

Internet Engineering Task Force (IETF)
INTERNET-DRAFT [draft-diaz-lzip-03](#)
Category: Informational
Expiration date: 2021-10-26

A. Diaz
GNU Project
April 2021

Lzip Compressed Format and the application/lzip Media Type

Abstract

Lzip is a lossless compressed data format designed for data sharing, long-term archiving, and parallel compression/decompression. Lzip uses a simplified form of the Lempel-Ziv-Markov chain-Algorithm (LZMA) stream format, chosen to maximize safety and interoperability. Lzip can achieve higher compression ratios than gzip. This document describes the lzip format and registers a media type and content encoding to be used when transporting lzip-compressed content via Multipurpose Internet Mail Extensions (MIME).

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are candidates for any level of Internet Standard; see [Section 2 of RFC 7841](#).

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at <http://www.rfc-editor.org/info/rfc<rfc-no>>.

Comments are solicited and should be addressed to the lzip's mailing list at lzip-bug@nongnu.org and/or the author.

[draft-diaz-lzip-03](#)

application/lzip

April 2021

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Internet-Draft Boilerplate

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#). Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

Table of Contents

1.	Introduction	3
1.1.	Purpose	3
1.2.	Compliance	4
2.	File Format	4
3.	Format of the LZMA stream in lzip files	5
3.1.	What is coded	6
3.2.	The coding contexts	8
3.3.	The range decoder	10
3.4.	Decoding and verifying the LZMA stream	10
3.5.	Compression	10
4.	IANA Considerations	10
4.1.	The 'application/lzip' Media Type	11

4.2.	Content Encoding	12
5.	Security Considerations	12
6.	References	13
6.1.	Normative References	13
6.2.	Informative References	13
Appendix A.	Reference Source Code	13
	Acknowledgments	23
	Author's Address	23

[1.](#) Introduction

Lzip is a lossless compressed data format similar to gzip [[RFC1952](#)]. Lzip is designed for data sharing, long-term archiving, parallel compression/decompression, and limited random access to the data. Lzip can achieve higher compression ratios than gzip.

Lzip uses a simplified form of the Lempel-Ziv-Markov chain-Algorithm (LZMA) stream format, chosen to maximize safety and interoperability. The original LZMA algorithm and stream format were developed by Igor Pavlov.

LZMA is much like deflate (the algorithm of gzip) with two main differences that account for its higher compression ratio. First, LZMA can use a dictionary size thousands of times larger than deflate. Second, LZMA uses a range encoder as its last stage instead of the less efficient (but faster) Huffman coding used by deflate.

[1.1.](#) Purpose

The purpose of this document is to define a lossless compressed data format that is a) independent of the CPU type, operating system, file system, and character set and b) is suitable for file compression and pipe and streaming compression, using the LZMA algorithm. The text of the specification assumes a basic background in programming at the level of bits and other primitive data representations.

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, and hence can be used in data communications.

The data format defined by this specification allows efficient

parallel compression/decompression, and random access to blocks of compressed data by means of multimember files and a distributed index.

This specification is intended for use by implementors of software to compress data into lzip format and/or decompress data from lzip format. The lzip format is supported by one free software reference implementation (the lzip tool) written in portable C++ (C++11), and by several free software implementations written in portable C (C99), all of them available at [[LZIP](#)]. The reference implementation has been in stable status since 2009, and is widely deployed.

Also, to enable the transport of a data object compressed with lzip, this document registers a media type that can be used to identify such content when it is used in a payload encoded using Multipurpose Internet Mail Extensions (MIME).

[1.2.](#) Compliance

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.

[2.](#) File Format

Perfection is reached, not when there is no longer anything to add, but when there is no longer anything to take away.

-- Antoine de Saint-Exupery

In the diagram below, a box like this:

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

represents one byte; a box like this:

```
+=====+
```

```

|           |
+=====+

```

represents a variable number of bytes.

A lzip file consists of a series of "members" (compressed data sets). The members simply appear one after another in the file, with no additional information before, between, or after them.

Each member has the following structure:

```

+---+---+---+---+---+---+---+=====+
| ID string | VN | DS | LZMA stream | (more-->)
+---+---+---+---+---+---+---+=====+

+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| CRC32 |   Data size   |   Member size   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

All multibyte values are stored in little endian order.

ID string (the "magic" bytes)

A four byte string, identifying the lzip format, with the value "LZIP" (0x4C, 0x5A, 0x49, 0x50).

VN (version number, 1 byte)

Just in case something needs to be modified in the future. 1 for now.

DS (coded dictionary size, 1 byte)

The dictionary size is calculated by taking a power of 2 (the base size) and subtracting from it a fraction between 0/16 and 7/16 of the base size.

Bits 4-0 contain the base 2 logarithm of the base size (12 to 29).

Bits 7-5 contain the numerator of the fraction (0 to 7) to subtract from the base size to obtain the dictionary size.

Example: $0xD3 = 2^{19} - 6 * 2^{15} = 512 \text{ KiB} - 6 * 32 \text{ KiB} = 320 \text{ KiB}$

Valid values for dictionary size range from 4 KiB to 512 MiB.

LZMA stream

The LZMA stream, finished by an end of stream marker. Uses default values for encoder properties (see [Section 3](#) for a complete description).

CRC32 (4 bytes)

Cyclic Redundancy Check (CRC) of the uncompressed original data.

Data size (8 bytes)

Size of the uncompressed original data.

Member size (8 bytes)

Total size of the member, including header and trailer. This field acts as a distributed index, allows the verification of stream integrity, and facilitates safe recovery of undamaged members from multimember files.

[3.](#) Format of the LZMA stream in lzip files

The LZMA algorithm has three parameters, called "special LZMA properties", to adjust it for some kinds of binary data. These parameters are; 'literal_context_bits' (with a default value of 3), 'literal_pos_state_bits' (with a default value of 0), and 'pos_state_bits' (with a default value of 2). As a general purpose compressor, lzip only uses the default values for these parameters. In particular 'literal_pos_state_bits' has been optimized away and does not even appear in the code.

Lzip finishes the LZMA stream with an "End Of Stream" (EOS) marker (the distance-length pair 0xFFFFFFFFU, 2), which in conjunction with the 'member size' field in the member trailer allows the verification of stream integrity. The LZMA stream in lzip files always has these two features (default properties and EOS marker) and is referred to in this document as LZMA-302eos. The EOS marker is the only marker allowed in lzip files. This variant of the LZMA stream format has been chosen to maximize safety and interoperability.

The second stage of LZMA is a range encoder that uses a different probability model for each type of symbol; distances, lengths, literal bytes, etc. Range encoding conceptually encodes all the

symbols of the message into one number. Unlike Huffman coding, which assigns to each symbol a bit-pattern and concatenates all the bit-patterns together, range encoding can compress one symbol to less than one bit. Therefore the compressed data produced by a range encoder can't be split in pieces that could be described individually.

It seems that the only way of describing the LZMA-302eos stream is describing the algorithm that decodes it. And given the many details about the range decoder that need to be described accurately, the source code of a real decoder seems the only appropriate reference to use.

What follows is a description of the decoding algorithm for LZMA-302eos streams using as reference the source code of "lzd", an educational decompressor for lzip files which can be downloaded from the lzip download directory. Lzd is written in C++11 and its source code is included in [appendix A](#).

[3.1](#). What is coded

The LZMA stream includes literals, matches, and repeated matches (matches reusing a recently used distance). There are 7 different coding sequences:

Bit sequence	Name	Description
0 + byte	literal	literal byte
1 + 0 + len + dis	match	distance-length pair
1 + 1 + 0 + 0	shortrep	1 byte match at latest used distance
1 + 1 + 0 + 1 + len	rep0	len bytes match at latest used distance
1 + 1 + 1 + 0 + len	rep1	len bytes match at second latest used distance
1 + 1 + 1 + 1 + 0 + len	rep2	len bytes match at third latest used distance
1 + 1 + 1 + 1 + 1 + len	rep3	len bytes match at fourth latest used distance

In the following tables, multibit sequences are coded in normal order, from most significant bit (MSB) to least significant bit (LSB), except where noted otherwise.

[draft-diaz-lzip-03](#)

application/lzip

April 2021

Lengths (the 'len' in the table above) are coded as follows:

Bit sequence	Description
-----	-----
0 + 3 bits	lengths from 2 to 9
1 + 0 + 3 bits	lengths from 10 to 17
1 + 1 + 8 bits	lengths from 18 to 273

The coding of distances is a little more complicated, so I'll begin by explaining a simpler version of the encoding.

Imagine you need to encode a number from 0 to $2^{32} - 1$, and you want to do it in a way that produces shorter codes for the smaller numbers. You may first encode the position of the most significant bit that is set to 1, which you may find by making a bit scan from the left (from the MSB). A position of 0 means that the number is 0 (no bit is set), 1 means the LSB is the first bit set (the number is 1), and 32 means the MSB is set (i.e., the number is $\geq 0x80000000$). Then, if the position is ≥ 2 , you encode the remaining position - 1 bits. Let's call these bits "direct bits" because they are coded directly by value instead of indirectly by position.

The inconvenient of this simple method is that it needs 6 bits to encode the position, but it just uses 33 of the 64 possible values, wasting almost half of the codes.

The intelligent trick of LZMA is that it encodes in what it calls a "slot" the position of the most significant bit set, along with the value of the next bit, using the same 6 bits that would take to encode the position alone. This seems to need 66 slots (twice the number of positions), but for positions 0 and 1 there is no next bit, so the number of slots needed is 64 (0 to 63).

The 6 bits representing this "slot number" are then context-coded. If the distance is ≥ 4 , the remaining bits are encoded as follows. 'direct_bits' is the amount of remaining bits (from 1 to 30) needed to form a complete distance, and is calculated as $(\text{slot} \gg 1) - 1$. If a distance needs 6 or more direct_bits, the last 4 bits are encoded separately. The last piece (all the direct_bits for distances 4 to 127, or the last 4 bits for distances ≥ 128) is context-coded in reverse order (from LSB to MSB). For distances ≥ 128 , the 'direct_bits - 4' part is encoded with fixed 0.5

probability.

Bit sequence	Description
-----	-----
slot	distances from 0 to 3
slot + direct_bits	distances from 4 to 127
slot + (direct_bits - 4) + 4 bits	distances from 128 to $2^{32} - 1$

[3.2.](#) The coding contexts

These contexts ('Bit_model' in the source), are integers or arrays of integers representing the probability of the corresponding bit being 0.

The indices used in these arrays are:

state

A state machine ('State' in the source) with 12 states (0 to 11), coding the latest 2 to 4 types of sequences processed. The initial state is 0.

pos_state

Value of the 2 least significant bits of the current position in the decoded data.

literal_state

Value of the 3 most significant bits of the latest byte decoded.

len_state

Coded value of the current match length (length - 2), with a maximum of 3. The resulting value is in the range 0 to 3.

In the following table, '!literal' is any sequence except a literal byte. 'rep' is any one of 'rep0', 'rep1', 'rep2', or 'rep3'. The types of previous sequences corresponding to each state are:

State	Types of previous sequences
-----	-----
0	literal, literal, literal
1	match, literal, literal
2	rep or (!literal, shortrep), literal, literal

```

3      literal, shortrep, literal, literal
4      match, literal
5      rep or (!literal, shortrep), literal
6      literal, shortrep, literal
7      literal, match
8      literal, rep
9      literal, shortrep
10     !literal, match
11     !literal, (rep or shortrep)

```

The contexts for decoding the type of coding sequence are:

Name	Indices	Used when
-----	-----	-----
bm_match	state, pos_state	sequence start
bm_rep	state	after sequence 1
bm_rep0	state	after sequence 11
bm_rep1	state	after sequence 111
bm_rep2	state	after sequence 1111
bm_len	state, pos_state	after sequence 110

The contexts for decoding distances are:

Name	Indices	Used when
-----	-----	-----
bm_dis_slot	len_state, bit tree	distance start
bm_dis	reverse bit tree	after slots 4 to 13
bm_align	reverse bit tree	for distances >= 128, after fixed probability bits

There are two separate sets of contexts for lengths ('Len_model' in

the source). One for normal matches, the other for repeated matches. The contexts in each Len_model are (see 'decode_len' in the source):

Name	Indices	Used when
-----	-----	-----
choice1	none	length start
choice2	none	after sequence 1
bm_low	pos_state, bit tree	after sequence 0
bm_mid	pos_state, bit tree	after sequence 10
bm_high	bit tree	after sequence 11

The context array 'bm_literal' is special. In principle it acts as a normal bit tree context, the one selected by 'literal_state'. But if the previous decoded byte was not a literal, two other bit tree contexts are used depending on the value of each bit in 'match_byte' (the byte at the latest used distance), until a bit is decoded that is different from its corresponding bit in 'match_byte'. After the first difference is found, the rest of the byte is decoded using the normal bit tree context. (See 'decode_matched' in the source).

[3.3.](#) The range decoder

The LZMA stream is consumed one byte at a time by the range decoder. (See 'normalize' in the source). Every byte consumed produces a variable number of decoded bits, depending on how well these bits agree with their context. (See 'decode_bit' in the source).

The range decoder state consists of two unsigned 32-bit variables; 'range' (representing the most significant part of the range size not yet decoded), and 'code' (representing the current point within 'range'). 'range' is initialized to $2^{32} - 1$, and 'code' is initialized to 0.

The range encoder produces a first 0 byte that must be ignored by the range decoder. This is done by shifting 5 bytes in the initialization of 'code' instead of 4. (See the 'Range_decoder' constructor in the source).

[3.4.](#) Decoding and verifying the LZMA stream

After decoding the member header and obtaining the dictionary size, the range decoder is initialized and then the LZMA decoder enters a loop (see 'decode_member' in the source) where it invokes the range decoder with the appropriate contexts to decode the different coding sequences (matches, repeated matches, and literal bytes), until the "End Of Stream" marker is decoded.

Once the "End Of Stream" marker has been decoded, the decompressor must read and decode the member trailer, and verify that the three integrity factors (CRC, data size, and member size) match those calculated by the LZMA decoder.

[3.5.](#) Compression

Compression consists in describing the uncompressed data as a succession of coding sequences from the set shown in [Section 3.1](#), and then encoding them using a range encoder. The fast encoder in the reference implementation shows how this can be done in almost the simplest way possible; issuing the longest match found, or a literal byte if no match is found, and repeating until all the data have been compressed. More sophisticated choosing of the coding sequences may achieve higher compression ratios.

[4.](#) IANA Considerations

IANA is asked to make two registrations, as described below.

[4.1.](#) The 'application/lzip' Media Type

The 'application/lzip' media type identifies a block of data that is

compressed using `gzip` compression. The data is a stream of bytes as described in this document. IANA is asked to add the following entry to the standards tree of the "Media Types" registry:

Type name: `application`

Subtype name: `gzip`

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: `binary`

Security considerations: See [Section 5](#) of this RFC

Interoperability considerations: N/A

Published specification: this RFC

Applications that use this media type:

Any application where data size is an issue

Fragment Identifier Considerations: N/A

Restrictions on Usage: N/A

Provisional Registrations: N/A

Additional information:

Deprecated alias names for this type: `application/x-gzip`

Magic number(s): First 4 bytes (`0x4C`, `0x5A`, `0x49`, `0x50`)

File extension(s): `.gz` `.tgz`

Macintosh File Type Code(s): N/A

Object Identifier(s) or OID(s): N/A

Intended Usage: `COMMON`

Other Information & Comments: See [\[LZIP\]](#)

Author: Antonio Diaz Diaz

Change Controller: IETF

[draft-diaz-lzip-03](#)

application/lzip

April 2021

[4.2.](#) Content Encoding

IANA is asked to add the following entry to the "HTTP Content Coding Registry" within the "Hypertext Transfer Protocol (HTTP) Parameters" registry:

Name: lzip

Description: A stream of bytes compressed using lzip compression

Pointer to specification text: this RFC

[5.](#) Security Considerations

Lzip is a compressed format. Decompression of the data may expand it to a size more than 7000 times larger, risking an out-of-memory or out-of-disc-space condition. Since both the gzip and lzip formats contain a single data stream, any steps already taken to avoid such attacks on application/gzip should also work on application/lzip. One possible measure, already implemented in some applications, is to set a limit to the decompressed size and stop the decompression or warn the user if the limit is surpassed.

This media type does not employ any kind of "active content", but it can be used to compress any other media type (for example application/postscript) which could then be interpreted by the application.

A lzip stream does not store any metadata. Any data appended to the end of the stream is easily detected, and an error can be signaled. It is not apparent how this media type could be used to help violate a recipient's privacy.

The lzip media type does not need itself any external security mechanisms. But again, it can be used to compress other media types requiring them (for example a media type defined for storage of sensitive medical information).

Because of the nature of the decoding algorithm used by lzip, it is easy to protect the decoder from invalid memory accesses caused by corruption in the input data (intentional or not). Size limits need to be checked at just one place in the decompressor (the decoding of rep0) to prevent buffer overflows. This inherent safety has been

extensively tested in the reference implementation and makes possible the unique data recovery capabilities provided by lziprecover.

The lzip data stream does not contain any size fields that could be faked to be smaller than the actual decompressed data in an attempt to trigger a buffer overflow. (The 'data size' field in the lzip trailer is only used to verify the size of the data produced as an error detection measure).

Diaz

Informational

[Page 12]

[draft-diaz-lzip-03](#)

application/lzip

April 2021

[6.](#) References

[6.1.](#) Normative References

[LZIP] Diaz, A., "Lzip", <<http://www.nongnu.org/lzip/lzip.html>>.

[6.2.](#) Informative References

[RFC1952] Deutsch, P., "GZIP file format specification version 4.3", [RFC 1952](#), DOI 10.17487/RFC1952, May 1996, <<http://www.rfc-editor.org/info/rfc1952>>.

[Appendix A.](#) Reference Source Code

<CODE BEGINS>

```
/* Lzd - Educational decompressor for the lzip format
   Copyright (C) 2013-2021 Antonio Diaz Diaz.
```

This program is free software. Redistribution and use in source and binary forms, with or without modification, are permitted provided

that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

*/

/*

Exit status: 0 for a normal exit, 1 for environmental problems (file not found, invalid flags, I/O errors, etc), 2 to indicate a corrupt or invalid input file.

*/

Diaz

Informational

[Page 13]

[draft-diaz-lzip-03](#)

application/lzip

April 2021

```
#include <algorithm>
#include <cerrno>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <stdint.h>
#include <unistd.h>
#if defined(__MSVCRT__) || defined(__OS2__) || defined(__DJGPP__)
#include <fcntl.h>
#include <io.h>
#endif
#define PROGVERSION "1.2"
```

```
class State
{
    int st;

public:
    enum { states = 12 };
    State() : st( 0 ) {}
    int operator()() const { return st; }
    bool is_char() const { return st < 7; }
```



```

void set_char()
{
    const int next[states] = { 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 4, 5 };
    st = next[st];
}
void set_match()      { st = ( st < 7 ) ? 7 : 10; }
void set_rep()        { st = ( st < 7 ) ? 8 : 11; }
void set_short_rep() { st = ( st < 7 ) ? 9 : 11; }
};

```

```

enum {
    min_dictionary_size = 1 << 12,
    max_dictionary_size = 1 << 29,
    literal_context_bits = 3,
    literal_pos_state_bits = 0,                      // not used
    pos_state_bits = 2,
    pos_states = 1 << pos_state_bits,
    pos_state_mask = pos_states - 1,

    len_states = 4,
    dis_slot_bits = 6,
    start_dis_model = 4,
    end_dis_model = 14,
    modeled_distances = 1 << ( end_dis_model / 2 ),    // 128
    dis_align_bits = 4,
    dis_align_size = 1 << dis_align_bits,

```

```

len_low_bits = 3,
len_mid_bits = 3,
len_high_bits = 8,
len_low_symbols = 1 << len_low_bits,
len_mid_symbols = 1 << len_mid_bits,
len_high_symbols = 1 << len_high_bits,
max_len_symbols = len_low_symbols+len_mid_symbols+len_high_symbols,

min_match_len = 2,                      // must be 2

bit_model_move_bits = 5,
bit_model_total_bits = 11,
bit_model_total = 1 << bit_model_total_bits };

```

```

struct Bit_model
{
    int probability;
    Bit_model() : probability( bit_model_total / 2 ) {}
};

struct Len_model
{
    Bit_model choice1;
    Bit_model choice2;
    Bit_model bm_low[pos_states][len_low_symbols];
    Bit_model bm_mid[pos_states][len_mid_symbols];
    Bit_model bm_high[len_high_symbols];
};

class CRC32
{
    uint32_t data[256];          // Table of CRCs of all 8-bit messages.

public:
    CRC32()
    {
        for( unsigned n = 0; n < 256; ++n )
        {
            unsigned c = n;
            for( int k = 0; k < 8; ++k )
                { if( c & 1 ) c = 0xEDB88320U ^ ( c >> 1 ); else c >>= 1; }
            data[n] = c;
        }
    }

    void update_buf( uint32_t & crc, const uint8_t * const buffer,
                    const int size ) const
    {
        for( int i = 0; i < size; ++i )
            crc = data[(crc^buffer[i])&0xFF] ^ ( crc >> 8 );
    }
};

```

```

    }
};

```

```

const CRC32 crc32;

typedef uint8_t Lzip_header[6];           // 0-3 magic bytes
                                           // 4 version
                                           // 5 coded dictionary size
typedef uint8_t Lzip_trailer[20];
      // 0-3 CRC32 of the uncompressed data
      // 4-11 size of the uncompressed data
      // 12-19 member size including header and trailer

class Range_decoder
{
    unsigned long long member_pos;
    uint32_t code;
    uint32_t range;

public:
    Range_decoder() : member_pos( 6 ), code( 0 ), range( 0xFFFFFFFFU )
    {
        for( int i = 0; i < 5; ++i ) code = ( code << 8 ) | get_byte();
    }

    uint8_t get_byte() { ++member_pos; return std::getc( stdin ); }
    unsigned long long member_position() const { return member_pos; }

    unsigned decode( const int num_bits )
    {
        unsigned symbol = 0;
        for( int i = num_bits; i > 0; --i )
        {
            range >>= 1;
            symbol <<= 1;
            if( code >= range ) { code -= range; symbol |= 1; }
            if( range <= 0x00FFFFFFU ) // normalize
                { range <<= 8; code = ( code << 8 ) | get_byte(); }
        }
        return symbol;
    }

    unsigned decode_bit( Bit_model & bm )
    {
        unsigned symbol;
        const uint32_t bound =
            ( range >> bit_model_total_bits ) * bm.probability;
        if( code < bound )
        {
            range = bound;

```

[draft-diaz-lzip-03](#)

application/lzip

April 2021

```
        bm.probability +=
            ( bit_model_total - bm.probability ) >> bit_model_move_bits;
        symbol = 0;
    }
    else
    {
        range -= bound;
        code -= bound;
        bm.probability -= bm.probability >> bit_model_move_bits;
        symbol = 1;
    }
    if( range <= 0x00FFFFFFU ) // normalize
    { range <<= 8; code = ( code << 8 ) | get_byte(); }
    return symbol;
}

unsigned decode_tree( Bit_model bm[], const int num_bits )
{
    unsigned symbol = 1;
    for( int i = 0; i < num_bits; ++i )
        symbol = ( symbol << 1 ) | decode_bit( bm[symbol] );
    return symbol - ( 1 << num_bits );
}

unsigned decode_tree_reversed( Bit_model bm[], const int num_bits )
{
    unsigned symbol = decode_tree( bm, num_bits );
    unsigned reversed_symbol = 0;
    for( int i = 0; i < num_bits; ++i )
    {
        reversed_symbol = ( reversed_symbol << 1 ) | ( symbol & 1 );
        symbol >>= 1;
    }
    return reversed_symbol;
}

unsigned decode_matched( Bit_model bm[], const unsigned match_byte )
{
    unsigned symbol = 1;
    for( int i = 7; i >= 0; --i )
    {
        const unsigned match_bit = ( match_byte >> i ) & 1;
        const unsigned bit = decode_bit( bm[symbol+(match_bit<<8)+0x100] );
```

```

symbol = ( symbol << 1 ) | bit;
if( match_bit != bit )
{
    while( symbol < 0x100 )
        symbol = ( symbol << 1 ) | decode_bit( bm[symbol] );
    break;
}
}

```

```

    return symbol & 0xFF;
}

unsigned decode_len( Len_model & lm, const int pos_state )
{
    if( decode_bit( lm.choice1 ) == 0 )
        return decode_tree( lm.bm_low[pos_state], len_low_bits );
    if( decode_bit( lm.choice2 ) == 0 )
        return len_low_symbols +
            decode_tree( lm.bm_mid[pos_state], len_mid_bits );
    return len_low_symbols + len_mid_symbols +
        decode_tree( lm.bm_high, len_high_bits );
}
};

```

```

class LZ_decoder
{
    unsigned long long partial_data_pos;
    Range_decoder rdec;
    const unsigned dictionary_size;
    uint8_t * const buffer;          // output buffer
    unsigned pos;                    // current pos in buffer
    unsigned stream_pos;             // first byte not yet written to stdout
    uint32_t crc_;
    bool pos_wrapped;

    void flush_data();

    uint8_t peek( const unsigned distance ) const
    {
        if( pos > distance ) return buffer[pos - distance - 1];
        if( pos_wrapped ) return buffer[dictionary_size+pos-distance-1];
    }
}

```

```

        return 0;                                // prev_byte of first byte
    }

    void put_byte( const uint8_t b )
    {
        buffer[pos] = b;
        if( ++pos >= dictionary_size ) flush_data();
    }

public:
    explicit LZ_decoder( const unsigned dict_size )
        :
        partial_data_pos( 0 ),
        dictionary_size( dict_size ),
        buffer( new uint8_t[dictionary_size] ),
        pos( 0 ),
        stream_pos( 0 ),
        crc_( 0xFFFFFFFFFU ),

```

Diaz

Informational

[Page 18]

[draft-diaz-lzip-03](#)

application/lzip

April 2021

```

        pos_wrapped( false )
    {}

    ~LZ_decoder() { delete[] buffer; }

    unsigned crc() const { return crc_ ^ 0xFFFFFFFFFU; }
    unsigned long long data_position() const
    { return partial_data_pos + pos; }
    uint8_t get_byte() { return rdec.get_byte(); }
    unsigned long long member_position() const
    { return rdec.member_position(); }

    bool decode_member();
};

void LZ_decoder::flush_data()
{
    if( pos > stream_pos )
    {
        const unsigned size = pos - stream_pos;
        crc32.update_buf( crc_, buffer + stream_pos, size );
        if( std::fwrite( buffer + stream_pos, 1, size, stdout ) != size )

```

```

        { std::fprintf(stderr, "Write error: %s\n", std::strerror(errno));
          std::exit( 1 ); }
    if( pos >= dictionary_size )
        { partial_data_pos += pos; pos = 0; pos_wrapped = true; }
    stream_pos = pos;
}
}

```

```

bool LZ_decoder::decode_member()          // Returns false if error
{
    Bit_model bm_literal[1<<literal_context_bits][0x300];
    Bit_model bm_match[State::states][pos_states];
    Bit_model bm_rep[State::states];
    Bit_model bm_rep0[State::states];
    Bit_model bm_rep1[State::states];
    Bit_model bm_rep2[State::states];
    Bit_model bm_len[State::states][pos_states];
    Bit_model bm_dis_slot[len_states][1<<dis_slot_bits];
    Bit_model bm_dis[modeled_distances-end_dis_model+1];
    Bit_model bm_align[dis_align_size];
    Len_model match_len_model;
    Len_model rep_len_model;
    unsigned rep0 = 0;          // rep[0-3] latest four distances
    unsigned rep1 = 0;          // used for efficient coding of
    unsigned rep2 = 0;          // repeated distances
    unsigned rep3 = 0;
}

```

State state;

```

while( !std::feof( stdin ) && !std::ferror( stdin ) )
{
    const int pos_state = data_position() & pos_state_mask;
    if( rdec.decode_bit( bm_match[state()][pos_state] ) == 0 )// 1st bit
    {
        // literal byte
        const uint8_t prev_byte = peek( 0 );
        const int literal_state = prev_byte >> ( 8 - literal_context_bits);
        Bit_model * const bm = bm_literal[literal_state];
        if( state.is_char() )
            put_byte( rdec.decode_tree( bm, 8 ) );
    }
}

```

```

else
    put_byte( rdec.decode_matched( bm, peek( rep0 ) ) );
    state.set_char();
    continue;
}
// match or repeated match
int len;
if( rdec.decode_bit( bm_rep[state()] ) != 0 )           // 2nd bit
{
    if( rdec.decode_bit( bm_rep0[state()] ) == 0 )       // 3rd bit
    {
        if( rdec.decode_bit( bm_len[state()][pos_state] ) == 0 ) // 4th bit
        { state.set_short_rep(); put_byte( peek( rep0 ) ); continue; }
    }
    else
    {
        unsigned distance;
        if( rdec.decode_bit( bm_rep1[state()] ) == 0 )   // 4th bit
            distance = rep1;
        else
        {
            if( rdec.decode_bit( bm_rep2[state()] ) == 0 ) // 5th bit
                distance = rep2;
            else
                { distance = rep3; rep3 = rep2; }
            rep2 = rep1;
        }
        rep1 = rep0;
        rep0 = distance;
    }
    state.set_rep();
    len = min_match_len + rdec.decode_len( rep_len_model, pos_state );
}
else // match
{
    rep3 = rep2; rep2 = rep1; rep1 = rep0;
    len = min_match_len + rdec.decode_len( match_len_model, pos_state );
    const int len_state = std::min( len-min_match_len, len_states-1 );
}

```

```

rep0 = rdec.decode_tree( bm_dis_slot[len_state], dis_slot_bits );
if( rep0 >= start_dis_model )
{

```



```

const unsigned dis_slot = rep0;
const int direct_bits = ( dis_slot >> 1 ) - 1;
rep0 = ( 2 | ( dis_slot & 1 ) ) << direct_bits;
if( dis_slot < end_dis_model )
    rep0 += rdec.decode_tree_reversed( bm_dis + (rep0 - dis_slot),
                                       direct_bits );
else
{
    rep0 +=
        rdec.decode(direct_bits - dis_align_bits) << dis_align_bits;
    rep0 += rdec.decode_tree_reversed( bm_align, dis_align_bits );
    if( rep0 == 0xFFFFFFFFU )                // marker found
    {
        flush_data();
        return ( len == min_match_len );    // End Of Stream marker
    }
}
}
state.set_match();
if( rep0 >= dictionary_size || ( rep0 >= pos && !pos_wrapped ) )
{ flush_data(); return false; }
}
for( int i = 0; i < len; ++i ) put_byte( peek( rep0 ) );
}
flush_data();
return false;
}

```

```

int main( const int argc, const char * const argv[] )
{
    if( argc > 2 || ( argc == 2 && std::strcmp( argv[1], "-d" ) != 0 ) )
    {
        std::printf(
            "Lzd %s - Educational decompressor for the lzip format.\n"
            "Study the source to learn how a lzip decompressor works.\n"
            "See the lzip manual for an explanation of the code.\n"
            "\nUsage: %s [-d] < file.lz > file\n"
            "Lzd decompresses from standard input to standard output.\n"
            "\nCopyright (C) 2021 Antonio Diaz Diaz.\n"
            "License 2-clause BSD.\n"
            "This is free software: you are free to change and redistribute "\
            "it.\nThere is NO WARRANTY, to the extent permitted by law.\n"
            "Report bugs to lzip-bug@nongnu.org\n"
            "Lzd home page: http://www.nongnu.org/lzip/lzd.html\n",
            PROGVERSION, argv[0] );
        return 0;
    }
}

```

[draft-diaz-lzip-03](#)

application/lzip

April 2021

```
#if defined(__MSVCRT__) || defined(__OS2__) || defined(__DJGPP__)
    setmode( STDIN_FILENO, O_BINARY );
    setmode( STDOUT_FILENO, O_BINARY );
#endif

for( bool first_member = true; ; first_member = false )
{
    Lzip_header header;                // verify header
    for( int i = 0; i < 6; ++i ) header[i] = std::getc( stdin );
    if( std::feof( stdin ) || std::memcmp( header, "LZIP\x01", 5 ) != 0 )
    {
        if( first_member )
        { std::fputs( "Bad magic number (file not in lzip format).\n",
                      stderr ); return 2; }

        break;                        // ignore trailing data
    }
    unsigned dict_size = 1 << ( header[5] & 0x1F );
    dict_size -= ( dict_size / 16 ) * ( ( header[5] >> 5 ) & 7 );
    if( dict_size < min_dictionary_size || dict_size > max_dictionary_size )
    { std::fputs( "Invalid dictionary size in member header.\n",
                  stderr ); return 2; }

    LZ_decoder decoder( dict_size );    // decode LZMA stream
    if( !decoder.decode_member() )
    { std::fputs( "Data error\n", stderr ); return 2; }

    Lzip_trailer trailer;               // verify trailer
    for( int i = 0; i < 20; ++i ) trailer[i] = decoder.get_byte();
    int retval = 0;
    unsigned crc = 0;
    for( int i = 3; i >= 0; --i ) crc = ( crc << 8 ) + trailer[i];
    if( crc != decoder.crc() )
    { std::fputs( "CRC mismatch\n", stderr ); retval = 2; }

    unsigned long long data_size = 0;
    for( int i = 11; i >= 4; --i )
        data_size = ( data_size << 8 ) + trailer[i];
    if( data_size != decoder.data_position() )
    { std::fputs( "Data size mismatch\n", stderr ); retval = 2; }

    unsigned long long member_size = 0;
    for( int i = 19; i >= 12; --i )
        member_size = ( member_size << 8 ) + trailer[i];
}
```

```
if( member_size != decoder.member_position() )
    { std::fputs( "Member size mismatch\n", stderr ); retval = 2; }
if( retval ) return retval;
}

if( std::fclose( stdout ) != 0 )
    { std::fprintf( stderr, "Error closing stdout: %s\n",
                    std::strerror( errno ) ); return 1; }
```

Diaz

Informational

[Page 22]

[draft-diaz-lzip-03](#)

application/lzip

April 2021

```
    return 0;
}
<CODE ENDS>
```

Acknowledgments

The ideas embodied in lzip are due to (at least) the following people: Abraham Lempel and Jacob Ziv (for the LZ algorithm), Andrey Markov (for the definition of Markov chains), G.N.N. Martin (for the definition of range encoding), and Igor Pavlov (for putting all the above together in LZMA).

Author's Address

Antonio Diaz Diaz
GNU Project
Email: antonio@gnu.org

Expiration date: 2021-10-26

