Network Working Group
Request for Comments: 1952
Category: Informational

P. Deutsch
Aladdin Enterprises

May 1996

GZIP file format specification version 4.3

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 that is compatible with the widely used GZIP utility. The format includes a cyclic redundancy check value for detecting data corruption. The format presently uses the DEFLATE method of compression but can be easily extended to use other compression methods. The format can be implemented readily in a manner not covered by patents.

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 compress or decompress a data stream (as opposed to a randomly accessible file) to produce another 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;
- Compresses data with efficiency comparable to the best currently available general-purpose compression methods, and in particular considerably better than the "compress" program;
- Can be implemented readily in a manner not covered by patents, and hence can be practiced freely;
- Is compatible with the file format produced by the current widely used gzip utility, in that conforming decompressors will be able to read data produced by the existing gzip compressor.

The data format defined by this specification does not attempt to:

- Provide random access to compressed data;
- Compress specialized data (e.g., raster graphics) as well as the best currently available specialized algorithms.

1.2 Intended audience

This specification is intended for use by implementors of software to compress data into gzip format and/or decompress data from gzip 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 compression method and a file format (the latter assuming only that a file can store a sequence of arbitrary bytes). It does not specify any particular interface to a file system or anything about character sets or encodings (except for file names and comments, which are optional).

1.4 Compliance

Unless otherwise indicated below, a compliant decompressor must be able to accept and decompress any file that conforms to all the specifications presented here; a compliant compressor must produce files that conform to all the specifications presented here. The material in the appendices is not part of the specification per se and is not relevant to compliance.

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

There have been no technical changes to the gzip format since version 4.1 of this specification. In version 4.2, some terminology was changed, and the sample CRC code was rewritten for clarity and to eliminate the requirement for the caller to do pre- and post-conditioning. Version 4.3 is a conversion of the specification 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|
+--------+
```

This document does not address the issue of the order in which bits of a byte are transmitted on a bit-sequential medium, since the data format described here is byte- rather than bit-oriented.

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

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

2.2 File format

A gzip file consists of a series of "members" (compressed data sets). The format of each member is specified in the following section. The members simply appear one after another in the file, with no additional information before, between, or after them.

2.3 Member format

Each member has the following structure:

```
+---+---+---+---+---+---+---+---+---+---+
|ID1|ID2|CM |FLG|     MTIME     |XFL|OS | (more-->)
+---+---+---+---+---+---+---+---+---+---+
```

(if FLG.FEXTRA set)

```
+---+---+=================================+
| XLEN  |...XLEN bytes of "extra field"...| (more-->)
+---+---+=================================+
```

(if FLG.FNAME set)

```
+=========================================+
|...original file name, zero-terminated...| (more-->)
+=========================================+
```

(if FLG.FCOMMENT set)

```
+===================================+
|...file comment, zero-terminated...| (more-->)
+===================================+
```

(if FLG.FHCRC set)

```
+---+---+
| CRC16 |
+---+---+

+=======================+
|...compressed blocks...| (more-->)
+=======================+

  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
|     CRC32     |     ISIZE     |
+---+---+---+---+---+---+---+---+
```

2.3.1 Member header and trailer

ID1 (IDentification 1)

ID2 (IDentification 2)

These have the fixed values ID1 = 31 (0x1f, \037), ID2 = 139 (0x8b, \213), to identify the file as being in gzip format.

CM (Compression Method)

This identifies the compression method used in the file. CM = 0-7 are reserved. CM = 8 denotes the “deflate” compression method, which is the one customarily used by gzip and which is documented elsewhere.

FLG (FLaGs)

This flag byte is divided into individual bits as follows:

```
bit 0   FTEXT
bit 1   FHCRC
bit 2   FEXTRA
bit 3   FNAME
bit 4   FCOMMENT
bit 5   reserved
bit 6   reserved
bit 7   reserved
```

If FTEXT is set, the file is probably ASCII text. This is an optional indication, which the compressor may set by checking a small amount of the input data to see whether any non-ASCII characters are present. In case of doubt, FTEXT is cleared, indicating binary data. For systems which have different file formats for ascii text and binary data, the decompressor can use FTEXT to choose the appropriate format. We deliberately do not specify the algorithm used to set this bit, since a compressor always has the option of leaving it cleared and a decompressor always has the option of ignoring it and letting some other program handle issues of data conversion.

If FHCRC is set, a CRC16 for the gzip header is present, immediately before the compressed data. The CRC16 consists of the two least significant bytes of the CRC32 for all bytes of the gzip header up to and not including the CRC16. [The FHCRC bit was never set by versions of gzip up to 1.2.4, even though it was documented with a different meaning in gzip 1.2.4.]

If FEXTRA is set, optional extra fields are present, as described in a following section.

If FNAME is set, an original file name is present, terminated by a zero byte. The name must consist of ISO 8859-1 (LATIN-1) characters; on operating systems using EBCDIC or any other character set for file names, the name must be translated to the ISO LATIN-1 character set. This is the original name of the file being compressed, with any directory components removed, and, if the file being compressed is on a file system with case insensitive names, forced to lower case. There is no original file name if the data was compressed from a source other than a named file; for example, if the source was stdin on a Unix system, there is no file name.

If FCOMMENT is set, a zero-terminated file comment is present. This comment is not interpreted; it is only intended for human consumption. The comment must consist of ISO 8859-1 (LATIN-1) characters. Line breaks should be denoted by a single line feed character (10 decimal).

Reserved FLG bits must be zero.

MTIME (Modification TIME)

This gives the most recent modification time of the original file being compressed. The time is in Unix format, i.e., seconds since 00:00:00 GMT, Jan. 1, 1970. (Note that this may cause problems for MS-DOS and other systems that use local rather than Universal time.) If the compressed data did not come from a file, MTIME is set to the time at which compression started. MTIME = 0 means no time stamp is available.

XFL (eXtra FLags)

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

```
XFL = 2 - compressor used maximum compression,
          slowest algorithm
XFL = 4 - compressor used fastest algorithm
```

OS (Operating System)

This identifies the type of file system on which compression took place. This may be useful in determining end-of-line convention for text files. The currently defined values are as follows:

```
  0 - FAT filesystem (MS-DOS, OS/2, NT/Win32)
  1 - Amiga
  2 - VMS (or OpenVMS)
  3 - Unix
  4 - VM/CMS
  5 - Atari TOS
  6 - HPFS filesystem (OS/2, NT)
  7 - Macintosh
  8 - Z-System
  9 - CP/M
 10 - TOPS-20
 11 - NTFS filesystem (NT)
 12 - QDOS
 13 - Acorn RISCOS
255 - unknown
```

XLEN (eXtra LENgth)

If FLG.FEXTRA is set, this gives the length of the optional extra field. See below for details.

CRC32 (CRC-32)

This contains a Cyclic Redundancy Check value of the uncompressed data computed according to CRC-32 algorithm used in the ISO 3309 standard and in section 8.1.1.6.2 of ITU-T recommendation

V.42. (See http://www.iso.ch for ordering ISO documents. See gopher://info.itu.ch for an online version of ITU-T V.42.)

ISIZE (Input SIZE)

This contains the size of the original (uncompressed) input data modulo 2^32.

Extra field If the FLG.FEXTRA bit is set, an "extra field" is present in the header, with total length XLEN bytes. It consists of a series of subfields, each of the form:

```
+---+---+---+---+==================================+
|SI1|SI2|  LEN  |... LEN bytes of subfield data ...|
+---+---+---+---+==================================+
```

SI1 and SI2 provide a subfield ID, typically two ASCII letters with some mnemonic value. Jean-Loup Gailly <gzip@prep.ai.mit.edu> is maintaining a registry of subfield IDs; please send him any subfield ID you wish to use. Subfield IDs with SI2 = 0 are reserved for future use. The following IDs are currently defined:

```
SI1         SI2         Data
----------  ----------  ----
0x41 ('A')  0x70 ('P')  Apollo file type information
```

LEN gives the length of the subfield data, excluding the 4 initial bytes.

Compliance A compliant compressor must produce files with correct ID1, ID2, CM, CRC32, and ISIZE, but may set all the other fields in the fixed-length part of the header to default values (255 for OS, 0 for all others). The compressor must set all reserved bits to zero.

A compliant decompressor must check ID1, ID2, and CM, and provide an error indication if any of these have incorrect values. It must examine FEXTRA/XLEN, FNAME, FCOMMENT and FHCRC at least so it can skip over the optional fields if they are present. It need not examine any other part of the header or trailer; in particular, a decompressor may ignore FTEXT and OS and always produce binary output, and still be compliant. A compliant decompressor must give an error indication if any reserved bit is non-zero, since such a bit could indicate the presence of a new field that would cause subsequent data to be interpreted incorrectly.

3 References

[1] "Information Processing - 8-bit single-byte coded graphic character sets - Part 1: Latin alphabet No.1" (ISO 8859-1:1987). The ISO 8859-1 (Latin-1) character set is a superset of 7-bit ASCII. Files defining this character set are available as iso_8859-1.* in ftp://ftp.uu.net/graphics/png/documents/

[2] ISO 3309

[3] ITU-T recommendation V.42

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

[5] Gailly, J.-L., GZIP documentation, available as gzip-*.tar in ftp://prep.ai.mit.edu/pub/gnu/

[6] Sarwate, D.V., "Computation of Cyclic Redundancy Checks via Table Look-Up", Communications of the ACM, 31(8), pp.1008-1013.

[7] Schwaderer, W.D., "CRC Calculation", April 85 PC Tech Journal, pp.118-133.

[8] ftp://ftp.adelaide.edu.au/pub/rocksoft/papers/crc_v3.txt, describing the CRC concept.

4 Security Considerations

Any data compression method involves the reduction of redundancy in the data. Consequently, any corruption of the data is likely to have severe effects and be difficult to correct. Uncompressed text, on the other hand, will probably still be readable despite the presence of some corrupted bytes.

It is recommended that systems using this data format provide some means of validating the integrity of the compressed data, such as by setting and checking the CRC-32 check value.

5 Acknowledgements

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

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

6 Author's Address

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>
```


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>
```

7 Appendix: Jean-Loup Gailly's gzip utility

The most widely used implementation of gzip compression, and the original documentation on which this specification is based, were created by Jean-Loup Gailly <gzip@prep.ai.mit.edu>. Since this implementation is a de facto standard, we mention some more of its features here. Again, the material in this section is not part of the specification per se, and implementations need not follow it to be compliant.

When compressing or decompressing a file, gzip preserves the protection, ownership, and modification time attributes on the local file system, since there is no provision for representing protection attributes in the gzip file format itself. Since the file format includes a modification time, the gzip decompressor provides a command line switch that assigns the modification time from the file, rather than the local modification time of the compressed input, to the decompressed output.

8 Appendix: Sample CRC Code

The following sample code represents a practical implementation of the CRC (Cyclic Redundancy Check). (See also ISO 3309 and ITU-T V.42 for a formal specification.)

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 exclusive-OR operator.
>>      Bitwise right shift operator. When applied to an
        unsigned quantity, as here, right shift inserts zero
        bit(s) at the left.
!       Logical NOT operator.
++      "n++" increments the variable n.
0xNNN   0x introduces a hexadecimal (base 16) constant.
        Suffix L indicates a long value (at least 32 bits).

/* Table of CRCs of all 8-bit messages. */
unsigned long crc_table[256];

/* Flag: has the table been computed? Initially false. */
int crc_table_computed = 0;
```

```
/* Make the table for a fast CRC. */
void make_crc_table(void)
{
  unsigned long c;
  int n, k;

  for (n = 0; n < 256; n++) {
    c = (unsigned long) n;
    for (k = 0; k < 8; k++) {
      if (c & 1) {
        c = 0xedb88320L ^ (c >> 1);
      } else {
        c = c >> 1;
      }
    }
    crc_table[n] = c;
  }
  crc_table_computed = 1;
}

/*
   Update a running crc with the bytes buf[0..len-1] and return
 the updated crc. The crc should be initialized to zero. Pre- and
 post-conditioning (one's complement) is performed within this
 function so it shouldn't be done by the caller. Usage example:

   unsigned long crc = 0L;

   while (read_buffer(buffer, length) != EOF) {
     crc = update_crc(crc, buffer, length);
   }
   if (crc != original_crc) error();
*/
unsigned long update_crc(unsigned long crc,
                unsigned char *buf, int len)
{
  unsigned long c = crc ^ 0xffffffffL;
  int n;

  if (!crc_table_computed)
    make_crc_table();
  for (n = 0; n < len; n++) {
    c = crc_table[(c ^ buf[n]) & 0xff] ^ (c >> 8);
  }
  return c ^ 0xffffffffL;
}
```

```
/* Return the CRC of the bytes buf[0..len-1]. */
unsigned long crc(unsigned char *buf, int len)
{
  return update_crc(0L, buf, len);
}
```