Network Working Group
Request for Comments: 1950
Category: Informational

P. Deutsch
Aladdin Enterprises
J-L. Gailly
Info-ZIP
May 1996

ZLIB Compressed Data Format Specification version 3.3

Status of This Memo

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

IESG Note:

The IESG takes no position on the validity of any Intellectual Property Rights statements contained in this document.

Notices

A pointer to the latest version of this and related documentation in HTML format can be found at the URL <ftp://ftp.uu.net/graphics/png/documents/zlib/zdoc-index.html>.

Abstract

This specification defines a lossless compressed data format. The data can be produced or consumed, even for an arbitrarily long sequentially presented input data stream, using only an *a priori* bounded amount of intermediate storage. The format presently uses the DEFLATE compression method but can be easily extended to use other compression methods. It can be implemented readily in a manner not covered by patents. This specification also defines the ADLER-32 checksum (an extension and improvement of the Fletcher checksum), used for detection of data corruption, and provides an algorithm for computing it.

Contents

1 Introduction

1.1 Purpose

The purpose of this specification is to define a lossless compressed data format that:

- Is independent of CPU type, operating system, file system, and character set, and hence can be used for interchange;
- Can be produced or consumed, even for an arbitrarily long sequentially presented input data stream, using only an *a priori* bounded amount of intermediate storage, and hence can be used in data communications or similar structures such as Unix filters;
- Can use a number of different compression methods;
- Can be implemented readily in a manner not covered by patents, and hence can be practiced freely.

The data format defined by this specification does not attempt to allow random access to compressed data.

1.2 Intended audience

This specification is intended for use by implementors of software to compress data into zlib format and/or decompress data from zlib format.

The text of the specification assumes a basic background in programming at the level of bits and other primitive data representations.

1.3 Scope

The specification specifies a compressed data format that can be used for in-memory compression of a sequence of arbitrary bytes.

1.4 Compliance

Unless otherwise indicated below, a compliant decompressor must be able to accept and decompress any data set that conforms to all the specifications presented here; a compliant compressor must produce data sets that conform to all the specifications presented here.

1.5 Definitions of terms and conventions used

byte: 8 bits stored or transmitted as a unit (same as an octet). (For this specification, a byte is exactly 8 bits, even on machines which store a character on a number of bits different from 8.) See below, for the numbering of bits within a byte.

1.6 Changes from previous versions

Version 3.1 was the first public release of this specification. In version 3.2, some terminology was changed and the Adler-32 sample code was rewritten for clarity. In version 3.3, the support for a preset dictionary was introduced, and the specification was converted to RFC style.

2 Detailed specification

2.1 Overall conventions

In the diagrams below, a box like this:

```
+---+
|   | <-- the vertical bars might be missing
+---+
```

represents one byte; a box like this:

```
+==============+
|              |
+==============+
```

represents a variable number of bytes.

Bytes stored within a computer do not have a "bit order", since they are always treated as a unit. However, a byte considered as an integer between 0 and 255 does have a most- and least-significant bit, and since we write numbers with the most-significant digit on the left, we also write bytes with the most-significant bit on the left. In the diagrams below, we number the bits of a byte so that bit 0 is the least-significant bit, i.e., the bits are numbered:

```
+--------+
|76543210|
+--------+
```

Within a computer, a number may occupy multiple bytes. All multi-byte numbers in the format described here are stored with the MOST-significant byte first (at the lower memory address). For example, the decimal number 520 is stored as:

```
    0        1
+--------+--------+
|00000010|00001000|
+--------+--------+
 ^        ^
 |        |
 |        + less significant byte = 8
 + more significant byte = 2 x 256
```

2.2 Data format

A zlib stream has the following structure:

```
  0   1
+---+---+
|CMF|FLG|   (more-->)
+---+---+
```

(if FLG.FDICT set)

```
  0   1   2   3
+---+---+---+---+
|     DICTID    |   (more-->)
+---+---+---+---+
```

```
+=====================+---+---+---+---+
|...compressed data...|    ADLER32    |
+=====================+---+---+---+---+
```

Any data which may appear after ADLER32 are not part of the zlib stream.

CMF (Compression Method and flags)

This byte is divided into a 4-bit compression method and a 4-bit information field depending on the compression method.

```
bits 0 to 3  CM     Compression method
bits 4 to 7  CINFO  Compression info
```

CM (Compression method)

This identifies the compression method used in the file. CM = 8 denotes the "deflate" compression method with a window size up to 32K. This is the method used by gzip and PNG (see references [1] and [2] in Chapter 3, below, for the reference documents). CM = 15 is reserved. It might be used in a future version of this specification to indicate the presence of an extra field before the compressed data.

CINFO (Compression info)

For CM = 8, CINFO is the base-2 logarithm of the LZ77 window size, minus eight (CINFO=7 indicates a 32K window size). Values of CINFO above 7 are not allowed in this version of the specification. CINFO is not defined in this specification for CM not equal to 8.

FLG (FLaGs)

This flag byte is divided as follows:

```
bits 0 to 4  FCHECK  (check bits for CMF and FLG)
bit  5       FDICT   (preset dictionary)
bits 6 to 7  FLEVEL  (compression level)
```

The FCHECK value must be such that CMF and FLG, when viewed as a 16-bit unsigned integer stored in MSB order (CMF*256 + FLG), is a multiple of 31.

FDICT (Preset dictionary)

If FDICT is set, a DICT dictionary identifier is present immediately after the FLG byte. The dictionary is a sequence of bytes which are initially fed to the compressor without producing any compressed output. DICT is the Adler-32 checksum of this sequence of bytes (see the definition of ADLER32 below). The decompressor can use this identifier to determine which dictionary has been used by the compressor.

FLEVEL (Compression level)

These flags are available for use by specific compression methods. The "deflate" method (CM = 8) sets these flags as follows:

```
0 - compressor used fastest algorithm
1 - compressor used fast algorithm
2 - compressor used default algorithm
3 - compressor used maximum compression, slowest algorithm
```

The information in FLEVEL is not needed for decompression; it is there to indicate if recompression might be worthwhile.

compressed data

For compression method 8, the compressed data is stored in the deflate compressed data format as described in the document "DEFLATE Compressed Data Format Specification" by L. Peter Deutsch. (See reference [3] in Chapter 3, below)

Other compressed data formats are not specified in this version of the zlib specification.

ADLER32 (Adler-32 checksum)

This contains a checksum value of the uncompressed data (excluding any dictionary data) computed according to Adler-32 algorithm. This algorithm is a 32-bit extension and improvement of the Fletcher algorithm, used in the ITU-T X.224 / ISO 8073 standard. See references [4] and [5] in Chapter 3, below)

Adler-32 is composed of two sums accumulated per byte: s1 is the sum of all bytes, s2 is the sum of all s1 values. Both sums are done modulo 65521. s1 is initialized to 1, s2 to zero. The Adler-32 checksum is stored as s2*65536 + s1 in most-significant-byte first (network) order.

2.3 Compliance

A compliant compressor must produce streams with correct CMF, FLG and ADLER32, but need not support preset dictionaries. When the zlib data format is used as part of another standard data format, the compressor may use only preset dictionaries that are specified by this other data format. If this other format does not use the preset dictionary feature, the compressor must not set the FDICT flag.

A compliant decompressor must check CMF, FLG, and ADLER32, and provide an error indication if any of these have incorrect values. A compliant decompressor must give an error indication if CM is not one of the values defined in this specification (only the value 8 is permitted in this version), since another value could indicate the presence of new features that would cause subsequent data to be interpreted incorrectly. A compliant decompressor must give an error indication if FDICT is set and DICTID is not the identifier of a known preset dictionary. A decompressor may ignore FLEVEL and still be compliant. When the zlib data format is being used as a part of another standard format, a compliant decompressor must support all the preset dictionaries specified by the other format. When the other format does not use the preset dictionary feature, a compliant decompressor must reject any stream in which the FDICT flag is set.

3 References

[1] Deutsch, L.P.,"GZIP Compressed Data Format Specification", available in ftp://ftp.uu.net/pub/archiving/zip/doc/

[2] Thomas Boutell, "PNG (Portable Network Graphics) specification", available in ftp://ftp.uu.net/graphics/png/documents/

[3] Deutsch, L.P.,"DEFLATE Compressed Data Format Specification", available in ftp://ftp.uu.net/pub/archiving/zip/doc/

[4] Fletcher, J. G., "An Arithmetic Checksum for Serial Transmissions," IEEE Transactions on Communications, Vol. COM-30, No. 1, January 1982, pp. 247-252.

[5] ITU-T Recommendation X.224, Annex D, "Checksum Algorithms," November, 1993, pp. 144, 145. (Available from gopher://info.itu.ch). ITU-T X.244 is also the same as ISO 8073.

4 Source code

Source code for a C language implementation of a "zlib" compliant library is available at ftp://ftp.uu.net/pub/archiving/zip/zlib/.

5 Security Considerations

A decoder that fails to check the ADLER32 checksum value may be subject to undetected data corruption.

6 Acknowledgements

Trademarks cited in this document are the property of their respective owners.

Jean-Loup Gailly and Mark Adler designed the zlib format and wrote the related software described in this specification. Glenn Randers-Pehrson converted this document to RFC and HTML format.

7 Authors' Addresses

L. Peter Deutsch

Aladdin Enterprises
203 Santa Margarita Ave.
Menlo Park, CA 94025

Phone: (415) 322-0103 (AM only)
FAX: (415) 322-1734
EMail: <ghost@aladdin.com>

Jean-Loup Gailly

EMail: <gzip@prep.ai.mit.edu>

Questions about the technical content of this specification can be sent by email to

Jean-Loup Gailly <gzip@prep.ai.mit.edu> and
Mark Adler <madler@alumni.caltech.edu>

Editorial comments on this specification can be sent by email to

L. Peter Deutsch <ghost@aladdin.com> and
Glenn Randers-Pehrson <randeg@alumni.rpi.edu>

8 Appendix: Rationale

8.1 Preset dictionaries

A preset dictionary is specially useful to compress short input sequences. The compressor can take advantage of the dictionary context to encode the input in a more compact manner. The decompressor can be initialized with the appropriate context by virtually decompressing a compressed version of the dictionary without producing any output. However for certain compression algorithms such as the deflate algorithm this operation can be achieved without actually performing any decompression.

The compressor and the decompressor must use exactly the same dictionary. The dictionary may be fixed or may be chosen among a certain number of predefined dictionaries, according to the kind of input data. The decompressor can determine which dictionary has been chosen by the compressor by checking the dictionary identifier. This document does not specify the contents of predefined dictionaries, since the optimal dictionaries are application specific. Standard data formats using this feature of the zlib specification must precisely define the allowed dictionaries.

8.2 The Adler-32 algorithm

The Adler-32 algorithm is much faster than the CRC32 algorithm yet still provides an extremely low probability of undetected errors.

The modulo on unsigned long accumulators can be delayed for 5552 bytes, so the modulo operation time is negligible. If the bytes are a, b, c, the second sum is 3a + 2b + c + 3, and so is position and order sensitive, unlike the first sum, which is just a checksum. That 65521 is prime is important to avoid a possible large class of two-byte errors that leave the check unchanged. (The Fletcher checksum uses 255, which is not prime and which also makes the Fletcher check insensitive to single byte changes 0 <-> 255.)

The sum s1 is initialized to 1 instead of zero to make the length of the sequence part of s2, so that the length does not have to be checked separately. (Any sequence of zeroes has a Fletcher checksum of zero.)

9 Appendix: Sample code

The following C code computes the Adler-32 checksum of a data buffer. It is written for clarity, not for speed. The sample code is in the ANSI C programming language. Non C users may find it easier to read with these hints:

```
&       Bitwise AND operator.
>>      Bitwise right shift operator. When applied to an
        unsigned quantity, as here, right shift inserts zero bit(s)
        at the left.
<<      Bitwise left shift operator. Left shift inserts zero
        bit(s) at the right.
++      "n++" increments the variable n.
%       modulo operator: a % b is the remainder of a divided by b.

#define BASE 65521 /* largest prime smaller than 65536 */

/*
   Update a running Adler-32 checksum with the bytes buf[0..len-1]
 and return the updated checksum. The Adler-32 checksum should be
 initialized to 1.

 Usage example:

   unsigned long adler = 1L;

   while (read_buffer(buffer, length) != EOF) {
     adler = update_adler32(adler, buffer, length);
   }
   if (adler != original_adler) error();
*/
unsigned long update_adler32(unsigned long adler,
   unsigned char *buf, int len)
{
  unsigned long s1 = adler & 0xffff;
  unsigned long s2 = (adler >> 16) & 0xffff;
  int n;
```

```
  for (n = 0; n < len; n++) {
    s1 = (s1 + buf[n]) % BASE;
    s2 = (s2 + s1)     % BASE;
  }
  return (s2 << 16) + s1;
}

/* Return the adler32 of the bytes buf[0..len-1] */

unsigned long adler32(unsigned char *buf, int len)
{
  return update_adler32(1L, buf, len);
}
```