



# 72


"Beauty can trick you."

## File format Collections

File formats are usually represented in record/structure format. Almost all documents use Assembly language's record format or C's structure format or sometimes Pascal's record format. In file formats, mostly we would come across the jargons: BYTE, WORD & DWORD. BYTE can be viewed as signed or unsigned char; WORD can be viewed as signed or unsigned int; DWORD can be viewed as signed or unsigned long.

### 72.1 File Formats Encyclopedia

The file formats encyclopedia found on CD  has lots of file formats. For a quick and neat description, I strongly suggest you to have a look on CD .

In this chapter, I give you few file formats that I think will be useful. Most of them are from File Formats Encyclopedia and official documentations. For a full description, have a look on CD .

### 72.2 ARJ

#### 72.2.1 Glimpse

Following documentation gives you overall picture about ARJ file format.

The ARJ program by Robert K. Jung is a "newcomer" which compares well to PKZip and LhArc in both compression and speed. An ARJ archive contains two types of header blocks, one archive main header at the head of the archive and local file headers before each archived file.

OFFSET	Count	TYPE	Description
0000h	1	word	ID=0EA60h
0002h	1	word	Basic header size (0 if end of archive)
0004h	1	byte	Size of header including extra data
0005h	1	byte	Archiver version number
0006h	1	byte	Minimum version needed to extract
0007h	1	byte	Host OS (see table 0002)
0008h	1	byte	Internal flags, bitmapped : 0 - no password / password 1 - reserved 2 - file continues on next disk 3 - file start position field is available 4 - path translation ( "\ " to "/" )

OFFSET	Count	TYPE	Description
0009h	1	byte	Compression method : 0 - stored 1 - compressed most 2 - compressed 3 - compressed faster 4 - compressed fastest
000Ah	1	byte	File type : 0 - binary 1 - 7-bit text 2 - comment header 3 - directory 4 - volume label
000Bh	1	byte	reserved
000Ch	1	dword	Date/Time of original file in MS-DOS format
0010h	1	dword	Compressed size of file
0014h	1	dword	Original size of file
0018h	1	dword	Original file's CRC-32
001Ah	1	word	Filespec position in filename
001Ch	1	word	File attributes
001Eh	1	word	Host data (currently not used)
?	1	dword	Extended file starting position when used (see above)
	?	char	ASCIIZ file name
	?	char	Comment
????h	1	dword	Basic header CRC-32
????h	1	word	Size of first extended header (0 if none) = "SIZ"
????h+"SIZ"+2	1	dword	Extended header CRC-32
????h+"SIZ"+6	?	byte	Compressed file

(Table 0002)

## ARJ HOST-OS types

- 0 - MS-DOS
- 1 - PRIMOS
- 2 - UNIX
- 3 - AMIGA
- 4 - MAC-OS (System xx)
- 5 - OS/2
- 6 - APPLE GS
- 7 - ATARI ST
- 8 - NeXT
- 9 - VAX VMS

## 774 A to Z of C

### 72.2.2 Official documentation

ARJ archives contains two types of header blocks:

Archive main header - This is located at the head of the archive

Local file header - This is located before each archived file

Structure of main header (low order byte first):	
Bytes	Description
2	header id (main and local file) = 0x60 0xEA
2	basic header size (from 'first_hdr_size' thru 'comment' below) = first_hdr_size + strlen(filename) + 1 + strlen(comment) + 1 = 0 if end of archive maximum header size is 2600
1	first_hdr_size (size up to and including 'extra data')
1	archiver version number
1	minimum archiver version to extract
1	host OS (0 = MSDOS, 1 = PRIMOS, 2 = UNIX, 3 = AMIGA, 4 = MAC-OS) (5 = OS/2, 6 = APPLE GS, 7 = ATARI ST, 8 = NEXT) (9 = VAX VMS)
1	arj flags (0x01 = NOT USED)(0x02 = OLD_SECURED_FLAG) (0x04 = VOLUME_FLAG) indicates presence of succeeding Volume (0x08 = NOT USED)(0x10 = PATHSYM_FLAG) indicates archive name translated ("\\" changed to "/") (0x20 = BACKUP_FLAG) indicates backup type archive (0x40 = SECURED_FLAG)
1	security version (2 = current)
1	file type (must equal 2)
1	reserved
4	date time when original archive was created
4	date time when archive was last modified
4	archive size (currently used only for secured archives)
4	security envelope file position
2	filespec position in filename
2	length in bytes of security envelope data
2	(currently not used)
?	(currently none)
?	filename of archive when created (null-terminated string)
?	archive comment (null-terminated string)
4	basic header CRC
2	1st extended header size (0 if none)
?	1st extended header (currently not used)
4	1st extended header's CRC (not present when 0 extended header size)

<b>Structure of local file header (low order byte first):</b>	
<b>Bytes</b>	<b>Description</b>
2	header id (main and local file) = 0x60 0xEA
2	basic header size (from 'first_hdr_size' thru 'comment' below) = first_hdr_size + strlen(filename) + 1 + strlen(comment) + 1 = 0 if end of archive maximum header size is 2600
1	first_hdr_size (size up to and including 'extra data')
1	archiver version number
1	minimum archiver version to extract
1	host OS (0 = MSDOS, 1 = PRIMOS, 2 = UNIX, 3 = AMIGA, 4 = MAC-OS) (5 = OS/2, 6 = APPLE GS, 7 = ATARI ST, 8 = NEXT) (9 = VAX VMS)
1	arj flags (0x01 = GARBLED_FLAG) indicates passworded file (0x02 = NOT USED) (0x04 = VOLUME_FLAG) indicates continued file to next volume (file is split) (0x08 = EXTFILE_FLAG) indicates file starting position field (for split files) (0x10 = PATHSYM_FLAG) indicates filename translated ("\" changed to "/") (0x20 = BACKUP_FLAG) indicates file marked as backup
1	method (0 = stored, 1 = compressed most ... 4 compressed fastest)
1	file type (0 = binary, 1 = 7-bit text)(3 = directory, 4 = volume label)
1	reserved
4	date time modified
4	compressed size
4	original size (this will be different for text mode compression)
4	original file's CRC
2	filespec position in filename
2	file access mode
2	host data (currently not used)
?	extra data
4	bytes for extended file starting position when used (these bytes are present when EXTFILE_FLAG is set). 0 bytes otherwise.
?	filename (null-terminated string)
?	comment (null-terminated string)
4	basic header CRC
2	1st extended header size (0 if none)
?	1st extended header (currently not used)
4	1st extended header's CRC (not present when 0 extended header size)
	...
?	compressed file

## 776 A to Z of C

Time stamp format:								
31 30 29 28 27 26 25	24 23 22 21	20 19 18 17 16						
<---- year-1980 ---->	<- month ->	<--- day ---->						
15 14 13 12 11	10 9 8 7 6 5	4 3 2 1 0						
<--- hour --->	<---- minute --->	<- second/2 ->						

### 72.3 BMP

Windows bitmap files are stored in a device-independent bitmap (DIB) format that allows Windows to display the bitmap on any type of display device. The term "device independent" means that the bitmap specifies pixel color in a form independent of the method used by a display to represent color. The default filename extension of a Windows DIB file is .BMP.

#### Bitmap-File Structures

Each bitmap file contains a bitmap-file header, a bitmap-information header, a color table, and an array of bytes that defines the bitmap bits. The file has the following form:

BITMAPFILEHEADER	bmfh;
BITMAPINFOHEADER	bmih;
RGBQUAD	aColors[];
BYTE	aBitmapBits[];

The bitmap-file header contains information about the type, size, and layout of a device-independent bitmap file. The header is defined as a BITMAPFILEHEADER structure.

The bitmap-information header, defined as a BITMAPINFOHEADER structure, specifies the dimensions, compression type, and color format for the bitmap.

The color table, defined as an array of RGBQUAD structures, contains as many elements as there are colors in the bitmap. The color table is not present for bitmaps with 24 color bits because each pixel is represented by 24-bit red-green-blue (RGB) values in the actual bitmap data area. The colors in the table should appear in order of importance. This helps a display driver render a bitmap on a device that cannot display as many colors as there are in the bitmap. If the DIB is in Windows version 3.0 or later format, the driver can use the biClrImportant member of the BITMAPINFOHEADER structure to determine which colors are important.

The BITMAPINFO structure can be used to represent a combined bitmap-information header and color table. The bitmap bits, immediately following the color table, consist of an array of BYTE values representing consecutive rows, or "scan lines," of the bitmap. Each scan line consists of consecutive bytes representing the pixels in the scan line, in left-to-right order. The number of bytes representing a scan line depends on the color format and the width, in pixels,

of the bitmap. If necessary, a scan line must be zero-padded to end on a 32-bit boundary. However, segment boundaries can appear anywhere in the bitmap. The scan lines in the bitmap are stored from bottom up. This means that the first byte in the array represents the pixels in the lower-left corner of the bitmap and the last byte represents the pixels in the upper-right corner.

The `biBitCount` member of the `BITMAPINFOHEADER` structure determines the number of bits that define each pixel and the maximum number of colors in the bitmap. These members can have any of the following values:

Value	Meaning
1	Bitmap is monochrome and the color table contains two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the color table. If the bit is set, the pixel has the color of the second entry in the table.
4	Bitmap has a maximum of 16 colors. Each pixel in the bitmap is represented by a 4-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.
8	Bitmap has a maximum of 256 colors. Each pixel in the bitmap is represented by a 1-byte index into the color table. For example, if the first byte in the bitmap is 0x1F, the first pixel has the color of the thirty-second table entry.
24	Bitmap has a maximum of $2^{24}$ colors. The <code>bmiColors</code> (or <code>bmciColors</code> ) member is NULL, and each 3-byte sequence in the bitmap array represents the relative intensities of red, green, and blue, respectively, for a pixel.

The `biClrUsed` member of the `BITMAPINFOHEADER` structure specifies the number of color indexes in the color table actually used by the bitmap. If the `biClrUsed` member is set to zero, the bitmap uses the maximum number of colors corresponding to the value of the `biBitCount` member. An alternative form of bitmap file uses the `BITMAPCOREINFO`, `BITMAPCOREHEADER`, and `RGBTRIPLE` structures.

### Bitmap Compression

Windows versions 3.0 and later support run-length encoded (RLE) formats for compressing bitmaps that use 4 bits per pixel and 8 bits per pixel.

Compression reduces the disk and memory storage required for a bitmap.

### Compression of 8-Bits-per-Pixel Bitmaps

When the `biCompression` member of the `BITMAPINFOHEADER` structure is set to `BI_RLE8`, the DIB is compressed using a run-length encoded format for a 256-color bitmap. This format uses two modes: encoded mode and absolute mode. Both modes can occur anywhere throughout a single bitmap.

## 778 A to Z of C

### Encoded Mode

A unit of information in encoded mode consists of two bytes. The first byte specifies the number of consecutive pixels to be drawn using the color index contained in the second byte. The first byte of the pair can be set to zero to indicate an escape that denotes the end of a line, the end of the bitmap, or a delta. The interpretation of the escape depends on the value of the second byte of the pair, which must be in the range 0x00 through 0x02.

Following are the meanings of the escape values that can be used in the second byte:

Second byte	Meaning
0	End of line.
1	End of bitmap.
2	Delta. The two bytes following the escape contain unsigned values indicating the horizontal and vertical offsets of the next pixel from the current position.

### Absolute Mode

Absolute mode is signaled by the first byte in the pair being set to zero and the second byte to a value between 0x03 and 0xFF. The second byte represents the number of bytes that follow, each of which contains the color index of a single pixel. Each run must be aligned on a word boundary.

Following is an example of an 8-bit RLE bitmap (the two-digit hexadecimal values in the second column represent a color index for a single pixel):

Compressed data	Expanded data
03 04	04 04 04
05 06	06 06 06 06 06
00 03 45 56 67 00	45 56 67
02 78	78 78
00 02 05 01	Move 5 right and 1 down
02 78	78 78
00 00	End of line
09 1E	1E 1E 1E 1E 1E 1E 1E 1E 1E
00 01	End of RLE bitmap

### Compression of 4-Bits-per-Pixel Bitmaps

When the `biCompression` member of the `BITMAPINFOHEADER` structure is set to `BI_RLE4`, the DIB is compressed using a run-length encoded format for a 16-color bitmap. This format uses two modes: encoded mode and absolute mode.

#### Encoded Mode

A unit of information in encoded mode consists of two bytes. The first byte of the pair contains the number of pixels to be drawn using the color indexes in the second byte.

The second byte contains two color indexes, one in its high-order nibble (that is, its low-order 4 bits) and one in its low-order nibble.

The first pixel is drawn using the color specified by the high-order nibble, the second is drawn using the color in the low-order nibble, the third is drawn with the color in the high-order nibble, and so on, until all the pixels specified by the first byte have been drawn.

The first byte of the pair can be set to zero to indicate an escape that denotes the end of a line, the end of the bitmap, or a delta. The interpretation of the escape depends on the value of the second byte of the pair. In encoded mode, the second byte has a value in the range 0x00 through 0x02. The meaning of these values is the same as for a DIB with 8 bits per pixel.

### Absolute Mode

In absolute mode, the first byte contains zero, the second byte contains the number of color indexes that follow, and subsequent bytes contain color indexes in their high- and low-order nibbles, one color index for each pixel. Each run must be aligned on a word boundary.

Following is an example of a 4-bit RLE bitmap (the one-digit hexadecimal values in the second column represent a color index for a single pixel):

Compressed data	Expanded data
03 04	0 4 0
05 06	0 6 0 6 0
00 06 45 56 67 00	4 5 5 6 6 7
04 78	7 8 7 8
00 02 05 01	Move 5 right and 1 down
04 78	7 8 7 8
00 00	End of line
09 1E	1 E 1 E 1 E 1 E 1
00 01	End of RLE bitmap

### Bitmap Example

The following example is a text dump of a 16-color bitmap (4 bits per pixel):

```
Win3DIBFile
  BitmapFileHeader
    Type      19778
    Size      3118
    Reserved1  0
    Reserved2  0
    OffsetBits 118
  BitmapInfoHeader
    Size      40
    Width     80
    Height    75
```





db	MajorVersion+'0'
db	(MinorVersion / 10)+'0',(MinorVersion mod 10)+'0'
db	' - 19 October 1987',ODH,OAH
db	'Copyright (c) 1987 Borland International', Odh,Oah
db	0,1ah ; null & ctrl-Z = end
dw	HeaderSize ; size of header
db	fname ; font name
dw	DataSize ; font file size
db	MajorVersion,MinorVersion ; version #'s
db	1,0 ; minimal version #'s
db	(HeaderSize - \$) DUP (0) ; pad out to header size

At offset 80h starts data for the file:

80h	'+' flags stroke file type
81h-82h	number chars in font file (n)
83h	undefined
84h	ASCII value of first char in file
85h-86h	offset to stroke definitions (8+3n)
87h	scan flag (normally 0)
88h	distance from origin to top of capital
89h	distance from origin to baseline
90h	distance from origin to bottom descender
91h-95h	undefined
96h	offsets to individual character definitions
96h+2n	width table (one word per character)
96h+3n	start of character definitions

The individual character definitions consist of a variable number of words describing the operations required to render a character. Each word consists of an (x,y) coordinate pair and a two-bit opcode, encoded as shown here:

<b>Byte 1</b>	7	6 5 4 3 2 1 0 bit #
	op1	<seven bit signed X coord>

<b>Byte 2</b>	7	6 5 4 3 2 1 0 bit #
	op2	<seven bit signed Y coord>

## 72.5 COM

The COM files are raw binary executables and are a leftover from the old CP/M machines with 64K RAM. A COM program can only have a size of less than one segment (64K), including code and static data since no fixups for segment relocation or anything else is included. One method to check for a COM file is to check if the first byte in the file could be

## 782 A to Z of C

a valid jump or call opcode, but this is a very weak test since a COM file is not required to start with a jump or a call. In principle, a COM file is just loaded at offset 100h in the segment and then executed.

OFFSET	Count	TYPE	Description
0000h	1	byte	ID=0E9h ID=0Ebh

Those are not safe ways to determine whether a file is a COM file or not, but most COM files start with a jump.

## 72.6 CUR

A cursor-resource file contains image data for cursors used by Windows applications. The file consists of a cursor directory identifying the number and types of cursor images in the file, plus one or more cursor images. The default filename extension for a cursor-resource file is .CUR.

### Cursor Directory

Each cursor-resource file starts with a cursor directory. The cursor directory, defined as a CURSORDIR structure, specifies the number of cursors in the file and the dimensions and color format of each cursor image. The CURSORDIR structure has the following form:

```
typedef struct _CURSORDIR {  
    WORD    cdReserved;  
    WORD    cdType;  
    WORD    cdCount;  
    CURSORDIRENTRY cdEntries[];  
} CURSORDIR;
```

Following are the members in the CURSORDIR structure:

cdReserved	Reserved; must be zero.
cdType	Specifies the resource type. This member must be set to 2.
cdCount	Specifies the number of cursors in the file.
cdEntries	Specifies an array of CURSORDIRENTRY structures containing information about individual cursors. The cdCount member specifies the number of structures in the array.

A CURSORDIRENTRY structure specifies the dimensions and color format of a cursor image. The structure has the following form:

```

typedef struct _CURSORDIRENTRY {
    BYTE  bWidth;
    BYTE  bHeight;
    BYTE  bColorCount;
    BYTE  bReserved;
    WORD  wXHotspot;
    WORD  wYHotspot;
    DWORD lBytesInRes;
    DWORD dwImageOffset;
} CURSORDIRENTRY;

```

Following are the members in the CURSORDIRENTRY structure:

bWidth	Specifies the width of the cursor, in pixels.
bHeight	Specifies the height of the cursor, in pixels.
bColorCount	Reserved; must be zero.
bReserved	Reserved; must be zero.
wXHotspot	Specifies the x-coordinate, in pixels, of the hot spot.
wYHotspot	Specifies the y-coordinate, in pixels, of the hot spot.
lBytesInRes	Specifies the size of the resource, in bytes.
dwImageOffset	Specifies the offset, in bytes, from the start of the file to the cursor image.

### Cursor Image

Each cursor-resource file contains one cursor image for each image identified in the cursor directory. A cursor image consists of a cursor-image header, a color table, an XOR mask, and an AND mask. The cursor image has the following form:

```

BITMAPINFOHEADER  crHeader;
RGBQUAD           crColors[];
BYTE              crXOR[];
BYTE              crAND[];

```

The cursor hot spot is a single pixel in the cursor bitmap that Windows uses to track the cursor. The crXHotspot and crYHotspot members specify the x- and y-coordinates of the cursor hot spot. These coordinates are 16-bit integers.

The cursor-image header, defined as a BITMAPINFOHEADER structure, specifies the dimensions and color format of the cursor bitmap. Only the biSize through biBitCount members and the biSizeImage member are used. The biHeight member specifies the combined height of the XOR and AND masks for the cursor. This value is twice the height of the XOR mask. The biPlanes and biBitCount members must be 1. All other members (such as biCompression and biClrImportant) must be set to zero.

## 784 A to Z of C

The color table, defined as an array of RGBQUAD structures, specifies the colors used in the XOR mask. For a cursor image, the table contains exactly two structures, since the biBitCount member in the cursor-image header is always 1.

The XOR mask, immediately following the color table, is an array of BYTE values representing consecutive rows of a bitmap. The bitmap defines the basic shape and color of the cursor image. As with the bitmap bits in a bitmap file, the bitmap data in a cursor-resource file is organized in scan lines, with each byte representing one or more pixels, as defined by the color format. For more information about these bitmap bits, see Section "Bitmap-File Formats."

The AND mask, immediately following the XOR mask, is an array of BYTE values representing a monochrome bitmap with the same width and height as the XOR mask. The array is organized in scan lines, with each byte representing 8 pixels.

When Windows draws a cursor, it uses the AND and XOR masks to combine the cursor image with the pixels already on the display surface. Windows first applies the AND mask by using a bitwise AND operation; this preserves or removes existing pixel color. Window then applies the XOR mask by using a bitwise XOR operation. This sets the final color for each pixel.

The following illustration shows the XOR and the AND masks that create a cursor (measuring 8 pixels by 8 pixels) in the form of an arrow:

Following are the bit-mask values necessary to produce black, white, inverted, and transparent results:

Pixel result	AND mask	XOR mask
Black	0	0
White	0	1
Transparent	1	0
Inverted	1	1

### Windows Cursor Selection

If a cursor-resource file contains more than one cursor image, Windows determines the best match for a particular display by examining the width and height of the cursor images.

## 72.7 DBF (General Format of .dbf files in Xbase languages)

Applies for / supported by:	
FS = FlagShip	D3 = dBaseIII+
Fb = FoxBase	D4 = dBaseIV
Fp = FoxPro	D5 = dBaseV
CL = Clipper	

1. DBF Structure		
Byte	Description	
0..n	.dbf header (see 2 for size, byte 8)	
n+1	1st record of fixed length (see 2&3) 2nd record (see 2 for size, byte 10) ... last record	if dbf is not empty
last	optional: 0x1a (eof byte)	

2. DBF Header (variable size, depending on field count)				
Byte	Size	Contents	Description	Applies for (supported by)
00	1	0x03	plain .dbf	FS, D3, D4, D5, Fb, Fp, CL
		0x04	plain .dbf	D4, D5 (FS)
		0x05	plain .dbf	D5, Fp (FS)
		0x43	with .dbv memo var size	FS
		0xB3	with .dbv and .dbt memo	FS
		0x83	with .dbt memo	FS, D3, D4, D5, Fb, Fp, CL
		0x8B	with .dbt memo in D4 format	D4, D5
		0x8E	with SQL table	D4, D5
		0xF5	with .fmp memo	Fp
01	3	YYMMDD	Last update digits	all
04	4	ulong	Number of records in file	all
08	2	ushort	Header size in bytes	all
10	2	ushort	Record size in bytes	all
12	2	0,0	Reserved	all
14	1	0x01	Begin transaction	D4, D5
		0x00	End Transaction	D4, D5
		0x00	ignored	FS, D3, Fb, Fp, CL
15	1	0x01	Encrypted	D4, D5
		0x00	normal visible	all
16	12	0 (1)	multi-user environment use	D4,D5
28	1	0x01	production index exists	Fp, D4, D5
		0x00	index upon demand	all
29	1	n	language driver ID	D4, D5
		0x01	codepage 437 DOS USA	Fp
		0x02	codepage 850 DOS Multi ling	Fp
		0x03	codepage 1251 Windows ANSI	Fp
		0xC8	codepage 1250 Windows EE	Fp
		0x00	ignored	FS, D3, Fb, Fp, CL
30	2	0,0	reserved	all
32	n*32		Field Descriptor, see (2a)	all
+1	1	0x0D	Header Record Terminator	all

## 786 A to Z of C

<b>2a. Field descriptor array in dbf header (fix 32 bytes for each field)</b>				
Byte	Size	Contents	Description	Applies for (supported by)
0	11	ASCII	field name, 0x00 termin.	all
11	1	ASCII	field type (see 2b)	all
12	4	n,n,n,n	fld address in memory	D3
		n,n,0,0	offset from record begin	Fp
		0,0,0,0	ignored	FS, D4, D5, Fb, CL
16	1	byte	Field length, bin (see 2b)	all \ FS,CL: for C field type,
17	1	byte	decimal count, bin	all / both used for fld lng
18	2	0,0	reserved	all
20	1	byte	Work area ID	D4, D5
		0x00	unused	FS, D3, Fb, Fp, CL
21	2	n,n	multi-user dBase	D3, D4, D5
		0,0	ignored	FS, Fb, Fp, CL
23	1	0x01	Set Fields	D3, D4, D5
		0x00	ignored	FS, Fb, Fp, CL
24	7	0..0	reserved	all
31	1	0x01	Field is in .mdx index	D4, D5
		0x00	ignored	FS, D3, Fb, Fp, CL

<b>2b. Field type and size in dbf header, field descriptor (1 byte)</b>			
Size	Type	Description/Storage	Applies for (supported by)
C 1..n	Char	ASCII (OEM code page chars)	all
		rest= space, not \0 term.	
		n = 1..64kb (using deci count) FS	
		n = 1..32kb (using deci count) Fp, CL	
		n = 1..254	all
D 8	Date	8 Ascii digits (0..9) in the	all
		YYYYMMDD format	
F 1..n	Numeric	Ascii digits (-.0123456789)	FS, D4, D5, Fp
		variable pos. of float.point	
		n = 1..20	
N 1..n	Numeric	Ascii digits (-.0123456789)	all
		fix posit/no float.point	
		n = 1..20	FS, Fp, CL
		n = 1..18	D3, D4, D5, Fb
L 1	Logical	Ascii chars (YyNnTtFf space)	FS, D3, Fb, Fp, CL
		Ascii chars (YyNnTtFf ?)	D4, D5 (FS)
M 10	Memo	10 digits repres. the start	all
		block posit. in .dbt file, or	
		10spaces if no entry in memo	

Size	Type	Description/Storage	Applies for (supported by)
V 10	Variable	Variable, bin/asc data in .dbv	FS
		4bytes bin= start pos in memo	
		4bytes bin= block size	
		1byte = subtype	
		1byte = reserved (0x1a)	
		10spaces if no entry in .dbv	
P 10	Picture	binary data in .ftp structure like M	Fp
B 10	Binary	binary data in .dbt	D5
		structure like M	
G 10	General	OLE objects	D5, Fp
		structure like M	
2 2	short int	binary int max +/- 32767	FS
4 4	long int	binary int max +/- 2147483647	FS
8 8	double	binary signed double IEEE	FS

3. Each Dbf record (fix length)			
Byte	Size	Description	Applies for (supported by)
0	1	deleted flag "*" or not deleted " "	all
1..n	1..	x-times contents of fields, fixed length, unterminated. For n, see (2) byte 10..11	all

Courtesy:multisoft Datentechnik GmbH



## 72.8 EXE

### 72.8.1 Old EXE format (EXE MZ)

.EXE - DOS EXE File Structure		
Offset	Size	Description
00	word	"MZ" or "ZM"- Link file .EXE signature (Mark Zbikowski?)
02	word	length of image mod 512
04	word	size of file in 512 byte pages
06	word	number of relocation items following header
08	word	size of header in 16 byte paragraphs, used to locate the beginning of the load module
0A	word	min # of paragraphs needed to run program
0C	word	max # of paragraphs the program would like
0E	word	offset in load module of stack segment (in paras)
10	word	initial SP value to be loaded
12	word	negative checksum of pgm used while by EXEC loads pgm
14	word	program entry point, (initial IP value)
16	word	offset in load module of the code segment (in paras)
18	word	offset in .EXE file of first relocation item overlay number (0 for root program)
1A	word	

- relocation table and the program load module follow the header
- relocation entries are 32 bit values representing the offset into the load module needing patched
- once the relocatable item is found, the CS register is added to the value found at the calculated offset

Registers at load time of the EXE file are as follows:

AX:	contains number of characters in command tail, or 0
BX:CX	32 bit value indicating the load module memory size
DX	zero
SS:SP	set to stack segment if defined else, SS = CS and SP=FFFFh or top of memory.
DS	set to segment address of EXE header
ES	set to segment address of EXE header
CS:IP	far address of program entry point, (label on "END" statement of program)

### 72.8.2 New EXE format (EXE NE)

The Windows (new-style) executable-file header contains information that the loader requires for segmented executable files. This information includes the linker version number, data specified by the linker, data specified by the resource compiler, tables of segment data, tables of resource data, and so on. The following illustration shows the Windows executable-file header: The following sections describe the entries in the Windows executable-file header.

### Information Block

The information block in the Windows header contains the linker version number, the lengths of various tables that further describe the executable file, the offsets from the beginning of the header to the beginning of these tables, the heap and stack sizes, and so on. The following list summarizes the contents of the header information block (the locations are relative to the beginning of the block):

Location	Description
00h	Specifies the signature word. The low byte contains "N" (4Eh) and the high byte contains "E" (45h).
02h	Specifies the linker version number.
03h	Specifies the linker revision number.
04h	Specifies the offset to the entry table (relative to the beginning of the header).
06h	Specifies the length of the entry table, in bytes.
08h	Reserved.
0Ch	Specifies flags that describe the contents of the executable file. This value can be one or more of the following bits:

Bit	Meaning
0	The linker sets this bit if the executable-file format is SINGLEDATA. An executable file with this format contains one data segment. This bit is set if the file is a dynamic-link library (DLL).
1	The linker sets this bit if the executable-file format is MULTIPLEDATA. An executable file with this format contains multiple data segments. This bit is set if the file is a Windows application. If neither bit 0 nor bit 1 is set, the executable-file format is NOAUTODATA. An executable file with this format does not contain an automatic data segment.
2	Reserved.
3	Reserved.
8	Reserved.
9	Reserved.
11	If this bit is set, the first segment in the executable file contains code that loads the application.
13	If this bit is set, the linker detects errors at link time but still creates an executable file.
14	Reserved.
15	If this bit is set, the executable file is a library module.

If bit 15 is set, the CS:IP registers point to an initialization procedure called with the value in the AX register equal to the module handle. The initialization procedure must execute a far return to the caller. If the procedure is successful, the value in AX is nonzero. Otherwise, the value in AX is zero. The value in the DS register is set to the library's data segment if SINGLEDATA is set. Otherwise, DS is set to the data segment of the application that loads the library.

## 790 A to Z of C

0Eh	Specifies the automatic data segment number. (0Eh is zero if the SINGLEDATA and MULTIPLEDATA bits are cleared.)
10h	Specifies the initial size, in bytes, of the local heap. This value is zero if there is no local allocation.
12h	Specifies the initial size, in bytes, of the stack. This value is zero if the SS register value does not equal the DS register value.
14h	Specifies the segment:offset value of CS:IP.
18h	Specifies the segment:offset value of SS:SP.

The value specified in SS is an index to the module's segment table. The first entry in the segment table corresponds to segment number 1. If SS addresses the automatic data segment and SP is zero, SP is set to the address obtained by adding the size of the automatic data segment to the size of the stack.

1Ch	Specifies the number of entries in the segment table.
1Eh	Specifies the number of entries in the module-reference table.
20h	Specifies the number of bytes in the nonresident-name table.
22h	Specifies a relative offset from the beginning of the Windows header to the beginning of the segment table.
24h	Specifies a relative offset from the beginning of the Windows header to the beginning of the resource table.
26h	Specifies a relative offset from the beginning of the Windows header to the beginning of the resident-name table.
28h	Specifies a relative offset from the beginning of the Windows header to the beginning of the module-reference table.
2Ah	Specifies a relative offset from the beginning of the Windows header to the beginning of the imported-name table.
2Ch	Specifies a relative offset from the beginning of the file to the beginning of the nonresident-name table.
30h	Specifies the number of movable entry points.
32h	Specifies a shift count that is used to align the logical sector. This count is log <sub>2</sub> of the segment sector size. It is typically 4, although the default count is 9. (This value corresponds to the /alignment [/a] linker switch. When the linker command line contains /a:16, the shift count is 4. When the linker command line contains /a:512, the shift count is 9.)
34h	Specifies the number of resource segments.
36h	Specifies the target operating system, depending on which bits are set:

Bit	Meaning
0	Operating system format is unknown.
1	Reserved.
2	Operating system is Microsoft Windows.
3	Reserved.
4	Reserved.

37h Specifies additional information about the executable file. It can be one or more of the following values:

Bit	Meaning
1	If this bit is set, the executable file contains a Windows 2.x application that runs in version 3.x protected mode.
2	If this bit is set, the executable file contains a Windows 2.x application that supports proportional fonts.
3	If this bit is set, the executable file contains a fast-load area.

38h	Specifies the offset, in sectors, to the beginning of the fast-load area. (Only Windows uses this value.)
3Ah	Specifies the length, in sectors, of the fast-load area. (Only Windows uses this value.)
3Ch	Reserved.
3Eh	Specifies the expected version number for Windows. (Only Windows uses this value.)

### Segment Table

The segment table contains information that describes each segment in an executable file. This information includes the segment length, segment type, and segment-relocation data. The following list summarizes the values found in the segment table (the locations are relative to the beginning of each entry):

Location	Description
00h	Specifies the offset, in sectors, to the segment data (relative to the beginning of the file). A value of zero means no data exists.
02h	Specifies the length, in bytes, of the segment, in the file. A value of zero indicates that the segment length is 64K, unless the selector offset is also zero.
04h	Specifies flags that describe the contents of the executable file. This value can be one or more of the following:

Bit	Meaning
0	If this bit is set, the segment is a data segment. Otherwise, the segment is a code segment.
1	If this bit is set, the loader has allocated memory for the segment.
2	If this bit is set, the segment is loaded.
3	Reserved.
4	If this bit is set, the segment type is MOVABLE. Otherwise, the segment type is FIXED.
5	If this bit is set, the segment type is PURE or SHAREABLE. Otherwise, the segment type is IMPURE or NONSHAREABLE.
6	If this bit is set, the segment type is PRELOAD. Otherwise, the segment type is LOADONCALL.
7	If this bit is set and the segment is a code segment, the segment type is EXECUTEONLY. If this bit is set and the segment is a data segment, the segment type is READONLY.

## 792 A to Z of C

Bit	Meaning
8	If this bit is set, the segment contains relocation data.
9	Reserved.
10	Reserved.
11	Reserved.
12	If this bit is set, the segment is discardable.
13	Reserved.
14	Reserved.
15	Reserved.

06h Specifies the minimum allocation size of the segment, in bytes. A value of zero indicates that the minimum allocation size is 64K.

### Resource Table

The resource table describes and identifies the location of each resource in the executable file. The table has the following form:

WORD	rscAlignShift;
TYPEINFO	rscTypes[];
WORD	rscEndTypes;
BYTE	rscResourceNames[];
BYTE	rscEndNames;

Following are the members in the resource table:

rscAlignShift	Specifies the alignment shift count for resource data. When the shift count is used as an exponent of 2, the resulting value specifies the factor, in bytes, for computing the location of a resource in the executable file.
rscTypes	Specifies an array of TYPEINFO structures containing information about resource types. There must be one TYPEINFO structure for each type of resource in the executable file.
rscEndTypes	Specifies the end of the resource type definitions. This member must be zero.
RscResourceNames	Specifies the names (if any) associated with the resources in this table. Each name is stored as consecutive bytes; the first byte specifies the number of characters in the name.
rscEndNames	Specifies the end of the resource names and the end of the resource table. This member must be zero.

### Type Information

The TYPEINFO structure has the following form:

```
typedef struct _TYPEINFO {
    WORD      rtTypeID;
    WORD      rtResourceCount;
    DWORD     rtReserved;
    NAMEINFO  rtNameInfo[];
} TYPEINFO;
```

Following are the members in the TYPEINFO structure:

rtTypeID	Specifies the type identifier of the resource. This integer value is either a resource-type value or an offset to a resource-type name. If the high bit in this member is set (0x8000), the value is one of the following resource-type values:
----------	---

Value	Resource type
RT_ACCELERATOR	Accelerator table
RT_BITMAP	Bitmap
RT_CURSOR	Cursor
RT_DIALOG	Dialog box
RT_FONT	Font component
RT_FONTDIR	Font directory
RT_GROUP_CURSOR	Cursor directory
RT_GROUP_ICON	Icon directory
RT_ICON	Icon
RT_MENU	Menu
RT_RCDATA	Resource data
RT_STRING	String table

If the high bit of the value in this member is not set, the value represents an offset, in bytes relative to the beginning of the resource table, to a name in the rscResourceNames member.

rtResourceCount        Specifies the number of resources of this type in the executable file.

rtReserved        Reserved.

rtNameInfo        Specifies an array of NAMEINFO structures containing information about individual resources.

The rtResourceCount member specifies the number of structures in the array.

### Name Information

The NAMEINFO structure has the following form:

## 794 A to Z of C

```
typedef struct _NAMEINFO {  
    WORD rnOffset;  
    WORD rnLength;  
    WORD rnFlags;  
    WORD rnID;  
    WORD rnHandle;  
    WORD rnUsage;  
} NAMEINFO;
```

Following are the members in the NAMEINFO structure:

**rnOffset** Specifies an offset to the contents of the resource data (relative to the beginning of the file). The offset is in terms of alignment units specified by the **rscAlignShift** member at the beginning of the resource table.

**rnLength** Specifies the resource length, in bytes.

**rnFlags** Specifies whether the resource is fixed, preloaded, or shareable. This member can be one or more of the following values:

Value	Meaning
0x0010	Resource is movable (MOVEABLE). Otherwise, it is fixed.
0x0020	Resource can be shared (PURE).
0x0040	Resource is preloaded (PRELOAD). Otherwise, it is loaded on demand.

**rnID** Specifies or points to the resource identifier. If the identifier is an integer, the high bit is set (8000h). Otherwise, it is an offset to a resource string, relative to the beginning of the resource table.

**rnHandle** Reserved.

**rnUsage** Reserved.

### Resident-Name Table

The resident-name table contains strings that identify exported functions in the executable file. As the name implies, these strings are resident in system memory and are never discarded. The resident-name strings are case-sensitive and are not null-terminated. The following list summarizes the values found in the resident-name table (the locations are relative to the beginning of each entry):

Location	Description
00h	Specifies the length of a string. If there are no more strings in the table, this value is zero.
01h - xxh	Specifies the resident-name text. This string is case-sensitive and is not null-terminated.
xxh + 01h	Specifies an ordinal number that identifies the string. This number is an index into the entry table.

The first string in the resident-name table is the module name.

## Module-Reference Table

The module-reference table contains offsets for module names stored in the imported-name table. Each entry in this table is 2 bytes long.

## Imported-Name Table

The imported-name table contains the names of modules that the executable file imports. Each entry contains two parts: a single byte that specifies the length of the string and the string itself. The strings in this table are not null-terminated.

## Entry Table

The entry table contains bundles of entry points from the executable file (the linker generates each bundle). The numbering system for these ordinal values is 1-based--that is, the ordinal value corresponding to the first entry point is 1. The linker generates the densest possible bundles under the restriction that it cannot reorder the entry points. This restriction is necessary because other executable files may refer to entry points within a given bundle by their ordinal values. The entry-table data is organized by bundle, each of which begins with a 2-byte header. The first byte of the header specifies the number of entries in the bundle (a value of 00h designates the end of the table). The second byte specifies whether the corresponding segment is movable or fixed. If the value in this byte is 0FFh, the segment is movable. If the value in this byte is 0FEh, the entry does not refer to a segment but refers, instead, to a constant defined within the module. If the value in this byte is neither 0FFh nor 0FEh, it is a segment index.

For movable segments, each entry consists of 6 bytes and has the following form:

Location	Description
00h	Specifies a byte value. This value can be a combination of the following bits:

Bit(s)	Meaning
0	If this bit is set, the entry is exported.
1	If this bit is set, the segment uses a global (shared) data segment.
3-7	If the executable file contains code that performs ring transitions, these bits specify the number of words that compose the stack. At the time of the ring transition, these words must be copied from one ring to the other.

01h	Specifies an int 3fh instruction.
03h	Specifies the segment number.
04h	Specifies the segment offset.

For fixed segments, each entry consists of 3 bytes and has the following form:

Location	Description
00h	Specifies a byte value. This value can be a combination of the following bits:



## 796 A to Z of C

Bit(s)	Meaning
0	If this bit is set, the entry is exported.
1	If this bit is set, the entry uses a global (shared) data segment. (This may be set only for SINGLEDATA library modules.)
3-7	If the executable file contains code that performs ring transitions, these bits specify the number of words that compose the stack. At the time of the ring transition, these words must be copied from one ring to the other.

01h	Specifies an offset.
-----	----------------------

### Nonresident-Name Table

The nonresident-name table contains strings that identify exported functions in the executable file. As the name implies, these strings are not always resident in system memory and are discardable. The nonresident-name strings are case-sensitive; they are not null-terminated. The following list summarizes the values found in the nonresident-name table (the specified locations are relative to the beginning of each entry):

Location	Description
00h	Specifies the length, in bytes, of a string. If this byte is 00h, there are no more strings in the table.
01h - xxh	Specifies the nonresident-name text. This string is case-sensitive and is not null-terminated.
xx + 01h	Specifies an ordinal number that is an index to the entry table.

The first name that appears in the nonresident-name table is the module description string (which was specified in the module-definition file).

### Code Segments and Relocation Data

Code and data segments follow the Windows header. Some of the code segments may contain calls to functions in other segments and may, therefore, require relocation data to resolve those references. This relocation data is stored in a relocation table that appears immediately after the code or data in the segment. The first 2 bytes in this table specify the number of relocation items the table contains. A relocation item is a collection of bytes specifying the following information:

Address type (segment only, offset only, segment and offset)
Relocation type (internal reference, imported ordinal, imported name)
Segment number or ordinal identifier (for internal references)
Reference-table index or function ordinal number (for imported ordinals)
Reference-table index or name-table offset (for imported names)

Each relocation item contains 8 bytes of data, the first byte of which specifies one of the following relocation-address types:

Value	Meaning
0	Low byte at the specified offset
2	16-bit selector
3	32-bit pointer
5	16-bit offset
11	48-bit pointer
13	32-bit offset

The second byte specifies one of the following relocation types:

Value	Meaning
0	Internal reference
1	Imported ordinal
2	Imported name
3	OSFIXUP

The third and fourth bytes specify the offset of the relocation item within the segment. If the relocation type is imported ordinal, the fifth and sixth bytes specify an index to a module's reference table and the seventh and eighth bytes specify a function ordinal value. If the relocation type is imported name, the fifth and sixth bytes specify an index to a module's reference table and the seventh and eighth bytes specify an offset to an imported-name table. If the relocation type is internal reference and the segment is fixed, the fifth byte specifies the segment number, the sixth byte is zero, and the seventh and eighth bytes specify an offset to the segment. If the relocation type is internal reference and the segment is movable, the fifth byte specifies 0FFh, the sixth byte is zero; and the seventh and eighth bytes specify an ordinal value found in the segment's entry table.

## 72.9 GIF

The Graphics Interchange Format (tm) was created by CompuServe Inc. as a standard for the storage and transmission of raster-based graphics information, i.e. images. A GIF file may contain several images, which are to be displayed overlapping and without any delay between the images. The image data itself is compressed using a LZW scheme. Please note that the LZW algorithm is patented by UniSys and that since Jan.1995 royalties to CompuServe are due for every software that implements GIF images. The GIF file consists of a global GIF header, one or more image blocks and optionally some GIF extensions.

## 798 A to Z of C

OFFSET	Count	TYPE	Description
0000h	6	char	ID='GIF87a', ID='GIF89a' This ID may be viewed as a version number
0006h	1	word	Image width
0008h	1	word	Image height
000Ah	1	byte	bit mapped 0-2 - bits per pixel -1 3 - reserved 4-6 - bits of color resolution 7 - Global color map follows image descriptor
000Bh	1	byte	Color index of screen background
000Ch	1	byte	reserved

The global color map immediately follows the screen descriptor and has the size (2\*\*BitsPerPixel), and has the RGB colors for each color index. 0 is none, 255 is full intensity. The bytes are stored in the following format :

OFFSET	Count	TYPE	Description
0000h	1	byte	Red component
0001h	1	byte	Green component
0002h	1	byte	Blue component

After the first picture, there may be more pictures attached in the file which overlay the first picture or parts of the first picture. The Image Descriptor defines the actual placement and extents of the following image within the space defined in the Screen Descriptor. Each Image Descriptor is introduced by an image separator character. The role of the Image Separator is simply to provide a synchronization character to introduce an Image Descriptor, the image separator is defined as ",", 02Ch, Any characters encountered between the end of a previous image and the image separator character are to be ignored.

The format of the Image descriptor looks like this :

OFFSET	Count	TYPE	Description
0000h	1	char	Image separator ID=','
0001h	1	word	Left offset of image
0003h	1	word	Upper offset of image
0005h	1	word	Width of image
0007h	1	word	Height of image
0009h	1	byte	Palette description - bitmapped 0-2 - Number of bits per pixel-1 3-5 - reserved (0) 6 - Interlaced / sequential image 7 - local / global color map, ignore bits 0-2

To provide for some possibility of an extension of the GIF files, a special extension block introducer can be added after the GIF data block. The block has the following structure :

OFFSET	Count	TYPE	Description
0000h	1	char	ID='!'
0001h	1	byte	Extension ID
0002h	?	rec	
	1	word	Byte count
	?	byte	Extra data
????h	1	byte	Zero byte count - terminates extension block.

## 72.10 ICO

An icon-resource file contains image data for icons used by Windows applications. The file consists of an icon directory identifying the number and types of icon images in the file, plus one or more icon images. The default filename extension for an icon-resource file is .ICO.

### Icon Directory

Each icon-resource file starts with an icon directory. The icon directory, defined as an ICONDIR structure, specifies the number of icons in the resource and the dimensions and color format of each icon image. The ICONDIR structure has the following form:

```
typedef struct ICONDIR {
    WORD        idReserved;
    WORD        idType;
    WORD        idCount;
    ICONDIRENTRY idEntries[1];
} ICONHEADER;
```

Following are the members in the ICONDIR structure:

idReserved	Reserved; must be zero.
idType	Specifies the resource type. This member is set to 1.
idCount	Specifies the number of entries in the directory.
idEntries	Specifies an array of ICONDIRENTRY structures containing information about individual icons. The idCount member specifies the number of structures in the array.

The ICONDIRENTRY structure specifies the dimensions and color format for an icon. The structure has the following form:

```
struct IconDirectoryEntry {
    BYTE bWidth;
    BYTE bHeight;
    BYTE bColorCount;
    BYTE bReserved;
    WORD wPlanes;
    WORD wBitCount;
    DWORD dwBytesInRes;
    DWORD dwImageOffset;
};
```

## 800 A to Z of C

Following are the members in the ICONDIRENTRY structure:

bWidth	Specifies the width of the icon, in pixels. Acceptable values are 16, 32, and 64.
bHeight	Specifies the height of the icon, in pixels. Acceptable values are 16, 32, and 64.
bColorCount	Specifies the number of colors in the icon. Acceptable values are 2, 8, and 16.
bReserved	Reserved; must be zero.
wPlanes	Specifies the number of color planes in the icon bitmap.
wBitCount	Specifies the number of bits in the icon bitmap.
dwBytesInRes	Specifies the size of the resource, in bytes.
dwImageOffset	Specifies the offset, in bytes, from the beginning of the file to the icon image.

### Icon Image

Each icon-resource file contains one icon image for each image identified in the icon directory. An icon image consists of an icon-image header, a color table, an XOR mask, and an AND mask. The icon image has the following form:

BITMAPINFOHEADER	icHeader;
RGBQUAD	icColors[];
BYTE	icXOR[];
BYTE	icAND[];

The icon-image header, defined as a BITMAPINFOHEADER structure, specifies the dimensions and color format of the icon bitmap. Only the biSize through biBitCount members and the biSizeImage member are used. All other members (such as biCompression and biClrImportant) must be set to zero. The color table, defined as an array of RGBQUAD structures, specifies the colors used in the XOR mask. As with the color table in a bitmap file, the biBitCount member in the icon-image header determines the number of elements in the array. For more information about the color table, see Section "Bitmap-File Formats."

The XOR mask, immediately following the color table, is an array of BYTE values representing consecutive rows of a bitmap. The bitmap defines the basic shape and color of the icon image. As with the bitmap bits in a bitmap file, the bitmap data in an icon-resource file is organized in scan lines, with each byte representing one or more pixels, as defined by the color format. For more information about these bitmap bits, see Section "Bitmap-File Formats."

The AND mask, immediately following the XOR mask, is an array of BYTE values, representing a monochrome bitmap with the same width and height as the XOR mask. The array is organized in scan lines, with each byte representing 8 pixels.

When Windows draws an icon, it uses the AND and XOR masks to combine the icon image with the pixels already on the display surface. Windows first applies the AND mask by using a

bitwise AND operation; this preserves or removes existing pixel color. Windows then applies the XOR mask by using a bitwise XOR operation. This sets the final color for each pixel.

The following illustration shows the XOR and AND masks that create a monochrome icon (measuring 8 pixels by 8 pixels) in the form of an uppercase K:

**Windows Icon Selection**

Windows detects the resolution of the current display and matches it against the width and height specified for each version of the icon image. If Windows determines that there is an exact match between an icon image and the current device, it uses the matching image. Otherwise, it selects the closest match and stretches the image to the proper size.

If an icon-resource file contains more than one image for a particular resolution, Windows uses the icon image that most closely matches the color capabilities of the current display. If no image matches the device capabilities exactly, Windows selects the image that has the greatest number of colors without exceeding the number of display colors. If all images exceed the color capabilities of the current display, Windows uses the icon image with the least number of colors.

**72.11 JPEG**

Format of a JPEG block (all data is in Motorola byte order) :

OFFSET	Count	TYPE	Description
0000h	1	word	Block ID OFFD8h - JPEG signature block(4 chars="JFIF") OFFC0h - JPEG color information OFFC1h - JPEG color information
0002h	1	word	Block size in bytes, without ID word.

Format of JPEG color information (motorola byte order) :

OFFSET	Count	TYPE	Description
0000h	1	byte	1=Grayscale image
0001h	1	word	Height
0003h	1	word	Width

Another try for JPEG identification could be this one :

OFFSET	Count	TYPE	Description
0000h	1	dword	ID=FFD9FFE0h ID=FFD8FFE0h Big endian JPEG file (Intel) ID=E0FFD8FFh Little endian JPEG file (Motorola)

## 802 A to Z of C

### 72.12 LZH

The LHArc/LHA archiver is a multi platform archiver made by Haruyasu Yoshizaki, which has a relatively good compression. It uses more or less the same technology like the ZIP programs by Phil Katz. There was a hack named "ICE", which had only the graphic characters displayed on decompression changed.

OFFSET	Count	TYPE	Description
0000h	1	byte	Size of archived file header
0001h	1	byte	Checksum of remaining bytes
0002h	3	char	ID='-lh' ID='-lz'
0005h	1	char	Compression methods used (see table 0005)
0006h	1	char	ID='-.'
0007h	1	dword	Compressed size
000Bh	1	dword	Uncompressed size
000Fh	1	dword	Original file date/time (see table 0009)
0013h	1	word	File attribute
0015h	1	byte	Filename / path length in bytes ="LEN"
0016h	"LEN"	char	Filename / path
0018h +"LEN"	1	word	CRC-16 of original file

(Table 0005)

LHArc compression types

- "0" - No compression
- "1" - LZW, 4K buffer, Huffman for upper 6 bits of position
- "2" - unknown
- "3" - unknown
- "4" - LZW, Arithmetic Encoding
- "5" - LZW, Arithmetic Encoding
- "s" - LHa 2.x archive?
- "\" - LHa 2.x archive?
- "d" - LHa 2.x archive?

### 72.13 MIDI

The MIDI file format is used to store MIDI song data on disk. The discussed version of the MIDI file spec is the approved MIDI Manufacturers' Associations format version 0.06 of (3/88). The contact address is listed in the addresses file. Version 1.0 is technically identical but the description has been rewritten. The description was made by Dave Oppenheim, most of the text was taken right out of his document.

MIDI files contain one or more MIDI streams, with time information for each event. Song, sequence, and track structures, tempo and time signature information, are all

supported. Track names and other descriptive information may be stored with the MIDI data. This format supports multiple tracks and multiple sequences so that if the user of a program which supports multiple tracks intends to move a file to another one, this format can allow that to happen.

The MIDI files are block oriented files, currently only 2 block types are defined, header and track data. Opposed to the IFF and RIFF formats, no global header is given, so that the validation must be done by adding the different block sizes.

A MIDI file always starts with a header block, and is followed by one or more track block.

The format of the header block :

OFFSET	Count	TYPE	Description
0000h	4	char	ID='MThd'
0004h	1	dword	Length of header data (=6)
0008h	1	word	Format specification 0 - one, single multi-channel track 1 - one or more simultaneous tracks 2 - one or more sequentially independent single-track patterns
000Ah	1	word	Number of track blocks in the file
000Ch	1	int	Unit of delta-time values. If negative : Absolute of high byte : Number of frames per second. Low byte : Resolution within one frame If positive, division of a quarter-note.

**The track data format :**

The MTrk block type is where actual song data is stored. It is simply a stream of MIDI events (and non-MIDI events), preceded by delta-time values.

Some numbers in MTrk blocks are represented in a form called a variable-length quantity. These numbers are represented 7 bits per byte, most significant bits first. All bytes except the last have bit 7 set, and the last byte has bit 7 clear. If the number is between 0 and 127, it is thus represented exactly as one byte. Since this explanation might not be too clear, some examples :

Number (hex)	Representation (hex)
00000000	00
00000040	40
0000007F	7F
00000080	81 00
00002000	C0 00
00003FFF	FF 7F
001FFFFFF	FF FF 7F
08000000	C0 80 80 00
0FFFFFFF	FF FF FF 7F



## 804 A to Z of C

The largest number which is allowed is 0FFFFFFF so that the variable-length representation must fit in 32 bits in a routine to write variable-length numbers.

Each track block contains one or more MIDI events, each event consists of a delta-time and the number of the event. The delta-time is stored as a variable-length quantity and represents the time to delay before the following event. A delta-time of 0 means, that the event occurs simultaneous with the previous event or occurs right at the start of a track. The delta-time unit is specified in the header block.

Format of track information block :

OFFSET	Count	TYPE	Description
0000h	4	char	ID='MTrk'
0004h	1	dword	Length of header data
0008h	?	rec	<delta-time>, <event>

Three types of events are defined, MIDI event, system exclusive event and meta event. The first event in a file must specify status; delta-time itself is not an event. Meta events are non-MIDI informations.

The format of the meta event :

OFFSET	Count	TYPE	Description
0000h	1	byte	ID=FFh
0001h	1	byte	Type (<=128)
0002h	?	?	Length of the data, 0 if no data stored as variable length quantity
	?	byte	Data

A few meta-events are defined. It is not required for every program to support every meta-event. Meta-events initially defined include:

FF 00 02 ssss Sequence Number

This optional event, which must occur at the beginning of a track, before any nonzero delta-times, and before any transmittable MIDI events, specifies the number of a sequence.

FF 01 len text Text Event

Any amount of text describing anything. It is a good idea to put a text event right at the beginning of a track, with the name of the track, a description of its intended orchestration, and any other information which the user wants to put there. Programs on a computer which does not support non-ASCII characters should ignore those characters with the hi-bit set. Meta event types 01 through 0F are reserved for various types of text events, each of which meets the specification of text events(above) but is used for a different purpose:

FF 02 len text Copyright Notice

Contains a copyright notice as printable ASCII text. The notice should contain the characters (C), the year of the copyright, and the owner of the copyright. If several pieces of music are in the same MIDI file, all of the copyright notices should be placed together in this event so that

it will be at the beginning of the file. This event should be the first event in the first track block, at time 0.

FF 03 len text Sequence/Track Name

If in a format 0 track, or the first track in a format 1 file, the name of the sequence. Otherwise, the name of the track.

FF 04 len text Instrument Name

A description of the type of instrumentation to be used in that track.

FF 05 len text Lyric

A lyric to be sung. Generally, each syllable will be a separate lyric event which begins at the event's time.

FF 06 len text Marker

Normally in a format 0 track, or the first track in a format 1 file. The name of that point in the sequence, such as a rehearsal letter or section name ("First Verse", etc.).

FF 07 len text Cue Point

A description of something happening on a film or video screen or stage at that point in the musical score ("Car crashes into house", "curtain opens", "she slaps his face", etc.)

FF 2F 00 End of Track

This event is not optional. It is included so that an exact ending point may be specified for the track, so that it has an exact length, which is necessary for tracks which are looped or concatenated.

FF 51 03 tttttt Set Tempo, in microseconds per MIDI quarter-note

This event indicates a tempo change. Another way of putting "microseconds per quarter-note" is "24ths of a microsecond per MIDI clock". Representing tempos as time per beat instead of beat per time allows absolutely exact dword-term synchronization with a time-based sync protocol such as SMPTE time code or MIDI time code. This amount of accuracy provided by this tempo resolution allows a four-minute piece at 120 beats per minute to be accurate within 500 usec at the end of the piece. Ideally, these events should only occur where MIDI clocks would be located Q this convention is intended to guarantee, or at least increase the likelihood, of compatibility with other synchronization devices so that a time signature/tempo map stored in this format may easily be transferred to another device.

FF 54 05 hr mn se fr ff SMPTE Offset

This event, if present, designates the SMPTE time at which the track block is supposed to start. It should be present at the beginning of the track, that is, before any nonzero delta-times, and before any transmittable MIDI events. The hour must be encoded with the SMPTE format, just as it is in MIDI Time Code. In a format 1 file, the SMPTE Offset must be stored with the tempo map, and has no meaning in any of the other tracks. The ff field contains fractional frames, in 100ths of a frame, even in SMPTE-based tracks which specify a different frame subdivision for delta-times.

## 806 A to Z of C

FF 58 04 nn dd cc bb Time Signature

The time signature is expressed as four numbers. nn and dd represent the numerator and denominator of the time signature as it would be notated. The denominator is a negative power of two: 2 represents a quarter-note, 3 represents an eighth-note, etc. The cc parameter expresses the number of MIDI clocks in a metronome click. The bb parameter expresses the number of notated 32nd-notes in a MIDI quarter-note (24 MIDI Clocks).

FF 59 02 sf mi Key Signature

sf = -7: 7 flats  
sf = -1: 1 flat  
sf = 0: key of C  
sf = 1: 1 sharp  
sf = 7: 7 sharps

mi = 0: major key  
mi = 1: minor key

FF 7F len data Sequencer-Specific Meta-Event

Special requirements for particular sequencers may use this event type: the first byte or bytes of data is a manufacturer ID. However, as this is an interchange format, growth of the spec proper is preferred to use of this event type. This type of event may be used by a sequencer which elects to use this as its only file format; sequencers with their established feature-specific formats should probably stick to the standard features when using this format.

The system exclusive event is used as an escape to specify arbitrary bytes to be transmitted. The system exclusive event has two forms, to compensate for some manufacturer-specific modes, the F7h event is used if a F0h is to be transmitted. Each system exclusive event must end with an F7h event.

The format of a system exclusive event :

OFFSET	Count	TYPE	Description
0000h	1	byte	ID=F0h, ID=F7h
0001h	?	?	Length as variable length qty.
	?	byte	bytes to be transmitted

## 72.14 PCX

The PCX files are created by the programs of the ZSoft Paintbrush family and the FRIEZE package by the same manufacturer. A PCX file contains only one image, the data for this image and possibly palette information for this image. The encoding scheme used for PCX encoding is a simple RLE mechanism, see ALGRTHMS.txt for further information. A PCX image is stored from the upper scan line to the lower scan line.

The size of a decoded scan line is always an even number, thus one additional byte should always be allocated for the decoding buffer.

The header has a fixed size of 128 bytes and looks like this :