

File: APPNOTE.TXT - .ZIP File Format Specification
Version: 6.3.2
Revised: September 28, 2007
Copyright (c) 1989 - 2007 PKWARE Inc., All Rights Reserved.

The use of certain technological aspects disclosed in the current APPNOTE is available pursuant to the below section entitled "Incorporating PKWARE Proprietary Technology into Your Product".

I. Purpose

This specification is intended to define a cross-platform, interoperable file storage and transfer format. Since its first publication in 1989, PKWARE has remained committed to ensuring the interoperability of the .ZIP file format through publication and maintenance of this specification. We trust that all .ZIP compatible vendors and application developers that have adopted and benefited from this format will share and support this commitment to interoperability.

II. Contacting PKWARE

PKWARE, Inc.
648 N. Plankinton Avenue, Suite 220
Milwaukee, WI 53203
+1-414-289-9788
+1-414-289-9789 FAX
zipformat@pkware.com

III. 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 or omission cannot 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.

If the version of this file is marked as a NOTIFICATION OF CHANGE, the content defines an Early Feature Specification (EFS) change to the .ZIP file format that may be subject to modification prior to publication of the Final Feature Specification (FFS). This

document may also contain information on Planned Feature Specifications (PFS) defining recognized future extensions.

IV. Change Log

Version	Change Description	Date
-----	-----	-----
5.2	-Single Password Symmetric Encryption storage	06/02/2003
6.1.0	-Smartcard compatibility -Documentation on certificate storage	01/20/2004
6.2.0	-Introduction of Central Directory Encryption for encrypting metadata -Added OS/X to Version Made By values	04/26/2004
6.2.1	-Added Extra Field placeholder for POSZIP using ID 0x4690 -Clarified size field on "zip64 end of central directory record"	04/01/2005
6.2.2	-Documented Final Feature Specification for Strong Encryption -Clarifications and typographical corrections	01/06/2006
6.3.0	-Added tape positioning storage parameters -Expanded list of supported hash algorithms -Expanded list of supported compression algorithms -Expanded list of supported encryption algorithms -Added option for Unicode filename storage -Clarifications for consistent use of Data Descriptor records -Added additional "Extra Field" definitions	09/29/2006
6.3.1	-Corrected standard hash values for SHA-256/384/512	04/11/2007

-Documented InfoZIP "Extra Field"
values for UTF-8 file name and
file comment storage

V. General Format of a .ZIP file

Files stored in arbitrary order. Large .ZIP files can span multiple volumes or be split into user-defined segment sizes. All values are stored in little-endian byte order unless otherwise specified.

Overall .ZIP file format:

```
[local file header 1]
[file data 1]
[data descriptor 1]
.
.
.
[local file header n]
[file data n]
[data descriptor n]
[archive decryption header]
[archive extra data record]
[central directory]
[zip64 end of central directory record]
[zip64 end of central directory locator]
[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	
file name length	2 bytes	
extra field length	2 bytes	
file name (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. For ZIP64(tm) format archives, the compressed and uncompressed sizes are 8 bytes each.

When compressing files, compressed and uncompressed sizes should be stored in ZIP64 format (as 8 byte values) when a files size exceeds 0xFFFFFFFF. However ZIP64 format may be used regardless of the size of a file. When extracting, if the zip64 extended information extra field is present for the file the compressed and uncompressed sizes will be 8 byte values.

Although not originally assigned a signature, the value 0x08074b50 has commonly been adopted as a signature value for the data descriptor record. Implementers should be aware that ZIP files may be encountered with or without this signature marking data descriptors and should account for either case when reading ZIP files to ensure compatibility. When writing ZIP files, it is recommended to include the signature value marking the data descriptor record. When the signature is used, the fields currently defined for the data descriptor record will immediately follow the signature.

An extensible data descriptor will be released in a future version of this APPNOTE. This new record is intended to resolve conflicts with the use of this record going forward, and to provide better support for streamed file processing.

When the Central Directory Encryption method is used, the data descriptor record is not required, but may be used. If present, and bit 3 of the general purpose bit field is set to indicate its presence, the values in fields of the data descriptor record should be set to binary zeros.

D. Archive decryption header:

The Archive Decryption Header is introduced in version 6.2 of the ZIP format specification. This record exists in support of the Central Directory Encryption Feature implemented as part of

the Strong Encryption Specification as described in this document. When the Central Directory Structure is encrypted, this decryption header will precede the encrypted data segment. The encrypted data segment will consist of the Archive extra data record (if present) and the encrypted Central Directory Structure data. The format of this data record is identical to the Decryption header record preceding compressed file data. If the central directory structure is encrypted, the location of the start of this data record is determined using the Start of Central Directory field in the Zip64 End of Central Directory record. Refer to the section on the Strong Encryption Specification for information on the fields used in the Archive Decryption Header record.

E. Archive extra data record:

archive extra data signature	4 bytes	(0x08064b50)
extra field length	4 bytes	
extra field data	(variable size)	

The Archive Extra Data Record is introduced in version 6.2 of the ZIP format specification. This record exists in support of the Central Directory Encryption Feature implemented as part of the Strong Encryption Specification as described in this document. When present, this record immediately precedes the central directory data structure. The size of this data record will be included in the Size of the Central Directory field in the End of Central Directory record. If the central directory structure is compressed, but not encrypted, the location of the start of this data record is determined using the Start of Central Directory field in the Zip64 End of Central Directory record.

F. Central directory structure:

```
[file header 1]
.
.
.
[file header n]
[digital signature]
```

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
file name 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
file name (variable size)	
extra field (variable size)	
file comment (variable size)	

Digital signature:

header signature	4 bytes	(0x05054b50)
size of data	2 bytes	
signature data (variable size)		

With the introduction of the Central Directory Encryption feature in version 6.2 of this specification, the Central Directory Structure may be stored both compressed and encrypted. Although not required, it is assumed when encrypting the Central Directory Structure, that it will be compressed for greater storage efficiency. Information on the Central Directory Encryption feature can be found in the section describing the Strong Encryption Specification. The Digital Signature record will be neither compressed nor encrypted.

G. Zip64 end of central directory record

zip64 end of central dir signature	4 bytes	(0x06064b50)
size of zip64 end of central directory record	8 bytes	
version made by	2 bytes	
version needed to extract	2 bytes	
number of this disk	4 bytes	
number of the disk with the start of the central directory	4 bytes	
total number of entries in the central directory on this disk	8 bytes	
total number of entries in the central directory	8 bytes	
size of the central directory	8 bytes	
offset of start of central directory with respect to the starting disk number	8 bytes	
zip64 extensible data sector	(variable size)	

The value stored into the "size of zip64 end of central directory record" should be the size of the remaining record and should not include the leading 12 bytes.

Size = SizeOfFixedFields + SizeOfVariableData - 12.

The above record structure defines Version 1 of the zip64 end of central directory record. Version 1 was implemented in versions of this specification preceding 6.2 in support of the ZIP64 large file feature. The introduction of the Central Directory Encryption feature implemented in version 6.2 as part of the Strong Encryption Specification defines Version 2 of this record structure. Refer to the section describing the Strong Encryption Specification for details on the version 2 format for this record.

Special purpose data may reside in the zip64 extensible data sector field following either a V1 or V2 version of this record. To ensure identification of this special purpose data it must include an identifying header block consisting of the following:

Header ID - 2 bytes
Data Size - 4 bytes

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

Data Size identifies the number of bytes that follow for this data block type.

Multiple special purpose data blocks may be present, but each must be preceded by a Header ID and Data Size field. Current mappings of Header ID values supported in this field are as defined in APPENDIX C.

H. Zip64 end of central directory locator

zip64 end of central dir locator signature	4 bytes	(0x07064b50)
number of the disk with the start of the zip64 end of central directory	4 bytes	
relative offset of the zip64 end of central directory record	8 bytes	
total number of disks	4 bytes	

I. End of central directory 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 directory on this disk	2 bytes	

total number of entries in
the central directory 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)

J. 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 - MVS (OS/390 - Z/OS) | 12 - VSE |
| 13 - Acorn Risc | 14 - VFAT |
| 15 - alternate MVS | 16 - BeOS |
| 17 - Tandem | 18 - OS/400 |
| 19 - OS/X (Darwin) | 20 thru 255 - unused |

The lower byte indicates the ZIP specification version (the version of this document) supported by 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 supported ZIP specification version needed to extract the file, mapped as above. This value is based on the specific format features a ZIP program must support to be able to extract the file. If multiple features are applied to a file, the minimum version should be set to the feature having the highest value. New features or feature changes affecting the published format specification will be implemented using higher version numbers than the last published value to avoid conflict.

Current minimum feature versions are as defined below:

- 1.0 - Default value
- 1.1 - File is a volume label
- 2.0 - File is a folder (directory)
- 2.0 - File is compressed using Deflate compression
- 2.0 - File is encrypted using traditional PKWARE encryption
- 2.1 - File is compressed using Deflate64(tm)
- 2.5 - File is compressed using PKWARE DCL Implode
- 2.7 - File is a patch data set
- 4.5 - File uses ZIP64 format extensions
- 4.6 - File is compressed using BZIP2 compression*
- 5.0 - File is encrypted using DES
- 5.0 - File is encrypted using 3DES
- 5.0 - File is encrypted using original RC2 encryption
- 5.0 - File is encrypted using RC4 encryption
- 5.1 - File is encrypted using AES encryption
- 5.1 - File is encrypted using corrected RC2 encryption**
- 5.2 - File is encrypted using corrected RC2-64 encryption**
- 6.1 - File is encrypted using non-OAEP key wrapping***
- 6.2 - Central directory encryption
- 6.3 - File is compressed using LZMA
- 6.3 - File is compressed using PPMd+
- 6.3 - File is encrypted using Blowfish
- 6.3 - File is encrypted using Twofish

* Early 7.x (pre-7.2) versions of PKZIP incorrectly set the version needed to extract for BZIP2 compression to be 50 when it should have been 46.

** Refer to the section on Strong Encryption Specification for additional information regarding RC2 corrections.

*** Certificate encryption using non-OAEP key wrapping is the intended mode of operation for all versions beginning with 6.1. Support for OAEP key wrapping should only be used for backward compatibility when sending ZIP files to be opened by versions of PKZIP older than 6.1 (5.0 or 6.0).

+ Files compressed using PPMd should set the version needed to extract field to 6.3, however, not all ZIP programs enforce this and may be unable to decompress data files compressed using PPMd if this value is set.

When using ZIP64 extensions, the corresponding value in the zip64 end of central directory record should also be set. This field should be set appropriately to indicate whether Version 1 or Version 2 format is in use.

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 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. |

(For Method 14 - LZMA)

- Bit 1: If the compression method used was type 14, LZMA, then this bit, if set, indicates an end-of-stream (EOS) marker is used to mark the end of the compressed data stream. If clear, then an EOS marker is not present and the compressed data size must be known to extract.

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: Strong encryption. If this bit is set, you should set the version needed to extract value to at least 50 and you must also set bit 0. If AES encryption is used, the version needed to extract value must be at least 51.

Bit 7: Currently unused.

Bit 8: Currently unused.

Bit 9: Currently unused.

Bit 10: Currently unused.

Bit 11: Language encoding flag (EFS). If this bit is set, the filename and comment fields for this file must be encoded using UTF-8. (see APPENDIX D)

Bit 12: Reserved by PKWARE for enhanced compression.

Bit 13: Used when encrypting the Central Directory to indicate selected data values in the Local Header are masked to hide their actual values. See the section describing the Strong Encryption Specification for details.

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 Data Compression Library Imploding (old IBM TERSE)
- 11 - Reserved by PKWARE
- 12 - File is compressed using BZIP2 algorithm
- 13 - Reserved by PKWARE
- 14 - LZMA (EFS)
- 15 - Reserved by PKWARE
- 16 - Reserved by PKWARE
- 17 - Reserved by PKWARE
- 18 - File is compressed using IBM TERSE (new)
- 19 - IBM LZ77 z Architecture (PFS)
- 97 - WavPack compressed data
- 98 - PPMd version I, Rev 1

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. If encrypting the central directory and general purpose bit flag 13 is set indicating masking, the value stored in the Local Header will be zero.

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. When encrypting the central directory, if the local header is not in ZIP64 format and general purpose bit flag 13 is set indicating masking, the value stored in the Local Header will be zero.

compressed size: (4 bytes)

uncompressed size: (4 bytes)

The size of the file compressed and uncompressed, respectively. When a decryption header is present it will be placed in front of the file data and the value of the compressed file size will include the bytes of the decryption header. 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. If an archive is in ZIP64 format and the value in this field is 0xFFFFFFFF, the size will be in the corresponding 8 byte ZIP64 extended information extra field. When encrypting the central directory, if the local header is not in ZIP64 format and general purpose bit flag 13 is set indicating masking, the value stored for the uncompressed size in the Local Header will be zero.

file name length: (2 bytes)

extra field length: (2 bytes)

file comment length: (2 bytes)

The length of the file name, 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 file name length is set to zero.

disk number start: (2 bytes)

The number of the disk on which this file begins. If an archive is in ZIP64 format and the value in this field is 0xFFFF, the size will be in the corresponding 4 byte zip64 extended information extra field.

internal file attributes: (2 bytes)

Bits 1 and 2 are reserved for use by PKWARE.

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.

The 0x0002 bit of this field indicates, if set, that a 4 byte variable record length control field precedes each logical record indicating the length of the record. The record length control field is stored in little-endian byte order. This flag is independent of text control characters, and if used in conjunction with text data, includes any control characters in the total length of the record. This value is provided for mainframe data transfer support.

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. If an archive is in ZIP64 format and the value in this field is 0xFFFFFFFF, the size will be in the corresponding 8 byte zip64 extended information extra field.

file name: (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 file name field. If encrypting the central directory and general purpose bit flag 13 is set indicating masking, the file name stored in the Local Header

will not be the actual file name. A masking value consisting of a unique hexadecimal value will be stored. This value will be sequentially incremented for each file in the archive. See the section on the Strong Encryption Specification for details on retrieving the encrypted file name.

extra field: (Variable)

This is for expansion. If additional information needs to be stored for special needs or for specific platforms, 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:

0x0001	Zip64 extended information extra field
0x0007	AV Info
0x0008	Reserved for extended language encoding data (PFS) (see APPENDIX D)
0x0009	OS/2
0x000a	NTFS
0x000c	OpenVMS
0x000d	UNIX
0x000e	Reserved for file stream and fork descriptors
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
0x0017	Strong Encryption Header

0x0018	Record Management Controls
0x0019	PKCS#7 Encryption Recipient Certificate List
0x0065	IBM S/390 (Z390), AS/400 (I400) attributes - uncompressed
0x0066	Reserved for IBM S/390 (Z390), AS/400 (I400) attributes - compressed
0x4690	POSZIP 4690 (reserved)

Third party mappings commonly used are:

0x07c8	Macintosh
0x2605	ZipIt Macintosh
0x2705	ZipIt Macintosh 1.3.5+
0x2805	ZipIt Macintosh 1.3.5+
0x334d	Info-ZIP Macintosh
0x4341	Acorn/SparkFS
0x4453	Windows NT security descriptor (binary ACL)
0x4704	VM/CMS
0x470f	MVS
0x4b46	FWKCS MD5 (see below)
0x4c41	OS/2 access control list (text ACL)
0x4d49	Info-ZIP OpenVMS
0x4f4c	Xceed original location extra field
0x5356	AOS/VS (ACL)
0x5455	extended timestamp
0x554e	Xceed unicode extra field
0x5855	Info-ZIP UNIX (original, also OS/2, NT, etc)
0x6375	Info-ZIP Unicode Comment Extra Field
0x6542	BeOS/BeBox
0x7075	Info-ZIP Unicode Path Extra Field
0x756e	ASi UNIX
0x7855	Info-ZIP UNIX (new)
0xa220	Microsoft Open Packaging Growth Hint
0xfd4a	SMS/QDOS

Detailed descriptions of Extra Fields defined by third party mappings will be documented as information on these data structures is made available to PKWARE. PKWARE does not guarantee the accuracy of any published third party data.

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 file name, 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.

-Zip64 Extended Information Extra Field (0x0001):

The following is the layout of the zip64 extended information "extra" block. If one of the size or offset fields in the Local or Central directory record is too small to hold the required data, a Zip64 extended information record is created. The order of the fields in the zip64 extended information record is fixed, but the fields will only appear if the corresponding Local or Central directory record field is set to 0xFFFF or 0xFFFFFFFF.

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

	Value	Size	Description
	-----	----	-----
(ZIP64)	0x0001	2 bytes	Tag for this "extra" block type
	Size	2 bytes	Size of this "extra" block
	Original		
	Size	8 bytes	Original uncompressed file size
	Compressed		
	Size	8 bytes	Size of compressed data
	Relative Header		
	Offset	8 bytes	Offset of local header record
	Disk Start		
	Number	4 bytes	Number of the disk on which this file starts

This entry in the Local header must include BOTH original and compressed file size fields. If encrypting the central directory and bit 13 of the general purpose bit flag is set indicating masking, the value stored in the Local Header for the original file size will be zero.

-OS/2 Extra Field (0x0009):

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
	-----	----	-----
(OS/2)	0x0009	2 bytes	Tag for this "extra" block type
	TSize	2 bytes	Size for the following data block

BSize	4 bytes	Uncompressed Block Size
CType	2 bytes	Compression type
EACRC	4 bytes	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[].

-NTFS Extra Field (0x000a):

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.)

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

	Value	Size	Description
	-----	----	-----
(NTFS)	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 tag 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

-OpenVMS Extra Field (0x000c):

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
-----	----	-----

(VMS)	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	OpenVMS attribute tag value #1
	Size1	2 bytes	Size of attribute #1, in bytes
	(var.)	Size1	Attribute #1 data
	.		
	.		
	.		
	TagN	2 bytes	OpenVMS attribute tag 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.

-UNIX Extra Field (0x000d):

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
	-----	----	-----
(UNIX)	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, and the major and minor device node numbers for character and block device nodes. Since device nodes cannot be either symbolic or hard links, only one set of variable length data is stored. Link files will have the name of the original file stored. This name is NOT NULL terminated. Its size can be determined by checking TSize -

12. Device entries will have eight bytes stored as two 4 byte entries (in little endian format). The first entry will be the major device number, and the second the minor device number.

-PATCH Descriptor Extra Field (0x000f):

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
-----	----	-----
(Patch) 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 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 auto detection
1	Treat as a self-patch
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

Patch support is provided by PKPatchMaker(tm) technology and is covered under U.S. Patents and Patents Pending. The use or implementation in a product of certain technological aspects set forth in the current APPNOTE, including those with regard to strong encryption, patching, or extended tape operations requires a license from PKWARE. Please contact PKWARE with regard to acquiring a license.

-PKCS#7 Store for X.509 Certificates (0x0014):

This field contains information about each of the certificates files may be signed with. When the Central Directory Encryption feature is enabled for a ZIP file, this record will appear in the Archive Extra Data Record, otherwise it will 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
	-----	----	-----
(Store)	0x0014	2 bytes	Tag for this "extra" block type
	TSize	2 bytes	Size of the store data
	TData	TSize	Data about the store

-X.509 Certificate ID and Signature for individual file (0x0015):

This field contains the information about which certificate in the PKCS#7 store was used to sign a 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
	-----	----	-----
(CID)	0x0015	2 bytes	Tag for this "extra" block type
	TSize	2 bytes	Size of data that follows
	TData	TSize	Signature Data

-X.509 Certificate ID and Signature for central directory (0x0016):

This field contains the information about which certificate in the PKCS#7 store was used to sign the central directory structure. When the Central Directory Encryption feature is enabled for a ZIP file, this record will appear in the Archive Extra Data Record, otherwise it will appear in the first central directory record.

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

	Value	Size	Description
	-----	----	-----
(CDID)	0x0016	2 bytes	Tag for this "extra" block type
	TSize	2 bytes	Size of data that follows
	TData	TSize	Data

-Strong Encryption Header (0x0017):

	Value	Size	Description
	-----	----	-----
	0x0017	2 bytes	Tag for this "extra" block type
	TSize	2 bytes	Size of data that follows
	Format	2 bytes	Format definition for this record
	AlgID	2 bytes	Encryption algorithm identifier
	Bitlen	2 bytes	Bit length of encryption key
	Flags	2 bytes	Processing flags
	CertData	TSize-8	Certificate decryption extra field data (refer to the explanation for CertData in the section describing the Certificate Processing Method under the Strong Encryption Specification)

-Record Management Controls (0x0018):

	Value	Size	Description
	-----	----	-----
(Rec-CTL)	0x0018	2 bytes	Tag for this "extra" block type
	CSize	2 bytes	Size of total extra block data
	Tag1	2 bytes	Record control attribute 1
	Size1	2 bytes	Size of attribute 1, in bytes
	Data1	Size1	Attribute 1 data
	.		
	.		
	.		
	TagN	2 bytes	Record control attribute N
	SizeN	2 bytes	Size of attribute N, in bytes
	DataN	SizeN	Attribute N data

-PKCS#7 Encryption Recipient Certificate List (0x0019):

This field contains information about each of the certificates used in encryption processing and it can be used to identify who is allowed to decrypt encrypted files. This field should only appear in the archive extra data record. This field is not required and serves only to aide archive modifications by preserving public encryption key data. Individual security requirements may dictate that this data be omitted to deter information exposure.

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

	Value	Size	Description
	-----	----	-----
(CStore)	0x0019	2 bytes	Tag for this "extra" block type
	TSize	2 bytes	Size of the store data
	TData	TSize	Data about the store

TData:

	Value	Size	Description
	-----	----	-----
	Version	2 bytes	Format version number - must 0x0001 at this time
	CStore	(var)	PKCS#7 data blob

-MVS Extra Field (0x0065):

The following is the layout of the MVS "extra" block.
Note: Some fields are stored in Big Endian format.
All text is in EBCDIC format unless otherwise specified.

	Value	Size	Description
	-----	----	-----
(MVS)	0x0065	2 bytes	Tag for this "extra" block type
	TSize	2 bytes	Size for the following data block
	ID	4 bytes	EBCDIC "Z390" 0xE9F3F9F0 or "T4MV" for TargetFour
	(var)	TSize-4	Attribute data (see APPENDIX B)

-OS/400 Extra Field (0x0065):

The following is the layout of the OS/400 "extra" block.
Note: Some fields are stored in Big Endian format.
All text is in EBCDIC format unless otherwise specified.

	Value	Size	Description
	-----	----	-----
(OS400)	0x0065	2 bytes	Tag for this "extra" block type
	TSize	2 bytes	Size for the following data block
	ID	4 bytes	EBCDIC "I400" 0xC9F4F0F0 or "T4MV" for TargetFour
	(var)	TSize-4	Attribute data (see APPENDIX A)

Third-party Mappings:

-ZipIt Macintosh Extra Field (long) (0x2605):

The following is the layout of the ZipIt extra block for Macintosh. The local-header and central-header versions are identical. This block must be present if the file is stored MacBinary-encoded and it should not be used if the file is not stored MacBinary-encoded.

	Value	Size	Description
	-----	----	-----
(Mac2)	0x2605	Short	tag for this extra block type
	TSize	Short	total data size for this block
	"ZPIT"	beLong	extra-field signature
	FnLen	Byte	length of FileName
	FileName	variable	full Macintosh filename
	FileType	Byte[4]	four-byte Mac file type string
	Creator	Byte[4]	four-byte Mac creator string

-ZipIt Macintosh Extra Field (short, for files) (0x2705):

The following is the layout of a shortened variant of the ZipIt extra block for Macintosh (without "full name" entry). This variant is used by ZipIt 1.3.5 and newer for entries of files (not directories) that do not have a MacBinary encoded file. The local-header and central-header versions are identical.

	Value	Size	Description
	-----	----	-----
(Mac2b)	0x2705	Short	tag for this extra block type
	TSize	Short	total data size for this block (12)
	"ZPIT"	beLong	extra-field signature
	FileType	Byte[4]	four-byte Mac file type string
	Creator	Byte[4]	four-byte Mac creator string
	fdFlags	beShort	attributes from FInfo.frFlags, may be omitted
	0x0000	beShort	reserved, may be omitted

-ZipIt Macintosh Extra Field (short, for directories) (0x2805):

The following is the layout of a shortened variant of the ZipIt extra block for Macintosh used only for directory entries. This variant is used by ZipIt 1.3.5 and newer to save some optional Mac-specific information about directories. The local-header and central-header versions are identical.

	Value	Size	Description
	-----	----	-----
(Mac2c)	0x2805	Short	tag for this extra block type
	TSize	Short	total data size for this block (12)
	"ZPIT"	beLong	extra-field signature
	frFlags	beShort	attributes from DInfo.frFlags, may be omitted
	View	beShort	ZipIt view flag, may be omitted

The View field specifies ZipIt-internal settings as follows:

Bits of the Flags:

bit 0 if set, the folder is shown expanded (open)
 when the archive contents are viewed in ZipIt.
 bits 1-15 reserved, zero;

-FWKCS MD5 Extra Field (0x4b46):

The FWKCS Contents_Signature System, used in automatically identifying files independent of file name, 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 .ZIP file central directory to add this extra field for a file, it also replaces the central directory entry for that file's uncompressed file length with a measured value.

FWKCS provides an option to strip this extra field, if present, from a .ZIP file central directory. In adding this extra field, FWKCS preserves .ZIP file 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.
 11.76-77: "The MD5 algorithm is being placed in the public domain for review and possible adoption as a standard."

-Info-ZIP Unicode Comment Extra Field (0x6375):

Stores the UTF-8 version of the file comment as stored in the central directory header. (Last Revision 20070912)

	Value	Size	Description
	-----	----	-----
(UCom)	0x6375	Short	tag for this extra block type ("uc")
	TSize	Short	total data size for this block
	Version	1 byte	version of this extra field, currently 1
	ComCRC32	4 bytes	Comment Field CRC32 Checksum
	UnicodeCom	Variable	UTF-8 version of the entry comment

Currently Version is set to the number 1. If there is a need

to change this field, the version will be incremented. Changes may not be backward compatible so this extra field should not be used if the version is not recognized.

The ComCRC32 is the standard zip CRC32 checksum of the File Comment field in the central directory header. This is used to verify that the comment field has not changed since the Unicode Comment extra field was created. This can happen if a utility changes the File Comment field but does not update the UTF-8 Comment extra field. If the CRC check fails, this Unicode Comment extra field should be ignored and the File Comment field in the header should be used instead.

The UnicodeCom field is the UTF-8 version of the File Comment field in the header. As UnicodeCom is defined to be UTF-8, no UTF-8 byte order mark (BOM) is used. The length of this field is determined by subtracting the size of the previous fields from TSize. If both the File Name and Comment fields are UTF-8, the new General Purpose Bit Flag, bit 11 (Language encoding flag (EFS)), can be used to indicate both the header File Name and Comment fields are UTF-8 and, in this case, the Unicode Path and Unicode Comment extra fields are not needed and should not be created. Note that, for backward compatibility, bit 11 should only be used if the native characters of the paths and comments being zipped up are already in UTF-8. It is expected that the same file comment storage method, either general purpose bit 11 or extra fields, be used in both the Local and Central Directory Header for a file.

-Info-ZIP Unicode Path Extra Field (0x7075):

Stores the UTF-8 version of the file name field as stored in the local header and central directory header. (Last Revision 20070912)

	Value	Size	Description
	-----	----	-----
(UPath)	0x7075	Short	tag for this extra block type ("up")
	TSize	Short	total data size for this block
	Version	1 byte	version of this extra field, currently 1
	NameCRC32	4 bytes	File Name Field CRC32 Checksum
	UnicodeName	Variable	UTF-8 version of the entry File Name

Currently Version is set to the number 1. If there is a need to change this field, the version will be incremented. Changes may not be backward compatible so this extra field should not be used if the version is not recognized.

The NameCRC32 is the standard zip CRC32 checksum of the File Name field in the header. This is used to verify that the header File Name field has not changed since the Unicode Path extra field was created. This can happen if a utility renames the File Name but does not update the UTF-8 path extra field. If the CRC check fails, this UTF-8 Path Extra Field should be ignored and the File Name field in the header should be used instead.

The UnicodeName is the UTF-8 version of the contents of the File Name field in the header. As UnicodeName is defined to be UTF-8, no UTF byte order mark (BOM) is used. The length of this field is determined by subtracting the size of the previous fields from TSize. If both the File Name and Comment fields are UTF-8, the new General Purpose Bit Flag, bit 11 (Language encoding flag (EFS)), can be used to indicate that both the header File Name and Comment fields are UTF-8, and, in this case, the Unicode Path and Unicode Comment extra fields are not needed and should not be created. Note that, for backward compatibility, bit 11 should only be used if the native characters of the paths and comments being zipped up are already in UTF-8. It is expected that the same file name storage method, either general purpose bit 11 or extra fields, be used in both the Local and Central Directory Header for a file.

-Microsoft Open Packaging Growth Hint (0xa220):

Value	Size	Description
-----	----	-----
0xa220	Short	tag for this extra block type
TSize	Short	size of Sig + PadVal + Padding
Sig	Short	verification signature (A028)
PadVal	Short	Initial padding value
Padding	variable	filled with NULL characters

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. If an archive is in ZIP64 format and the value in this field is 0xFFFF, the size will be in the corresponding 4 byte zip64 end of central directory field.

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

The number of the disk on which the central directory starts. If an archive is in ZIP64 format and the value in this field is 0xFFFF, the size will be in the corresponding 4 byte zip64 end of central directory field.

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

The number of central directory entries on this disk. If an archive is in ZIP64 format and the value in this field is 0xFFFF, the size will be in the corresponding 8 byte zip64 end of central directory field.

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

The total number of files in the .ZIP file. If an archive is in ZIP64 format and the value in this field is 0xFFFF, the size will be in the corresponding 8 byte zip64 end of central directory field.

size of the central directory: (4 bytes)

The size (in bytes) of the entire central directory. If an archive is in ZIP64 format and the value in this field is 0xFFFFFFFF, the size will be in the corresponding 8 byte zip64 end of central directory field.

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. If an archive is in ZIP64 format and the value in this field is 0xFFFFFFFF, the size will be in the corresponding 8 byte zip64 end of central directory field.

.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. ZIP file comment data is stored unsecured. No encryption or data authentication is applied to this area at this time. Confidential information should not be stored in this section.

zip64 extensible data sector (variable size)

(currently reserved for use by PKWARE)

K. Splitting and Spanning ZIP files

Spanning is the process of segmenting a ZIP file across multiple removable media. This support has typically only been provided for DOS formatted floppy diskettes.

File splitting is a newer derivative of spanning. Splitting follows the same segmentation process as spanning, however, it does not require writing each segment to a unique removable medium and instead supports placing all pieces onto local or non-removable locations such as file systems, local drives, folders, etc...

A key difference between spanned and split ZIP files is that all pieces of a spanned ZIP file have the same name. Since each piece is written to a separate volume, no name collisions occur and each segment can reuse the original .ZIP file name given to the archive.

Sequence ordering for DOS spanned archives uses the DOS volume label to determine segment numbers. Volume labels for each segment are written using the form PKBACK#xxx, where xxx is the segment number written as a decimal value from 001 - nnn.

Split ZIP files are typically written to the same location and are subject to name collisions if the spanned name format is used since each segment will reside on the same drive. To avoid name collisions, split archives are named as follows.

Segment 1 = filename.z01
Segment n-1 = filename.z(n-1)
Segment n = filename.zip

The .ZIP extension is used on the last segment to support quickly reading the central directory. The segment number n should be a decimal value.

Spanned ZIP files may be PKSFx Self-extracting ZIP files. PKSFx files may also be split, however, in this case the first segment must be named filename.exe. The first segment of a split PKSFx archive must be large enough to include the entire executable program.

Capacities for split archives are as follows.

Maximum number of segments = 4,294,967,295 - 1
Maximum .ZIP segment size = 4,294,967,295 bytes
Minimum segment size = 64K
Maximum PKSFx segment size = 2,147,483,647 bytes

Segment sizes may be different however by convention, all segment sizes should be the same with the exception of the last, which may be smaller. Local and central directory header records must never be split across a segment boundary. When writing a header record, if the number of bytes remaining within a segment is less than the size of the header record, end the current segment and write the header at the start

of the next segment. The central directory may span segment boundaries, but no single record in the central directory should be split across segments.

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.

A special spanning marker may also appear in spanned/split archives if the spanning or splitting process starts but only requires one segment. In this case the 0x08074b50 signature will be replaced with the temporary spanning marker signature of 0x30304b50. Split archives can only be uncompressed by other versions of PKZIP that know how to create a split archive.

The signature value 0x08074b50 is also used by some ZIP implementations as a marker for the Data Descriptor record. Conflict in this alternate assignment can be avoided by ensuring the position of the signature within the ZIP file to determine the use for which it is intended.

L. 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) The entries in the central directory may not necessarily be in the same order that files appear in the .ZIP file.
- 4) If one of the fields in the end of central directory record is too small to hold required data, the field should be set to -1 (0xFFFF or 0xFFFFFFFF) and the ZIP64 format record should be created.
- 5) The end of central directory record and the Zip64 end of central directory locator record must reside on the same disk when splitting or spanning an archive.

VI. Explanation of compression methods

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 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.

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 $S(j)[0], \dots, S(j)[m]$, 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 $\{ N(j), S(j)[0], \dots, S(j)[N(j)-1] \}$, 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  $B(N(\text{Last-Character}))$  bits from the input
      stream, and assign this value to I.
      Copy the value of  $S(\text{Last-Character})[I]$  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 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:

- 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 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

```

Tokenizing - Method 7

This method is not used by PKZIP.

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

Extra			Extra			Extra			Extra		
Code	Bits	Length	Code	Bits	Lengths	Code	Bits	Lengths	Code	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

Extra			Extra			Extra			Extra		
Code	Bits	Dist	Code	Bits	Dist	Code	Bits	Distance	Code	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

```

Enhanced Deflating - Method 9

The Enhanced Deflating algorithm is similar to Deflate but uses a sliding dictionary of up to 64K. Deflate64(tm) is supported by the Deflate extractor.

BZIP2 - Method 12

BZIP2 is an open-source data compression algorithm developed by Julian Seward. Information and source code for this algorithm can be found on the internet.

LZMA - Method 14 (EFS)

LZMA is a block-oriented, general purpose data compression algorithm developed and maintained by Igor Pavlov. It is a derivative of LZ77 that utilizes Markov chains and a range coder. Information and source code for this algorithm can be found on the internet. Consult with the author of this algorithm for information on terms or restrictions on use.

Support for LZMA within the ZIP format is defined as follows:

The Compression method field within the ZIP Local and Central Header records will be set to the value 14 to indicate data was compressed using LZMA.

The Version needed to extract field within the ZIP Local and Central Header records will be set to 6.3 to indicate the minimum ZIP format version supporting this feature.

File data compressed using the LZMA algorithm must be placed immediately following the Local Header for the file. If a standard ZIP encryption header is required, it will follow the Local Header and will precede the LZMA compressed file

data segment. The location of LZMA compressed data segment within the ZIP format will be as shown:

```
[local header file 1]
[encryption header file 1]
[LZMA compressed data segment for file 1]
[data descriptor 1]
[local header file 2]
```

The encryption header and data descriptor records may be conditionally present. The LZMA Compressed Data Segment will consist of an LZMA Properties Header followed by the LZMA Compressed Data as shown:

```
[LZMA properties header for file 1]
[LZMA compressed data for file 1]
```

The LZMA Compressed Data will be stored as provided by the LZMA compression library. Compressed size, uncompressed size and other file characteristics about the file being compressed must be stored in standard ZIP storage format.

The LZMA Properties Header will store specific data required to decompress the LZMA compressed Data. This data is set by the LZMA compression engine using the function `WriteCoderProperties()` as documented within the LZMA SDK.

Storage fields for the property information within the LZMA Properties Header are as follows:

```
LZMA Version Information 2 bytes
LZMA Properties Size 2 bytes
LZMA Properties Data variable, defined by "LZMA Properties Size"
```

LZMA Version Information - this field identifies which version of the LZMA SDK was used to compress a file. The first byte will store the major version number of the LZMA SDK and the second byte will store the minor number.

LZMA Properties Size - this field defines the size of the remaining property data. Typically this size should be determined by the version of the SDK. This size field is included as a convenience and to help avoid any ambiguity should it arise in the future due to changes in this compression algorithm.

LZMA Property Data - this variable sized field records the required values for the decompressor as defined by the LZMA SDK. The data stored in this field should be obtained using the `WriteCoderProperties()` in the version of the SDK defined by the "LZMA Version Information" field.

The layout of the "LZMA Properties Data" field is a function of the LZMA compression algorithm. It is possible that this layout may be

changed by the author over time. The data layout in version 4.32 of the LZMA SDK defines a 5 byte array that uses 4 bytes to store the dictionary size in little-endian order. This is preceded by a single packed byte as the first element of the array that contains the following fields:

PosStateBits
LiteralPosStateBits
LiteralContextBits

Refer to the LZMA documentation for a more detailed explanation of these fields.

Data compressed with method 14, LZMA, may include an end-of-stream (EOS) marker ending the compressed data stream. This marker is not required, but its use is highly recommended to facilitate processing and implementers should include the EOS marker whenever possible. When the EOS marker is used, general purpose bit 1 must be set. If general purpose bit 1 is not set, the EOS marker is not present.

WavPack - Method 97

Information describing the use of compression method 97 is provided by WinZIP International, LLC. This method relies on the open source WavPack audio compression utility developed by David Bryant. Information on WavPack is available at www.wavpack.com. Please consult with the author of this algorithm for information on terms and restrictions on use.

WavPack data for a file begins immediately after the end of the local header data. This data is the output from WavPack compression routines. Within the ZIP file, the use of WavPack compression is indicated by setting the compression method field to a value of 97 in both the local header and the central directory header. The Version needed to extract and version made by fields use the same values as are used for data compressed using the Deflate algorithm.

An implementation note for storing digital sample data when using WavPack compression within ZIP files is that all of the bytes of the sample data should be compressed. This includes any unused bits up to the byte boundary. An example is a 2 byte sample that uses only 12 bits for the sample data with 4 unused bits. If only 12 bits are passed as the sample size to the WavPack routines, the 4 unused bits will be set to 0 on extraction regardless of their original state. To avoid this, the full 16 bits of the sample data size should be provided.

PPMd - Method 98

PPMd is a data compression algorithm developed by Dmitry Shkarin which includes a carryless rangecoder developed by Dmitry Subbotin.

This algorithm is based on predictive phrase matching on multiple order contexts. Information and source code for this algorithm can be found on the internet. Consult with the author of this algorithm for information on terms or restrictions on use.

Support for PPMd within the ZIP format currently is provided only for version I, revision 1 of the algorithm. Storage requirements for using this algorithm are as follows:

Parameters needed to control the algorithm are stored in the two bytes immediately preceding the compressed data. These bytes are used to store the following fields:

Model order - sets the maximum model order, default is 8, possible values are from 2 to 16 inclusive

Sub-allocator size - sets the size of sub-allocator in MB, default is 50, possible values are from 1MB to 256MB inclusive

Model restoration method - sets the method used to restart context model at memory insufficiency, values are:

- 0 - restarts model from scratch - default
- 1 - cut off model - decreases performance by as much as 2x
- 2 - freeze context tree - not recommended

An example for packing these fields into the 2 byte storage field is illustrated below. These values are stored in Intel low-byte/high-byte order.

```
wPPMd = (Model order - 1) +  
        ((Sub-allocator size - 1) << 4) +  
        (Model restoration method << 12)
```

VII. Traditional PKWARE Encryption

The following information discusses the decryption steps required to support traditional PKWARE encryption. This form of encryption is considered weak by today's standards and its use is recommended only for situations with low security needs or for compatibility with older .ZIP applications.

Decryption

PKWARE is grateful to Mr. Roger Schlafly for his expert contribution towards the development of PKWARE's traditional 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:

- 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 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
```

VIII. Strong Encryption Specification

The Strong Encryption technology defined in this specification is covered under a pending patent application. The use or implementation in a product of certain technological aspects set forth in the current APPNOTE, including those with regard to strong encryption, patching, or extended tape operations requires a license from PKWARE. Portions of this Strong Encryption technology are available for use at no charge. Contact PKWARE for licensing terms and conditions. Refer to section II of this APPNOTE (Contacting PKWARE) for information on how to contact PKWARE.

Version 5.x of this specification introduced support for strong encryption algorithms. These algorithms can be used with either a password or an X.509v3 digital certificate to encrypt each file. This format specification supports either password or certificate based encryption to meet the security needs of today, to enable interoperability between users within both PKI and non-PKI environments, and to ensure interoperability between different

computing platforms that are running a ZIP program.

Password based encryption is the most common form of encryption people are familiar with. However, inherent weaknesses with passwords (e.g. susceptibility to dictionary/brute force attack) as well as password management and support issues make certificate based encryption a more secure and scalable option. Industry efforts and support are defining and moving towards more advanced security solutions built around X.509v3 digital certificates and Public Key Infrastructures(PKI) because of the greater scalability, administrative options, and more robust security over traditional password based encryption.

Most standard encryption algorithms are supported with this specification. Reference implementations for many of these algorithms are available from either commercial or open source distributors. Readily available cryptographic toolkits make implementation of the encryption features straight-forward. This document is not intended to provide a treatise on data encryption principles or theory. Its purpose is to document the data structures required for implementing interoperable data encryption within the .ZIP format. It is strongly recommended that you have a good understanding of data encryption before reading further.

The algorithms introduced in Version 5.0 of this specification include:

- RC2 40 bit, 64 bit, and 128 bit
- RC4 40 bit, 64 bit, and 128 bit
- DES
- 3DES 112 bit and 168 bit

Version 5.1 adds support for the following:

- AES 128 bit, 192 bit, and 256 bit

Version 6.1 introduces encryption data changes to support interoperability with Smartcard and USB Token certificate storage methods which do not support the OAEP strengthening standard.

Version 6.2 introduces support for encrypting metadata by compressing and encrypting the central directory data structure to reduce information leakage. Information leakage can occur in legacy ZIP applications through exposure of information about a file even though that file is stored encrypted. The information exposed consists of file characteristics stored within the records and fields defined by this specification. This includes data such as a files name, its original size, timestamp and CRC32 value.

Version 6.3 introduces support for encrypting data using the Blowfish and Twofish algorithms. These are symmetric block ciphers developed

by Bruce Schneier. Blowfish supports using a variable length key from 32 to 448 bits. Block size is 64 bits. Implementations should use 16 rounds and the only mode supported within ZIP files is CBC. Twofish supports key sizes 128, 192 and 256 bits. Block size is 128 bits. Implementations should use 16 rounds and the only mode supported within ZIP files is CBC. Information and source code for both Blowfish and Twofish algorithms can be found on the internet. Consult with the author of these algorithms for information on terms or restrictions on use.

Central Directory Encryption provides greater protection against information leakage by encrypting the Central Directory structure and by masking key values that are replicated in the unencrypted Local Header. ZIP compatible programs that cannot interpret an encrypted Central Directory structure cannot rely on the data in the corresponding Local Header for decompression information.

Extra Field records that may contain information about a file that should not be exposed should not be stored in the Local Header and should only be written to the Central Directory where they can be encrypted. This design currently does not support streaming. Information in the End of Central Directory record, the Zip64 End of Central Directory Locator, and the Zip64 End of Central Directory records are not encrypted. Access to view data on files within a ZIP file with an encrypted Central Directory requires the appropriate password or private key for decryption prior to viewing any files, or any information about the files, in the archive.

Older ZIP compatible programs not familiar with the Central Directory Encryption feature will no longer be able to recognize the Central Directory and may assume the ZIP file is corrupt. Programs that attempt streaming access using Local Headers will see invalid information for each file. Central Directory Encryption need not be used for every ZIP file. Its use is recommended for greater security. ZIP files not using Central Directory Encryption should operate as in the past.

This strong encryption feature specification is intended to provide for scalable, cross-platform encryption needs ranging from simple password encryption to authenticated public/private key encryption.

Encryption provides data confidentiality and privacy. It is recommended that you combine X.509 digital signing with encryption to add authentication and non-repudiation.

Single Password Symmetric Encryption Method:

The Single Password Symmetric Encryption Method using strong encryption algorithms operates similarly to the traditional PKWARE encryption defined in this format. Additional data structures are added to support the processing needs of the strong algorithms.

The Strong Encryption data structures are:

1. General Purpose Bits - Bits 0 and 6 of the General Purpose bit flag in both local and central header records. Both bits set indicates strong encryption. Bit 13, when set indicates the Central Directory is encrypted and that selected fields in the Local Header are masked to hide their actual value.

2. Extra Field 0x0017 in central header only.

Fields to consider in this record are:

Format - the data format identifier for this record. The only value allowed at this time is the integer value 2.

AlgId - integer identifier of the encryption algorithm from the following range

- 0x6601 - DES
- 0x6602 - RC2 (version needed to extract < 5.2)
- 0x6603 - 3DES 168
- 0x6609 - 3DES 112
- 0x660E - AES 128
- 0x660F - AES 192
- 0x6610 - AES 256
- 0x6702 - RC2 (version needed to extract >= 5.2)
- 0x6720 - Blowfish
- 0x6721 - Twofish
- 0x6801 - RC4
- 0xFFFF - Unknown algorithm

Bitlen - Explicit bit length of key

32 - 448 bits

Flags - Processing flags needed for decryption

- 0x0001 - Password is required to decrypt
- 0x0002 - Certificates only
- 0x0003 - Password or certificate required to decrypt

Values > 0x0003 reserved for certificate processing

3. Decryption header record preceding compressed file data.

-Decryption Header:

Value	Size	Description
-----	----	-----
IVSize	2 bytes	Size of initialization vector (IV)
IVData	IVSize	Initialization vector for this file

Size	4 bytes	Size of remaining decryption header data
Format	2 bytes	Format definition for this record
AlgID	2 bytes	Encryption algorithm identifier
Bitlen	2 bytes	Bit length of encryption key
Flags	2 bytes	Processing flags
ErdSize	2 bytes	Size of Encrypted Random Data
ErdData	ErdSize	Encrypted Random Data
Reserved1	4 bytes	Reserved certificate processing data
Reserved2	(var)	Reserved for certificate processing data
VSize	2 bytes	Size of password validation data
VData	VSize-4	Password validation data
VCRC32	4 bytes	Standard ZIP CRC32 of password validation data

IVData - The size of the IV should match the algorithm block size. The IVData can be completely random data. If the size of the randomly generated data does not match the block size it should be complemented with zero's or truncated as necessary. If IVSize is 0, then IV = CRC32 + Uncompressed File Size (as a 64 bit little-endian, unsigned integer value).

Format - the data format identifier for this record. The only value allowed at this time is the integer value 3.

AlgId - integer identifier of the encryption algorithm from the following range

- 0x6601 - DES
- 0x6602 - RC2 (version needed to extract < 5.2)
- 0x6603 - 3DES 168
- 0x6609 - 3DES 112
- 0x660E - AES 128
- 0x660F - AES 192
- 0x6610 - AES 256
- 0x6702 - RC2 (version needed to extract >= 5.2)
- 0x6720 - Blowfish
- 0x6721 - Twofish
- 0x6801 - RC4
- 0xFFFF - Unknown algorithm

Bitlen - Explicit bit length of key

32 - 448 bits

Flags - Processing flags needed for decryption

- 0x0001 - Password is required to decrypt
- 0x0002 - Certificates only
- 0x0003 - Password or certificate required to decrypt

Values > 0x0003 reserved for certificate processing

ErdData - Encrypted random data is used to store random data that is used to generate a file session key for encrypting

each file. SHA1 is used to calculate hash data used to derive keys. File session keys are derived from a master session key generated from the user-supplied password. If the Flags field in the decryption header contains the value 0x4000, then the ErdData field must be decrypted using 3DES. If the value 0x4000 is not set, then the ErdData field must be decrypted using AlgId.

Reserved1 - Reserved for certificate processing, if value is zero, then Reserved2 data is absent. See the explanation under the Certificate Processing Method for details on this data structure.

Reserved2 - If present, the size of the Reserved2 data structure is located by skipping the first 4 bytes of this field and using the next 2 bytes as the remaining size. See the explanation under the Certificate Processing Method for details on this data structure.

VSize - This size value will always include the 4 bytes of the VCRC32 data and will be greater than 4 bytes.

VData - Random data for password validation. This data is VSize in length and VSize must be a multiple of the encryption block size. VCRC32 is a checksum value of VData. VData and VCRC32 are stored encrypted and start the stream of encrypted data for a file.

4. Useful Tips

Strong Encryption is always applied to a file after compression. The block oriented algorithms all operate in Cypher Block Chaining (CBC) mode. The block size used for AES encryption is 16. All other block algorithms use a block size of 8. Two ID's are defined for RC2 to account for a discrepancy found in the implementation of the RC2 algorithm in the cryptographic library on Windows XP SP1 and all earlier versions of Windows. It is recommended that zero length files not be encrypted, however programs should be prepared to extract them if they are found within a ZIP file.

A pseudo-code representation of the encryption process is as follows:

```
Password = GetUserPassword()  
MasterSessionKey = DeriveKey(SHA1>Password))  
RD = CryptographicStrengthRandomData()  
For Each File  
    IV = CryptographicStrengthRandomData()  
    VData = CryptographicStrengthRandomData()  
    VCRC32 = CRC32(VData)  
    FileSessionKey = DeriveKey(SHA1(IV + RD))  
    ErdData = Encrypt(RD,MasterSessionKey,IV)
```

Encrypt(VData + VCRC32 + FileData, FileSessionKey,IV)
Done

The function names and parameter requirements will depend on the choice of the cryptographic toolkit selected. Almost any toolkit supporting the reference implementations for each algorithm can be used. The RSA BSAFE(r), OpenSSL, and Microsoft CryptoAPI libraries are all known to work well.

Single Password - Central Directory Encryption:

Central Directory Encryption is achieved within the .ZIP format by encrypting the Central Directory structure. This encapsulates the metadata most often used for processing .ZIP files. Additional metadata is stored for redundancy in the Local Header for each file. The process of concealing metadata by encrypting the Central Directory does not protect the data within the Local Header. To avoid information leakage from the exposed metadata in the Local Header, the fields containing information about a file are masked.

Local Header:

Masking replaces the true content of the fields for a file in the Local Header with false information. When masked, the Local Header is not suitable for streaming access and the options for data recovery of damaged archives is reduced. Extra Data fields that may contain confidential data should not be stored within the Local Header. The value set into the Version needed to extract field should be the correct value needed to extract the file without regard to Central Directory Encryption. The fields within the Local Header targeted for masking when the Central Directory is encrypted are:

Field Name	Mask Value
-----	-----
compression method	0
last mod file time	0
last mod file date	0
crc-32	0
compressed size	0
uncompressed size	0
file name (variable size)	Base 16 value from the range 1 - 0xFFFFFFFFFFFFFFFF represented as a string whose size will be set into the file name length field

The Base 16 value assigned as a masked file name is simply a sequentially incremented value for each file starting with 1 for the first file. Modifications to a ZIP file may cause different values to be stored for each file. For compatibility, the file name field in the Local Header should never be left blank. As of Version 6.2 of this specification, the Compression Method and Compressed Size fields are not yet masked.

Fields having a value of 0xFFFF or 0xFFFFFFFF for the ZIP64 format should not be masked.

Encrypting the Central Directory:

Encryption of the Central Directory does not include encryption of the Central Directory Signature data, the Zip64 End of Central Directory record, the Zip64 End of Central Directory Locator, or the End of Central Directory record. The ZIP file comment data is never encrypted.

Before encrypting the Central Directory, it may optionally be compressed. Compression is not required, but for storage efficiency it is assumed this structure will be compressed before encrypting. Similarly, this specification supports compressing the Central Directory without requiring that it also be encrypted. Early implementations of this feature will assume the encryption method applied to files matches the encryption applied to the Central Directory.

Encryption of the Central Directory is done in a manner similar to that of file encryption. The encrypted data is preceded by a decryption header. The decryption header is known as the Archive Decryption Header. The fields of this record are identical to the decryption header preceding each encrypted file. The location of the Archive Decryption Header is determined by the value in the Start of the Central Directory field in the Zip64 End of Central Directory record. When the Central Directory is encrypted, the Zip64 End of Central Directory record will always be present.

The layout of the Zip64 End of Central Directory record for all versions starting with 6.2 of this specification will follow the Version 2 format. The Version 2 format is as follows:

The leading fixed size fields within the Version 1 format for this record remain unchanged. The record signature for both Version 1 and Version 2 will be 0x06064b50. Immediately following the last byte of the field known as the Offset of Start of Central Directory With Respect to the Starting Disk Number will begin the new fields defining Version 2 of this record.

New fields for Version 2:

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

Value	Size	Description
-----	----	-----
Compression Method	2 bytes	Method used to compress the Central Directory
Compressed Size	8 bytes	Size of the compressed data
Original Size	8 bytes	Original uncompressed size
AlgId	2 bytes	Encryption algorithm ID
BitLen	2 bytes	Encryption key length
Flags	2 bytes	Encryption flags

HashID	2 bytes	Hash algorithm identifier
Hash Length	2 bytes	Length of hash data
Hash Data	(variable)	Hash data

The Compression Method accepts the same range of values as the corresponding field in the Central Header.

The Compressed Size and Original Size values will not include the data of the Central Directory Signature which is compressed or encrypted.

The AlgId, BitLen, and Flags fields accept the same range of values the corresponding fields within the 0x0017 record.

Hash ID identifies the algorithm used to hash the Central Directory data. This data does not have to be hashed, in which case the values for both the HashID and Hash Length will be 0. Possible values for HashID are:

Value	Algorithm
-----	-----
0x0000	none
0x0001	CRC32
0x8003	MD5
0x8004	SHA1
0x8007	RIPEMD160
0x800C	SHA256
0x800D	SHA384
0x800E	SHA512

When the Central Directory data is signed, the same hash algorithm used to hash the Central Directory for signing should be used. This is recommended for processing efficiency, however, it is permissible for any of the above algorithms to be used independent of the signing process.

The Hash Data will contain the hash data for the Central Directory. The length of this data will vary depending on the algorithm used.

The Version Needed to Extract should be set to 62.

The value for the Total Number of Entries on the Current Disk will be 0. These records will no longer support random access when encrypting the Central Directory.

When the Central Directory is compressed and/or encrypted, the End of Central Directory record will store the value 0xFFFFFFFF as the value for the Total Number of Entries in the Central Directory. The value stored in the Total Number of Entries in the Central Directory on this Disk field will be 0. The actual values will be stored in the equivalent fields of the Zip64 End of Central Directory record.

Decrypting and decompressing the Central Directory is accomplished in the same manner as decrypting and decompressing a file.

Certificate Processing Method:

The Certificate Processing Method of for ZIP file encryption defines the following additional data fields:

1. Certificate Flag Values

Additional processing flags that can be present in the Flags field of both the 0x0017 field of the central directory Extra Field and the Decryption header record preceding compressed file data are:

- 0x0007 - reserved for future use
- 0x000F - reserved for future use
- 0x0100 - Indicates non-OAEP key wrapping was used. If this field is set, the version needed to extract must be at least 61. This means OAEP key wrapping is not used when generating a Master Session Key using ErdData.
- 0x4000 - ErdData must be decrypted using 3DES-168, otherwise use the same algorithm used for encrypting the file contents.
- 0x8000 - reserved for future use

2. CertData - Extra Field 0x0017 record certificate data structure

The data structure used to store certificate data within the section of the Extra Field defined by the CertData field of the 0x0017 record are as shown:

Value	Size	Description
-----	----	-----
RCount	4 bytes	Number of recipients.
HashAlg	2 bytes	Hash algorithm identifier
HSize	2 bytes	Hash size
SRList	(var)	Simple list of recipients hashed public keys

RCount This defines the number intended recipients whose public keys were used for encryption. This identifies the number of elements in the SRList.

HashAlg This defines the hash algorithm used to calculate the public key hash of each public key used for encryption. This field currently supports only the following value for SHA-1

0x8004 - SHA1

HSize This defines the size of a hashed public key.

SRList This is a variable length list of the hashed public keys for each intended recipient. Each element in this list is HSize. The total size of SRList is determined using RCount * HSize.

3. Reserved1 - Certificate Decryption Header Reserved1 Data:

Value	Size	Description
-----	----	-----
RCount	4 bytes	Number of recipients.

RCount This defines the number intended recipients whose public keys were used for encryption. This defines the number of elements in the REList field defined below.

4. Reserved2 - Certificate Decryption Header Reserved2 Data Structures:

Value	Size	Description
-----	----	-----
HashAlg	2 bytes	Hash algorithm identifier
HSize	2 bytes	Hash size
REList	(var)	List of recipient data elements

HashAlg This defines the hash algorithm used to calculate the public key hash of each public key used for encryption. This field currently supports only the following value for SHA-1

0x8004 - SHA1

HSize This defines the size of a hashed public key defined in REHData.

REList This is a variable length of list of recipient data. Each element in this list consists of a Recipient Element data structure as follows:

Recipient Element (REList) Data Structure:

Value	Size	Description
-----	----	-----
RESize	2 bytes	Size of REHData + REKData
REHData	HSize	Hash of recipients public key
REKData	(var)	Simple key blob

RESize This defines the size of an individual REList

element. This value is the combined size of the REHData field + REKData field. REHData is defined by HSize. REKData is variable and can be calculated for each REList element using RESize and HSize.

REHData Hashed public key for this recipient.

REKData Simple Key Blob. The format of this data structure is identical to that defined in the Microsoft CryptoAPI and generated using the CryptExportKey() function. The version of the Simple Key Blob supported at this time is 0x02 as defined by Microsoft.

Certificate Processing - Central Directory Encryption:

Central Directory Encryption using Digital Certificates will operate in a manner similar to that of Single Password Central Directory Encryption. This record will only be present when there is data to place into it. Currently, data is placed into this record when digital certificates are used for either encrypting or signing the files within a ZIP file. When only password encryption is used with no certificate encryption or digital signing, this record is not currently needed. When present, this record will appear before the start of the actual Central Directory data structure and will be located immediately after the Archive Decryption Header if the Central Directory is encrypted.

The Archive Extra Data record will be used to store the following information. Additional data may be added in future versions.

Extra Data Fields:

- 0x0014 - PKCS#7 Store for X.509 Certificates
- 0x0016 - X.509 Certificate ID and Signature for central directory
- 0x0019 - PKCS#7 Encryption Recipient Certificate List

The 0x0014 and 0x0016 Extra Data records that otherwise would be located in the first record of the Central Directory for digital certificate processing. When encrypting or compressing the Central Directory, the 0x0014 and 0x0016 records must be located in the Archive Extra Data record and they should not remain in the first Central Directory record. The Archive Extra Data record will also be used to store the 0x0019 data.

When present, the size of the Archive Extra Data record will be included in the size of the Central Directory. The data of the Archive Extra Data record will also be compressed and encrypted along with the Central Directory data structure.

Certificate Processing Differences:

The Certificate Processing Method of encryption differs from the Single Password Symmetric Encryption Method as follows. Instead of using a user-defined password to generate a master session key, cryptographically random data is used. The key material is then wrapped using standard key-wrapping techniques. This key material is wrapped using the public key of each recipient that will need to decrypt the file using their corresponding private key.

This specification currently assumes digital certificates will follow the X.509 V3 format for 1024 bit and higher RSA format digital certificates. Implementation of this Certificate Processing Method requires supporting logic for key access and management. This logic is outside the scope of this specification.

OAEP Processing with Certificate-based Encryption:

OAEP stands for Optimal Asymmetric Encryption Padding. It is a strengthening technique used for small encoded items such as decryption keys. This is commonly applied in cryptographic key-wrapping techniques and is supported by PKCS #1. Versions 5.0 and 6.0 of this specification were designed to support OAEP key-wrapping for certificate-based decryption keys for additional security.

Support for private keys stored on Smartcards or Tokens introduced a conflict with this OAEP logic. Most card and token products do not support the additional strengthening applied to OAEP key-wrapped data. In order to resolve this conflict, versions 6.1 and above of this specification will no longer support OAEP when encrypting using digital certificates.

Versions of PKZIP available during initial development of the certificate processing method set a value of 61 into the version needed to extract field for a file. This indicates that non-OAEP key wrapping is used. This affects certificate encryption only, and password encryption functions should not be affected by this value. This means values of 61 may be found on files encrypted with certificates only, or on files encrypted with both password encryption and certificate encryption. Files encrypted with both methods can safely be decrypted using the password methods documented.

IX. Change Process

In order for the .ZIP file format to remain a viable definition, this specification should be considered as open for periodic review and revision. Although this format was originally designed with a certain level of extensibility, not all changes in technology (present or future) were or will be necessarily considered in its design. If your application requires new definitions to the extensible sections in this format, or if you would like to submit new data structures, please forward your request to zipformat@pkware.com. All submissions will be reviewed by the ZIP File Specification Committee for possible inclusion into

future versions of this specification. Periodic revisions to this specification will be published to ensure interoperability. We encourage comments and feedback that may help improve clarity or content.

X. Incorporating PKWARE Proprietary Technology into Your Product

PKWARE is committed to the interoperability and advancement of the .ZIP format. PKWARE offers a free license for certain technological aspects described above under certain restrictions and conditions. However, the use or implementation in a product of certain technological aspects set forth in the current APPNOTE, including those with regard to strong encryption, patching, or extended tape operations requires a license from PKWARE. Please contact PKWARE with regard to acquiring a license.

XI. Acknowledgements

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.

XII. 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.

APPENDIX A - AS/400 Extra Field (0x0065) Attribute Definitions

Field Definition Structure:

a. field length including length	2 bytes
b. field code	2 bytes
c. data	x bytes

Field Code	Description	
4001	Source type i.e. CLP etc	
4002	The text description of the library	
4003	The text description of the file	
4004	The text description of the member	
4005	x'F0' or 0 is PF-DTA, x'F1' or 1 is PF_SRC	
4007	Database Type Code	1 byte
4008	Database file and fields definition	
4009	GZIP file type	2 bytes
400B	IFS code page	2 bytes
400C	IFS Creation Time	4 bytes
400D	IFS Access Time	4 bytes
400E	IFS Modification time	4 bytes
005C	Length of the records in the file	2 bytes
0068	GZIP two words	8 bytes

APPENDIX B - z/OS Extra Field (0x0065) Attribute Definitions

Field Definition Structure:

a. field length including length	2 bytes
b. field code	2 bytes
c. data	x bytes

Field Code	Description	
0001	File Type	2 bytes
0002	NonVSAM Record Format	1 byte
0003	Reserved	
0004	NonVSAM Block Size	2 bytes Big Endian
0005	Primary Space Allocation	3 bytes Big Endian
0006	Secondary Space Allocation	3 bytes Big Endian
0007	Space Allocation Type1 byte flag	
0008	Modification Date	Retired with PKZIP 5.0 +
0009	Expiration Date	Retired with PKZIP 5.0 +
000A	PDS Directory Block Allocation	3 bytes Big Endian binary va
000B	NonVSAM Volume List	variable
000C	UNIT Reference	Retired with PKZIP 5.0 +
000D	DF/SMS Management Class	8 bytes EBCDIC Text Value

000E	DF/SMS Storage Class	8 bytes EBCDIC Text Value
000F	DF/SMS Data Class	8 bytes EBCDIC Text Value
0010	PDS/PDSE Member Info.	30 bytes
0011	VSAM sub-filetype	2 bytes
0012	VSAM LRECL	13 bytes EBCDIC "(num_avg nur
0013	VSAM Cluster Name	Retired with PKZIP 5.0 +
0014	VSAM KSDS Key Information	13 bytes EBCDIC "(num_length
0015	VSAM Average LRECL	5 bytes EBCDIC num_value padc
0016	VSAM Maximum LRECL	5 bytes EBCDIC num_value padc
0017	VSAM KSDS Key Length	5 bytes EBCDIC num_value padc
0018	VSAM KSDS Key Position	5 bytes EBCDIC num_value padc
0019	VSAM Data Name	1-44 bytes EBCDIC text string
001A	VSAM KSDS Index Name	1-44 bytes EBCDIC text string
001B	VSAM Catalog Name	1-44 bytes EBCDIC text string
001C	VSAM Data Space Type	9 bytes EBCDIC text string
001D	VSAM Data Space Primary	9 bytes EBCDIC num_value left
001E	VSAM Data Space Secondary	9 bytes EBCDIC num_value left
001F	VSAM Data Volume List	variable EBCDIC text list of
0020	VSAM Data Buffer Space	8 bytes EBCDIC num_value left
0021	VSAM Data CISIZE	5 bytes EBCDIC num_value left
0022	VSAM Erase Flag	1 byte flag
0023	VSAM Free CI %	3 bytes EBCDIC num_value left
0024	VSAM Free CA %	3 bytes EBCDIC num_value left
0025	VSAM Index Volume List	variable EBCDIC text list of
0026	VSAM Ordered Flag	1 byte flag
0027	VSAM REUSE Flag	1 byte flag
0028	VSAM SPANNED Flag	1 byte flag
0029	VSAM Recovery Flag	1 byte flag
002A	VSAM WRITECHK Flag	1 byte flag
002B	VSAM Cluster/Data SHROPTS	3 bytes EBCDIC "n,y"
002C	VSAM Index SHROPTS	3 bytes EBCDIC "n,y"
002D	VSAM Index Space Type	9 bytes EBCDIC text string
002E	VSAM Index Space Primary	9 bytes EBCDIC num_value left
002F	VSAM Index Space Secondary	9 bytes EBCDIC num_value left
0030	VSAM Index CISIZE	5 bytes EBCDIC num_value left
0031	VSAM Index IMBED	1 byte flag
0032	VSAM Index Ordered Flag	1 byte flag
0033	VSAM REPLICATE Flag	1 byte flag
0034	VSAM Index REUSE Flag	1 byte flag
0035	VSAM Index WRITECHK Flag	1 byte flag Retired with PKZI
0036	VSAM Owner	8 bytes EBCDIC text string
0037	VSAM Index Owner	8 bytes EBCDIC text string
0038	Reserved	
0039	Reserved	
003A	Reserved	
003B	Reserved	
003C	Reserved	
003D	Reserved	
003E	Reserved	
003F	Reserved	
0040	Reserved	
0041	Reserved	
0042	Reserved	

0043	Reserved		
0044	Reserved		
0045	Reserved		
0046	Reserved		
0047	Reserved		
0048	Reserved		
0049	Reserved		
004A	Reserved		
004B	Reserved		
004C	Reserved		
004D	Reserved		
004E	Reserved		
004F	Reserved		
0050	Reserved		
0051	Reserved		
0052	Reserved		
0053	Reserved		
0054	Reserved		
0055	Reserved		
0056	Reserved		
0057	Reserved		
0058	PDS/PDSE Member TTR Info.	6 bytes	Big Endian
0059	PDS 1st LMOD Text TTR	3 bytes	Big Endian
005A	PDS LMOD EP Rec #	4 bytes	Big Endian
005B	Reserved		
005C	Max Length of records	2 bytes	Big Endian
005D	PDSE Flag	1 byte	flag
005E	Reserved		
005F	Reserved		
0060	Reserved		
0061	Reserved		
0062	Reserved		
0063	Reserved		
0064	Reserved		
0065	Last Date Referenced	4 bytes	Packed Hex "yyyymmdd"
0066	Date Created	4 bytes	Packed Hex "yyyymmdd"
0068	GZIP two words	8 bytes	
0071	Extended NOTE Location	12 bytes	Big Endian
0072	Archive device UNIT	6 bytes	EBCDIC
0073	Archive 1st Volume	6 bytes	EBCDIC
0074	Archive 1st VOL File Seq#	2 bytes	Binary

APPENDIX C - Zip64 Extensible Data Sector Mappings (EFS)

-Z390 Extra Field:

The following is the general layout of the attributes for the ZIP 64 "extra" block for extended tape operations. Portions of this extended tape processing technology is covered under a pending patent application. The use or implementation in a product of certain technological aspects set forth in the current APPNOTE, including those with regard to strong encryption,

patching or extended tape operations, requires a license from PKWARE. Please contact PKWARE with regard to acquiring a license.

Note: some fields stored in Big Endian format. All text is in EBCDIC format unless otherwise specified.

	Value	Size	Description
	-----	----	-----
(Z390)	0x0065	2 bytes	Tag for this "extra" block type
	Size	4 bytes	Size for the following data block
	Tag	4 bytes	EBCDIC "Z390"
	Length71	2 bytes	Big Endian
	Subcode71	2 bytes	Enote type code
	FMEPos	1 byte	
	Length72	2 bytes	Big Endian
	Subcode72	2 bytes	Unit type code
	Unit	1 byte	Unit
	Length73	2 bytes	Big Endian
	Subcode73	2 bytes	Volumel type code
	FirstVol	1 byte	Volume
	Length74	2 bytes	Big Endian
	Subcode74	2 bytes	FirstVol file sequence
	FileSeq	2 bytes	Sequence

APPENDIX D - Language Encoding (EFS)

The ZIP format has historically supported only the original IBM PC character encoding set, commonly referred to as IBM Code Page 437. This limits storing file name characters to only those within the original MS-DOS range of values and does not properly support file names in other character encodings, or languages. To address this limitation, this specification will support the following change.

If general purpose bit 11 is unset, the file name and comment should conform to the original ZIP character encoding. If general purpose bit 11 is set, the filename and comment must support The Unicode Standard, Version 4.1.0 or greater using the character encoding form defined by the UTF-8 storage specification. The Unicode Standard is published by the The Unicode Consortium (www.unicode.org). UTF-8 encoded data stored within ZIP files is expected to not include a byte order mark (BOM).

Applications may choose to supplement this file name storage through the use of the 0x0008 Extra Field. Storage for this optional field is currently undefined, however it will be used to allow storing extended information on source or target encoding that may further assist applications with file name, or file content encoding tasks. Please contact PKWARE with any requirements on how this field should be used.

The 0x0008 Extra Field storage may be used with either setting for general purpose bit 11. Examples of the intended usage for this field is to store whether "modified-UTF-8" (JAVA) is used, or UTF-8-MAC. Similarly, other

commonly used character encoding (code page) designations can be indicated through this field. Formalized values for use of the 0x0008 record remain undefined at this time. The definition for the layout of the 0x0008 field will be published when available. Use of the 0x0008 Extra Field provides for storing data within a ZIP file in an encoding other than IBM Code Page 437 or UTF-8.

General purpose bit 11 will not imply any encoding of file content or password. Values defining character encoding for file content or password must be stored within the 0x0008 Extended Language Encoding Extra Field.

Ed Gordon of the Info-ZIP group has defined a pair of "extra field" records that can be used to store UTF-8 file name and file comment fields. These records can be used for cases when the general purpose bit 11 method for storing UTF-8 data in the standard file name and comment fields is not desirable. A common case for this alternate method is if backward compatibility with older programs is required.

Definitions for the record structure of these fields are included above in the section on 3rd party mappings for "extra field" records. These records are identified by Header ID's 0x6375 (Info-ZIP Unicode Comment Extra Field) and 0x7075 (Info-ZIP Unicode Path Extra Field).

The choice of which storage method to use when writing a ZIP file is left to the implementation. Developers should expect that a ZIP file may contain either method and should provide support for reading data in either format. Use of general purpose bit 11 reduces storage requirements for file name data by not requiring additional "extra field" data for each file, but can result in older ZIP programs not being able to extract files. Use of the 0x6375 and 0x7075 records will result in a ZIP file that should always be readable by older ZIP programs, but requires more storage per file to write file name and/or file comment fields.