Windows Help File Format,

Annotation File Format,

SHG and MRB File Format

This documentation describes the file format parsed by HELPDECO, because Microsoft did not publish the file formats used by WinHelp and MultiMedia Viewers, and created by HC30, HC31, HCP, HCRTF, HCW, MVC, MMVC and WMVC. This way it is not an official reference, but the result of many weekends of work dumping 500+ help files and trying to understand what all the bytes may mean.

I would like to thank Pete Davis, who first tried to describe 'The Windows Help File Format' in Dr. Dobbs Journal, Sep/Oct 1993, and Holger Haase, who did a lot of work on picture file formats and Bent Lynggaard for the information on free lists in help files and unused bytes in B+ trees.

* Revision 1: Fixed hash value calculation and |FONT, minor additions
* Revision 2: Transparent bitmaps, {button}, and {mci} commands
* Revision 3: Unknown in Paragraphinfo changed, minor additions
* Revision 4: CTXOMAP corrected, bitmap dimensions dpi - not PelsPerMeter
* Revision 5: MacroData in HotspotInfo added, Annotation file format added
* Revision 6: [MACROS] section / internal file |Rose added, MVB font structure
* Revision 7: [GROUPS] section \*.GRP and [CHARTAB] section \*.tbl file format
* Revision 8: free list, clarified TOPICPOS/TOPICOFFSET
* Revision 9: B+ tree unused bytes and what I found out about GID files

# Windows Help File Format

## Headers

A help file starts with a header, the only structure at a fixed place:

|  |  |  |
| --- | --- | --- |
| Long | Magic | 0x00035F3F |
| Long | DirectoryStart | offset of FILEHEADER of internal directory |
| Long | FirstFreeBlock | offset of FREEHEADER or -1L if no free list |
| Long | EntireFileSize | size of entire help file in bytes |
| Char | HelpFileContent[EntireFileSize-16] | the remainder of the help file |

At offset DirectoryStart the FILEHEADER of the internal directory is located:

|  |  |  |
| --- | --- | --- |
| Long | ReservedSpace | size reserved including FILEHEADER |
| Long | UsedSpace | size of internal file in bytes |
| Unsigned char | FileFlags | normally 4 |
| Char | FileContent[UsedSpace] | the bytes contained in the internal file |
| Char | FreeSpace[ReservedSpace-UsedSpace-9] |  |

The FILEHEADER of the internal directory is followed by UsedSpace bytes containing the internal directory which is used to associate FileNames and FileOffsets. The directory is structured as a B+ tree.

A B+ tree is made of leaf-pages and index-pages of fixed size, one of which is the root-page. All entries are contained in leaf-pages. If more entries are required than fit into a single leaf-page, index-pages are used to locate the leaf-page which contains the required entry.

A B+ tree starts with a BTREEHEADER telling you the size of the B+ tree pages, the root-page, the number of levels, and the number of all entries in this B+ tree. You must follow (NLevels-1) index-pages before you reach a leaf-page.

|  |  |  |
| --- | --- | --- |
| unsigned short | Magic | 0x293B |
| unsigned short | Flags | bit 0x0002 always 1, bit 0x0400 1 if directory |
| unsigned short | PageSize | 0x0400=1k if directory, 0x0800=2k else, or 4k |
| char | Structure[16] | string describing format of data: |
| 'L' = long (indexed) |
| 'F' = NUL-terminated string (indexed) |
| 'i' = NUL-terminated string (indexed) |
| '2' = short |
| '4' = long |
| 'z' = NUL-terminated string |
| '!' = long count value, count/8 \* record |
| long filenumber |
| long TopicOffset |
| short | MustBeZero | 0 |
| short | PageSplits | number of page splits B+ tree has suffered |
| short | RootPage | page number of B+ tree root page |
| short | MustBeNegOne | 0xFFFF |
| short | TotalPages | number of B+ tree pages |
| short | NLevels | number of levels of B+ tree |
| long | TotalBtreeEntries | number of entries in B+ tree |
| Char | Page[TotalPages][PageSize] | the pages the B+ tree is made of |

If NLevel is greater than 1, RootPage is the page number of an index-page. Index-pages start with a BTREEINDEXHEADER and are followed by an array of BTREEINDEX structures, in case of the internal directory containing pairs of FileNames and PageNumbers.

(STRINGZ is a NUL-terminated string, sizeof(STRINGZ) is strlen(string)+1).

PageNumber gets you to the next page containing entries lexically starting at FileName, but less than the next FileName. PreviousPage gets you to the next page if the desired FileName is lexically before the first FileName.

|  |  |  |
| --- | --- | --- |
| unsigned short | Unused | number of free bytes at end of this page |
| short | NEntries | number of entries in this index-page |
| short | PreviousPage | page number of previous page |
| struct | DIRECTORYINDEXENTRY[NEntries] | and this is the structure of directory index-pages |
| STRINGZ | FileName | varying length NUL-terminated string |
| short | PageNumber | page number of page dealing with FileName and above |

After NLevels-1 of index-pages you will reach a leaf-page starting with a BTREENODEHEADER followed by an array of BTREELEAF structures, in case of the internal directory containing pairs of FileNames and FileOffsets.

You may follow the PreviousPage entry in all NLevels-1 index-pages to reach the first leaf-page, then iterate thru all entries and use NextPage to follow the double linked list of leaf-pages until NextPage is -1 to retrieve a sorted list of all TotalBtreeEntries entries contained in the B+ tree.

|  |  |  |
| --- | --- | --- |
| unsigned short | Unused | number of free bytes at end of this page |
| short | NEntries | number of entries in this leaf-page |
| short | PreviousPage | page number of previous leaf-page or -1 if first |
| short | NextPage | page number of next leaf-page or -1 if last |
| struct | DIRECTORYLEAFENTRY[NEntries] | and this is the structure of directory leaf-pages |
| STRINGZ | FileName | varying length NUL-terminated string |
| Long | FileOffset | offset of FILEHEADER of internal file FileName relative to beginning of help file |

At offset FirstFreeBlock the first FREEHEADER is located. It contains:

|  |  |  |
| --- | --- | --- |
| Long | FreeSpace | number of bytes unused, including this header |
| Long | NextFreeBlock | offset of next FREEHEADER or -1L if end of list |
| Char | Unused[FreeSpace-8] | unused bytes |

All unused portions of the help file are linked together using FREEHEADERs.

Now that you are able to locate the position of an internal file in the help file, let's describe what they contain. Remember that each FileOffset first takes you to the FILEHEADER of the internal file. The structures described next are located just behind this FILEHEADER.

## |SYSTEM

The first one to start with is the **|SYSTEM** file. This is the SYSTEMHEADER, the structure of the first bytes of this internal file:

|  |  |  |
| --- | --- | --- |
| Short | Magic | 0x036C |
| Short | Minor | help file format version number |
| 15 = HC30 Windows 3.0 help file |
| 21 = HC31 Windows 3.1 help file |
| 27 = WMVC/MMVC media view file |
| 33 = MVC or HCW 4.00 Windows 95 |
| Short | Major | 1 |
| time\_t | GenDate | help file created seconds after 1.1.1980, or 0 |
| unsigned short | Flags | see below |

Use Minor and Flags to find out how the help file was compressed:

* Minor <= 16: not compressed, TopicBlockSize 2k
* Minor > 16:
  + Flags=0: not compressed, TopicBlockSize 4k
  + Flags=4: LZ77 compressed, TopicBlockSize 4k
  + Flags=8: LZ77 compressed, TopicBlockSize 2k

Additionally the help file may use phrase compression (oldstyle or Hall).

If Minor is 16 or less, the help file title follows the SYSTEMHEADER:

|  |  |
| --- | --- |
| STRINGZ | HelpFileTitle |

If Minor is above 16, one or more SYSTEMREC records follow instead up to the internal end of the |SYSTEM file:

|  |  |  |
| --- | --- | --- |
| unsigned short | RecordType | type of data in record |
| unsigned short | DataSize | size of data |
| Char | Data[DataSize] | Dependent on RecordType |

There are different RecordTypes defined, each storing different Data. They mainly contain what was specified in the help project file.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **RecordType** | | **Data** | | |
| 1 | TITLE | STRINGZ | Title | Help File Title |
| 2 | COPYRIGHT | STRINGZ | Copyright | copyright notice shown in AboutBox |
| 3 | CONTENTS | TOPICOFFSET | Contents | topic offset of starting topic |
| 4 | CONFIG | STRINGZ | Macro | all macros executed on opening |
| 5 | ICON | Windows \*.ICO file |  | See WIN31WH on icon file format |
| 6 | WINDOW | struct |  | Windows defined in the HPJ-file |
| 6 | WINDOW | typedef struct |  | Viewer 2.0 Windows defined in MVP-file |
| 8 | CITATION | STRINGZ | Citation | the Citation printed |
| 9 | LCID | short LCID[4] | Language ID | Windows 95 (HCW 4.00) |
| 10 | CNT | STRINGZ | ContentFileName | CNT file name, Windows 95 (HCW 4.00) |
| 11 | CHARSET | unsigned short | Charset | charset, Windows 95 (HCW 4.00) |
| 12 | DEFFONT | struct |  | default dialog font, Windows 95 (HCW 4.00) |
| 12 | FTINDEX | STRINGZ | dtype | Multimedia Help Files dtypes (*only for MVP*) |
| 13 | GROUPS | STRINGZ | Group | defined GROUPs, Multimedia Help File |
| 14 | INDEX\_S | STRINGZ | IndexSeparators | separators, Windows 95 (HCW 4.00) |
| 14 | KEYINDEX | struct |  | Multimedia Help Files (*only for MVP*) |
| 18 | LANGUAGE | STRINGZ | language | defined language, Multimedia Help Files |
| 19 | DLLMAPS | struct |  | defined DLLMAPS, Multimedia Help Files |

Remark: Here MVP refers to variants of the HLP format used by old Microsoft Multimedia Viewer

**WINDOW Structure (First one):**

struct

{

struct

{

unsigned short TypeIsValid : 1;

unsigned short NameIsValid : 1;

unsigned short CaptionIsValid : 1;

unsigned short XIsValid : 1;

unsigned short YIsValid : 1;

unsigned short WithIsValid : 1;

unsigned short HeigthIsValid : 1;

unsigned short MaximizeWindow : 1;

unsigned short RGBIsValid : 1;

unsigned short RGBNSRIsValid : 1;

unsigned short WindowsAlwaysOnTop : 1;

unsigned short AutoSizeHeight : 1;

} Flags;

char Type[10]; // type of window

char Name[9]; // window name

char Caption[51]; // caption of window

short X; // x coordinate of window(0..1000)

short Y; // y coordinate of window(0..1000)

short Width; // width of window(0..1000)

short Height; // height of window(0..1000)

short Maximize; // maximize flag and window styles

COLORREF Rgb; // color of scrollable region

COLORREF RgbNsr; // color of non scrollable region

} Window;

**WINDOW Structure (2nd one):**

typedef struct

{

unsigned short Flags;

char Type[10]; /\* type of window \*/

char Name[9]; /\* window name \*/

char Caption[51]; /\* caption for window \*/

unsigned char MoreFlags;

short X; /\* x coordinate of window (0..1000) \*/

short Y; /\* y coordinate of window (0..1000) \*/

short Width; /\* width of window (0..1000) \*/

short Height; /\* height of window (0..1000) \*/

short Maximize; /\* maximize flag and window styles \*/

COLORREF Rgb1;

char Unknown;

COLORREG Rgb2;

COLORREF Rgb3;

short X2;

short Y2;

short Width2;

short Height2;

short X3;

short Y3;

} Window;

Remark: both WINDOW structures are associated with the same RecordType. You’ll choose the best one based on the data size.

**DEFFONT Structure:**

struct

{

unsigned char HeightInPoints;

unsigned char Charset;

STRINGZ FontName;

} DefFont;

**Remark:** this structure is filled only for MVP files. For regular Help Files, the RecordType Id 12 gives DEFFONT data.

**KEYINDEX Structure:**

struct

{

char btreename[10]; // btreename[1] is footnote character

char mapname[10];

char dataname[10];

char title[80];

} KeyIndex;

**DLLMAPS Structure:**

struct

{

STRINGZ Win16RetailDLL;

STRINGZ Win16DebugDLL;

STRINGZ Win32RetailDLL;

STRINGZ Win32DebugDLL;

} DLLNames;

## |Phrases

If the help file is phrase compressed, it contains an internal file named **|Phrases**. Windows 3.0 help files generated with HC30 use the following uncompressed structure to store phrases. A phrase is not NUL-terminated, instead use the next PhraseOffset to locate the end of the phrase string (there is one more phrase offset stored than phrases are defined to allow for this).

unsigned short NumPhrases number of phrases in table

unsigned short OneHundred 0x0100

unsigned short PhraseOffset[NumPhrases+1] PhraseOffset[0]==2\*(NumPhrases+1)

char Phrase[NumPhrases][PhraseOffset[PhraseNum+1]-PhraseOffset[PhraseNum]]

Windows 3.1 help files generated using HC31 and later always LZ77 compress the Phrase character array. Read NumPhrases, OneHundred, DecompressedSize, and NumPhrases+1 PhraseOffset values. Allocate DecompressedSize bytes for the Phrase character array and decompress the UsedSpace-2\*NumPhrases-10 remaining bytes into the allocated space to retrieve the phrase strings.

unsigned short NumPhrases number of phrases in table

unsigned short OneHundred 0x0100

long DecompressedSize

unsigned short PhraseOffset[NumPhrases+1] PhraseOffset[0]==2\*(NumPhrases+1)

---- the remaining part is LZ77 compressed

char Phrase[NumPhrases][PhraseOffset[PhraseNum+1]-PhraseOffset[PhraseNum]]

The LZ77 decompression algorithm can best be described like this:

Take the next byte

Start at the least significant bit

If the bit is cleared

Copy 1 byte from source to destination

Else

Get the next WORD into the struct { unsigned pos:12; unsigned len:4; }

Copy len+3 bytes from destination-pos-1 to destination

Loop until all bits are done

Loop until all bytes are consumed

See end of this file for a detailed algorithm.

Some MVBs use a slightly different layout of internal |Phrases file:

unsigned short EightHundred 0x0800

unsigned short NumPhrases number of phrases in table

unsigned short OneHundred 0x0100

long DecompressedSize

char unused[30]

unsigned short PhraseOffset[NumPhrases+1] PhraseOffset[0]==2\*(NumPhrases+1)

---- the remaining part is LZ77 compressed

char Phrase[NumPhrases][PhraseOffset[PhraseNum+1]-PhraseOffset[PhraseNum]]

## |PhrIndex

Windows 95 (HCW 4.00) may use Hall compression and the internal files |PhrIndex and |PhrImage to store phrases. Both must be used to build a table of phrases and PhraseOffsets. |PhrIndex starts with this header:

long Magic 1L

long NEntries

long CompressedSize

long PhrImageSize

long PhrImageCompressedSize

long Always0 0L

unsigned short BitCount:4

unsigned short UnknownBits:12

unsigned short Always4A00 not really always

The remaining data is bitcompressed. Use this algorithm to build a table

of PhraseOffsets:

short n,i; long mask=0,\*ptr=(long \*)(&always4A00+1);

int GetBit(void)

{

ptr+=(mask<0);

mask=mask\*2+(mask<=0);

return (\*ptr&mask)!=0;

}

PhaseOffset[0]=0;

for(i=0;i<NEntries;i++)

{

for(n=1;GetBit();n+=1<<BitCount) ;

if(GetBit()) n+=1;

if(BitCount>1) if(GetBit()) n+=2;

if(BitCount>2) if(GetBit()) n+=4;

if(BitCount>3) if(GetBit()) n+=8;

if(BitCount>4) if(GetBit()) n+=16;

PhraseOffset[i+1]=PhraseOffset[i]+n;

}

Just behind the bitcompressed phrase length information (on a 32-bit boundary, that's why GetBit consumed longs) follow NumPhrases bits (one bit for each phrase). It is assumed that this information is used for the full text search capability to exclude certain phrases.

## |PhrImage

The |PhrImage file stores the phrases. A phrase is not NUL-terminated. Use PhraseOffset[NumPhrase] and PhraseOffset[NumPhrase+1] to locate beginning and end of the phrase string. We generated one more PhraseOffset to allow for this. |PhrImage is LZ77 compressed if PhrImageCompressedSize is not equal to PhrImageSize. Otherwise you may take it as stored.

## |FONT

The next internal file described is the |FONT file, which uses this header:

unsigned short NumFacenames number of face names

unsigned short NumDescriptors number of font descriptors

unsigned short FacenamesOffset start of array of face names

relative to &NumFacenames

unsigned short DescriptorsOffset start of array of font descriptors

relative to &NumFacenames

--- only if FacenamesOffset >= 12

unsigned short NumStyles number of style descriptors

unsigned short StyleOffset start of array of style descriptors

relative to &NumFacenames

--- only if FacenamesOffset >= 16

unsigned short NumCharMapTables number of character mapping tables

unsigned short CharMapTableOffset start of array of character mapping

table names relative to &NumFacenames

The face name array is located at FacenamesOffset and contains strings, which are Windows font names or in case of multimedia files a Windows font name concatenated with ',' and the character mapping table number. Short strings are NUL-terminated, but a string may use all bytes for characters.

char FaceName[NumFacenames][(DescriptorsOffset-FacenamesOffset)/NumFacenames]

At DescriptorsOffset is an array located describing all fonts used in the help file. If this kind of descriptor appears in a help file, any metric value is given in HalfPoints.

struct oldfont

{

struct

{

unsigned char Bold:1

unsigned char Italic:1

unsigned char Underline:1

unsigned char StrikeOut:1

unsigned char DoubleUnderline:1

unsigned char SmallCaps:1

}

Attributes

unsigned char HalfPoints PointSize \* 2

unsigned char FontFamily font family. See values below

unsigned short FacenameIndex index into FaceName array

unsigned char FGRGB[3] RGB values of foreground

unsigned char BGRGB[3] unused background RGB Values

}

FontDescriptor[NumDescriptors]

#define FAM\_MODERN 0x01 This is a different order than

#define FAM\_ROMAN 0x02 FF\_ROMAN, FF\_SWISS, etc. of

#define FAM\_SWISS 0x03 windows !

#define FAM\_TECH 0x03

#define FAM\_NIL 0x03

#define FAM\_SCRIPT 0x04

#define FAM\_DECOR 0x05

Multimedia MVB files use different structures to store font descriptors. Assume this structure for descriptors if FacenamesOffset is at least 12. If this kind of descriptor is used, any metric is given in twips.

struct newfont

{

unsigned char unknown1

short FacenameIndex

unsigned char FGRGB[3]

unsigned char BGRGB[3]

unsigned char unknown5

unsigned char unknown6

unsigned char unknown7

unsigned char unknown8

unsigned char unknown9

long Height

unsigned char mostlyzero[12]

short Weight

unsigned char unknown10

unsigned char unknown11

unsigned char Italic

unsigned char Underline

unsigned char StrikeOut

unsigned char DoubleUnderline

unsigned char SmallCaps

unsigned char unknown17

unsigned char unknown18

unsigned char PitchAndFamily Same values as windows LOGFONT

}

FontDescriptor[NumDescriptors]

Assume this structure for descriptors if FacenamesOffset is at least 16. If this kind of descriptor is used, any metric is given in twips.

struct mvbfont

{

short FacenameIndex index into Facename array

short StyleNumber 0 if not used

unsigned char unknown3

unsigned char unknown4

unsigned char FGRGB[3]

unsigned char BGRGB[3]

long Height negative (incl. external leading)

unsigned char mostlyzero[12]

short Weight

unsigned char unknown10

unsigned char unknown11

unsigned char Italic

unsigned char Underline

unsigned char StrikeOut

unsigned char DoubleUnderline

unsigned char SmallCaps

unsigned char unknown17

unsigned char unknown18

unsigned char PitchAndFamily Same values as windows LOGFONT

unsigned char unknown20

unsigned char unknown21

}

FontDescriptor[NumDescriptors]

If FacenamesOffset is at least 12, the |FONT file supports character styles.

StyleNumber-1 of the FontDescriptor indexes into this array located at

StyleOffset in |FONT.

struct

{

short StyleNum

short BasedOnStyleNum 0 if not used

struct Font struct newfont or struct mvbfont

char unknown[35]

char StyleName[65]

}

Style[NumStyles]

If FacenamesOffset is at least 16, the |FONT file supports character mapping

tables.

The array of character mapping table file names is located in |FONT at

CharMapTableOffset and contains strings of the internal filename of the

character mapping table concatenated with ',' and the character mapping table

number. The entries are not sorted by character mapping table numbers. Short

strings are NUL-terminated, but a string may use up all bytes.

char CharMapTableName[NumCharMapTables][32]

## |TOMAP

Windows 3.0 (HC30) uses topic numbers that start at 16 for the first topic to identify topics. To retrieve the location of the TOPICLINK for the TOPICHEADER of a certain topic (in |TOPIC explained later), use the |TOMAP file.

It contains an array of topic positions. Index with TopicNumber (do not subtract 16). TopicPos[0] points to the topic specified as INDEX in the help project.

TOPICPOS TopicPos[UsedSpace/4]

## |CONTEXT

Windows 3.1 (HC31) uses hash values of context names to identify topics. To get the location of the topic, search the B+ tree of the internal file

|CONTEXT:

Structure of |CONTEXT index-page entries:

struct

{

long HashValue

short PageNumber

}

CONTEXTINDEXENTRY[NEntries]

Structure of |CONTEXT leaf-page entries:

struct

{

long HashValue hash value of context id

TOPICOFFSET TopicOffset position

}

CONTEXTLEAFENTRY[NEntries]

To calculate the HashValue hash from a context id ptr do this:

signed char table[256]=

{

'\x00', '\xD1', '\xD2', '\xD3', '\xD4', '\xD5', '\xD6', '\xD7',

'\xD8', '\xD9', '\xDA', '\xDB', '\xDC', '\xDD', '\xDE', '\xDF',

'\xE0', '\xE1', '\xE2', '\xE3', '\xE4', '\xE5', '\xE6', '\xE7',

'\xE8', '\xE9', '\xEA', '\xEB', '\xEC', '\xED', '\xEE', '\xEF',

'\xF0', '\x0B', '\xF2', '\xF3', '\xF4', '\xF5', '\xF6', '\xF7',

'\xF8', '\xF9', '\xFA', '\xFB', '\xFC', '\xFD', '\x0C', '\xFF',

'\x0A', '\x01', '\x02', '\x03', '\x04', '\x05', '\x06', '\x07',

'\x08', '\x09', '\x0A', '\x0B', '\x0C', '\x0D', '\x0E', '\x0F',

'\x10', '\x11', '\x12', '\x13', '\x14', '\x15', '\x16', '\x17',

'\x18', '\x19', '\x1A', '\x1B', '\x1C', '\x1D', '\x1E', '\x1F',

'\x20', '\x21', '\x22', '\x23', '\x24', '\x25', '\x26', '\x27',

'\x28', '\x29', '\x2A', '\x0B', '\x0C', '\x0D', '\x0E', '\x0D',

'\x10', '\x11', '\x12', '\x13', '\x14', '\x15', '\x16', '\x17',

'\x18', '\x19', '\x1A', '\x1B', '\x1C', '\x1D', '\x1E', '\x1F',

'\x20', '\x21', '\x22', '\x23', '\x24', '\x25', '\x26', '\x27',

'\x28', '\x29', '\x2A', '\x2B', '\x2C', '\x2D', '\x2E', '\x2F',

'\x50', '\x51', '\x52', '\x53', '\x54', '\x55', '\x56', '\x57',

'\x58', '\x59', '\x5A', '\x5B', '\x5C', '\x5D', '\x5E', '\x5F',

'\x60', '\x61', '\x62', '\x63', '\x64', '\x65', '\x66', '\x67',

'\x68', '\x69', '\x6A', '\x6B', '\x6C', '\x6D', '\x6E', '\x6F',

'\x70', '\x71', '\x72', '\x73', '\x74', '\x75', '\x76', '\x77',

'\x78', '\x79', '\x7A', '\x7B', '\x7C', '\x7D', '\x7E', '\x7F',

'\x80', '\x81', '\x82', '\x83', '\x0B', '\x85', '\x86', '\x87',

'\x88', '\x89', '\x8A', '\x8B', '\x8C', '\x8D', '\x8E', '\x8F',

'\x90', '\x91', '\x92', '\x93', '\x94', '\x95', '\x96', '\x97',

'\x98', '\x99', '\x9A', '\x9B', '\x9C', '\x9D', '\x9E', '\x9F',

'\xA0', '\xA1', '\xA2', '\xA3', '\xA4', '\xA5', '\xA6', '\xA7',

'\xA8', '\xA9', '\xAA', '\xAB', '\xAC', '\xAD', '\xAE', '\xAF',

'\xB0', '\xB1', '\xB2', '\xB3', '\xB4', '\xB5', '\xB6', '\xB7',

'\xB8', '\xB9', '\xBA', '\xBB', '\xBC', '\xBD', '\xBE', '\xBF',

'\xC0', '\xC1', '\xC2', '\xC3', '\xC4', '\xC5', '\xC6', '\xC7',

'\xC8', '\xC9', '\xCA', '\xCB', '\xCC', '\xCD', '\xCE', '\xCF'

}

for(hash=0L;\*ptr;ptr++) hash=(hash\*43)+table[(unsigned char)\*ptr];

Remember that only 0-9, A-Z, a-z, \_ and . are legal characters for context ids in Win 3.1 (HC31). Only Windows 95 (HCRTF) allows nearly all characters. The hash value for an empty string is 1.

## |CTXOMAP

If your help project file had a [MAP] section, the internal file |CTXOMAP contains an array to assign map ids to topic offsets.

short NEntries

struct

{

long MapID

TOPICOFFSET TopicOffset

}

CTXOMAPENRTY[NEntries]

## |xWBTREE, |xWDATA, |xWMAP, |xKWBTREE, |xKWDATA, |xKWMAP

To locate a keyword assigned using a x-footnote (x may be A-Z, a-z), use the |xWDATA, |xWBTREE and |xWMAP internal files. |xWBTREE tells you how often a certain Keyword is defined in the help file.

Structure of |xWBTREE index page entries:

struct

{

STRINGZ Keyword

short PageNumber

}

xWBTREEINDEXENTRY[NEntries]

Structure of |xWBTREE leaf page entries:

struct

{

STRINGZ Keyword

short Count number of times keyword is referenced

long KWDataOffset this is the offset into |xWDATA

}

xWBTREELEAFENTRY[NEntries]

KWBTREE files in WinHlp32 GID files are structured differently (they have a different description in the structure field of the BTREEHEADER) and pack former KWBTREE and KWDATA files into one:

Structure of |xWBTREE leaf page entries in Win95 GID files:

struct

{

STRINGZ Keyword

long Size size of following record

struct

{

long FileNumber ?

long TopicOffset this is the offset into |xWDATA

}

record[Size/8]

}

xWBTREELEAFENTRY[NEntries]

The |xWDATA contains an array of topic offsets. The KWDataOffset from the |xWBTREE tells you where to seek to in the |xWDATA file to read Count topic offsets.

TOPICOFFSET KeywordTopicOffset[UsedSpace/4]

And the topic offset retrieved tells you which location the Keyword was assigned to. It is -1L if the Keyword is assigned to a macro using the [MACROS] section of HCRTF 4.0 (see description of |Rose file).

The |xWMAP contains an array that tells you where to find the n-th keyword in the |xWBTREE. You don't need to use this file but it allows for faster scrolling lists of alphabetically ordered Keywords. (WinHelp search dialog).

struct

{

long KeywordNumber number of first keyword on leaf-page

unsigned short PageNum B+ tree page number

}

xWMAP[UsedSpace/6]

Similarily |xKWBTREE B+ tree and |xKWDATA, |xKWMAP files (where x may be 0-9, A-Z, a-z) are built from K-x:footnotes and [KEYINDEX] declarations of multimedia files.

## |TTLBTREE

If you want to know the topic title assigned using the $-footnote, take a look into the |TTLBTREE internal file, which contains topic titles ordered by topic offsets in a B+ tree. (It is used by WinHelp to display the topic titles in the search dialog).

Structure of |TTLBTREE index page entries:

struct

{

TOPICOFFSET TopicOffset

short PageNumber

}

TTLBTREEINDEXENTRY[NEntries]

Structure of |TTLBTREE leaf page entries:

struct

{

TOPICOFFSET TopicOffset

STRINGZ TopicTitle

}

TTLBTREELEAFENTRY[NEntries]

## |CFn

The |CFn (where n is integer) internal file lists the macros defined in [CONFIG:n] sections of the help project file (HCW 4.00). The file contains as many macro strings as were specified one after another:

STRINGZ Macro[]

## |Rose

The |Rose internal file contains all definitions from the [MACROS] section of a Windows 95 (HCW 4.00) help project file. It is build using a B+ tree. Keywords only appear using hash values but are listed in the |KWBTREE with a TopicPos in the associated |KWDATA array of -1L.

Structure of |Rose index page entries:

struct

{

long KeywordHash

short PageNumber

}

RoseINDEXENTRY[NEntries]

Structure of |Rose leaf page entries:

struct

{

long KeywordHash

STRINGZ Macro

STRINGZ TopicTitle not a real topic title but the string

displayed in the search dialog where

normally topic titles are listed

}

RoseLEAFENTRY[NEntries]

## |TopicId

The |TopicId internal file lists the ContextName assigned to a specific topic offset if the help file was created using the /a option of HCRTF and is built using a B+ tree.

Structure of |TopicId index-page entries:

struct

{

TOPICOFFSET TopicOffset

short PageNumber

}

TopicIdINDEXENTRY[NEntries]

Structure of |TopicId leaf-page entries:

struct

{

TOPICOFFSET TopicOffset

STRINGZ ContextName

}

TopicIdLEAFENTRY[NEntries]

## |Petra

The |Petra internal file contains a B+ tree mentioning the names of the RTF source files the help file was built from for each topic if the help file was created using the /a option of HCRTF.

Structure of |Petra index-page entries:

struct

{

TOPICOFFSET TopicOffset

short PageNumber

}

PetraINDEXENTRY[NEntries]

Structure of |Petra leaf-page entries:

struct

{

TOPICOFFSET TopicOffset

STRINGZ RTFSourceFileName

}

PetraLEAFENTRY[NEntries]

## |Viola

The |Viola internal file contains a B+ tree specifying the default Windows assigned to topics using the > footnote available in HCRTF 4.00.

Structure of |VIOLA index-page entries:

struct

{

TOPICOFFSET TopicOffset

short PageNumber

}

VIOLAINDEXENTRY[NEntries]

Structure of |VIOLA leaf-page entries:

struct

{

TOPICOFFSET TopicOffset

long DefaultWindowNumber

}

VIOLALEAFENTRY[NEntries]

## \*.GID

I have not investigated GID files, as they are created by WinHlp32 and are not needed for help file reconstruction. But they are based on the same file format as Windows help files, so HELPDECO may be used to display their content. Notice the difference between |xWBTREE files stored in \*.GID files and regular files.

## |WinPos

This file has been seen in WinHlp32 GID files, but always contained an empty Btree (with an unknown 'a' in the BTREEHEADER structure).

## |Pete

This file has been seen in WinHlp32 GID files but is currently not understood.

## |Flags

This file has been seen in WinHlp32 GID files but is currently not understood.

## |CntJump

This B+ tree stored in WinHlp32 GID files contains the jump references of the \*.CNT file.

## |CntText

This B+ tree stored in WinHlp32 GID files contains the topic titles of the jumps from the \*.CNT file.

## \*.GRP

MediaView compilers create \*.GRP internal files from group + footnotes assigned to topics. All \*.GRP files follow this structure:

struct

{

unsigned long Magic /\* 0x000A3333 \*/

unsigned long BitmapSize /\* max. 64000 equalling 512000 topics \*/

unsigned long LastTopic /\* first topic in help file has topic number 0 \*/

unsigned long FirstTopic /\* first topic in help file has topic number 0 \*/

unsigned long TopicsUsed /\* in this group \*/

unsigned long TopicCount /\* in whole help file \*/

unsigned long GroupType /\* 1 or 2, see below \*/

unsigned long Unknown[3]

unsigned char Bitmap[BitmapSize] /\* only if GroupType equals 2 \*/

}

GROUP

Starting with the first topic of the help file using TopicNumber 0, a topic is included in a group if TopicNumber is in the range of FirstTopic to LastTopic. If GroupType equals 2 it is additionally required that the corresponding bit starting with lsb of Bitmap[0] is set in the Bitmap.

(Bitmap[TopicNumber>>3]&(1<<(TopicNumber&7))!=0).

## \*.tbl

MediaView compilers store character mapping tables listed in the [CHARTAB] section in internal \*.tbl files using the following binary structure:

struct

{

unsigned short Magic /\* 0x5555 \*/

unsigned short Size

unsigned short Unknown1[2]

unsigned short Entries

unsigned short Ligatures

unsigned short LigLen

unsigned short Unknown2[13]

struct

{

unsigned short class

unsigned short order

unsigned char normal

unsigned char clipboard

unsigned char mac

unsigned char macclipboard

unsigned short unused

}

charentry[Entries]

unsigned char Ligature[Ligatures][LigLen]

}

CHARTAB

A character mapping table is assigned to a font by appending ,x (where x is a decimal number) to the font name and the same ,x to the character mapping table name (in the CHARMAP section of the internal |FONT file).

## |TOPIC

And now to the interesting part, the internal file named |TOPIC. It's divided into blocks of TopicBlockSize bytes, each beginning with a TOPICBLOCKHEADER:

TOPICPOS LastTopicLink points to last topic link in previous block or -1L

TOPICPOS FirstTopicLink points to first topic link in this block

TOPICPOS LastTopicHeader points to topic link of last topic header or 0L, -1L

----

char PlainOrCompressedData[TopicBlockSize-12]

Read the first 12 bytes into a TOPICBLOCKHEADER structure. The remaining TopicBlockSize-12 bytes of each topic block may be compressed using the LZ77 algorithm described above.

Decompress them into a buffer of DecompressSize bytes size if the Flags value contained in the internal |SYSTEM file is 4 or 8 and Minor is greater than 16 (DecompressSize is 16k this way), else they are not compressed and you should copy them as delivered (DecompressSize=TopicBlockSize-12).

Do not decompress to more than DecompressSize bytes. As this would cause ambiguous values for TOPICPOS, the help compilers will not compress more, but fill the remaining topic block with 0es. Data will continue in the next topic block.

### TOPICPOS

A TOPICPOS is used to locate the position of TOPICLINKs in |TOPIC and contains the TopicBlockNumber in its higher bits and an offset into the decompression buffer in its lower bits.

How many bits are used for TopicBlockNumber and TopicBlockOffset depends on the compression method used and the TopicBlockSize:

* (TOPICPOS-sizeof(TOPICBLOCKHEADER))%DecompressSize = TopicBlockOffset
* (TOPICPOS-sizeof(TOPICBLOCKHEADER))/DecompressSize = TopicBlockNumber

A TOPICPOS below sizeof(TOPICBLOCKHEADER) is invalid.

### TOPICLINK

A TOPICLINK (located inside the buffer after decompression, the first of it pointed to by TOPICBLOCKHEADERs FirstTopicLink field) looks like this:

long BlockSize Size of TOPICLINK + LinkData1 + compressed LinkData2

long DataLen2 length of decompressed LinkData2

TOPICPOS PrevBlock Windows 3.0 (HC30): Number of bytes previous

TOPICLINK is located before this TOPICLINC,

including eventually skipped TOPICBLOCKHEADER and

unused bytes.

Windows 3.1 (HC31): TOPICPOS of previous TOPICLINK

TOPICPOS NextBlock Windows 3.0 (HC30): Number of bytes next TOPICLINK

is located behind this TOPICLINK, incl. eventually

skipped TOPICBLOCKHEADER and unused bytes.

Windows 3.1 (HC31): TOPICPOS of next TOPICLINK

long DataLen1 includes size of TOPICLINK

unsigned char RecordType See below

----

char LinkData1[DataLen1-11]

char LinkData2[BlockSize-DataLen1]

LinkData2 may be compressed using Phrase compression. If you find DataLen2>BlockSize-DataLen1 use the following algorithm to decompress:

If your help file contains a |Phrases internal file:

Take the next character. If it's value is 0 or above 15 emit it. Else multiply it with 256, subtract 256 and add the value of the next character. Divide by 2 to get the phrase number. Emit the phrase from the |Phrase file and append a space if the division had a remainder (the number was odd).

If the help file doesn't contain a |Phrases file but instead a |PhrIndex and |PhrImage, it uses Hall compression and the decompression of LinkData2is a bit more difficult:

Take the next character (ch). If ch is even emit the phrase number ch/2. Else if the least two bits are 01 multiply by 64, add 64 and the value of the next character. Emit the Phrase using this number. If the least three bits are 011 copy the next ch/8+1 characters. If the least four bits are 0111 emit ch/16+1 spaces. If the least four bits are 1111 emit ch/16+1 NUL's.

If DataLen2<=BlockSize-DataLen1 the DataLen2 bytes of LinkData2 are stored uncompressed (makes a difference for Hall compression only).

If DataLen2<BlockSize-DataLen1 the remaining BlockSize-DataLen1-DataLen2 bytes are unused, but must be read from the |TOPIC file (this can only happen in Hall-compressed help files).

Now that you know how to decompress the topic data, let's see what you get.

If the TOPICLINK RecordType is 2 you got a topic header in LinkData1. In Windows 3.0 (HC30) the TOPICHEADER is structured like this:

long BlockSize size of topic, including internal topic links

long PrevTopicNumber -1L or 0xFFFF at the beginning of a browse sequence

long NextTopicNumber -1L or 0xFFFF at the end of a browse sequence

In Windows Version 3.1 (HC31) and later it looks like this:

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

long TopicNum topic number

TOPICPOS NonScroll start of non-scrolling region (topic offset) or -1L

TOPICPOS Scroll start of scrolling region (topic offset)

TOPICPOS NextTopic start of next type 2 record

The LinkData2 of Topic RecordType 2 contains NUL terminated strings. The first string is the topic title, the next strings contain all macros to be executed on opening this topic (specified using the ! footnote).

If the TOPICLINK RecordType is 1, you have a Windows 3.0 displayable text record, a RecordType of 0x20 is Windows 3.1 displayable text and 0x23 is a Windows 3.1 table record. A displayable text record may contain multiple paragraphs, but all have the same paragraph formatting. A table record stores all rows and columns of a table and may contain multiple paragraphs of different formatting.

Data inside LinkData1 is sometimes stored as compressed shorts or longs:

* A compressed unsigned short is made of a single byte. Divide by two to get the value if it's even. Divide by two and add 128 times the value of the next byte if it's odd.
* A compressed signed short is made of a single byte. Divide by two and subtract 64 to get the value if it's even. Divide by two, add 128 times the value of the next byte and subtract 16384 if it's odd.
* A compressed unsigned long is made of a 2 byte value. Divide by two to get its value if it's even. Divide by two and add 32768 times the value of the next 2 bytes if it's odd.
* A compressed signed long is made of a 2 byte value. Divide by two and subtract 16384 to get its value if it's even. Divide by two, add 32768 times the value of the next 2 bytes and subtract 67108864 if it's odd.

The structure of LinkData1 in RecordType 1, 0x20, and 0x23 is difficult to describe, as some values are only stored if a certain condition is met and is therefore of variable size. I try to describe them as a C-structure and note which fields are not present under certain circumstances. Don't declare this structure. Write a parser which reads a value only if its condition is met.

The metric used (GapWidth, LeftIndent, etc.) is dependend upon the FontDescriptor used (See |FONT file). It may be HalfPoints or Twips.

compressed long TopicSize

struct only in records type 0x20 and 0x23

{

compressed unsigned short TopicLength

struct only in records type 0x23

{

unsigned char NumberOfColumns

unsigned char TableType 0,2=variable width, 1,3=normal

struct only for TableType 0 and 2

{

short MinTableWidth

}

ForTableType0or2only

struct

{

short GapWidth LeftMargin if first column

short ColWidth relative in variable width tables

Sum of all GapWidth/ColWidth values

is 32767 in variable width tables

}

Column[NumberOfColumns]

}

RecordType0x23only

}

RecordType0x20or0x23only

struct

{

struct only in RecordType 0x23

{

short column -1 if end of topic, don't continue

short unknown

char always0

}

RecordType0x23only

unsigned char unknownUnsignedChar

char unknownBiasedChar

unsigned short id

struct

{

unsigned short UnknownFollows:1

unsigned short SpacingAboveFollows:1

unsigned short SpacingBelowFollows:1

unsigned short SpacingLinesFollows:1

unsigned short LeftIndentFollows:1

unsigned short RightIndentFollows:1

unsigned short FirstlineIndentFollows:1

unsigned short unused:1

unsigned short BorderinfoFollows:1

unsigned short TabinfoFollows:1

unsigned short RightAlignedParagraph:1

unsigned short CenterAlignedParagraph:1

}

bits

compressed long Unknown only if UnknownFollows set

compressed short SpacingAbove only if SpacingAboveFollows set

compressed short SpacingBelow only if SpacingBelowFollows set

compressed short SpacingLines only if SpacingLinesFollows set

compressed short LeftIndent only if LeftIndentFollows set

compressed short RightIndent only if RightIndentFollows set

compressed short FirstlineIndent only if FirstlineIndentFollows set

struct only if BorderinfoFollows set

{

unsigned char BorderBox:1

unsigned char BorderTop:1

unsigned char BorderLeft:1

unsigned char BorderBottom:1

unsigned char BorderRight:1

unsigned char BorderThick:1

unsigned char BorderDouble:1

unsigned char BorderUnknown:1

short BorderWidth

}

Borderinfo

struct only if TabinfoFollows set

{

compressed short NumberOfTabStops

struct

{

compressed unsigned short TabStop position is lower 14 bits

struct only if TabStop bit 0x4000 set

{

compressed unsigned short TabType 1=right, 2=center

}

onlyIfTabStopBit0x4000set

}

Tab[NumberOfTabStops]

}

Tabinfo

}

Paragraphinfo

Behind this structure LinkData1 contains character formatting information. Always output the next string (NUL terminated) from LinkData2 (use Phrase decompression if required), than read the next formatting command, set up the required font, color or position before displaying the next string.

Sometimes the string is of zero length, as multiple formatting commands are required before output.

0xFF: end of character formatting. Proceed with next Paragraphinfo if

RecordType is 0x23, else you are done.

0x20: long vfldNumber 0 = {vfld} n = {vfld n}

0x21: short dtypeNumber 0 = {dtype} n = {dtype n}

0x80: short FontNumber index into Descriptor array of internal |FONT file

0x81: line break no firstlineindent/spacingabove on next paragraph

0x82: end of paragraph next paragraph has same Paragraphinfo as this one

0x83: TAB jump to next tab stop

0x86: ewc or bmc or bmcwd or bmct or button or mci

0x87: ewl or bml or bmlwd or bmlt or button or mci\_left

0x88: ewr or bmr or bmrwd or bmrt or button or mci\_right

unsigned char Type 5=embedded, 3 or 0x22=picture

compressed long PictureSize size of union

struct only if Type = 0x22

{

compressed word NumberOfHotspots Add to TopicPos if counting

}

OnlyIfTypeIs0x22

union

{

struct

{

short PictureIsEmbedded 0=bmc/bmr/bml or 1=bmcwd/bmlwd/bmrwd

short PictureNumber only if PictureIsEmbedded = 0

char EmbeddedPicture[PictureSize-4]

only if PictureIsEmbedded = 1

See 'Format of Pictures' section

}

Type3or0x22

struct

{

short unknown1

short unknown2

short unknown3

STRINGZ Embedded Format of string depends on statement

DLLName,WindowClass,Param if ewc/ewr/ewl

!Label,Macro if button

\*n,m,[helpfilename+]filename if mci/mci\_left/mci\_right

n=0x8400

n+=2 if NOPLAYBAR specified

n+=8 if NOMENU specified

m=0

m+=1 if PLAY specified

n+=2 if REPEAT specified

[helpfilename+] if not EXTERNAL

}

Type5only

}

PictureData size of union is PictureSize

0x89: end of hotspot switch back from underlined green

0x8B: non-break-space the blank does not appear in LinkData2

0x8C: non-break-hyphen the hyphen itself is stored in LinkData2

0xC8: macro start with underlined green

0xCC: macro without font change

short Length

char MacroString[Length-3]

0xE0: popup jump start with underlined green

0xE1: topic jump start with underlined green

TOPICOFFSET TopicOffset

0xE2: popup jump start with underlined green

0xE3: topic jump start with underlined green

0xE3: topic jump start with underlined green

0xE6: popup jump without font change

0xE7: topic jump without font change

TOPICOFFSET TopicOffset

0xEA: popup jump into external file start with underlined green

0xEB: popup jump into external file without font change

0xEE: topic jump into external file / secondary window start with underlined green

0xEF: topic jump into external file / secondary window without font change

short SizeOfFollowingStruct

struct

{

unsigned char Type 0, 1, 4 or 6

TOPICOFFSET TopicOffset

unsigned char WindowNumber only if Type = 1

STRINGZ NameOfExternalFile only if Type = 4 or 6

STRINGZ WindowName only if Type = 6

}

Continue outputting strings from LinkData2 and parsing formatting commands from LinkData1 until the 'end of character formatting' command is found.

### TOPICOFFSET

A TOPICOFFSET is used since WinHelp 3.1 to locate a cursor-like position, even in the middle of a topic. The position must be unique for hotspots (tabbing).

And it needs to be unique for every scrollable position (going 'Back' to a topic that was scrolled). And it needs to quickly give you the topic block to read from the help file.

Like a TOPICPOS, a TOPICOFFSET is divided into a TopicBlockNumber in it's 17 higher bits (TOPICPOS/32768) and a CharacterCount in it's 15 lower bits (TOPICPOS%32768) counting all characters and the number of hotspots in pictures appearing in all TOPICLINKs in the topic block before this position.

If you got a TopicOffset, seek to the TopicBlock in |TOPIC as told by the TopicBlockNumber, read in and decompress the whole block. Use FirstTopicLink to locate the first TOPICLINK in this decompressed block (CharacterCount is 0 at this place) and follow the list of TOPICLINKs up to the desired position, adding TopicLength of every RecordType 0x20 and 0x23 you come across, until adding TopicLength would exceed the desired CharacterPosition.

Your position is located in this TL\_DISPLAY or TL\_TABLE TOPICLINK. Expand LinkData2 if phrase compressed and follow the formatting procedure described above incrementing CharacterCount on every character (and NUL-terminator) passed. Add the NumberOfHotspots if a picture is included.

If a TOPICLINK crosses a topic block, this has no effect on the TopicBlockNumber for this TOPICLINK (i.e. a TOPICOFFSET pointing into the second part has the TopicBlockNumber of the beginning of the TOPICLINK).

If you didn't come across a TOPICHEADER (TOPICLINK RecordType 2) in this process, the beginning of the topic is located in a previous block. The LastTopicHeader field of the TOPICBLOCKHEADER of the current block tells you where to find it.

### WALKING TOPICS

To follow all topics contained in the help file, set the current TOPICPOS to 12 (that's FirstTopicLink of the first TOPICBLOCKHEADER at offset 0 in|TOPIC) and load it's TopicBlock ((12-12)/DecompressSize = 0) and decompress.

The TOPICLINK is located at TopicBlockOffset ((12-12)%DecompressSize = 0) in the decompression buffer. The first TOPICLINK contains the TOPICHEADER of the first topic.

In Windows 3.0 (HC30) help files you move from one TOPICLINK to the next by adding NextBlock to the current TOPICPOS. If the next TOPICLINK is located in the next topic block, the value of NextBlock handles the jump over the intervening TOPICBLOCKHEADER and possibly unused bytes nicely.

In Windows 3.1 (HC31) and later you move from one TOPICLINK to the next by setting the current position to NextBlock, which also handles the jump from one topic block to the other nicely.

The last TOPICLINK has NextBlock set to 0 or -1L. The last TOPICLINK does not contain any usable data.

# Format of Pictures

Inside help files Bitmaps and Metafiles are stored in lP- or lp-format. This is the format of SHG/MRB files that SHED/MRBC produce and may contain multiple pictures at different resolutions, each with optional additional hotspot data.

Pictures may be embedded in LinkData2 of |TOPIC or appear as |bm<x> files (or bm<x> in case of Windows 3.0 HC30). Each picture starts with this header data. The PictureOffset tells you where to look for the desired picture.

short Magic 0x506C (SHG,lP) or 0x706C (MRB,lp)

short NumberOfPictures >1 if multi-resolution-bitmap

long PictureOffset[NumberOfPictures] relative to &Magic

You shouldn't depend on Magic lP/lp upon reading, as there are some MRBs

flagged like SHG, but please write correct values.

Seek to PictureOffset and you will find this:

char PictureType 5=DDB 6=DIB 8=metafile

char PackingMethod 0=uncompressed 1=RunLen 2=LZ77 3=both

If PictureType is 5 or 6 the picture is a bitmap described by:

compressed unsigned long Xdpi resolution in dpi, not PelsPerMeter

compressed unsigned long Ydpi resolution in dpi, not PelsPerMeter

compressed unsigned short Planes

compressed unsigned short BitCount

compressed unsigned long Width

compressed unsigned long Height

compressed unsigned long ColorsUsed

compressed unsigned long ColorsImportant 1 if bitmap is transparent

compressed unsigned long CompressedSize

compressed unsigned long HotspotSize 0 if none are defined

unsigned long CompressedOffset relative to &PictureType

unsigned long HotspotOffset relative to &PictureType

If PictureType is 6 a color palette follows immediatly

COLORREF palette[ColorsUsed] or 1<<BitCount if ColorsUsed=0

If PackingMethod is 0 copy CompressedSize bytes starting at CompressedOffset

to retrieve the bitmap data. If PackingMethod is 1 seek to CompressedOffset,

and decode CompressedSize bytes using the RunLen algorithm:

n=getc(f); if(n&0x80) copy n&0x7F bytes, else copy next byte n times.

If PackingMethod is 2 use the LZ77 algorithm described above and if Packing-

Method is 3 first use LZ77, then RunLen to decompress.

If PictureType is 8 the picture is a metafile described by:

compressed unsigned short MappingMode

unsigned short Width

unsigned short Height

compressed unsigned long DecompressedSize can be used to allocate buffer

compressed unsigned long CompressedSize

compressed unsigned long HotspotSize 0 if none are defined

unsigned long CompressedOffset relative to &PictureType

unsigned long HotspotOffset relative to &PictureType

Seek to CompressedOffset and decompress CompressedSize bytes as described

above to retrieve metafile data.

If HotspotSize or HotspotOffset is 0, no hotspots are defined. Otherwise

seek to HotspotOffset and retrieve HotspotSize bytes of hotspot definition

as declared below. Each macro hotspot contributes data to MacroData in a

way not fully understood at this moment.

unsigned char Always1

unsigned short NumberOfHotspots

unsigned long SizeOfMacroData

struct

{

unsigned char id0,id1,id2;

unsigned short x,y,w,h;

unsigned long hash;

}

Hotspot[NumberOfHotspots]

char MacroData[SizeOfMacroData] if SizeOfMacroData>0 the first byte

of MacroData is always 2.

struct

{

STRINGZ HotspotName

STRINGZ ContextNameOrMacro

}

StringData[NumberOfHotspots]

Possible values of id0,id1,id2 are:

0xC8 0x00 0x00 macro visible

0xCC 0x04 0x00 macro invisible

0xE2 0x00 0x00 popup jump visible

0xE3 0x00 0x00 topic jump visible

0xE6 0x04 0x00 popup jump invisible

0xE7 0x04 0x00 topic jump invisible

0xEA 0x00 0x00 popup jump into external file visible

0xEB 0x00 0x00 topic jump into external file / secondary window visible

0xEE 0x04 0x00 popup jump into external file invisible

0xEF 0x04 0x00 topic jump into external file / secondary window invisible

The hash field is only used if id0 = 0xE2, 0xE3, 0xE6, 0xE7. It is 1 if

id0 = 0xC8 or 0xCC.

The ContextNameOrMacro contains a macro if id0 = 0xC8 or 0xCC, otherwise

it contains a ContextName (id0 = 0xE2, 0xE3, 0xE6, 0xE7) or the complete

reference ContextName>Window@File (id0 = 0xEA, 0xEB, 0xEE, 0xEF) (@File

may be missing if target is in same file).

# Annotation file format

An annotation file created by WinHelp uses the same basic file format as

a Windows help file. The first 16 bytes contain the same header as a help

file, with same Magic. DirectoryStart points to a FILEHEADER of an internal

directory formatted the same way as a help file internal directory. There

are just internal files of different name and format used to collect the

annotations.

@VERSION

The first internal file described contains (after the usual FILEHEADER) 6

bytes of version info:

0x08 0x62 0x6D 0x66 0x01 0x00 (I've never seen other values)

@LINK

The @LINK internal file contains (after the usual FILEHEADER) the number of

annotations and the TOPICOFFSET of every annotation. The TopicOffset separates

into a TopicBlockNumber in it's upper bits and TopicBlockOffset pointing into

the decompression buffer in it's lower bits as explained above in the

description of the |TOPIC format and points the the first TOPICLINK following

the TOPICHEADER of the topic where the annotation belongs to.

unsigned short NumberOfAnnotations

struct

{

unsigned long TopicOffset

unsigned long Unknown1 // always 0

unsigned long Unknown2 // always 0

}

AnnotationTopicRef[NumberOfAnnotations]

n!0

For each annotation the ANN file also carrys an internal file with a name like 12345!0, where 12345 is the decimal representation of the TopicOffset (as listed in the @LINK array) where the annotation belongs to. These files contain the annotation text as unformatted, uncompressed plain ANSI characters, and are not NUL terminated.

That's all what I've seen in an annotation file.

# \*.CAC, \*.AUX

Multimedia files using extensions \*.CAC or \*.AUX are formatted like help files, but contain only auxiliary files, no |SYSTEM or |TOPIC.

Investigate them yourself. HELPDECO may be used to display or extract files contained in them.

# LZ77

You want to handle LZ77 compressed data in HLPs, MRBs, and SHGs yourself? Here is an algorithm to do it:

// LZ77 compression / decompression algorithm

// this is the compression Microsoft used in Windows \*.HLP and \*.MRB files

// so it works like Microsoft COMPRESS.EXE/EXPAND.EXE/LZEXPAND.DLL

//#define MSEXPAND

#include <stdio.h>

#include <stdlib.h>

#define N 4096

#define F 16

#define THRESHOLD 3

#define dad (node+1)

#define lson (node+1+N)

#define rson (node+1+N+N)

#define root (node+1+N+N+N)

#define NIL -1

char \*buffer;

int \*node;

int pos;

int insert(int i, int run)

{

int c, j, k, l, n, match;

int \*p;

k = l = 1;

match = THRESHOLD - 1;

p = &root[(unsigned char)buffer[i]];

lson[i] = rson[i] = NIL;

while ((j = \*p) != NIL)

{

for (n = min(k, l); n < run && (c = (buffer[j + n] - buffer[i + n])) == 0; n++);

if (n > match)

{

match = n;

pos = j;

}

if (c < 0)

{

p = &lson[j];

k = n;

}

else if (c > 0)

{

p = &rson[j];

l = n;

}

else

{

dad[j] = NIL;

dad[lson[j]] = lson + i - node;

dad[rson[j]] = rson + i - node;

lson[i] = lson[j];

rson[i] = rson[j];

break;

}

}

dad[i] = p - node;

\*p = i;

return match;

}

void delete(int z)

{

int j;

if (dad[z] != NIL)

{

if (rson[z] == NIL)

{

j = lson[z];

}

else if (lson[z] == NIL)

{

j = rson[z];

}

else

{

j = lson[z];

if (rson[j] != NIL)

{

do

{

j = rson[j];

} while (rson[j] != NIL);

node[dad[j]] = lson[j];

dad[lson[j]] = dad[j];

lson[j] = lson[z];

dad[lson[z]] = lson + j - node;

}

rson[j] = rson[z];

dad[rson[z]] = rson + j - node;

}

dad[j] = dad[z];

node[dad[z]] = j;

dad[z] = NIL;

}

}

void compress(FILE \*f, FILE \*out)

{

int ch, i, run, len, match, size, mask;

char buf[17];

buffer = malloc(N + F + (N + 1 + N + N + 256)\*sizeof(int)); // 28.5 k !

if (buffer)

{

#ifdef MSEXPAND

struct { long magic, magic2; int magic3; long filesize; } header;

header.magic = 0x44445A53L; // SZDD

header.magic2 = 0x3327F088L;

header.magic3 = 0x0041;

header.filesize = filelength(fileno(f));

fwrite(&header, sizeof(header), 1, out);

#endif

node = (int \*)(buffer + N + F);

for (i = 0; i < 256; i++) root[i] = NIL;

for (i = NIL; i < N; i++) dad[i] = NIL;

size = mask = 1;

buf[0] = 0;

i = N - F - F;

for (len = 0; len < F && (ch = getc(f)) != -1; len++)

{

buffer[i + F] = ch;

i = (i + 1)&(N - 1);

}

run = len;

do

{

ch = getc(f);

if (i >= N - F)

{

delete(i + F - N);

buffer[i + F] = buffer[i + F - N] = ch;

}

else

{

delete(i + F);

buffer[i + F] = ch;

}

match = insert(i, run);

if (ch == -1)

{

run--;

len--;

}

if (len++ >= run)

{

if (match >= THRESHOLD)

{

#ifdef MSEXPAND

buf[size++] = pos;

buf[size++] = ((pos >> 4) & 0xF0) + (match - 3);

#else

buf[0] |= mask;

\*(int \*)(buf + size) = ((match - 3) << 12) | ((i - pos - 1)&(N - 1));

size += 2;

#endif

len -= match;

}

else

{

#ifdef MSEXPAND

buf[0] |= mask;

#endif

buf[size++] = buffer[i];

len--;

}

if (!((mask += mask) & 0xFF))

{

fwrite(buf, size, 1, out);

size = mask = 1;

buf[0] = 0;

}

}

i = (i + 1)&(N - 1);

} while (len > 0);

if (size > 1) fwrite(buf, size, 1, out);

free(buffer);

}

}

void expand(FILE \*f, FILE \*out)

{

int bits, ch, i, j, len, mask;

char \*buffer;

#ifdef MSEXPAND

struct { long magic, magic2; int magic3; long filesize; } header;

i = fread(&header, 1, sizeof(header), f);

if (i != sizeof(header) || header.magic != 0x44445A53L || header.magic2 != 0x3327F088L || header.magic3 != 0x0041)

{

fwrite(&header, 1, i, out);

while ((ch = getc(f)) != -1) putc(ch, out);

return;

}

#endif

buffer = malloc(N);

if (buffer)

{

i = N - F;

while ((bits = getc(f)) != -1)

{

for (mask = 0x01; mask & 0xFF; mask <<= 1)

{

#ifdef MSEXPAND

if (!(bits&mask))

{

j = getc(f);

if (j == -1) break;

len = getc(f);

j += (len & 0xF0) << 4;

len = (len & 15) + 3;

#else

if (bits&mask)

{

j = getw(f);

len = ((j >> 12) & 15) + 3;

j = (i - j - 1)&(N - 1);

#endif

while (len--)

{

putc(buffer[i] = buffer[j], out);

j = (j + 1)&(N - 1);

i = (i + 1)&(N - 1);

}

}

else

{

ch = getc(f);

#ifndef MSEXPAND

if (ch == -1) break;

#endif

putc(buffer[i] = ch, out);

i = (i + 1)&(N - 1);

}

}

}

free(buffer);

}

}

}

# Conclusion

That's all I can tell you about the format of Windows 3.x/95 help files.

If you found out more, please let me know.

M. Winterhoff

[100326.2776@compuserve.com](mailto:100326.2776@compuserve.com)

# Annexes

My own researches over the Internet:

* <http://www.helpmvp.com/MVP-Info/history>
* <https://en.wikipedia.org/wiki/WinHelp>  
  <http://www.herdsoft.com/catalog/ehlp2rtf.html>
* <http://www.herdsoft.com/linux/themen/online_publishing_winhelp.html>
* <https://support.microsoft.com/en-us/kb/171958> --> This is the place where one can download the old Microsoft Help Workshop tools.
* <http://www.herdsoft.com/linux/produkte/winhelpcgi.html> --> converts .HLP and .MVB to .CHM (GPL: <http://www.herdsoft.com/ftp/winhelpcgi_1.0-1.tar.gz>)

Concerning “MVP”, what I understand from reading the HELPDECO.TXT and HELPFILE.TXT from the original author (M. Winterhoff), is that \*.MVP files were Multimedia Project files compiles by the multimedia help compiler into files in a format derived from standard HLP. This is what is called ‘MVP’ files in this text.

From <http://www.herdsoft.com/linux/produkte/winhelpcgi.html>, it seems the real extension for MVP compiled files is ‘MVB’, stating for “Multimedia Viewer 2.0 Tiles”

According to HELPDECO.C @line 6042, it’s enough for a file to have its extension start with ‘M’ for the Helpdeco tool to consider it was generated by the MVP compiler. And then depending, on this ‘mvp’ flag, the tool makes several decisions especially regarding the colliding System Record Types parsing.