Freeman, Inc.
 *
 ***********************************************************************/
```

The code in Listing 10.1 jumps to the beginning of the LE file, where the LE file header resides. The LE file header is shown in Table 10.1.

I'll be the first to admit that this is not a small header, but a lot of things need to be kept track of in an executable file. Some of the fields may not make all that much sense, so I'll go into more detail about them.

First of all, I'll talk about what fields aren't used and get them out of the way. All of the checksum fields are zero, so it appears that no checksum is done by the linker. The resource table will never exist because a VxD cannot have resources. So the ResourceTbl and NumResources fields will always be zero. The ModDirectTable and NumModDirect will also be zero. The module format directive table is used to extend the LE or LX executables. These are unused by VxDs and will therefore always be zero. NumInstPreload and NumInstDemand are not used by Windows or Win95. Preload and Demand pages are handled entirely by the loader in Win95. This is simply a carry over from OS/2. The ESPObjNum and ESP fields are both unused, as well.

Listing 10.1 (continued)

```
typedef struct tagMZHEADER
{
   char      MZMagic[2];                  /* Should always be 'MZ'        */
   char      Stuff[58];                   /* Stuff you don't care about   */
   long      LEOff;                       /* Offset to 'LE' Header        */
} MZHEADER;

BOOL SkipMZ(FILE *inFile)
{
   MZHEADER MZHeader;

   fread(&MZHeader, sizeof(MZHeader), 1, inFile);
   if (MZHeader.MZMagic[0] != 'M' || MZHeader.MZMagic[1] != 'Z')
   {
      printf("This is not an executable file!\n");
      return FALSE;
   }

   if (!MZHeader.LEOff)
   {
      printf("This is a DOS executable.\n");
      return FALSE;
   }

   fseek(inFile, MZHeader.LEOff, SEEK_SET);
   return TRUE;
}
```

Table 10.1 LEFileHeader record.

Field Name	*Data Type*	*Comment*
LEMagic	char[2]	Must be LE
ByteOrder	BYTE	Byte ordering (big-/little-endian)
WordOrder	BYTE	Word ordering (big-/little-endian)
FormatLevel	DWORD	Version of LE file
CPUType	WORD	Type of CPU required
OSType	WORD	Type of OS required
ModuleVer	DWORD	Version of this module
ModuleFlags	DWORD	Global flags for this module
NumPages	DWORD	Number of physical pages in entire module
EIPObjNum	DWORD	Object no. to which EIP register is relative
EIP	DWORD	Starting instruction pointer (EIP) address
ESPObjNum	DWORD	Object no. to which ESP register is relative
ESP	DWORD	Starting stack (ESP) address
PageSize	DWORD	Size of a page (usually 4Kb)
LastPageSize	DWORD	Used to read object page table entries
FixupSize	DWORD	Total size of the fixup information
FixupChecksum	DWORD	Checksum for fixup information
LoaderSize	DWORD	Size of loader section
LoaderChecksum	DWORD	Checksum for loader information
ObjTblOffset	DWORD	Offset to object table
NumObjects	DWORD	Number of objects in object table
ObjPageTbl	DWORD	Offset to object page table
ObjIterPage	DWORD	Offset to object iterated pages
ResourceTbl	DWORD	Offset to resource table
NumResources	DWORD	Number of entries in resource table
ResNameTable	DWORD	Resident name table offset
EntryTable	DWORD	Entry table offset
ModDirectTable	DWORD	Module directive table
NumModDirect	DWORD	Number of module directives
FixUpPageTable	DWORD	Offset to fixup page table

Although the import module and import procedure tables may exist, you'll find them empty because VxDs don't import functions dynamically. Therefore both tables are almost always empty (single NULL byte) and NumImports is almost always zero. I say "almost always" because it is "possible" to add imports. If you add an IMPORTS section to the .DEF file for the VxD, you will, in fact, populate these tables. There is no way of actually using these imports, because Windows has no dynamic-link support for VxDs.

Now I'll discuss the fields you will use.

LEMagic Contains the letters LE. They are, of course, quite magical, so be careful with them.

ByteOrder **and** WordOrder These flags let you know if the data in the file is stored in little-endian (0x00) or big-endian (0x01) format. ByteOrder indicates how a WORD is stored internally. For example, the WORD 0x1234 in little-endian notation would be stored internally with the byte order 0x34 0x12. In big-endian notation, it would be

Table 10.1 (continued)

Field Name	*Data Type*	*Comment*
FixUpRecTable	DWORD	Offset to fixup record table
ImportModTable	DWORD	Import module table offset
NumImports	DWORD	Number of entries in import module table
ImportProcTable	DWORD	Offset to import procedure name table
PerPageChecksum	DWORD	Offset to per-page checksum table
DataPages	DWORD	Offset to data pages
NumPreloadPages	DWORD	Number of pages to preload
NonResTable	DWORD	Nonresident name table
NonResSize	DWORD	Size, in BYTEs, of nonresident name table
NonResChecksum	DWORD	Checksum for nonresident name table
AutoDSObj	DWORD	Object no. of automatic data segment
DebugInfoOff	DWORD	Offset to debug information
DebugInfoLen	DWORD	Size of debug information area in BYTEs
NumInstPreload	DWORD	Instance pages in preload section
NumInstDemand	DWORD	Instance pages in demand section
HeapSize	DWORD	No. BYTEs to add to auto-data segment for heap

stored with the byte order 0x12 0x34. WordOrder works the same way. The DWORD 0x12345678 in little-endian notation would be stored as 0x5678 0x1234. ByteOrder and WordOrder let us know how the data is stored in the file. Because Intel uses little-endian notation exclusively, you'll find that the ByteOrder and WordOrder fields are always set to little-endian.

FormatLevel This should always be zero. If it's not zero, then the LE format itself has been modified in an incompatible format (i.e., the LE format has been changed).

CPUType This indicates the minimum CPU required. Valid values are:

0x01	80286
0x02	80386
0x03	80486

The first value is a carry-over from the LX file format used by OS/2. You won't find any VxDs that require less than a 386. I haven't seen one that's specific to the 486 yet.

OSType This tells us what OS is required to run this executable. Valid values are:

0x00	Unknown
0x01	OS/2
0x02	Windows
0x03	DOS 4.x
0x04	Windows 386 Enhanced mode

Obviously, Windows 386 Enhanced Mode is the only valid value you'll see. I list the others simply because they are defined in the LX file format specifications.

ModuleVer This is the version data for this particular module. According to the LX specifications, this value is specified at link time by the user. Any DWORD value would be valid. I've seen only 0x0000.

ModuleFlags These flags globally define aspects of the VxD. I will only cover those flags used by VxDs.

0x00000010 Module uses internal fixups. This means that each object has a preferred load address. ("Object", in the case of the LE file format, means segment. I'll discuss this later.) If the object can be loaded at that address, then no fixups need be applied. If it must be moved to another address, then fixups should be applied.

0x00000020 Module uses external fixups. This just means that there are no internal fixups, so the fixups *must* be applied at load time. I have never seen a VxD that used internal fixups, but because the internal fixups flag is specified, I'll assume it's possible to have external fixups.

0x00008000 Library module.

0x00028000 Virtual device driver module.

0x00038000 Physical device driver module.

The ModuleFlags value in Windows 3.x VxDs always seems to be 0x00008020. This means it has the Library Module attribute and uses external fixups.

Windows 95 VxDs, on the other hand, seem to have one of two values, 0x00028000 (virtual device driver) or 0x00038000 (physical device driver). I've seen the latter for network cards, but most VxDs in Windows 95 appear to have the virtual device driver flag.

NumPages The NumPages field specifies the number of pages contained in the VxD module.

EIPObjNum This provides the object (or segment) with which to initialize EIP.

EIP This is the offset within the object (segment) with which to initialize EIP.

PageSize This is the size of the page, in bytes, within this module. Where page counts are given, they are multiplied by this number to get the size in bytes. Page size is usually 4Kb. I've never seen a different value used.

LastPageSize This is the size of the last page in the module. This keeps the module from having to be an exact multiple of PageSize and saves a little space.

FixupSize This is the size of the fixup section. The fixup section includes the fixup page table, the fixup record table, import module name table, and import procedure name table. The fixup page table and fixup record table are used to map unresolved addresses in the code to the proper locations after the code has been loaded. The import module name and import procedure name tables are used to resolve calls to external functions. Of course, as mentioned before, Windows doesn't support dynamic linking for VxDs.

LoaderSize This is the size of all the objects (segments) that need to remain resident in memory while the program is resident in memory. It includes everything from the object table down to and including the entry table.

ObjTblOffset This is the offset to the object table. The object table is described below in greater detail. Objects, as mentioned earlier, are segments within the executable and contain code or data.

NumObjects Contains a count with the number of entries in the object table.

ObjPageTbl This is the offset to the object page table (described in greater detail in the section "Object Page Table").

ResNameTable This contains the offset to the resident name table (again, described in greater detail in the section "Resident or Nonresident Name Tables").

EntryTable Contains the offset to the entry table.

FixUpPageTable Offset to the fixup page table.

FixUpRecTable Offset to the fixup record table.

ImportModTable Offset to the import module table.

NumImports Number of imports in the import module table. This should always be zero.

ImportProcTable Offset to the import procedure name table.

DataPages Data pages offset.

NumPreloadPages Number of preload pages.

NonResTable Offset to the nonresident name table.

NonResSize Size of the nonresident name table.

That's pretty much it for the fields you're going to use from the header. As you start to look at the different areas of the LE file, you'll see that it's really just a large collection of parts, and parts of the header are all that are needed for parts of the LE file. It's really not quite as overwhelming as it may first appear.

Object Table

Again, "Objects" are really just segments. The object table just provides some basic information about each segment in the executable. Table 10.2 shows the OBJECTENTRY structure.

The VirtualSize field is the amount of space that Windows needs to allocate for the segment in memory.

The RelocAddr field is the base address to which the object is currently relocated. If the internal fixups for the module have been removed, this is the address at which the object will be allocated by the loader.

The ObjectFlags field may have the following bit values:

0x0001h	Readable
0x0002h	Writable
0x0004h	Executable
0x0008h	Resource object
0x0010h	Discardable object
0x0020h	Object is shared
0x0040h	Object has preload pages
0x0080h	Object has invalid pages
0x0100h	Object has zero-filled pages
0x0200h	Object is resident
0x0300h	Object is resident and contiguous
0x0400h	Object is resident and "long lockable"
0x0800h	Reserved for system use
0x1000h	16:16 alias required

Table 10.2 OBJECTENTRY *record.*

Field Name	*Data Type*	*Comments*
VirtualSize	DWORD	Amount of space to allocate for the object
RelocAddr	DWORD	Relocation base address for the object
ObjectFlags	DWORD	See below for a list.
PageTblIdx	DWORD	First object page table entry for the object
NumPgTblEntries	DWORD	No. of entries in object page table
Reserved	DWORD	Must be set to 0

0x2000h	Big/Default bit setting (see the following paragraph on bit settings)
0x4000h	Object is conforming for code
0x8000h	Object I/O privilege level (used for 16:16 alias objects)

The Big/Default bit setting, for data segments, controls the setting of the Big bit in the segment descriptor. (The Big bit, or B-bit, determines whether ESP or SP is used as the stack pointer.) For code segments, this bit controls the setting of the Default bit in the segment descriptor. (The Default bit, or D-bit, determines whether the default word size is 32 bits or 16 bits. It also affects the interpretation of the instruction stream.)

The `PageTblIdx` field specifies the number of the first entry for this object in the object page table. I'll discuss the object page table later.

The `NumPgTblEntries` field is the number of entries in the object page table for this object.

Object Page Table

The Object Page Table (OPT) provides information about logical pages in an object. A page can be either a pseudo-page, an enumerated page, or an iterated page.

The `OPTENTRY` structure (Table 10.3) shows the format for each page table entry.

The `PageDataOff` field has the offset to the page data in the `.EXE`. This field may be zero if the flags specify that it is a zero-filled page.

The `DataSize` field specifies the number of bytes that the page actually takes up. If it is smaller than the page size specified in the module header, then the remaining bytes in the page are filled with zeros.

The `OPTFlags` field has five possible flag values:

0x00h	Legal physical page in the module (offset from preload page section)
0x01h	Iterated data page (offset from iterated data pages section)
0x02h	Invalid page
0x03h	Zero-filled page
0x04h	Range of pages

Table 10.3 OPTENTRY record.

Field Name	*Data Type*	*Comments*
`PageDataOff`	`DWORD`	Offset to page data in `.EXE`
`DataSize`	`WORD`	Number of bytes for this page
`OPTFlags`	`WORD`	Flags for the OPT entry

Resident or Nonresident Name Tables

The structure of the resident and nonresident name tables is identical. Again, this is a place where I am adapting the names from the LX file format.

These tables aren't really the resident and nonresident name tables, per se. The resident name table, for example, contains a single entry, which is the name given in the .DEF file for the LIBRARY parameter. The nonresident name table contains two entries. The first entry is the description provided in the DESCRIPTION parameter of the .DEF file. The second entry is the single export specified in the .DEF file, usually the module name with _DDB appended to it.

The actual format for these tables is very simple. Each entry consists of a single byte with the length of the string. This is followed by the string (which is not null-terminated). The string is, in turn, followed by an ordinal number. The ordinal number is an index into the entry table (described in the following section).

Entry Table

The entry table contains information to resolve fixups to entry points within the module. There appears to only be one entry point within VxDs. This is the "vxdname_DDB" area defined in the EXPORTS section of the .DEF file. An ordinal number is used as an index into the entry table. Entry table items are numbered starting with one. There should only be one entry in this table and that entry will be 1.

Objects in the entry table are grouped in bundles. The ENTRYTABLEREC is seen in Table 10.4.

Count is the number of entries in the "bundle".

The Type field describes the type of data that's contained in the bundle. Valid values are:

0x01	16-bit entry
0x02	286 callgate entry
0x03	32-bit entry
0x04	Forwarder entry

Table 10.4 ENTRYTABLEREC record.

Field Name	*Data Type*	*Contents*
Count	BYTE	**No. entries in this bundle**
Type	BYTE	**Type of entries in this bundle**
Bundle	BYTE[]	**Bundle data**

The type of data in the Bundle field depends on the Type field. Only the 32-bit entry type is used, so it's the only one I'll look at. For the 32-bit entry, the bundle data contains an object number in the form of a WORD followed by an array of 32BITENTRY records (Table 10.5). The size of the array depends on the value of Count in ENTRYTABLEREC:

The only value for Flags is 0x01, which simply means it's an exported entry. Offset is the offset of the entry within the object defined for this bundle.

Fixup Page Table

The fixup page table is simply a list of offsets into the fixup record table (described in the following section). The offsets are to the beginning of the fixup data for a given page. Each entry in the object page table has one entry, plus one additional entry that points to the end of the fixup record table. Each offset is simply a DWORD that is added to the offset of the fixup record table. This provides you with the first entry in the fixup record table for a specific page number (or segment). Figure 10.3 shows how this works.

Table 10.5 32BITENTRY record.

Field Name	*Data Type*	*Comments*
Flags	BYTE	Flags
Offset	DWORD	Offset in the object for this entry

Figure 10.3 Layout of the fixup page table.

```
Page #1
Offset for page 1 in fixup record table
Page #2
Offset for page 2 in fixup record table
...
...
Page #n
Offset for page n in fixup record table
Offset to end of fixup record table
```

Fixup Record Table

The fixup record table contains a list of all fixups for the VxD. Entries in this table are grouped by page number. This allows the fixup page table to point to a group of fixup records for an individual page.

Each entry consists of a FIXUP record (Table 10.6).

Valid Type values are:

0x00	Byte fixup (8 bits)
0x02	16-bit selector (16 bits)
0x03	16:16 pointer (32 bits)
0x05	16-bit offset (16 bits)
0x06	16:32 pointer (48 bits)
0x07	32-bit offset (32 bits)
0x08	32-bit self-relative offset (32 bits)
0x0F	Type mask
0x10	Fixup to alias flag
0x20	List flag

Although the linker (LINK386.EXE) supports all of these fixup types, the VxD loader itself only supports 32-bit offsets and 32-bit self-relative offsets, types 0x07 and 0x08, respectively.

The two types that really require a description are 0x10 (fixup to alias flag) and 0x20 (list flag). The fixup to alias flag occurs for some entries of type 0x02, 0x03, and 0x06. In these cases, the fixup refers to the 16:16 alias for the object. For the list flag, the Fixup field contains a byte that says how many offsets are listed. The offsets then follow the end of the FIXUP record.

Valid Flags values are:

0x00	Internal reference
0x01	Imported reference by ordinal
0x02	Imported reference by name

Table 10.6 FIXUP *record.*

Field Name	*Data Type*	*Comment*
Type	BYTE	Type of fixup
Flags	BYTE	Specifies how the fixup is interpreted
Fixup	BYTE[]	Fixup information. Size and format depend on Type

0x03	Internal reference via entry table
0x04	Additive fixup
0x10	32-bit offset
0x20	32-bit additive fixup
0x40	16-bit object number/module ordinal flag
0x80	8-bit ordinal

The contents of the `Fixup` field depend on whether or not the list flag in the Type field is set. If the list flag isn't set, the `Fixup` field simply contains the value for the fixup; otherwise, the `Fixup` field begins with the number of fixups, in the form of a `WORD`, to follow. This is followed by an array of fixups.

LE Dump

Dump (Listings 10.2 and 10.3) is a fairly simple program. It basically provides various pieces of information from the LE Header and several of the internal structures, such as the object table, resident name table, and nonresident name table.

The idea is just to show you how to get around the LE file itself. If you actually want to write a disassembler, assembler, or linker, you're going to have a bit more work to do, but you'll be able to do it from the information given in this chapter.

Listing 10.2 LEDUMP.H — Headerfile for LEDUMP.C.

```
/**********************************************************************
 *
 * PROGRAM: LEDUMP.H
 *
 * PURPOSE: Header file for LEDUMP.C.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 10, LE File Format,
 * from Undocumented Windows File Formats, published by R&D Books,
 * an imprint of Miller Freeman, Inc.
 *
 **********************************************************************/

/* Mini-MZ Header. Need only 2 fields */
typedef struct tagMZHEADER
{
   char   MZMagic[2];
   char   Stuff[58];            /* Stuff you don't care about */
   long   LEOff;
} MZHEADER;
```

Where Do I Go from Here?

What can you do with this information? Well, I've already mentioned this, but disassemblers, assemblers, and linkers are the main tools that come to mind. A VxD disassembler is probably the most useful tool, in my mind, but then again I like to reverse engineer stuff, if you haven't noticed.

Listing 10.2 (continued)

```
/* LE Header structure */
typedef struct tagLEHEADER
{
    char            LEMagic[2];
    BYTE            ByteOrder;
    BYTE            WordOrder;
    DWORD           FormatLevel;
    WORD            CPUType;
    WORD            OSType;
    DWORD           ModuleVer;
    DWORD           ModuleFlags;
    DWORD           NumPages;
    DWORD           EIPObjNum;
    DWORD           EIP;
    DWORD           ESPObjNum;
    DWORD           ESP;
    DWORD           PageSize;
    DWORD           LastPageSize;
    DWORD           FixupSize;
    DWORD           FixupChecksum;
    DWORD           LoaderSize;
    DWORD           LoaderChecksum;
    DWORD           ObjTblOffset;
    DWORD           NumObjects;
    DWORD           ObjPageTbl;
    DWORD           ObjIterPage;
    DWORD           ResourceTbl;
    DWORD           NumResources;
    DWORD           ResNameTable;
    DWORD           EntryTable;
    DWORD           ModDirectTable;
    DWORD           NumModDirect;
    DWORD           FixUpPageTable;
    DWORD           FixUpRecTable;
    DWORD           ImportModTable;
    DWORD           NumImports;
    DWORD           ImportProcTable;
    DWORD           PerPageChecksum;
    DWORD           DataPages;
    DWORD           NumPreloadPages;
    DWORD           NonResTable;
    DWORD           NonResSize;
    DWORD           NonResChecksum;
    DWORD           AutoDSObj;
    DWORD           DebugInfoOff;
    DWORD           DebugInfoLen;
    DWORD           NumInstPreload;
    DWORD           NumInstDemand;
    DWORD           HeapSize;
} LEHEADER;
```

Listing 10.2 (continued)

```
/* Defines for Byte and Word Order */
#define ORD_LITTLEENDIAN    0x00
#define ORD_BIGENDIAN       0x01

/* Defines for CPU Type */
#define CPU_286    0x01
#define CPU_386    0x02
#define CPU_486    0x03

/* Defines for OS Type */
#define OS_UNKNOWN       0x00
#define OS_OS2           0x01
#define OS_WINDOWS       0x02
#define OS_DOS4X         0x03
#define OS_WINDOWS386    0x04

/* Defines for Module Flags */
#define MOD_PERPROCESSLIBINIT    0x00000004
#define MOD_INTERNALFIXUPS       0x00000010
#define MOD_EXTERNALFIXUPS       0x00000020
#define MOD_INCOMPAT_PM          0x00000100
#define MOD_COMPAT_PM            0x00000200
#define MOD_USES_PM              0x00000300
#define MOD_NOTLOADABLE          0x00002000
#define MOD_MODTYPEMASK          0x00038000
#define MOD_PROGRAMMOD           0x00000000
#define MOD_LIBMOD               0x00008000
#define MOD_PROTLIBMOD           0x00018000
#define MOD_PHYSDEVICEDRVR       0x00020000
#define MOD_VIRTDEVICEDRVR       0x00028000
#define MOD_PERPROCLIBTERM       0x40000000


/* Object Table Entry Records */
typedef struct tagOBJTBLENTRY
{
   DWORD          VirtualSize;
   DWORD          RelocBaseAddr;
   DWORD          ObjectFlags;
   DWORD          PageTableIndex;
   DWORD          NumPgTblEntries;
   DWORD          Reserved;
} OBJTBLENTRY;
```

Listing 10.2 (continued)

```
/* Defines for Object Flags */
#define OBJ_READABLE          0x0001
#define OBJ_WRITEABLE         0x0002
#define OBJ_EXECUTABLE        0x0004
#define OBJ_RESOURCE          0x0008
#define OBJ_DISCARDABLE       0x0010
#define OBJ_SHARED            0x0020
#define OBJ_PRELOAD           0x0040
#define OBJ_INVALID           0x0080
#define OBJ_ZEROFILLED        0x0100
#define OBJ_RESIDENT          0x0200
#define OBJ_RESIDENTCONTIG    0x0300
#define OBJ_RESIDENTLOCK      0x0400
#define OBJ_RESERVED          0x0800
#define OBJ_1616ALIAS         0x1000
#define OBJ_BIGDEFAULT        0x2000
#define OBJ_CONFORM           0x4000
#define OBJ_PRIVILEGE         0x8000


/* Object Page Table Entries */
typedef struct tagOBJPAGETBLENTRY
{
   DWORD              PageDataOffset;
   WORD               DataSize;
   WORD               ObjPageFlags;
} OBJPAGETBLENTRY;

/* Defines for Object Page Flags */
#define OPG_LEGAL           0x00
#define OPG_ITERATEDDATA    0x01
#define OPG_INVALID         0x02
#define OPG_ZEROFILL        0x03
#define OPG_RANGE           0x04


typedef struct tagFIXUPREC
{
   BYTE        Src;
   BYTE        Flags;
   WORD        SrcOff_Cnt;
} FIXUPREC;
```

Listing 10.2 (continued)

```
typedef struct tagPROCBYNAME
{
   WORD          ModuleOrd;
   DWORD         ProcNameOff;
} PROCBYNAME;

typedef struct tagPROCBYORD
{
   WORD    ModuleOrd;
   WORD    ImportOrd;
};
```

Listing 10.3 ***LEDUMP.C — Extract information from the LE header.***

```
/**********************************************************************
 *
 * PROGRAM: LEDUMP.C
 *
 * PURPOSE: Dump 'LE' file info.
 *
 * Copyright 1997, Mike Wallace and Pete Davis
 *
 * Chapter 10, LE File Format,
 * from Undocumented Windows File Formats, published by R&D Books,
 * an imprint of Miller Freeman, Inc.
 *
 **********************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <windows.h>
#include "ledump.h"

/* Make this big thing global since you'll need it everywhere. */
LEHEADER    LEHeader;
long        LEStart;               /* Keep track of start of LE Header */


BOOL SkipMZ(FILE *LEFile)
{
   MZHEADER MZHeader;

   fread(&MZHeader, sizeof(MZHeader), 1, LEFile);
   if (MZHeader.MZMagic[0] != 'M' || MZHeader.MZMagic[1] != 'Z')
   {
      printf("This is not an executable file!\n");
      return FALSE;
   }
```

Listing 10.3 (continued)

```
   if (!MZHeader.LEOff)
   {
      printf("This is a DOS executable.\n");
      return FALSE;
   }

   fseek(LEFile, MZHeader.LEOff, SEEK_SET);
   return TRUE;
}

void DumpHeaderInfo()
{
   DWORD Flags;

   Flags = LEHeader.ModuleFlags;    /* Get it into an easier to type variable */

   printf("CPU Required: ");
   switch(LEHeader.CPUType)
   {
      case CPU_286:
         printf("80286\n");
         break;
      case CPU_386:
         printf("80386\n");
         break;
      case CPU_486:
         printf("80486\n");
         break;
   }

   printf("OS Required: ");
   switch(LEHeader.OSType)
   {
      case OS_UNKNOWN:
         printf("Unknown\n");
         break;
      case OS_OS2:
         printf("OS/2\n");
         break;
      case OS_WINDOWS:
         printf("Windows\n");
         break;
      case OS_DOS4X:
         printf("DOS 4.x\n");
         break;
      case OS_WINDOWS386:
         printf("Windows 386 Enhanced Mode\n");
         break;
   }

   printf("Initial EIP: %lX:%08lX\n", LEHeader.EIPObjNum, LEHeader.EIP);
   printf("Initial ESP: %lX:%08lX\n", LEHeader.ESPObjNum, LEHeader.ESP);
```

Listing 10.3 (continued)

```
   printf("Flags: 0x%08lX\n", Flags);
   if (Flags & MOD_PERPROCESSLIBINIT)
      printf("- Per Process Library Initialization\n");
   if (Flags & MOD_INTERNALFIXUPS)
      printf("- Internal Fixups\n");
   if (Flags & MOD_EXTERNALFIXUPS)
      printf("- External Fixups\n");
   if ( (Flags & MOD_USES_PM) == MOD_USES_PM)
      printf("- Uses Presentation Manager\n");
   else
   {
      if (Flags & MOD_INCOMPAT_PM)
         printf("- Incompatible with Presentation Manager\n");
      if (Flags & MOD_COMPAT_PM)
         printf("- Compatible with Presentation Manager\n");
   }
   if (Flags & MOD_NOTLOADABLE)
      printf("- Module is not loadable.\n");
   if ( (Flags & MOD_PROTLIBMOD) == MOD_PROTLIBMOD)
      printf("- Protected Memory Library Module\n");
   else if ( (Flags & MOD_VIRTDEVICEDRVR) == MOD_VIRTDEVICEDRVR)
      printf("- Virtual Device Driver Module\n");
   else
   {
      if (Flags & MOD_LIBMOD)
         printf("- Library Module\n");
      if (Flags & MOD_PHYSDEVICEDRVR)
         printf("- Physical Device Driver\n");
   }
   if (Flags & MOD_PERPROCLIBTERM)
      printf("- Per-Process Library Termination\n");

   printf("\n");
}

void DumpObjectTable(FILE *LEFile)
{
   DWORD          i;
   OBJTBLENTRY    ote;

   printf("There are %ld objects in the Object Table.\n", LEHeader.NumObjects);

   fseek(LEFile, LEStart + LEHeader.ObjTblOffset, SEEK_SET);
   for (i=1; i<=LEHeader.NumObjects; i++)
   {
      fread(&ote, sizeof(ote), 1, LEFile);
      printf("Segment #: %ld  Size: 0x%08lX  Reloc Addr: 0x%08lX   # Pages: %ld\n",
            i, ote.VirtualSize, ote.RelocBaseAddr, ote.NumPgTblEntries);
      if (i == LEHeader.AutoDSObj)
         printf("Auto Data Segment\n");
      printf("Flags:\n");
      if (ote.ObjectFlags & OBJ_READABLE)
         printf("Readable\n");
      if (ote.ObjectFlags & OBJ_WRITEABLE)
         printf("Writeable\n");
      if (ote.ObjectFlags & OBJ_EXECUTABLE)
         printf("Executable\n");
```

Listing 10.3 (continued)

```
      if (ote.ObjectFlags & OBJ_RESOURCE)
         printf("Resource\n");
      if (ote.ObjectFlags & OBJ_DISCARDABLE)
         printf("Discardable\n");
      if (ote.ObjectFlags & OBJ_SHARED)
         printf("Shared\n");
      if (ote.ObjectFlags & OBJ_PRELOAD)
         printf("Preload\n");
      if (ote.ObjectFlags & OBJ_INVALID)
         printf("Invalid\n");
      if ( (ote.ObjectFlags & OBJ_RESIDENTCONTIG) == OBJ_RESIDENTCONTIG)
         printf("Resident & Contiguous\n");
      else
      {
         if (ote.ObjectFlags & OBJ_ZEROFILLED)
               printf("Zero-Filled\n");
         if (ote.ObjectFlags & OBJ_RESIDENT)
               printf("Resident\n");
      }
      if (ote.ObjectFlags & OBJ_RESERVED)
         printf("Reserved\n");
      if (ote.ObjectFlags & OBJ_1616ALIAS)
         printf("16:16 Alias Required\n");
      if (ote.ObjectFlags & OBJ_BIGDEFAULT)
         printf("'Big' bit for segment descriptor\n");
      if (ote.ObjectFlags & OBJ_CONFORM)
         printf("Object is conforming to code\n");
      if (ote.ObjectFlags & OBJ_PRIVILEGE)
         printf("Object I/O privilege level\n");
      printf("\n");
   }
}

void DumpNameTable(FILE *LEFile, DWORD size)
{
   DWORD    Curr;
   BYTE     len;
   char     Name[257];
   WORD     Ord;

   Curr = 0;
   while (Curr < size)
   {
      fread(&len, sizeof(len), 1, LEFile);
      Curr += (len + sizeof(len) + sizeof(Ord));
      fread(Name, len, 1, LEFile);
      Name[len] = 0x00;
      fread(&Ord, sizeof(Ord), 1, LEFile);
      printf("%u         %s\n", Ord, Name);
   }

}
```

Listing 10.3 (continued)

```
/* Dumps the Resident and Non-Resident Name Tables */
void DumpNameTables(FILE *LEFile)
{
   printf("\nResident Name Table\n");
   printf("Ordinal   Name\n");
   fseek(LEFile, LEStart + LEHeader.ResNameTable, SEEK_SET);
   DumpNameTable(LEFile, (LEHeader.EntryTable - LEHeader.ResNameTable) - 1);
   printf("\n");
   printf("Non-Resident Name Table\n");
   printf("Ordinal   Name\n");
   fseek(LEFile, LEHeader.NonResTable, SEEK_SET);
   DumpNameTable(LEFile, LEHeader.NonResSize - 1);

}
/*
void DumpImports(FILE *LEFile)
{
   FIXUPREC      FixupRec;
   PROCBYNAME    ProcByName;
   PROCBYORD     ProcByOrd;
   long          CurrLoc;

   fseek(LEFile, LEStart+LEHeader.FixupRecTable, SEEK_SET);


}
*/
void Check4HeaderSurprises()
{
   if (LEHeader.ByteOrder != ORD_LITTLEENDIAN)
      printf("Byte Order is not Little Endian!\n");
   if (LEHeader.WordOrder != ORD_LITTLEENDIAN)
      printf("Word Order is not Little Endian!\n");

   if (LEHeader.CPUType < CPU_386)
      printf("Doesn't require a 386 or better!\n");
   if (LEHeader.OSType != OS_WINDOWS386)
      printf("Not a Windows386 VxD!\n");

   if (LEHeader.ResourceTbl)
      printf("Resource Table Found!\n");
   if (LEHeader.NumResources)
      printf("Num Resources Found!\n");
   if (LEHeader.ModDirectTable)
      printf("Module Directive Table Found!\n");
   if (LEHeader.NumModDirect)
      printf("NumModDirect Found!\n");
   if (LEHeader.NumInstPreload)
      printf("Instance Preloads Found!\n");
   if (LEHeader.NumInstDemand)
      printf("Instance Demands Found!\n");
   if (LEHeader.FixupChecksum)
      printf("Fixup Checksum Found!\n");
   if (LEHeader.LoaderChecksum)
      printf("Loader Checksum Found!\n");
   if (LEHeader.PerPageChecksum)
      printf("Per-Page Checksum Found!\n");
   if (LEHeader.NonResChecksum)
      printf("Non-Resident Name Table Checksum Found!\n");
}
```

Listing 10.3 (continued)

```
void DumpLEFile(FILE *LEFile)
{
   LEStart = ftell(LEFile);

   fread(&LEHeader, sizeof(LEHeader), 1, LEFile);
   if (LEHeader.LEMagic[0] != 'L' || LEHeader.LEMagic[1] != 'E')
   {
      printf("This is not an 'LE' executable.\n\n");
      return;
   }

   DumpHeaderInfo();

   Check4HeaderSurprises();

   DumpObjectTable(LEFile);

   DumpNameTables(LEFile);

//   if (LEHeader.NumImports)
//      DumpImports(LEFile);

}

void Usage(void)
{
   printf("Usage: LEDump filename[.386]\n\n");
}

int main(int argc, char *argv[])
{

   char       filename[256];
   FILE       *LEFile;

   if (argc < 2) {
      Usage();
      return EXIT_FAILURE;
   }

   strcpy(filename, argv[1]);

   if (!strchr(filename, '.'))
      strcat(filename, ".386");

   if ((LEFile = fopen(filename, "rb")) == NULL) {
      printf("%s does not exist!\n", filename);
      return EXIT_FAILURE;
   }

   if (SkipMZ(LEFile))
   {
      printf("Dumping %s\n", filename);
      DumpLEFile(LEFile);
   }
   fclose(LEFile);
   return EXIT_SUCCESS;
}
```

Appendix A

Contents of the Companion Code Disk

The included code disk contains all of the source code mentioned in the book, as well as make files for Microsoft's and Borland's compilers. The files are organized by chapter with all the files for each chapter sorted in the same directory, except when there was a clear delineation in functionality. In such cases, we provided separate directories beneath the chapter directory. We will also keep the source code updated on the Internet at `http://www.mnsinc.com/peted/`. The executables will also be available on the web site. The following page contains a list of the companion code disk contents.

```
FILEFORMATS
CHAP1
                    SETVAL
                                        SETVAL.C
                                        MAKEFILE.BOR
                                        MAKEFILE.MS
                    DUMP
                                        DUMP.C
                                        MAKEFILE.BOR
                                        MAKEFILE.MS
CHAP3
                    MAKEFILE.BOR
                    MAKEFILE.MS
                    SHGDUMP.C
                    SHEDEDIT.H
CHAP4
                    HLPDUMP2.H
                    HLPDUMP2.C
                    WINHELP.H
                    MAKEFILE.BOR
                    MAKEFILE.MS

                    TOPICDMP
                                        TOPICDMP.C
                                        WHSTRUCT.H
                                        MAKEFILE.BOR
                                        MAKEFILE.MS
CHAP6
                    COMP.C
                    DECOMP.H
                    DECOMP.C
                    MAKEFILE.BOR
                    MAKEFILE.MS
CHAP7
                    RESTYPES.H
                    MAKEFILE.BOR
                    RES2RC.C
                    MAKEFILE.MS
CHAP8
                    PIFDUMP.C
                    PIFSTRUC.H
                    MAKEFILE.MS
                    MAKEFILE.BOR
CHAP9
                    SUCKW3
                                        SUCKW3.C
                                        SUCKW3.H
                                        MAKEFILE.BOR
                                        MAKEFILE.MS
                    W4DECOMP
                                        W4DECOMP.C
                                        W4DECOMP.H
                                        MAKEFILE.BOR
                                        MAKEFILE.MS
CHAP10
                    LEDUMP.C
                    LEDUMP.H
                    MAKEFILE.MS
                    MAKEFILE.BOR
```

Annotated Bibliography

1. Pete Davis, "`.mrb` and `.shg` File Formats", *Windows/DOS Developer's Joumal*, February 1994.

   A true work of art. I recommend all of his articles as required reading. Actually, I admit, it's a bit out of date. Most of the information was accurate. Enough so that many people were actually able to put it to good use.

2. Pete Davis, "Microsoft's Compression File Format", *Windows/DOS Developer's Journal*, July 1994.

   This article just covered the "SZDD" variation of the compression format. Two days after this article was available in stores, I found out about the older "SZ" and newer "KWAJ" algorithms from, as Dave Barry would say, "Alert Readers". I was pretty bummed.

3. Pete Davis, "Documenting Documentation: The Windows `.HLP` File Format, Part1", *Dr. Dobb's Journal*, September 1993.

4. Pete Davis, "Documenting Documentation: The Windows `.HLP` File Format, Part2", *Dr. Dobb's Journal*, October 1993.

   As with the other file formats I wrote articles about, this one is out of date and incomplete. It was a good start, I think.

5. Microsoft Corporation, `SHDFMT.DOC`.

   This file has a `README.1ST` from Rick Segal at Microsoft. The `README.1ST` warns that "This [the SHED file format] is gonna change. I promise." Well, he kept his word. They changed it quite a bit between when this document was written and the SHED editor came out. I say that because it's completely inaccurate. It's possible it was just inaccurate to begin with but hey, Microsoft wrote it, so I'm sure it was originally entirely accurate (or at least as much as they thought we deserved). I actually found this after I had done most of the reverse-engineering, so it wasn't really a great help. The only thing I could have done was try to adopt their naming convention, but I couldn't find many names that matched the function of the fields in my structures.

6. Charles Petzold, *Programming Windows 3.1,* Third Edition, Redmond WA : Microsoft Press, 1990.

   How do you write anything about Windows without in some way owing some of your knowledge to this book?

7. Alex Fedorov and Dmitry Rogatkin, "The Windows .RES File Format", *Dr. Dobb's Journal,* August 1993.

   This is the first public description of the .RES file format. Alex and Dmitry did a great job. I knew Alex before he wrote the article and he has sent me two issues of *ComputerPress,* a Russian magazine of which he is executive editor in Moscow. Can someone please translate this to English for me?

8. Mike Maurice, "The PIF File Format, or Topview (sort of) Lives!", *Dr. Dobb's Journal*, July 1993.

   Like several other file formats, I've never understood why Microsoft didn't just document it. On the other hand, I'm surprised it took so long for someone to do it. Thanks to Mike Maurice for doing a stand-up job.

9. Jim Mischel, *The Developer's Guide to WINHELP.EXE,* NewYork NY : John Wiley & Sons, Inc, 1994.

   This is the only really good book that covers almost all the major aspects of WinHelp. It's a necessity for any serious WinHelp authors or DLL developers. It's either this or use 20 different articles, books, etc., as your source of WinHelp information. Also a good source of undocumented WinHelp information.

10. Microsoft Windows Software Development Kit v3.1, 1992.

    If you don't know what this is, you probably bought the wrong book. In particular, Volume 3 "Message, Structures, and Macros", and Volume 4 "Resources". Some of the undocumented file formats are related to the documented file formats in some way or another, and this book is a good source for the documented ones.

11. Mark Nelson, The Data Compression Book, San Mateo CA : M&T Books, 1992.

    This is the best book I've ever read on data compression. (Okay, it's the only one I've ever read.) As much as data compression can be explained in English, it's done in this book. Data compression can be very complex, and most explanations have been cryptic to me. Mark Nelson did a really great job of making it understandable by your average programmer.

12. Matt Pietrek, "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format", *Microsoft Systems Journal*, Vol. 9 No. 3, March 1994.

    This is a really good description of the Portable Executable (PE) file format and gave some insights into the LE file format.

13. IBM Boca Programming Center, "IBM OS/2 32 bit Object Module Format (OMF) and Linear eXecutable Module Format(LX): Revision 6".

    This was passed along to me by Clive Turvey who also provided most of the information on the W4 file format. As the title declares, this is a description of both the OMF and the LX file formats for OS/2. It contains a fairly detailed description of the LX file format, upon which the LE file format was based. It's not exactly light reading material, but it appears to be very complete and gave me enough information to provide the LE file format.

14. Author Unknown (believed to be IBM), "LX - Linear eXecutable Module Format Description", June 3, 1992.

    I found this file on the Internet as LXEXE.ZIP (ftp.watcom.on.ca in the /pub/bbs/general directory). There was no credit as to the author, but it's almost identical to the IBM document sited above, so my guess is that it's either an earlier or later edit. This version has only the LX file format and not the OMF. The postmaster of the site informed me that the file had been obtained much earlier from a site he couldn't remember, so there's no real record of where it originated.

Index