The following is a copy of the file "APPNOTE.TXT" from the file PKZ193A.EXE. This file describes the .ZIP file format and the archiving/dearchiving algorithms used by PKZIP and PKUNZIP.

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.

General Format of a ZIP file

Files stored in arbitrary order. Large zipfiles can span multiple diskette media.

Overall zipfile format:

[local file header+file data] . . .
[central directory] end of central directory record

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. Central directory structure:

[file header] . . . end of central dir 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

zipfile comment length

2 bytes

zipfile comment (variable size)

C. Explanation of fields:

version made by

The upper byte indicates the host system (OS) for the file. 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 (F.A.T. file systems)

1 - Amiga

2 - VMS

3 - *nix

4 - VM/CMS

5 - Atari ST 6 - OS/2 H.P.F.S.

7 - Macintosh

8 - Z-System

9 - CP/M

10 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

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

general purpose bit flag:

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 Method 8)
bit 1: If this bit is set, the file was compressed with
the maximum compression version of the type 8
compression algorithm. (-ex option)
bit 2: If this bit is set, the file was compressed with
the fastest version of the type 8 compression
algorithm. (-es option)

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

The upper three bits are reserved and used internally

by the software when processing the zipfile. The

remaining bits are unused in version 1.0.

compression method:

(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 alogrithm
8 - The algorithm used by the latest version of
PKZIP/PKUNZIP.

date and time fields:

The date and time are encoded in standard MS-DOS

format.

CRC-32:

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.

compressed size:
uncompressed size:

The size of the file compressed and uncompressed,

respectively.

filename length:
extra field length:
file comment length:

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.

disk number start:

The number of the disk on which this file begins.

internal file attributes:

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:

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.

relative offset of local header:

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:

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.

extra field:

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 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 it's unique signature is present, in addition to the Header ID value being correct, before assuming that it is a block of known type.

file comment:

The comment for this file.

number of this disk:

The number of this disk, which contains central

directory end record.

number of the disk with the start of the central directory:

The number of the disk on which the central

directory starts.

total number of entries in the central dir on this disk:

The number of central directory entries on this disk.

total number of entries in the central dir:

The total number of files in the zipfile.

size of the central directory:

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

offset of start of central directory with respect to
the starting disk number:

Offset of the start of the central direcory on the

disk on which the central directory starts.

zipfile comment length:

The length of the comment for this zipfile.

zipfile comment:

The comment for this zipfile.

D. 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 boundries. Also, even though the central directory can span disk boundries, 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 zipfile.

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:

1) 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.

2) When the table becomes full, total clearing is not performed. Rather, when the compresser 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.

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.
```

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 ouput, 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:

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

2) Generate the Shannon-Fano trees:

```
Code <- 0
CodeIncrement <- 0
LastBitLength <- 0
i <- number of Shannon-Fano codes - 1     (either 255 or 63)

loop while i >= 0

Code = Code + CodeIncrement

if BitLength(i) <> LastBitLength then

    LastBitLength=BitLength(i)
```

CodeIncrement = 1 shifted left (16 - LastBitLength)

ShannonCode(i) = Code

i <- i - 1
end loop

3) 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).

4) 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 succesfully 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

Tokenizing - Method 7

Not currently supported by PKZIP or PKUNZIP.

?????? - Method 8

The ?????? algorithm is similiar to the Implode algorithm using
a sliding dictionary of up to 32K with secondary compression
from dynamic 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 1: Last Block bit This bit is set to 1 if this is the last
compressed block in the data.

Bits 2-3: 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		

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 + 255 (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)
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 alogrithm 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. These 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 288 literal characters, the first 256 represent the respective 8 bit character, code 257 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 Bits	Length	Code	Extra Bits	Lengths	Code	Extra Bits	Lengths	Code	Extra Bits	Length(s)
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	Dist	Code	Extra Bits	Dist	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
```

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 3, 32-bit keys. The key values are initialized using the supplied encryption password. After each byte is encrypted, the keys are then updated using psuedo-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:

1) Initialize the three 32-bit keys with the password.
2) Read and decrypt the 12-byte encryption header, further initializing the encryption keys.
3) 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 two bytes in Buffer should be the high-order word of the CRC for the file being decrypted, stored in Intel low-byte/high-byte order. 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 charcter 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:

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

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