Doc. A/100-5
09 March 2003

DTV APPLICATION SOFTWARE ENVIRONMENT LEVEL 1 (DASE-1)
PART 5: ZIP ARCHIVE RESOURCE FORMAT

ATSC Standard

Note that this documents is past the customary 5-year review point. No update of the document is in process.

Advanced Television Systems Committee
1750 K Street, N.W.
Suite 1200
Washington, D.C. 20006
www.atsc.org

Blank Page

Table of Contents

Table of Tables

DASE-1 ZIP Archive Resource Format

ATSC Standard

1. SCOPE

1.1 Status

This section describes the status of this document at the time of its publication. Other documents may supersede this document. The latest status of this document series is maintained by the ATSC.

This specification is an ATSC Standard, having passed ATSC Member Ballot on September 16, 2002. This document is an editorial revision of the Approved Proposed Standard (PS/100-5) dated November 5, 2002.

The ATSC believes that this specification is stable, that it has been substantially demonstrated in independent implementations, and that it defines criteria that are necessary for effective implementation and interoperability of Advanced Television Systems. A list of cumulative changes made to this specification may be found at the end of this document.

A list of current ATSC Standards and other technical documents can be found at http://www.atsc.org/standards.html.

1.2 Purpose

This document specifies a content type for the representation of ZIP archives that may be delivered to and processed by a DASE application environment embodied by a compliant receiver.[1]

1.3 Application

The behavior and facilities of this specification are intended to apply to terrestrial (over-the-air) broadcast systems and receivers. In addition, the same behavior and facilities may be applied to other transport systems (such as cable or satellite).

1.4 Organization

This specification is organized as follows:

Section 1	Describes purpose, application and organization of this specification
Section 2	Enumerates normative and informative references
Section 3	Defines acronyms, terminology, and conventions
Section 4	Specifies ZIP archive resource format
Annex A	Specifies field and sub-field semantics
Annex B	Specifies extension syntax and semantics
Annex C	Specifies checksum algorithm

[1] The user's attention is called to the possibility that compliance with this standard may require use of an invention covered by patent rights. By publication of this standard, no position is taken with respect to the validity of this claim, or of any patent rights in connection therewith. The patent holder has, however, filed a statement of willingness to grant a license under these rights on reasonable and nondiscriminatory terms and conditions to applicants desiring to obtain such a license. Details may be obtained from the publisher.

- Changes Cumulative changes to specification

Unless explicitly indicated otherwise, all annexes shall be interpreted as normative parts of this specification.

This specification makes use of certain notational devices to provide valuable informative and explanatory information in the context of normative and, occasionally, informative sections. These devices take the form of paragraphs labeled as *Example* or *Note*. In each of these cases, the material is to be considered informative in nature.

2. REFERENCES

This section defines the normative and informative references employed by this specification. With the exception of Section 2.1, this section and its subsections are informative; in contrast, Section 2.1 is normative.

2.1 Normative References

The following documents contain provisions which, through reference in this specification, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All referenced documents are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent edition of the referenced document.

When a conflict exists between this specification and a referenced document, this specification takes precedence.

> *Note:* This specification uses a reference notation based on acronyms or convenient labels for identifying a reference (as opposed to using numbers).

[DEFLATE]

DEFLATE Compressed Data Format, Version 1.3, RFC1951, IETF

[UTF-8]

UTF-8, A Transformation Format of ISO 10646, RFC2279, IETF

2.2 Informative References

[GZIP]

GZIP File Format Specification, Version 4.3, RFC1952, IETF

[LZW]

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

[MIME]

Multimedia Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, RFC2045, IETF

[MPEG-2]

Information technology – Generic coding of moving pictures and associated audio information: Systems, ISO/IEC 13818-1:1996, ISO

[NELSON]

The Data Compression Book, Second Edition, Mark Nelson and Jean-Loup Gailly, 1996, M&T Books, ISBN 1-55851-434-1

[ZIP]

APPNOTE.TXT – ZIP File Format Specification, Version 4.0, May 1, 2001, PKWARE, Inc.

2.3 Reference Acquisition

IETF Standards

Internet Engineering Task Force (IETF), c/o Corporation for National Research Initiatives, 1895 Preston White Drive, Suite 100, Reston, VA 20191-5434, USA; Phone: +1 703 620 8990; Fax: +1 703 758 5913; http://www.ietf.org/.

ISO Standards

International Organization for Standardization (ISO), 1, rue de Varembé, Case postale 56, CH-1211 Geneva 20, Switzerland; Phone: +41 22 749 01 11; Fax: +41 22 733 34 30; http://www.iso.ch/.

PKWARE Documents

PKWARE, Inc., 9025 N. Deerwood Drive, Brown Deer, WI 53223-2480 USA; Phone: +1 414 354 8699; Fax: +1 414 354 8559; http://www.pkware.com/.

3. DEFINITIONS

This section defines conformance keywords, acronyms and abbreviations, and terms as employed by this specification.

3.1 Conformance Keywords

As used in this document, the conformance keyword *shall* denotes a mandatory provision of the standard. The keyword *should* denotes a provision that is recommended but not mandatory. The keyword *may* denotes a feature whose presence does not preclude compliance, that may or may not be present at the option of the application or the application environment implementer.

3.2 Acronyms and Abbreviations

bslbf	Bit Serial Leftmost Bit First
uilsBf	Unsigned Integer Least Significant Byte First
uilsWBf	Unsigned Integer Least Significant Word and Byte First

3.3 Terms

octet: an 8-bit byte.

4. ZIP ARCHIVE RESOURCE FORMAT

A ZIP Archive Resource is employed for the purpose of storing (or *archiving*) a collection of resources, known as archive entries, where each archive entry may be compressed or otherwise transformed into an encoded form.

This section defines the format of a ZIP Archive Resource.

Note: All multiple-octet fields in a ZIP Archive data structure make use of the little-endian order prescribed by the Intel x86 architecture.

4.1 ZIP Archive Structure

A ZIP archive resource consists of the following sections in the specified order:

- Entry 1 … Entry *N*
- Directory

4.2 Entry Structure

An entry structure entry consists of the following sections in the specified order:

- Entry Header
- Entry Data
- Entry Trailer (Optional)

4.2.1 Entry Header

An entry header shall adhere to the syntax specified by Table 1 Entry Header. The entry header shall be octet aligned. The total size of an entry header shall not exceed 65536 octets.

Table 1 Entry Header

Syntax	*Number of bits*	*Format*
entryHeader() {		
ehMarker	32	uilsWBf
ehVersionDecoder	16	uilsBf
ehFlags	16	uilsBf
ehMethod	16	uilsBf
ehLastModTime	16	uilsBf
ehLastModDate	16	uilsBf
ehCRC32	32	uilsWBf
ehSizeCompressed	32	uilsWBf
ehSizeUncompressed	32	uilsWBf
ehPathnameLength	16	uilsBf
ehExtensionLength	16	uilsBf
for (i = 0; i < ehPathnameLength; i++) {		
ehPathname[i]	8	bslbf
}		
for (i = 0; i < ehExtensionLength; i++) {		
ehExtension[i]	8	bslbf
}		
}		

ehMarker: A marker which indicates the start of the entry header. This field shall be set to the constant value 0x04034b50.

> *Note:* This field is referred to as *local file header signature* by [ZIP].

ehVersionDecoder: A field which indicates version information related to the expected archive decoder. The lower octet indicates the version of the encoding system, with the major version being the octet's integer *value* divided by 10, and minor version being *value* modulo 10.

> *Note:* This field is referred to as *version needed to extract* by [ZIP].

ehFlags: A field which specifies general flags regarding this entry, the interpretation of which is specified in Section A.1, Entry Flags.

> *Note:* This field is referred to as *general purpose bit flag* by [ZIP].

ehMethod: A field which specifies the compression method applied to this entry, the interpretation of which is specified in Section 4.4, Compression Methods.

> *Note:* This field is referred to as *compression method* by [ZIP].

ehLastModTime: A field which specifies the last modified time of the entry as follows: bits 4-0 denote seconds divided by two (0-29), bits 10-5 denote minutes (0-59), and bits 15-11 denote hours (0-23).

> *Note:* This field is referred to as *last mod file time* by [ZIP]. The value of this field conforms to the format used by the MS-DOS FILEINFO.fiFileTime field.

ehLastModDate: A field which specifies the last modified date of the entry as follows: bits 4-0 denote day of month (1-31), bits 8-5 denote month (1-12), and bits 15-9 denote years relative to 1980.

> *Note:* This field is referred to as *last mod file date* by [ZIP]. The value of this field conforms to the format used by the MS-DOS FILEINFO.fiFileDate field.

ehCRC32: If bit 3 of the *ehFlags* is set, then this field is set to zero; otherwise, it is set to a value representing a checksum of the uncompressed entry data, where the checksum algorithm is that described in ANNEX C.

> *Note:* This field is referred to as *crc-32* by [ZIP].

ehSizeCompressed: : If bit 3 of the *ehFlags* is set, then this field is set to zero; otherwise, it is set to the length of the compressed entry data in octets.

> *Note:* This field is referred to as *compressed size* by [ZIP].

ehSizeUncompressed: If bit 3 of the *ehFlags* is set, then this field is set to zero; otherwise, it is set to the length of the uncompressed entry data in octets.

> *Note:* This field is referred to as *uncompressed size* by [ZIP].

ehPathnameLength: The length of the *ehPathname* field in octets.

> *Note:* This field is referred to as *filename length* by [ZIP].

ehExtensionLength: The length of the *ehExtension* field in octets.

> *Note:* This field is referred to as *extra field length* by [ZIP].

ehPathname[]: This field specifies the name of the entry optionally preceded by a relative path. The value of this field shall consist of a non-terminated character string which conforms to the following syntax:

```
pathname  : [ rel_path ] name
rel_path  : segment [ "/" segment ]* "/"
segment   : pchar+
name      : pchar+
pchar     : { any non-control character except '/' }
```

Note: This field is referred to as *filename* by [ZIP].

ehExtension[]: Used for extension fields in accordance with ANNEX B, Extensions. If present, the value of this field may be exposed to an application for application-defined usage; otherwise, the field shall be ignored.

Note: This field is referred to as *extra field* by [ZIP].

4.2.2 Entry Data

The entry data is composed of a sequence of octets which represent the original entry data after having been compressed in accordance with the compression method which applies to the entry. The entry data shall be octet aligned and shall follow immediately after the last octet of the entry header.

4.2.3 Entry Trailer

An optional entry trailer defined by Table 2 Entry Trailer may follow the entry data. If present, then bit 3 of the entry's *ehFlags* field shall be set ('1'). The entry trailer shall be octet aligned and shall follow immediately after the last octet of the entry data.

Note: An entry trailer is generated only when a ZIP archive is created by means of a non-seekable output stream. In such a case, it is not possible to update the values of the *ehCRC32*, *ehSizeCompressed*, and *ehSizeUncompressed* fields in the entry header subsequent to writing the entry data.

Table 2 Entry Trailer

Syntax	*Number of bits*	*Format*
entryTrailer() {		
etMarker	32	uilsWBf
etCRC32	32	uilsWBf
etSizeCompressed	32	uilsWBf
etSizeUncompressed	32	uilsWBf
}		

ehMarker: A marker which indicates the start of the entry trailer. This field shall be set to the constant value 0x08074b50.

Note: This field is not defined by [ZIP], but is used in common industry practice as a reliable means of locating the beginning of the entry trailer.

etCRC32: A field whose value represents a checksum of the uncompressed entry data, where the checksum algorithm is that described in ANNEX C.

Note: This field is referred to as *crc-32* by [ZIP].

etSizeCompressed: The length of the compressed entry data in octets.

Note: This field is referred to as *compressed size* by [ZIP].

etSizeUncompressed: The length of the uncompressed entry data in octets.

Note: This field is referred to as *uncompressed size* by [ZIP].

4.3 Directory Structure

A directory structure consists of the following sections in the specified order:

- Directory Entry 1 … Directory Entry *N*

- Directory Digital Signature (Optional)
- Directory Trailer

The directory shall contain an entry for each archive entry; however, the order of entries in the directory need not correspond to the order of archive entries.

4.3.1 Directory Entry Header

A directory entry shall adhere to the syntax specified by Table 3 Directory Entry. A directory entry shall be octet aligned.

Table 3 Directory Entry

Syntax	*Number of bits*	*Format*
directoryEntry() {		
deMarker	32	uilsWBf
deVersionEncoder	16	uilsBf
deVersionDecoder	16	uilsBf
deFlags	16	uilsBf
deMethod	16	uilsBf
deLastModTime	16	uilsBf
deLastModDate	16	uilsBf
deCRC32	32	uilsWBf
deSizeCompressed	32	uilsWBf
deSizeUncompressed	32	uilsWBf
dePathnameLength	16	uilsBf
deExtensionLength	16	uilsBf
deCommentLength	16	uilsBf
deSegment	16	uilsBf
deAttributesInternal	16	uilsBf
deAttributesExternal	32	uilsWBf
deOffset	32	uilsWBf
for (i = 0; i < dePathnameLength; i++) {		
dePathname[i]	8	bslbf
}		
for (i = 0; i < deExtensionLength; i++) {		
deExtension[i]	8	bslbf
}		
for (i = 0; i < deCommentLength; i++) {		
deComment[i]	8	bslbf
}		
}		

deMarker: A marker which indicates the start of the directory entry. This field shall be set to the constant value 0x02014b50.

> *Note:* This field is referred to as *central file header signature* by [ZIP].

deVersionEncoder: A field which indicates file attribute compatibility information and version information related to the actual archive encoder and encoding environment. The upper octet indicates a file attribute compatibility code in accordance with Section A.2, File Attributes. The lower octet indicates the version of the encoding system, with the major version being the octet's integer *value* divided by 10, and minor version being *value* modulo 10.

> *Note:* This field is referred to as *version made by* by [ZIP].

deVersionDecoder: A field which indicates version information related to the expected archive decoder. The lower octet indicates the version of the encoding system, with the major version being the octet's integer *value* divided by 10, and minor version being *value* modulo 10.

> *Note:* This field is referred to as *version needed to extract* by [ZIP].

deFlags: A field which specifies general flags regarding this entry, the interpretation of which is specified in Section A.1, Entry Flags.

> *Note:* This field is referred to as *general purpose bit flag* by [ZIP].

deMethod: A field which specifies the compression method applied to this entry, the interpretation of which is specified in Section 4.4, Compression Methods.

> *Note:* This field is referred to as *compression method* by [ZIP].

deLastModTime: A field which specifies the last modified time of the entry as follows: bits 4-0 denote seconds divided by two (0-23), bits 10-5 denote minutes (0-59), and bits 15-11 denote hours (0-23).

> *Note:* This field is referred to as *last mod file time* by [ZIP]. The value of this field conforms to the format used by the MS-DOS FILEINFO.fiFileTime field.

deLastModDate: A field which specifies the last modified date of the entry as follows: bits 4-0 denote day of month (1-31), bits 8-5 denote month (1-12), and bits 15-9 denote years relative to 1980.

> *Note:* This field is referred to as *last mod file date* by [ZIP]. The value of this field conforms to the format used by the MS-DOS FILEINFO.fiFileDate field.

deCRC32: A field whose value represents a checksum of the uncompressed entry data, where the checksum algorithm is that described in ANNEX C.

> *Note:* This field is referred to as *crc-32* by [ZIP].

deSizeCompressed: The length of the compressed entry data in octets.

> *Note:* This field is referred to as *compressed size* by [ZIP].

deSizeUncompressed: The length of the uncompressed entry data in octets.

> *Note:* This field is referred to as *uncompressed size* by [ZIP].

dePathnameLength: The length of the *dePathname* field in octets.

> *Note:* This field is referred to as *filename length* by [ZIP].

deExtensionLength: The length of the *deExtension* field in octets.

> *Note:* This field is referred to as *extra field length* by [ZIP].

deCommentLength: The length of the *deComment* field in octets.

> *Note:* This field is referred to as *file comment length* by [ZIP].

deSegment: The number of the archive segment within which this entry starts.

> *Note:* This field is referred to as *disk number start* by [ZIP].

deAttributesInternal: This field is used to indicate attributes about the entry data. If bit 0 is set, then the entry data is believed by the encoder to be of textual nature (i.e., encoded characters); if it is clear, then the entry data is believed by the encoder to be of another binary form. All other bits are reserved.

> *Note:* This field is referred to as *internal file attributes* by [ZIP].

deAttributesExternal: This field is used to indicate attributes about the entry which were known to the encoder at encoding time and which are encoder environment dependent.

The value of this field may be zero, indicating that no external attributes were specified. An encoder is not required to specify a value other than zero; a decoder is not required to make use of any information provided by this field.

> *Note:* This field is referred to as *external file attributes* by [ZIP]. If the encoding environment was MS-DOS, then the lower octet corresponds to the value of the FILEINFO.fiAttribute byte.

deOffset: The offset in octets from the start of the segment in which the entry appears to the entry's header.

> *Note:* This field is referred to as *relative offset of local header* by [ZIP].

dePathname[]: This field is identical to the corresponding entry header's *ehPathname* field.

> *Note:* This field is referred to as *filename* by [ZIP].

deExtension[]: Used for extension fields in accordance with ANNEX B, Extensions. If present, the value of this field may be exposed to an application for application-defined usage; otherwise, the field shall be ignored.

> *Note:* This field is referred to as *extra field* by [ZIP].

deComment[]: If non-empty, this field contains a non-terminated character string which provides a comment about the entry.

> *Note:* This field is referred to as *file comment* by [ZIP].

4.3.2 Directory Digital Signature

An optional directory digital signature defined by Table 4 Directory Digital Signature may follow the last directory entry. The directory digital signature shall be octet aligned and shall follow immediately after the last octet of the last directory entry.

If present, an application environment may ignore the directory digital signature.

Table 4 Directory Digital Signature

Syntax	*Number of bits*	*Format*
directoryDigitalSignature() {		
ddsMarker	32	uilsWBf
ddsSignatureLength	16	uilsBf
for (i = 0; i < dddsSignatureLength; i++) {		
ddsSignature[i]	8	bslbf
}		
}		

ddsMarker: A marker which indicates the start of the directory digital signature. This field shall be set to the constant value 0x05054b50.

> *Note:* This field is referred to as *header signature* by [ZIP].

ddsSignatureLength: The length of the *ddsSignature* field in octets.

> *Note:* This field is referred to as *size of data* by [ZIP].

ddsSignature[]: This field specifies signature data which applies to the directory.

> *Note:* This field is referred to as *signature data* by [ZIP].

4.3.3 Directory Trailer

A directory trailer defined by Table 5 Directory Trailer shall follow the directory digital signature, if present, or the last directory entry, if no digital signature is present. The directory trailer shall be octet aligned and shall follow immediately after the last octet of the digital signature or the last directory entry.

Table 5 Directory Trailer

Syntax	*Number of bits*	*Format*
directoryTrailer() {		
dtMarker	32	uilsWBf
dtSegmentTrailer	16	uilsBf
dtSegmentDirectory	16	uilsBf
dtEntriesLast	16	uilsBf
dtEntries	16	uilsBf
dtSize	32	uilsWBf
dtOffset	32	uilsWBf
dtCommentLength	16	uilsBf
for (i = 0; i < dtCommentLength; i++) {		
dtComment[i]	8	bslbf
}		
}		

dtMarker: A marker which indicates the start of the directory digital signature. This field shall be set to the constant value 0x06054b50.

Note: This field is referred to as *end of central dir signature* by [ZIP].

dtSegmentTrailer: The number of the archive segment within which the directory trailer starts.

Note: This field is referred to as *number of disk* by [ZIP].

dtSegmentDirectory: The number of the archive segment within which the directory starts.

Note: This field is referred to as *number of disk with the start of the central directory* by [ZIP].

dtEntriesLast: The number of directory entries in this segment.

Note: This field is referred to as *total number of entries in the central dir on this disk* by [ZIP].

dtEntries: The total number of directory entries in the archive.

Note: This field is referred to as *total number of entries in the central dir* by [ZIP].

Note: The number of directory entries corresponds to the number of archive entries.

dtSize: The size of the directory in octets.

Note: This field is referred to as *size of the central directory* by [ZIP].

dtOffset: The offset in octets from the start of the segment in which the directory appears to the beginning of the directory.

Note: This field is referred to as *offset of start of central directory with respect to the starting disk number* by [ZIP].

dtCommentLength: The length of the *deComment* field in octets.

Note: This field is referred to as *.ZIP file comment length* by [ZIP].

dtComment[]: If non-empty, this field contains a non-terminated character string which provides a comment about the archive.

> *Note:* This field is referred to as *.ZIP file comment* by [ZIP].

4.4 Compression Methods

Each entry may be compressed or encoded, where a distinct type of compression or encoding is referred to as a *compression method*. This section defines the compression methods which may be used with an entry.

Table 6 Compression Methods

Method	*Description*
0	Entry is stored without compression.
1	Entry is shrunk.
2	Entry is reduced with compression factor 1.
3	Entry is reduced with compression factor 2.
4	Entry is reduced with compression factor 3.
5	Entry is reduced with compression factor 4.
6	Entry is imploded.
7	Reserved for use with tokenizing compression algorithm.
8	Entry is deflated.
9	Reserved for use with enhanced deflation algorithm.
10	Reserved for use with date compression library imploding algorithm.
11-65535	Reserved.

The use of compression methods 7, 9 and 10 is not supported by this specification. If an entry is present which uses one of these methods, it shall be ignored.

4.4.1 No Compression Method

The use of method 0 indicates that *no compression* is applied to the entry data.

4.4.2 Shrinking Method

The use of method 1 indicates that a *shrinking* compression is applied to the entry data as further described below.

Shrinking is a Dynamic Ziv-Lempel-Welch compression algorithm with partial clearing. The initial code size is 9 bits; the maximum code size is 13 bits. Shrinking differs from conventional Dynamic Ziv-Lempel-Welch implementations as follows:

(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. 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.

Note: For additional information on this algorithm, see [LZW] and [NELSON], pp. 262-288.

4.4.3 Reduction Methods

The use of methods 2 through 5 indicates that a *reduction* compression is applied to the entry data as further described below.

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

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

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

The compressed data stream appears immediately after the follower sets appears. The compressed data stream can be interpreted for the probabilistic decompression as follows:

```
let last-character = 0.
loop until done
    if the follower set S(last-character) is empty then
        read 8 bits from the input stream, and copy this
        value to the output stream.
    otherwise if the follower set S(last-character) is non-empty then
        read 1 bit from the input stream.
        if this bit is not zero then
            read 8 bits from the input stream and copy this
            value to the output stream.
        otherwise if this bit is zero then
            read B(N(last-character)) bits from the input
            stream and assign this value to I.
            Copy the value of S(last-character)[I] to the
            output stream.
    assign the last value placed on the output stream to
    last-character.
end loop
```

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

The decompressed stream from above can then be expanded to recreate the original entry 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 DLE (144 decimal) to the output stream.
                let state = 0
        2:  let len = len + C
            let state = 3.
        3:  move backwards D(V,C) octets 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 octets 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:

(1) 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.
```

(2) 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 bit of x) * 256 + y + 1.
```

(3) 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 bit of x) * 256 + y + 1.
```

(4) 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 bit of x) * 256 + y + 1.
```

Compression methods 2 through 5 correspond to the use of compression factors 1 through 4, respectively.

4.4.4 Implosion Method

The use of method 6 indicates that an *implosion* compression is applied to the entry data as further described below.

The implosion algorithm is a combination of two distinct algorithms. The first algorithm compresses repeated octet 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 implosion algorithm can use a 4K or 8K sliding dictionary size. The dictionary size used is determined by bit 1 of the entry flags field; 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 entry. The number of trees stored is defined by bit 2 of the entry flags field; a 0 bit indicates two trees stored, a 1 bit indicates three trees are stored. If three 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, and the third represents the encoding of the Distance information. When two Shannon-Fano trees are stored, the Length tree is stored first, followed by the Distance tree.

If present, the Literal Shannon-Fano tree is used to represent the entire extended 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 are stored in a compressed format. The first octet of the tree data represents the number of octets of data representing the (compressed) Shannon-Fano tree minus 1. The remaining octets 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 entry.

(2) Generate the Shannon-Fano trees as follows:

```
let code = 0
let codeIncrement = 0
let lastBitLength = 0
let 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 by 16 - lastBitLength
    ShannonCode(i) = code
    i = i - 1
end loop
```

(3) Reverse the order of all the bits in the 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 entry.

Example: This example shows the encoding of a Shannon-Fano tree of size 8. Notice that the actual Shannon-Fano trees used for implosion is either 64 or 256 entries in size.

Input sequence:

```
0x02, 0x42, 0x01, 0x13
```

The first octet indicates 3 values in this table. Decoding the octets:

```
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:

Value	*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 *Value*, *Order Restored* and *Original Length* columns 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 specification. 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 octets 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
```

Note: For additional information on this algorithm, [NELSON], pp. 29-31.

4.4.5 Deflation Method

The use of method 8 indicates that a *deflation* compression is applied to the entry data as further described below.

An entry which uses the deflation compression method shall adhere to [DEFLATE].

4.4.6 Other Methods

Other compression methods may be used with an entry. An entry using an unsupported compression method may be ignored.

ANNEX A. FIELD AND SUBFIELD SEMANTICS

The entirety of this section and its subsections is normative.

A.1 Entry Flags

The fields *ehFlags* and *deFlags* consists of a bit field as described by Table 7 Entry Flags.

Table 7 Entry Flags

Bit	*Description*
0	If set, indicates entry is encrypted.
1	Compression method dependent, see Section A.1.1.
2	Compression method dependent, see Section A.1.1.
3	If set, indicates that an entry trailer follows entry data.
4	Reserved
5	Reserved
6-11	Unused
12-15	Reserved

A.1.1 Compression Method Dependent Flags

The interpretation of certain entry flags is dependent upon the compression method used for the entry. These interpretations are specified in the following subsections.

A.1.1.1 Implosion Method Entry Flags

If the entry is compressed with the implosion method (6), then the following interpretation of entry flags applies:

Table 8 Implosion Method Entry Flags

Bit	*Description*
1	If set, indicates use of 8K sliding dictionary; if clear, indicates use of 4K sliding dictionary.
2	If set, indicates use of three Shannon-Fano trees when constructing sliding dictionary; if clear, indicates use of 2 Shannon-Fano trees.

A.1.1.2 Deflation Method Entry Flags

If the entry is compressed with a deflation method (8 or 9), then the following interpretation of entry flags applies:

Table 9 Deflation Method Entry Flags

Bit 2	*Bit 1*	*Description*
0	0	Normal compression was used.
0	1	Maximum compression was used.
1	0	Fast compression was used.
1	1	Super fast compression was used.

A.2 File Attributes

The most significant octet of fields *ehVersionEncoder*, *deVersionEncoder* and *deVersionDecoder* consists of a value as described by Table 10 File Attributes.

Table 10 File Attributes

Value	*Description*
0	MS-DOS or OS/2 (FAT/VFAT/FAT32)
1	Amiga
2	OpenVMS
3	Unix
4	VM/CMS
5	Atari ST
6	OS/2 (HPFS)
7	Macintosh
8	Z-System
9	CP/M
10	Windows (NTFS)
11	MVS
12	VSE
13-255	Unused

ANNEX B. EXTENSIONS

The entirety of this section and its subsections is normative.

An extension field is composed of zero or more extension structures which adhere to Table 11 Extension.

The contents of an extension may be exposed to an application; however, no standard interpretation for any specific extension is prescribed by this specification.

Table 11 Extension

Syntax	*Number of bits*	*Format*
extension() {		
extIdentifier	16	uilsBf
extLength	16	uilsBf
for (i = 0; i < extLength; i++) {		
extData[i]	8	bslbf
}		
}		

extIdentifier: An identifier which indicates the specific type of extension. Extension identifiers in the range 0 through 31 are reserved.

> *Note:* This field is referred to as *extra field header ID* by [ZIP].

extLength: The length of the *extData* field in octets.

> *Note:* This field is referred to as *extra field data size* by [ZIP].

extData[]: The extension's data represented as sequence of octets, the interpretation of which depends upon the extension type as indicated by *extIdentifier*.

> *Note:* This field is referred to as *extra field data* by [ZIP].

B.1 MIME Header Extension

One or more MIME headers may be associated with an individual archive entry by making use of the extension defined by Table 12 MIME Header Extension. If used, this extension shall use an extension identifier (*extIdentifier*) set to the constant value 0xFFFF.

> *Note:* A MIME header consists of a name and value pair, where the name denotes the header's name, and the value denotes the header's value. See [MIME] for additional information regarding MIME headers.

> *Note:* The *mimeExtension* structure is encoded into the *extData* field of an extension with the *extLength* field denoting the length of the *mimeExtension* structure in octets.

A distinct archive entry shall not have more than one MIME header extension whose header name is the same string when compared using a case-insensitive comparison operation.

Table 12 MIME Header Extension

Syntax	*Number of bits*	*Format*
mimeExtension() {		
meMarker	32	uilsWBf
meNameLength	16	uilsBf
meValueLength	16	uilsBf
for (i = 0; i < meNameLength; i++) {		

```
        meName[i]                                         8          bslbf
    }
    for (i = 0; i < meValueLength; i++) {
        meValue[i]                                        8          bslbf
    }
}
```

meMarker: A marker which provides additional assurance that this extension can be properly identified as a MIME extension. This field shall be set to the constant value 0x4D494D45.

meNameLength: The length of the *meComment* field in octets.

meName[]: This field contains a non-empty, non-terminated character string which specifies a MIME header name. The character encoding of *meName* shall be in accordance with [UTF-8].

meValueLength: The length of the *meValue* field in octets.

meValue[]: This field contains a non-empty, non-terminated character string which specifies a MIME header value. The character encoding of *meValue* shall be in accordance with [UTF-8].

ANNEX C. CRC-32 CHECKSUM ALGORITHM

The entirety of this section is normative.

The 32-bit CRC checksum is calculated with the polynomial:

$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$

The initial value of a checksum is initialized to all ones; the final value is the ones-complement of the result of applying this polynomial to the initial value by processing each bit of the input buffer from most significant to least significant bit.

> *Note:* This checksum algorithm is identical to that described by [MPEG-2], Annex A, *CRC Decoder Model*. A sample implementation of this algorithm is available in [GZIP], Section 8, *Appendix: Sample CRC Code*.

CHANGES

This section is informative.

Changes from Candidate Standard to Standard

The following table enumerates the changes between the issuance of the candidate standard edition of this specification and the standard edition.

Table 13 Changes from Candidate Standard

Section	Description
1	Change status to standard.