OFFSET	Count	TYPE	Description
0000h	1	byte	Manufacturer. 10=ZSoft
0001h	1	byte	Version information 0=PC Paintbrush v2.5 2=PC Paintbrush v2.8 w palette information 3=PC Paintbrush v2.8 w/o palette information 4=PC Paintbrush/Windows 5=PC Paintbrush v3.0+
0002h	1	byte	Encoding scheme, 1 = RLE, none other known
0003h	1	byte	Bits per pixel
0004h	1	word	left margin of image
0006h	1	word	upper margin of image
0008h	1	word	right margin of image
000Ah	1	word	lower margin of image
000Ch	1	word	Horizontal DPI resolution
000Eh	1	word	Vertical DPI resolution
0010h	48	byte	Color palette setting for 16-color images 16 RGB triplets
0040h	1	byte	reserved
0041h	1	byte	Number of color planes = "NCP"
0042h	1	word	Number of bytes per scanline (always even, use instead of right margin-left margin). ="NBS"
0044h	1	word	Palette information 1=color/bw palette 2=grayscale image
0046h	1	word	Horizontal screen size
0048h	1	word	Vertical screen size
004Ah	54	byte	reserved, set to 0

The space needed to decode a single scan line is "NCP"*"NBS" bytes, the last byte may be a junk byte which is not displayed. After the image data, if the version number is 5 (or greater?) there possibly is a VGA color palette. The color ranges from 0 to 255, 0 is zero intensity, 255 is full intensity. The palette has the following format :

OFFSET	Count	TYPE	Description
0000h	1	byte	VGA palette ID (=0Ch)
0001h	768	byte	RGB triplets with palette information

72.15 PIF

The Program Information Files have stayed a long time with the PC. They origi- nated from IBMs Topview, were carried on by DoubleView and DesqView, and today they are used by Windows and Windows NT. The PIF files store additional information about executables that are foreign to the running multitasking system such as ressource usage, keyboard and mouse virtualization and hotkeys. The original (Topview) PIF had a size of

171h bytes, after that, there come the various extensions for the different operating environments. The different extensions are discussed in their own sections.

OFFSET	Count	TYPE	Description
0000h	1	byte	reserved
0001h	1	byte	Checksum
0002h	30	char	Title for the window
0020h	1	word	Maximum memory reserved for program
0022h	1	word	Minimum memory reserved for program
0024h	63	char	Path and filename of the program
0063h	1	byte	0 - Do not close window on exit other - Close window on exit
0064h	1	byte	Default drive (0=A: ??)
0065h	64	char	Default startup directory
00A5h	64	char	Parameters for program
00E5h	1	byte	Initial screen mode, 0 equals mode 3 ?
00E6h	1	byte	Text pages to reserve for program
00E7h	1	byte	First interrupt used by program
00E8h	1	byte	Last interrupt used by program
00E9h	1	byte	Rows on screen
00EAh	1	byte	Columns on screen
00EBh	1	byte	X position of window
00ECh	1	byte	Y position of window
00EDh	1	word	System memory ?? whatever
00EFh	64	char	?? Shared program path
012Fh	64	char	?? Shared program data file
016Fh	1	word	Program flags

72.16 RTF

RTF text is a form of encoding of various text formatting properties, document structures, and document properties, using the printable ASCII character set. Special characters can be also thus encoded, although RTF does not prevent the utilization of character codes outside the ASCII printable set. The main encoding mechanism of "control words" provides a name space that may be later used to expand the realm of RTF with macros, programming, etc.

1. BASIC INGREDIENTS

Control words are of the form:

\lettersequence <delimiter> where <delimiter>. is:

. a space: the space is part of the control word.

. a digit or - means that a parameter follows. The following digit sequence is then delimited by a space or any other non-letter-or-digit as for control words.

. any other non-letter-or digit: terminates the control word, but is not a part of the control word.

By "letter:, here we mean just the upper and lower case ASCII letters.

Control symbols consist of a \ character followed by a single non-letter. They require no further delimiting.

Notes: control symbols are compact, but there are not too many of them. The number of possible control words are not limited.

The parameter is partially incorporated in control symbols, so that a program that does not understand a control symbol can recognize and ignore the corresponding parameter as well.

In addition to control words and control symbols, there are also the braces:

{ group start, and

} group end. The text grouping will be used for formatting

and to delineate document structure - such as the footnotes, headers, title, and so on. The control words, control symbols, and braces constitute control information. All other characters in RTF text constitute "plain text".

Since the characters \, {, and } have specific uses in RTF, the control symbols \\,\{, and \} are provided to express the corresponding plain characters.

2. WHAT RTF TEXT MEANS (SEMANTICS)

The reader of a RTF stream will be concerned with:
Separating control information from plain text. Acting on control information. This is designed to be a relatively simple process, as described below. Some control information just contributes special characters to the plain text stream. Other information serves to change the "program state" which includes properties of the document as a whole and also a stack of "group states" that apply to parts. Note that the group state is saved by the { brace and is restored by the } brace. The current group state specifies:

1. the "destination" or part of the document that the plain text is building up.
2. the character formatting properties - such as bold or italic.
3. the paragraph formatting properties - such as justified.
4. the section formatting properties - such as number of columns.

Collecting and properly disposing of the remaining "plain text" as directed by the current group state.

In practice the RTF reader will proceed as follows:

0. read next char

1. if = {

stack current state. current state does not change.
continue.

2. if = }

unstack current state from stack. this will change the state in general.

3. if = \

collect control word/control symbol and parameter, if any. look up word/symbol in symbol table (a constant table) and act according to the description there. The different actions are listed below. Parameter is left available for use by the action.

Leave read pointer before or after the delimiter, as appropriate.

After the action, continue.

4. otherwise, write "plain text" character to current destination using current formatting properties.

Given a symbol table entry, the possible actions are as follows:

A. Change destination:
change destination to the destination described in the entry.
Most destination changes are legal only immediately after a {.
Other restrictions may also apply (for example, footnotes may not be nested.)
B. Change formatting property:
The symbol table entry will describe the property and whether the parameter is required.

C. Special character:
The symbol table entry will describe the character code..
goto 4.
D. End of paragraph
This could be viewed as just a special character.
E. End of section
This could be viewed as just a special character.
F. Ignore

3. SPECIAL CHARACTERS

The special characters are explained as they exist in Mac Word. Clearly, other characters may be added for interchange with other programs. If a character name is not recognized by a reader, according to the rules described above, it will be simply ignored.

\chpgn	current page number (as in headers)
\chftn	auto numbered footnote reference(footnote to follow in a group)
\chpict	placeholder character for picture (picture to follow in a group)
\chdate	current date (as in headers)
\chtime	current time (as in headers)
\\|	formula character
\~	non-breaking space
\-	non-required hyphen
_	non-breaking hyphen
\page	required page break
\line	required line break (no paragraph break)
\par	end of paragraph.
\sect	end of section and end of paragraph.
\tab	same as ASCII 9

For simplicity of operation, the ASCII codes 9 and 10 will be accepted as \tab and \par respectively. ASCII 13 will be ignored. The control code \<10> will be ignored. It may be used to include "soft" carriage returns for easier readability but which will have no effect on the interpretation.

4. DESTINATIONS

The change of destination will reset all properties to default. Changes are legal only at the beginning of a group (by group here we mean the text and controls enclosed in braces.)

\rtf<param>	The destination is the document. The parameter is the version number of the writer. This destination preceded by { the beginnings of RTF documents and the corresponding } marks the end. Legal only once after the initial {. Small scale interchange of RTF where other methods for marking the end of string are available, as in a string constant, need not include this identification but will start with this destination as the default.
\pict	The destination is a picture. The group must immediately follow a \chpict character. The plain text describes the picture as a hex dump (string of characters 0,1,...9, a, ..., e, f.)
\footnote	The destination is a footnote text. The group must immediately follow the footnote reference character(s).
\header	The destination is the header text for the current section. The group must precede the first plain text character in the section.
\headerl	Same as above, but header for left-hand pages.
\headerr	Same as above, but header for right-hand pages.
\headerf	Same as above, but header for first page.
\footer	Same as above, but footer.
\footerl	Same as above, but footer for left-hand pages.
\footerr	Same as above, but footer for right-hand pages.
\footerf	Same as above, but header for first page.
\ftnsep	Same as above, but text is footnote separator
\ftnsepc	Same as above, but text is separator for continued footnotes.

\ftncn	Same as above, but text is continued footnote notice.
\info	text is information block for the document. Parts of the text is further classified by "properties" of the text that are listed below - such as "title". These are not formatting properties, but a device to delimit and identify parts of the info from the text in the group.
\stylesheet	text is the style sheet for the document. More precisely, text between semicolons are taken to be style names which will be defined to stand for the formatting properties which are in effect.
\fonttbl	font table. See below.
\colortbl	color table. See below.
\comment	text will be ignored.

5. DOCUMENT FORMATTING PROPERTIES

(000 stands for a number which may be signed)

\paperw000	paper width in twips	12240
\paperh000	paper height	15840
\margl000	left margin	1800
\margr000	right margin	1800
\margt000	top margin	1440
\margb000	bottom margin	1440
\facingp	facing pages	
\gutter000	gutter width	
\deftab000	default tab width	720
\widowctrl	enable widow control	
\endnotes	footnotes at end of section	
\ftnbj	footnotes at bottom of page	default
\ftntj	footnotes beneath text (top just)	
\ftnstart000	starting footnote number	1
\ftnrestart	restart footnote numbers each page	
\pgnstart000	starting page number	1
\linestart000	starting line number	1
\landscape	printed in landscape format	

(the "next file" property will be encoded in the info text)

6. SECTION FORMATTING PROPERTIES

\sectd	reset to default section properties	
\nobreak	break code	
\colbreak	break code	default
\pagebreak	break code	
\evenbreak	break code	
\oddbreak	break code	
\pgnrestart	restart page numbers at 1	
\pgndec	page number format decimal	default

\pgnucrm	page number format uc roman	
\pgnlcrm	page number format lc roman	
\pgnucltr	page number format uc letter	
\pgnlcltr	page number format lc letter	
\pgnx000	auto page number x pos	720
\pgny000	auto page number y pos	720
\linemod000	line number modulus	
\linex000	line number - text distance	360
\linerestart	line number restart at 1	default
\lineppage	line number restart on each page	
\linecont	line number continued from prev section	
\headery000	header y position from top of page	720
\footery000	footer y position from bottom of page	720
\cols000	number of columns	1
\colsx000	space between columns	720
\endnhere	include endnotes in this section	
\titlepg	title page is special	

7. PARAGRAPH FORMATTING PROPERTIES

\pard	dreset to default para properties.
\s000	style
\ql	quad left (default)
\ql	right
\qj	justified
\qc	centered
\fi000	first line indent
\li000	left indent
\ri000	right indent
\sb000	space before
\sa000	space after
\sl000	space between lines
\keep	keep
\keepn	keep with next para
\sbys	side by side
\pagebb	page break before
\noline	no line numbering
\brdrt	border top
\brdrb	border bottom
\brdrl	border left
\brdrr	border right
\box	border all around
\brdrs	single thickness
\brdrth	thick
\brdrsh	shadow
\brdrdb	double

\tx000	tab position
\tqr	right flush tab (these apply to last specified pos)
\tqc	centered tab
\tqdec	decimal aligned tab
\tldot	leader dots
\tlhyph	leader hyphens
\tlul	leader underscore
\tlth	leader thick line

8. CHARACTER FORMATTING PROPERTIES

\plain	reset to default text properties.	
\b	bold	
\i	italic	
\strike	strikethrough	
\outl	outline	
\shad	shadow	
\scaps	small caps	
\caps	all caps	
\v	invisible text	
\f000	font number n	
\fs000	font size in half points	24
\ul	underline	
\ulw	word underline	
\uld	dotted underline	
\uldb	double underline	
\up000	superscript in half points	
\dn000	subscript in half points	

9. INFO GROUP

The plain text in the group is used to specify the various fields of the information block. The current field may be thought of as a particular setting of the "sub-destination" property of the text..

\title	following plain text is the title
\subject	following text is the subject
\operator	
\author	
\keywords	
\doccomm	comments (not to be confused with \comment)
\version	
\nextfile	following text is name of "next" file

The other properties assign their parameters directly to the

info block.

\verno000	internal version number
\creatim	creation time follows
\yr000	year to be assigned to previously specified timefield
\mo000	
\dy000	
\hr000	
\min000	
\sec000	
\revtim	revision time follows
\printtim	print time follows
\buptim	backup time follows
\edmins00	editing minutes
\nofpages000	
\nofwords000	
\noofchars000	
\id000	internal ID number

72.17 SCR

SCR files are Windows EXE files (EXE NE) with the extension SCR. Windows calls the .SCR file with two command-line options:

/s	to launch the screensaver
/c	to configure the screensaver

For the windows control panel to recognise the screensaver, the program's module description string must begin with SCRNSAVE: (in uppercase). So, if writing a Visual Basic screensaver, simply set the application title to something like "SCRNSAVE:My Screensaver"

To create a new screen saver simply write a program that checks the command-line option when starting and performs the appropriate action. The display should use a full-screen window (usually with a black background) and should end when any key is pressed or when the mouse is moved.

Compile the program to .SCR.

72.18 WAV

The Windows .WAV files are RIFF format files. Some programs expect the fmt block right behind the RIFF header itself, so your programs should write out this block as the first block in the RIFF file.

The subblocks for the wave files are
RiffBLOCK [data]

This block contains the raw sample data. The necessary information for playback is contained in the [fmt] block.

RiffBLOCK [fmt]
This block contains the data necessary for playback of the sound files. Note the blank after fmt !

OFFSET	Count	TYPE	Description
0000h	1	word	Format tag 1 = PCM (raw sample data) 2 etc. for APCDM, a-Law, u-Law ...
0002h	1	word	Channels (1=mono,2=stereo,...)
0004h	1	dword	Sampling rate
0008h	1	dword	Average bytes per second (=sampling rate*channels)
000Ch	1	word	Block alignment / reserved ??
000Eh	1	word	Bits per sample (8/12/16-bit samples)

RiffBLOCK [loop]
This block is for looped samples. Very few programs support this block, but if your program changes the wave file, it should preserve any unknown blocks.

OFFSET	Count	TYPE	Description
0000h	1	dword	Start of sample loop
0004h	1	dword	End of sample loop

72.19 ZIP

Following is the official documenation of PKZIP.

PKZIP® Application Note

File: APPNOTE.TXT - .ZIP File Format Specification
Version: 4.0
Revised: 11/01/2000

I. Disclaimer

II. General Format of a .ZIP file

A. Local file header

B. File data

C. Data descriptor

D. Central directory structure

E. Explanation of fields

I. Disclaimer

Although PKWARE will attempt to supply current and accurate information relating to its file formats, algorithms, and the subject programs, the possibility of error can not be eliminated. PKWARE therefore expressly disclaims any warranty that the information contained in the associated materials relating to the subject programs and/or the format of the files created or accessed by the subject programs and/or the algorithms used by the subject programs, or any other matter, is current, correct or accurate as delivered. Any risk of damage due to any possible inaccurate information is assumed by the user of the information. Furthermore, the information relating to the subject programs and/or the file formats created or accessed by the subject programs and/or the algorithms used by the subject programs is subject to change without notice.

II. General Format of a ZIP file

Files stored in arbitrary order. Large zipfiles can span multiple diskette media or be split into user-defined segment sizes. The minimum user-defined segment size for a split .ZIP file is 64K..

Overall zipfile format:

[local file header1]
[file data 1]
[data_descriptor 1]
.
.
.
[local file header n]
[file data n]
[data_descriptor n]
[central directory]

A. Local file header:

local file header signature 4 bytes (0x04034b50)

version needed to extract	2 bytes
general purpose bit flag	2 bytes
compression method	2 bytes
last mod file time	2 bytes
last mod file date	2 bytes
crc-32	4 bytes
compressed size	4 bytes
uncompressed size	4 bytes
filename length	2 bytes
extra field length	2 bytes
filename	(variable size)
extra field	(variable size)

B. File data:

Immediately following the local header for a file is the compressed or stored data for the file. The series of [local file header][file data][data descriptor] repeats for each file in the .ZIP archive.

C. Data descriptor:

crc-32	4 bytes
compressed size	4 bytes
uncompressed size	4 bytes

This descriptor exists only if bit 3 of the general purpose bit flag is set (see below). It is byte aligned and immediately follows the last byte of compressed data. This descriptor is used only when it was not possible to seek in the output zip file, e.g., when the output zip file was standard output or a non seekable device.

D. Central directory structure:

[file header 1]
.
.
.
[file header n]
[digital signature]
[end of central directory record]

File header:

central file header signature	4 bytes (0x02014b50)
version made by	2 bytes
version needed to extract	2 bytes
general purpose bit flag	2 bytes
compression method	2 bytes

last mod file time	2 bytes
last mod file date	2 bytes
crc-32	4 bytes
compressed size	4 bytes
uncompressed size	4 bytes
filename length	2 bytes
extra field length	2 bytes
file comment length	2 bytes
disk number start	2 bytes
internal file attributes	2 bytes
external file attributes	4 bytes
relative offset of local header	4 bytes
filename	(variable size)
extra field	(variable size)
file comment	(variable size)

End of central dir record:

end of central dir signature	4 bytes (0x06054b50)
number of this disk	2 bytes
number of the disk with the start of the central directory	2 bytes
total number of entries in the central dir on this disk	2 bytes
total number of entries in the central dir	2 bytes
size of the central directory	4 bytes
offset of start of central directory with respect to the starting disk number	4 bytes
.ZIP file comment length	2 bytes
.ZIP file comment	(variable size)

E. Explanation of fields:

version made by (2 bytes)

The upper byte indicates the compatibility of the file attribute information. If the external file attributes are compatible with MS-DOS and can be read by PKZIP for DOS version 2.04g then this value will be zero. If these attributes are not compatible, then this value will identify the host system on which the attributes are compatible. Software can use this information to determine the line record format for text files etc. The current mappings are:

0 - MS-DOS and OS/2 (FAT / VFAT / FAT32 file systems)
1 - Amiga
2 - OpenVMS
3 - Unix

4 - VM/CMS
5 - Atari ST
6 - OS/2 H.P.F.S.
7 - Macintosh
8 - Z-System
9 - CP/M
10 - Windows NTFS
11 thru 255 - unused

The lower byte indicates the version number of the software used to encode the file. The value/10 indicates the major version number, and the value mod 10 is the minor version number.

version needed to extract (2 bytes)

The minimum software version needed to extract the file, mapped as above.

general purpose bit flag: (2 bytes)

Bit 0: If set, indicates that the file is encrypted.

(For Method 6 - Imploding)

Bit 1: If the compression method used was type 6, Imploding, then this bit, if set, indicates an 8K sliding dictionary was used. If clear, then a 4K sliding dictionary was used.

Bit 2: If the compression method used was type 6, Imploding, then this bit, if set, indicates an 3 Shannon-Fano trees were used to encode the sliding dictionary output. If clear, then 2 Shannon-Fano trees were used.

(For Methods 8 and 9 - Deflating)

Bit 2	Bit 1	
0	0	Normal (-en) compression option was used.
0	1	Maximum (-exx/-ex) compression option was used.
1	0	Fast (-ef) compression option was used.
1	1	Super Fast (-es) compression option was used.

Note: Bits 1 and 2 are undefined if the compression method is any other.

Bit 3: If this bit is set, the fields crc-32, compressed size and uncompressed size are set to zero in the local header. The correct values are put in the data descriptor immediately following the compressed data. (Note: PKZIP version 2.04g for DOS only recognizes this bit for method 8 compression, newer versions of PKZIP recognize this bit for any compression method.)

Bit 4: Reserved for use with method 8, for enhanced deflating.

Bit 5: If this bit is set, this indicates that the file is compressed patched data.(Note: Requires PKZIP version 2.70 or greater)

Bit 6: Currently unused.
Bit 7: Currently unused.
Bit 8: Currently unused.
Bit 9: Currently unused.
Bit 10: Currently unused.
Bit 11: Currently unused.
Bit 12: Reserved by PKWARE for enhanced compression.
Bit 13: Reserved by PKWARE.
Bit 14: Reserved by PKWARE.
Bit 15: Reserved by PKWARE.

compression method: (2 bytes)

(see accompanying documentation for algorithm descriptions)

0 - The file is stored (no compression)
1 - The file is Shrunk
2 - The file is Reduced with compression factor 1
3 - The file is Reduced with compression factor 2
4 - The file is Reduced with compression factor 3
5 - The file is Reduced with compression factor 4
6 - The file is Imploded
7 - Reserved for Tokenizing compression algorithm
8 - The file is Deflated
9 - Enhanced Deflating using Deflate64(tm)
10 - PKWARE Date Compression Library Imploding

date and time fields: (2 bytes each)

The date and time are encoded in standard MS-DOS format. If input came from standard input, the date and time are those at which compression was started for this data.

CRC-32: (4 bytes)

The CRC-32 algorithm was generously contributed by David Schwaderer and can be found in his excellent book "C Programmers Guide to NetBIOS" published by Howard W. Sams & Co. Inc. The 'magic number' for the CRC is 0xdebb20e3. The proper CRC pre and post conditioning is used, meaning that the CRC register is pre-conditioned with all ones (a starting value of 0xffffffff) and the value is post-conditioned by taking the one's complement of the CRC residual. If bit 3 of the general purpose flag is set, this field is set to zero in the local header and the correct value is put in the data descriptor and in the central directory.

compressed size: (4 bytes)
uncompressed size: (4 bytes)

The size of the file compressed and uncompressed, respectively. If bit 3 of the general purpose bit flag is set, these fields are set to zero in the local header and the correct values are put in the data descriptor and in the central directory.

filename length: (2 bytes)
extra field length: (2 bytes)
file comment length: (2 bytes)

The length of the filename, extra field, and comment fields respectively. The combined length of any directory record and these three fields should not generally exceed 65,535 bytes. If input came from standard input, the filename length is set to zero.

disk number start: (2 bytes)

The number of the disk on which this file begins.

internal file attributes: (2 bytes)

The lowest bit of this field indicates, if set, that the file is apparently an ASCII or text file. If not set, that the file apparently contains binary data. The remaining bits are unused in version 1.0.

external file attributes: (4 bytes)

The mapping of the external attributes is host-system dependent (see 'version made by'). For MS-DOS, the low order byte is the MS-DOS directory attribute byte. If input came from standard input, this field is set to zero.

relative offset of local header: (4 bytes)

This is the offset from the start of the first disk on which this file appears, to where the local header should be found.

filename: (Variable)

The name of the file, with optional relative path. The path stored should not contain a drive or device letter, or a leading slash. All slashes should be forward slashes '/' as opposed to backwards slashes '\' for compatibility with Amiga and Unix file systems etc. If input came from standard input, there is no filename field.

extra field: (Variable)

This is for future expansion. If additional information needs to be stored in the future, it should be stored here. Earlier versions of the software can then safely skip this file, and find the next file or header. This field will be 0 length in version 1.0.

In order to allow different programs and different types of information to be stored in the 'extra' field in .ZIP files, the following structure should be used for all programs storing data in this field:

header1+data1 + header2+data2 . . .

Each header should consist of:

Header ID 2 bytes
Data Size 2 bytes

Note: all fields stored in Intel low-byte/high-byte order.

The Header ID field indicates the type of data that is in the following data block.

Header ID's of 0 thru 31 are reserved for use by PKWARE. The remaining ID's can be used by third party vendors for proprietary usage.

The current Header ID mappings defined by PKWARE are:

0x0007 AV Info
0x0009 OS/2
0x000A NTFS
0x000c OpenVMS
0x000d Unix
0x000f Patch Descriptor
0x0014 PKCS#7 Store for X.509 Certificates
0x0015 X.509 Certificate ID and Signature for individual file
0x0016 X.509 Certificate ID for Central Directory

Several third party mappings commonly used are:

0x4b46 FWKCS MD5 (see below)
0x07c8 Macintosh
0x4341 Acorn/SparkFS
0x4453 Windows NT security descriptor (binary ACL)
0x4704 VM/CMS
0x470f MVS
0x4c41 OS/2 access control list (text ACL)
0x4d49 Info-ZIP OpenVMS
0x5455 extended timestamp
0x5855 Info-ZIP Unix (original, also OS/2, NT, etc)
0x6542 BeOS/BeBox
0x756e ASi Unix
0x7855 Info-ZIP Unix (new)

0xfd4a SMS/QDOS

The Data Size field indicates the size of the following data block. Programs can use this value to skip to the next header block, passing over any data blocks that are not of interest.

Note: As stated above, the size of the entire .ZIP file header, including the filename, comment, and extra field should not exceed 64K in size.

In case two different programs should appropriate the same Header ID value, it is strongly recommended that each program place a unique signature of at least two bytes in size (and preferably 4 bytes or bigger) at the start of each data area. Every program should verify that its unique signature is present, in addition to the Header ID value being correct, before assuming that it is a block of known type.

-OS/2 Extra Field:

The following is the layout of the OS/2 attributes "extra" block. (Last Revision 09/05/95)

Note: all fields stored in Intel low-byte/high-byte order.

Value	Size	Description
0x0009	2 bytes	Tag for this "extra" block type
TSize	2 bytes	Size for the following data block
BSize	Long	Uncompressed Block Size
CType	2 bytes	Compression type
EACRC	Long	CRC value for uncompress block
(var)	variable	Compressed block

The OS/2 extended attribute structure (FEA2LIST) is compressed and then stored in it's entirety within this structure. There will only ever be one "block" of data in VarFields[].

-UNIX Extra Field:

The following is the layout of the Unix "extra" block.

Note: all fields are stored in Intel low-byte/high-byte order.

Value	Size	Description
0x000d	2 bytes	Tag for this "extra" block type
TSize	2 bytes	Size for the following data block
Atime	4 bytes	File last access time
Mtime	4 bytes	File last modification time
Uid	2 bytes	File user ID
Gid	2 bytes	File group ID

(var)	variable	Variable length data field

The variable length data field will contain file type specific data. Currently the only values allowed are the original "linked to" file names for hard or symbolic links.

-OpenVMS Extra Field:

The following is the layout of the OpenVMS attributes "extra" block.

Note: all fields stored in Intel low-byte/high-byte order.

Value	Size	Description
0x000c	2 bytes	Tag for this "extra" block type
TSize	2 bytes	Size of the total "extra" block
CRC	4 bytes	32-bit CRC for remainder of the block
Tag1	2 bytes	VMS attribute tag value #1
Size1	2 bytes	Size of attribute #1, in bytes
(var.)	Size1	Attribute #1 data
.		
.		
.		
TagN	2 bytes	VMS attribute tage value #N
SizeN	2 bytes	Size of attribute #N, in bytes
(var.)	SizeN	Attribute #N data

Rules:

1. There will be one or more of attributes present, which will each be preceded by the above TagX & SizeX values. These values are identical to the ATR$C_XXXX and ATR$S_XXXX constants which are defined in ATR.H under OpenVMS C. Neither of these values will ever be zero.

2. No word alignment or padding is performed.

3. A well-behaved PKZIP/OpenVMS program should never produce more than one sub-block with the same TagX value. Also, there will never be more than one "extra" block of type 0x000c in a particular directory record.

-NTFS Extra Field:

The following is the layout of the NTFS attributes "extra" block.

Note: At this time, the Mtime, Atime and Ctime values may be used on any Win32 system.

Value	Size	Description

0x000a	2 bytes	Tag for this "extra" block type
TSize	2 bytes	Size of the total "extra" block
Reserved	4 bytes	Reserved for future use
Tag1	2 bytes	NTFS attribute tag value #1
Size1	2 bytes	Size of attribute #1, in bytes
(var.)	Size1	Attribute #1 data
.		
.		
.		
TagN	2 bytes	NTFS attribute tage value #N
SizeN	2 bytes	Size of attribute #N, in bytes
(var.)	SizeN	Attribute #N data

For NTFS, values for Tag1 through TagN are as follows:
(currently only one set of attributes is defined for NTFS)

Tag	Size	Description
0x0001	2 bytes	Tag for attribute #1
Size1	2 bytes	Size of attribute #1, in bytes
Mtime	8 bytes	File last modification time
Atime	8 bytes	File last access time
Ctime	8 bytes	File creation time

-PATCH Descriptor Extra Field:

The following is the layout of the Patch Descriptor "extra" block.

Note: all fields stored in Intel low-byte/high-byte order.

Value	Size	Description
0x000f	2 bytes	Tag for this "extra" block type
TSize	2 bytes	Size of the total "extra" block
Version	2 bytes	Version of the descriptor
Flags	4 bytes	Actions and reactions (see below)
OldSize	4 bytes	Size of the file about to be patched
OldCRC	4 bytes	32-bit CRC of the file about to be patched
NewSize	4 bytes	Size of the resulting file
NewCRC	4 bytes	32-bit CRC of the resulting file

Actions and reactions

Bits	Description
0	Use for autodetection
1	Treat as selfpatch
2-3	RESERVED
4-5	Action (see below)
6-7	RESERVED

8-9	Reaction (see below) to absent file
10-11	Reaction (see below) to newer file
12-13	Reaction (see below) to unknown file
14-15	RESERVED
16-31	RESERVED

Actions

Action	Value
none	0
add	1
delete	2
patch	3

Reactions

Reaction	Value
ask	0
skip	1
ignore	2
fail	3

-PKCS#7 Store for X.509 Certificates

This field is contains the information about each certificate a file is signed with. This field should only appear in the first central directory record, and will be ignored in any other record.

Note: all fields stored in Intel low-byte/high-byte order.

Value	Size	Description
0x0014	2 bytes	Tag for this "extra" block type
SSize	2 bytes	Size of the stored data
SData	(variable)	Data about the store

SData

Value	Size	Description
Version	2 bytes	Version number, 0x0001 for now
StoreD	(variable)	Actual store data

The StoreD member is suitable for passing as the pbData member of a CRYPT_DATA_BLOB to the CertOpenStore() function in Microsoft's CryptoAPI. The SSize member above will be cbData + 6, where cbData is the cbData member of the same CRYPT_DATA_BLOB. The encoding type to pass to CertOpenStore() should be PKCS_7_ANS_ENCODING | X509_ASN_ENCODING.

-X.509 Certificate ID and Signature for individual file

This field contains the information about which certificate in the PKCS#7 Store was used to sign the particular file. It also contains the signature data. This field can appear multiple times, but can only appear once per certificate.

Note: all fields stored in Intel low-byte/high-byte order.

Value	Size	Description
0x0015	2 bytes	Tag for this "extra" block type
CSize	2 bytes	Size of Method
Method	(variable)	

Method

Value	Size	Description
Version	2 bytes	Version number, 0x0001 for now
AlgID	2 bytes	Algorithm ID used for signing
IDSize	2 bytes	Size of Certificate ID data
CertID	(variable)	Certificate ID data
SigSize	2 bytes	Size of Signature data
Sig	(variable)	Signature data

CertID

Value	Size	Description
Size1	4 bytes	Size of CertID, should be (IDSize - 4)
Size1	4 bytes	A bug in version one causes this value to appear twice.
IssSize	4 bytes	Issuer data size
Issuer	(variable)	Issuer data
SerSize	4 bytes	Serial Number size
Serial	(variable)	Serial Number data

The Issuer and IssSize members are suitable for creating a CRYPT_DATA_BLOB to be the Issuer member of a CERT_INFO struct. The Serial and SerSize members would be the SerialNumber member of the same CERT_INFO struct. This struct would be used to find the certificate in the store the file was signed with. Those structures are from the MS CryptoAPI.

Sig and SigSize are the actual signature data and size generated by signing the file with the MS CryptoAPI using a hash created with the given AlgID.

-X.509 Certificate ID and Signature for central directory

This field contains the information about which certificate in the PKCS#7 Store was used to sign the central directory. It should only appear with the first central directory record, along

with the store. The data structure is the same as the CID, except that SigSize will be 0, and there will be no Sig member.

This field is also kept after the last central directory record, as the signature data (ID 0x05054b50, it looks like a central directory record of a different type). This second copy of the data is the Signature Data member of the record, and will have a SigSize that is non-zero, and will have Sig data.

Note: all fields stored in Intel low-byte/high-byte order.

Value	Size	Description
0x0016	2 bytes	Tag for this "extra" block type
CSize	2 bytes	Size of Method
Method	(variable)	

- FWKCS MD5 Extra Field:

The FWKCS Contents_Signature System, used in automatically identifying files independent of filename, optionally adds and uses an extra field to support the rapid creation of an enhanced contents_signature:

Header ID = 0x4b46
Data Size = 0x0013
Preface = 'M','D','5'

followed by 16 bytes containing the uncompressed file's 128_bit MD5 hash[(1)], low byte first.

When FWKCS revises a zipfile central directory to add this extra field for a file, it also replaces the central directory entry for that file's uncompressed filelength with a measured value.

FWKCS provides an option to strip this extra field, if present, from a zipfile central directory. In adding this extra field, FWKCS preserves Zipfile Authenticity Verification; if stripping this extra field, FWKCS preserves all versions of AV through PKZIP version 2.04g.

FWKCS, and FWKCS Contents_Signature System, are trademarks of Frederick W. Kantor.

(1) R. Rivest, RFC1321.TXT, MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992. ll.76-77: "The MD5 algorithm is being placed in the public domain for review and possible adoption as a standard."

file comment: (Variable)

The comment for this file.

number of this disk: (2 bytes)

The number of this disk, which contains central directory end record.

number of the disk with the start of the central directory: (2 bytes)

The number of the disk on which the central directory starts.

total number of entries in the central dir on this disk: (2 bytes)

The number of central directory entries on this disk.

total number of entries in the central dir: (2 bytes)

The total number of files in the zipfile.

size of the central directory: (4 bytes)

The size (in bytes) of the entire central directory.

offset of start of central directory with respect to the starting disk number: (4 bytes)

Offset of the start of the central directory on the disk on which the central directory starts.

.ZIP file comment length: (2 bytes)

The length of the comment for this .ZIP file.

.ZIP file comment: (Variable)

The comment for this .ZIP file.

F. General notes:

1. All fields unless otherwise noted are unsigned and stored in Intel low-byte:high-byte, low-word:high-word order.

2. String fields are not null terminated, since the length is given explicitly.

3. Local headers should not span disk boundaries. Also, even though the central directory can span disk boundaries, no single record in the central directory should be split across disks.

4. The entries in the central directory may not necessarily be in the same order that files appear in the .ZIP file.

5. Spanned/Split archives created using PKZIP for Windows (V2.50 or greater), PKZIP Command Line (V2.50 or greater), or PKZIP Explorer will include a special spanning signature as the first 4 bytes of the first segment of the archive. This signature (0x08074b50) will be followed immediately by the local header signature for the first file in the archive. Spanned archives created with this special signature are compatible with all versions of PKZIP from PKWARE. Split archives can

only be uncompressed by other versions of PKZIP that know how to create a split archive.

III. UnShrinking - Method 1

Shrinking is a Dynamic Ziv-Lempel-Welch compression algorithm with partial clearing. The initial code size is 9 bits, and the maximum code size is 13 bits. Shrinking differs from conventional Dynamic Ziv-Lempel-Welch implementations in several respects:

a. The code size is controlled by the compressor, and is not automatically increased when codes larger than the current code size are created (but not necessarily used). When the decompressor encounters the code sequence 256 (decimal) followed by 1, it should increase the code size read from the input stream to the next bit size. No blocking of the codes is performed, so the next code at the increased size should be read from the input stream immediately after where the previous code at the smaller bit size was read. Again, the decompressor should not increase the code size used until the sequence 256,1 is encountered.

b. When the table becomes full, total clearing is not performed. Rather, when the compressor emits the code sequence 256,2 (decimal), the decompressor should clear all leaf nodes from the Ziv-Lempel tree, and continue to use the current code size. The nodes that are cleared from the Ziv-Lempel tree are then re-used, with the lowest code value re-used first, and the highest code value re-used last. The compressor can emit the sequence 256,2 at any time.

IV. Expanding - Methods 2-5

The Reducing algorithm is actually a combination of two distinct algorithms. The first algorithm compresses repeated byte sequences, and the second algorithm takes the compressed stream from the first algorithm and applies a probabilistic compression method.

The probabilistic compression stores an array of 'follower sets' S(j), for j=0 to 255, corresponding to each possible ASCII character. Each set contains between 0 and 32 characters, to be denoted as S(j)[0],...,S(j)[m], where m<32. The sets are stored at the beginning of the data area for a Reduced file, in reverse order, with S(255) first, and S(0) last.

The sets are encoded as { N(j), S(j)[0],...,S(j)[N(j)-1] }, where N(j) is the size of set S(j). N(j) can be 0, in which case the follower set for S(j) is empty. Each N(j) value is encoded in 6 bits, followed by N(j) eight bit character values corresponding to S(j)[0] to S(j)[N(j)-1] respectively. If N(j) is 0, then no values for S(j) are stored, and the value for N(j-1) immediately follows.

Immediately after the follower sets, is the compressed data stream. The compressed data stream can be interpreted for the probabilistic decompression as follows:

let Last-Character <- 0.

```
loop until done
   if the follower set S(Last-Character) is empty then
      read 8 bits from the input stream, and copy this
        value to the output stream.
   otherwise if the follower set S(Last-Character) is non-empty then
      read 1 bit from the input stream.
      if this bit is not zero then
         read 8 bits from the input stream, and copy this
           value to the output stream.
      otherwise if this bit is zero then
         read B(N(Last-Character)) bits from the input
           stream, and assign this value to I.
         Copy the value of S(Last-Character)[I] to the output stream.
   assign the last value placed on the output stream to
     Last-Character.
end loop
```

B(N(j)) is defined as the minimal number of bits required to encode the value N(j)-1.

The decompressed stream from above can then be expanded to re-create the original file as follows:

```
let State <- 0.

loop until done
   read 8 bits from the input stream into C.
   case State of
      0: if C is not equal to DLE (144 decimal) then
            copy C to the output stream.
         otherwise if C is equal to DLE then
            let State <- 1.

      1: if C is non-zero then
            let V <- C.
            let Len <- L(V)
            let State <- F(Len).
         otherwise if C is zero then
            copy the value 144 (decimal) to the output stream.
            let State <- 0

      2: let Len <- Len + C
         let State <- 3.

      3: move backwards D(V,C) bytes in the output stream
         (if this position is before the start of the output
          stream, then assume that all the data before the
          start of the output stream is filled with zeros).
         copy Len+3 bytes from this position to the output stream.
         let State <- 0.
```

```
   end case
end loop
```

The functions F,L, and D are dependent on the 'compression factor', 1 through 4, and are defined as follows:

For compression factor 1:

L(X) equals the lower 7 bits of X.
F(X) equals 2 if X equals 127 otherwise F(X) equals 3.
D(X,Y) equals the (upper 1 bit of X) * 256 + Y + 1.

For compression factor 2:

L(X) equals the lower 6 bits of X.
F(X) equals 2 if X equals 63 otherwise F(X) equals 3.
D(X,Y) equals the (upper 2 bits of X) * 256 + Y + 1.

For compression factor 3:

L(X) equals the lower 5 bits of X.
F(X) equals 2 if X equals 31 otherwise F(X) equals 3.
D(X,Y) equals the (upper 3 bits of X) * 256 + Y + 1.

For compression factor 4:

L(X) equals the lower 4 bits of X.
F(X) equals 2 if X equals 15 otherwise F(X) equals 3.
D(X,Y) equals the (upper 4 bits of X) * 256 + Y + 1.

V. Imploding - Method 6

The Imploding algorithm is actually a combination of two distinct algorithms. The first algorithm compresses repeated byte sequences using a sliding dictionary. The second algorithm is used to compress the encoding of the sliding dictionary output, using multiple Shannon-Fano trees.

The Imploding algorithm can use a 4K or 8K sliding dictionary size. The dictionary size used can be determined by bit 1 in the general purpose flag word; a 0 bit indicates a 4K dictionary while a 1 bit indicates an 8K dictionary.

The Shannon-Fano trees are stored at the start of the compressed file. The number of trees stored is defined by bit 2 in the general purpose flag word; a 0 bit indicates two trees stored, a 1 bit indicates three trees are stored. If 3 trees are stored, the first Shannon-Fano tree represents the encoding of the Literal characters, the second tree represents the encoding of the Length information, the third represents the encoding of the Distance information. When 2 Shannon-Fano trees are stored, the Length tree is stored first, followed by the Distance tree.

The Literal Shannon-Fano tree, if present is used to represent the entire ASCII character set, and contains 256 values. This tree is used to compress any data not compressed by the sliding dictionary algorithm. When this tree is present, the Minimum Match Length for the sliding dictionary is 3. If this tree is not present, the Minimum Match Length is 2.

The Length Shannon-Fano tree is used to compress the Length part of the (length,distance) pairs from the sliding dictionary output. The Length tree contains 64 values, ranging from the Minimum Match Length, to 63 plus the Minimum Match Length.

The Distance Shannon-Fano tree is used to compress the Distance part of the (length,distance) pairs from the sliding dictionary output. The Distance tree contains 64 values, ranging from 0 to 63, representing the upper 6 bits of the distance value. The distance values themselves will be between 0 and the sliding dictionary size, either 4K or 8K.

The Shannon-Fano trees themselves are stored in a compressed format. The first byte of the tree data represents the number of bytes of data representing the (compressed) Shannon-Fano tree minus 1. The remaining bytes represent the Shannon-Fano tree data encoded as:

High 4 bits: Number of values at this bit length + 1. (1 - 16)
Low 4 bits: Bit Length needed to represent value + 1. (1 - 16)

The Shannon-Fano codes can be constructed from the bit lengths using the following algorithm:

a. Sort the Bit Lengths in ascending order, while retaining the order of the original lengths stored in the file.

b. Generate the Shannon-Fano trees:

```
c.  Code <- 0
d.  CodeIncrement <- 0
e.  LastBitLength <- 0
f.  i <- number of Shannon-Fano codes - 1 (either 255 or 63)
g.
h.  loop while i >= 0
i.      Code = Code + CodeIncrement
j.      if BitLength(i) <> LastBitLength then
k.          LastBitLength=BitLength(i)
l.          CodeIncrement = 1 shifted left (16 - LastBitLength)
m.      ShannonCode(i) = Code
n.      i <- i - 1
        end loop
```

o. Reverse the order of all the bits in the above ShannonCode() vector, so that the most significant bit becomes the least significant bit. For example, the value 0x1234 (hex) would become 0x2C48 (hex).

p. Restore the order of Shannon-Fano codes as originally stored within the file.

Example:

This example will show the encoding of a Shannon-Fano tree of size 8. Notice that the actual Shannon-Fano trees used for Imploding are either 64 or 256 entries in size.

Example: 0x02, 0x42, 0x01, 0x13

The first byte indicates 3 values in this table. Decoding the bytes:

0x42 = 5 codes of 3 bits long
0x01 = 1 code of 2 bits long
0x13 = 2 codes of 4 bits long

This would generate the original bit length array of: (3, 3, 3, 3, 3, 2, 4, 4)

There are 8 codes in this table for the values 0 thru 7. Using the algorithm to obtain the Shannon-Fano codes produces:

Val	Sorted	Constructed Code	Reversed Value	Order Restored	Original Length
0:	2	1100000000000000	11	101	3
1:	3	1010000000000000	101	001	3
2:	3	1000000000000000	001	110	3
3:	3	0110000000000000	110	010	3
4:	3	0100000000000000	010	100	3
5:	3	0010000000000000	100	11	2
6:	4	0001000000000000	1000	1000	4
7:	4	0000000000000000	0000	0000	4

The values in the Val, Order Restored and Original Length columns now represent the Shannon-Fano encoding tree that can be used for decoding the Shannon-Fano encoded data. How to parse the variable length Shannon-Fano values from the data stream is beyond the scope of this document. (See the references listed at the end of this document for more information.) However, traditional decoding schemes used for Huffman variable length decoding, such as the Greenlaw algorithm, can be successfully applied.

The compressed data stream begins immediately after the compressed Shannon-Fano data. The compressed data stream can be interpreted as follows:

```
loop until done
   read 1 bit from input stream.

   if this bit is non-zero then (encoded data is literal data)
      if Literal Shannon-Fano tree is present
```

```
        read and decode character using Literal Shannon-Fano tree.
      otherwise
        read 8 bits from input stream.
      copy character to the output stream.
    otherwise (encoded data is sliding dictionary match)
      if 8K dictionary size
        read 7 bits for offset Distance (lower 7 bits of offset).
      otherwise
        read 6 bits for offset Distance (lower 6 bits of offset).

      using the Distance Shannon-Fano tree, read and decode the
       upper 6 bits of the Distance value.

      using the Length Shannon-Fano tree, read and decode
       the Length value.

      Length <- Length + Minimum Match Length

      if Length = 63 + Minimum Match Length
        read 8 bits from the input stream,
        add this value to Length.

      move backwards Distance+1 bytes in the output stream, and
       copy Length characters from this position to the output
       stream. (if this position is before the start of the output
       stream, then assume that all the data before the start of
       the output stream is filled with zeros).
end loop
```

VI. Tokenizing - Method 7

This method is not used by PKZIP.

VII. Deflating - Method 8

The Deflate algorithm is similar to the Implode algorithm using a sliding dictionary of up to 32K with secondary compression from Huffman/Shannon-Fano codes.

The compressed data is stored in blocks with a header describing the block and the Huffman codes used in the data block. The header format is as follows:

Bit 0: Last Block bit This bit is set to 1 if this is the last compressed block in the data.

Bits 1-2: Block type

00 (0) - Block is stored - All stored data is byte aligned. Skip bits until next byte, then next word = block length, followed by the ones compliment of the block length word. Remaining data in block is the stored data.

01 (1) - Use fixed Huffman codes for literal and distance codes.

Lit Code	Bits	Dist Code	Bits
0 - 143	8	0 - 31	5

144 - 255 9
256 - 279 7
280 - 287 8

Literal codes 286-287 and distance codes 30-31 are never used but participate in the huffman construction.

10 (2) - Dynamic Huffman codes. (See expanding Huffman codes)
11 (3) - Reserved - Flag a "Error in compressed data" if seen.

Expanding Huffman Codes

If the data block is stored with dynamic Huffman codes, the Huffman codes are sent in the following compressed format:

5 Bits: # of Literal codes sent - 256 (256 - 286)
All other codes are never sent
5 Bits: # of Dist codes - 1 (1 - 32)
4 Bits: # of Bit Length codes - 3 (3 - 19)

The Huffman codes are sent as bit lengths and the codes are built as described in the implode algorithm. The bit lengths themselves are compressed with Huffman codes. There are 19 bit length codes:

0 - 15: Represent bit lengths of 0 - 15
16: Copy the previous bit length 3 - 6 times.
The next 2 bits indicate repeat length (0 = 3, ... ,3 = 6)
Example: Codes 8, 16 (+2 bits 11), 16 (+2 bits 10) will expand to 12 bit lengths of 8 (1 + 6 + 5)
17: Repeat a bit length of 0 for 3 - 10 times. (3 bits of length)
18: Repeat a bit length of 0 for 11 - 138 times (7 bits of length)

The lengths of the bit length codes are sent packed 3 bits per value (0 - 7) in the following order:

16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15

The Huffman codes should be built as described in the Implode algorithm except codes are assigned starting at the shortest bit length, i.e. the shortest code should be all 0's rather than all 1's. Also, codes with a bit length of zero do not participate in the tree construction. The codes are then used to decode the bit lengths for the literal and distance tables.

The bit lengths for the literal tables are sent first with the number of entries sent described by the 5 bits sent earlier. There are up to 286 literal characters; the first 256 represent the respective 8 bit character, code 256 represents the End-Of-Block code, the remaining 29 codes represent copy lengths of 3 thru 258. There are up to 30 distance codes representing distances from 1 thru 32k as described below.

Length Codes

Code	Extra	Length	Code	Extra	Lengths	Code	Extra	Lengths	Code	Extra	Length(s)

	Bits			Bits			Bits			Bits	
257	0	3	265	1	11,12	273	3	35-42	281	5	131-162
258	0	4	266	1	13,14	274	3	43-50	282	5	163-194
259	0	5	267	1	15,16	275	3	51-58	283	5	195-226
260	0	6	268	1	17,18	276	3	59-66	284	5	227-257
261	0	7	269	2	19-22	277	4	67-82	285	0	258
262	0	8	270	2	23-26	278	4	83-98			
263	0	9	271	2	27-30	279	4	99-114			
264	0	10	272	2	31-34	280	4	115-130			

Distance Codes

Code	Extra Bits	Distance	Code	Extra Bits	Distance	Code	Extra Bits	Distance	Code	Extra Bits	Distance
0	0	1	8	3	17-24	16	7	257-384	24	11	4097-6144
1	0	2	9	3	25-32	17	7	385-512	25	11	6145-8192
2	0	3	10	4	33-48	18	8	513-768	26	12	8193-12288
3	0	4	11	4	49-64	19	8	769-1024	27	12	12289-16384
4	1	5,6	12	5	65-96	20	9	1025-1536	28	13	16385-24576
5	1	7,8	13	5	97-128	21	9	1537-2048	29	13	24577-32768
6	2	9-12	14	6	129-192	22	10	2049-3072			
7	2	13-16	15	6	193-256	23	10	3073-4096			

The compressed data stream begins immediately after the compressed header data. The compressed data stream can be interpreted as follows:

```
do
   read header from input stream.

   if stored block
      skip bits until byte aligned
      read count and 1's compliment of count
      copy count bytes data block
   otherwise
      loop until end of block code sent
         decode literal character from input stream
         if literal < 256
            copy character to the output stream
         otherwise
            if literal = end of block
               break from loop
            otherwise
               decode distance from input stream

               move backwards distance bytes in the output stream, and
                 copy length characters from this position to the
                 output stream.
      end loop
```

```
while not last block

if data descriptor exists
   skip bits until byte aligned
   read crc and sizes
endif
```

VIII. Decryption

The encryption used in PKZIP was generously supplied by Roger Schlafly. PKWARE is grateful to Mr. Schlafly for his expert help and advice in the field of data encryption.

PKZIP encrypts the compressed data stream. Encrypted files must be decrypted before they can be extracted.

Each encrypted file has an extra 12 bytes stored at the start of the data area defining the encryption header for that file. The encryption header is originally set to random values, and then itself encrypted, using three, 32-bit keys. The key values are initialized using the supplied encryption password. After each byte is encrypted, the keys are then updated using pseudo-random number generation techniques in combination with the same CRC-32 algorithm used in PKZIP and described elsewhere in this document.

The following is the basic steps required to decrypt a file:

a. Initialize the three 32-bit keys with the password.

b. Read and decrypt the 12-byte encryption header, further initializing the encryption keys.

c. Read and decrypt the compressed data stream using the encryption keys.

Step 1 - Initializing the encryption keys

```
Key(0) <- 305419896
Key(1) <- 591751049
Key(2) <- 878082192

loop for i <- 0 to length(password)-1
   update_keys(password(i))
end loop
```

Where update_keys() is defined as:

```
update_keys(char):
   Key(0) <- crc32(key(0),char)
   Key(1) <- Key(1) + (Key(0) & 000000ffH)
   Key(1) <- Key(1) * 134775813 + 1
   Key(2) <- crc32(key(2),key(1) >> 24)
end update_keys
```

Where crc32(old_crc,char) is a routine that given a CRC value and a character, returns an updated CRC value after applying the CRC-32 algorithm described elsewhere in this document.

Step 2 - Decrypting the encryption header

The purpose of this step is to further initialize the encryption keys, based on random data, to render a plaintext attack on the data ineffective.

Read the 12-byte encryption header into Buffer, in locations
  Buffer(0) thru Buffer(11).

```
loop for i <- 0 to 11
   C <- buffer(i) ^ decrypt_byte()
   update_keys(C)
   buffer(i) <- C
end loop
```

Where decrypt_byte() is defined as:

```
unsigned char decrypt_byte()
   local unsigned short temp
   temp <- Key(2) | 2
   decrypt_byte <- (temp * (temp ^ 1)) >> 8
end decrypt_byte
```

After the header is decrypted, the last 1 or 2 bytes in Buffer should be the high-order word/byte of the CRC for the file being decrypted, stored in Intel low-byte/high-byte order. Versions of PKZIP prior to 2.0 used a 2 byte CRC check; a 1 byte CRC check is used on versions after 2.0. This can be used to test if the password supplied is correct or not.

Step 3 - Decrypting the compressed data stream

The compressed data stream can be decrypted as follows:

```
loop until done
   read a character into C
   Temp <- C ^ decrypt_byte()
   update_keys(temp)
   output Temp
end loop
```

In addition to the above mentioned contributors to PKZIP and PKUNZIP, I would like to extend special thanks to Robert Mahoney for suggesting the extension .ZIP for this software.

References:

Fiala, Edward R., and Greene, Daniel H., "Data compression with finite windows", Communications of the ACM, Volume 32, Number 4, April 1989, pages 490-505.

Held, Gilbert, "Data Compression, Techniques and Applications, Hardware and Software Considerations", John Wiley & Sons, 1987.

Huffman, D.A., "A method for the construction of minimum-redundancy codes", Proceedings of the IRE, Volume 40, Number 9, September 1952, pages 1098-1101.

Nelson, Mark, "LZW Data Compression", Dr. Dobbs Journal, Volume 14, Number 10, October 1989, pages 29-37.

Nelson, Mark, "The Data Compression Book", M&T Books, 1991.

Storer, James A., "Data Compression, Methods and Theory", Computer Science Press, 1988

Welch, Terry, "A Technique for High-Performance Data Compression", IEEE Computer, Volume 17, Number 6, June 1984, pages 8-19.

Ziv, J. and Lempel, A., "A universal algorithm for sequential data compression", Communications of the ACM, Volume 30, Number 6, June 1987, pages 520-540.

Ziv, J. and Lempel, A., "Compression of individual sequences via variable-rate coding", IEEE Transactions on Information Theory, Volume 24, Number 5, September 1978, pages 530-536.

72.20 ZOO

The ZOO archive program by Raoul Dhesi is a file compression program now superceeded in both compression and speed by most other compression programs. The archive header looks like this :

OFFSET	Count	TYPE	Description
0000h	20	char	Archive header text, ^Z terminated, null padded
0014h	1	dword	ID=0FDC4A7DCh
0018h	1	dword	Offset of first file in archive
001Ch	1	dword	Offset of ????
0020h	1	byte	Version archive was made by
0021h	1	byte	Minimum version needed to extract

Each stored file has its own header, which looks like this :

OFFSET	Count	TYPE	Description
0000h	1	dword	ID=0FDC4A7DCh
0004h	1	byte	Type of directory entry
0005h	1	byte	Compression method : 0 - stored 1 - Crunched : LZW, 4K buffer, var len (9-13 bits)
0006h	1	dword	Offset of next directory entry
000Ah	1	dword	Offset of next header
000Dh	1	word	Original date / time of file

OFFSET	Count	TYPE	Description
0012h	1	word	CRC-16 of file
0014h	1	dword	Uncompressed size of file
0018h	1	dword	Compressed size of file
001Ch	1	byte	Version this file was compressed by
001Dh	1	byte	Minimum version needed to extract
001Eh	1	byte	Deleted flag 0 - file in archive 1 - file is considered deleted
001Fh	1	dword	Offset of comment field, 0 if none
0023h	1	word	Length of comment field
0025h	?	char	ASCIIZ path / filename