# Lzlib

Compression library for the lzip format
for Lzlib version 1.8, 17 May 2016

**by Antonio Diaz Diaz**

# Table of Contents

This manual is for Lzlib (version 1.8, 17 May 2016).

# 1 Introduction

Lzlib is a data compression library providing in-memory LZMA compression and decompression functions, including integrity checking of the decompressed data. The compressed data format used by the library is the lzip format. Lzlib is written in C.

The lzip file format is designed for data sharing and long-term archiving, taking into account both data integrity and decoder availability:

- The lzip format provides very safe integrity checking and some data recovery means. The lziprecover program can repair bit-flip errors (one of the most common forms of data corruption) in lzip files, and provides data recovery capabilities, including error-checked merging of damaged copies of a file. See Section "Data safety" in `lziprecover`.
- The lzip format is as simple as possible (but not simpler). The lzip manual provides the code of a simple decompressor along with a detailed explanation of how it works, so that with the only help of the lzip manual it would be possible for a digital archaeologist to extract the data from a lzip file long after quantum computers eventually render LZMA obsolete.
- Additionally the lzip reference implementation is copylefted, which guarantees that it will remain free forever.

A nice feature of the lzip format is that a corrupt byte is easier to repair the nearer it is from the beginning of the file. Therefore, with the help of lziprecover, losing an entire archive just because of a corrupt byte near the beginning is a thing of the past.

The functions and variables forming the interface of the compression library are declared in the file '`lzlib.h`'. Usage examples of the library are given in the files '`main.c`' and '`bbexample.c`' from the source distribution.

Compression/decompression is done by repeatedly calling a couple of read/write functions until all the data have been processed by the library. This interface is safer and less error prone than the traditional zlib interface.

Compression/decompression is done when the read function is called. This means the value returned by the position functions will not be updated until a read call, even if a lot of data is written. If you want the data to be compressed in advance, just call the read function with a *size* equal to 0.

If all the data to be compressed are written in advance, lzlib will automatically adjust the header of the compressed data to use the smallest possible dictionary size. This feature reduces the amount of memory needed for decompression and allows minilzip to produce identical compressed output as lzip.

Lzlib will correctly decompress a data stream which is the concatenation of two or more compressed data streams. The result is the concatenation of the corresponding decompressed data streams. Integrity testing of concatenated compressed data streams is also supported.

All the library functions are thread safe. The library does not install any signal handler. The decoder checks the consistency of the compressed data, so the library should never crash even in case of corrupted input.

In spite of its name (Lempel-Ziv-Markov chain-Algorithm), LZMA is not a concrete algorithm; it is more like "any algorithm using the LZMA coding scheme". For example,

the option '`-0`' of lzip uses the scheme in almost the simplest way possible; issuing the longest match it can find, or a literal byte if it can't find a match. Inversely, a much more elaborated way of finding coding sequences of minimum size than the one currently used by lzip could be developed, and the resulting sequence could also be coded using the LZMA coding scheme.

Lzlib currently implements two variants of the LZMA algorithm; fast (used by option '`-0`' of minilzip) and normal (used by all other compression levels).

The high compression of LZMA comes from combining two basic, well-proven compression ideas: sliding dictionaries (LZ77/78) and markov models (the thing used by every compression algorithm that uses a range encoder or similar order-0 entropy coder as its last stage) with segregation of contexts according to what the bits are used for.

The ideas embodied in lzlib 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), Igor Pavlov (for putting all the above together in LZMA), and Julian Seward (for bzip2's CLI).

# 2 Library version

`const char * LZ_version` ( *void* ) [Function]
Returns the library version as a string.

`const char * LZ_version_string` [Constant]
This constant is defined in the header file '`lzlib.h`'.

The application should compare LZ_version and LZ_version_string for consistency. If the first character differs, the library code actually used may be incompatible with the '`lzlib.h`' header file used by the application.

```
if( LZ_version()[0] != LZ_version_string[0] )
  error( "bad library version" );
```

# 3 Buffering

Lzlib internal functions need access to a memory chunk at least as large as the dictionary size (sliding window). For efficiency reasons, the input buffer for compression is twice or sixteen times as large as the dictionary size.

Finally, for safety reasons, lzlib uses two more internal buffers.

These are the four buffers used by lzlib, and their guaranteed minimum sizes:

- Input compression buffer. Written to by the '`LZ_compress_write`' function. For the normal variant of LZMA, its size is two times the dictionary size set with the '`LZ_compress_open`' function or 64 KiB, whichever is larger. For the fast variant, its size is 1 MiB.
- Output compression buffer. Read from by the '`LZ_compress_read`' function. Its size is 64 KiB.
- Input decompression buffer. Written to by the '`LZ_decompress_write`' function. Its size is 64 KiB.
- Output decompression buffer. Read from by the '`LZ_decompress_read`' function. Its size is the dictionary size set in the header of the member currently being decompressed or 64 KiB, whichever is larger.

# 4 Parameter limits

These functions provide minimum and maximum values for some parameters. Current values are shown in square brackets.

`int LZ_min_dictionary_bits ( void )` [Function]
Returns the base 2 logarithm of the smallest valid dictionary size [12].

`int LZ_min_dictionary_size ( void )` [Function]
Returns the smallest valid dictionary size [4 KiB].

`int LZ_max_dictionary_bits ( void )` [Function]
Returns the base 2 logarithm of the largest valid dictionary size [29].

`int LZ_max_dictionary_size ( void )` [Function]
Returns the largest valid dictionary size [512 MiB].

`int LZ_min_match_len_limit ( void )` [Function]
Returns the smallest valid match length limit [5].

`int LZ_max_match_len_limit ( void )` [Function]
Returns the largest valid match length limit [273].

# 5 Compression functions

These are the functions used to compress data. In case of error, all of them return -1 or 0, for signed and unsigned return values respectively, except '`LZ_compress_open`' whose return value must be verified by calling '`LZ_compress_errno`' before using it.

`struct LZ_Encoder * LZ_compress_open` ( *const int* `dictionary_size`, *const int* `match_len_limit`, *const unsigned long long* `member_size` ) [Function]

Initializes the internal stream state for compression and returns a pointer that can only be used as the *encoder* argument for the other LZ_compress functions, or a null pointer if the encoder could not be allocated.

The returned pointer must be verified by calling '`LZ_compress_errno`' before using it. If '`LZ_compress_errno`' does not return '`LZ_ok`', the returned pointer must not be used and should be freed with '`LZ_compress_close`' to avoid memory leaks.

*dictionary_size* sets the dictionary size to be used, in bytes. Valid values range from 4 KiB to 512 MiB. Note that dictionary sizes are quantized. If the specified size does not match one of the valid sizes, it will be rounded upwards by adding up to (*dictionary_size* / 8) to it.

*match_len_limit* sets the match length limit in bytes. Valid values range from 5 to 273. Larger values usually give better compression ratios but longer compression times.

If *dictionary_size* is 65535 and *match_len_limit* is 16, the fast variant of LZMA is chosen, which produces identical compressed output as `lzip -0`. (The dictionary size used will be rounded upwards to 64 KiB).

*member_size* sets the member size limit in bytes. Minimum member size limit is 100 kB. Small member size may degrade compression ratio, so use it only when needed. To produce a single-member data stream, give *member_size* a value larger than the amount of data to be produced, for example INT64_MAX.

`int LZ_compress_close` ( *struct LZ_Encoder * const* `encoder` ) [Function]

Frees all dynamically allocated data structures for this stream. This function discards any unprocessed input and does not flush any pending output. After a call to '`LZ_compress_close`', *encoder* can no more be used as an argument to any LZ_compress function.

`int LZ_compress_finish` ( *struct LZ_Encoder * const* `encoder` ) [Function]

Use this function to tell '`lzlib`' that all the data for this member have already been written (with the '`LZ_compress_write`' function). After all the produced compressed data have been read with '`LZ_compress_read`' and '`LZ_compress_member_finished`' returns 1, a new member can be started with '`LZ_compress_restart_member`'.

`int LZ_compress_restart_member` ( *struct LZ_Encoder * const* `encoder`, *const unsigned long long* `member_size` ) [Function]

Use this function to start a new member in a multimember data stream. Call this function only after '`LZ_compress_member_finished`' indicates that the current member has been fully read (with the '`LZ_compress_read`' function).

`int LZ_compress_sync_flush ( struct LZ_Encoder * const encoder )` [Function]

Use this function to make available to '`LZ_compress_read`' all the data already written with the '`LZ_compress_write`' function. First call '`LZ_compress_sync_flush`'. Then call '`LZ_compress_read`' until it returns 0.

Repeated use of '`LZ_compress_sync_flush`' may degrade compression ratio, so use it only when needed.

`int LZ_compress_read ( struct LZ_Encoder * const encoder, uint8_t * const buffer, const int size )` [Function]

The '`LZ_compress_read`' function reads up to *size* bytes from the stream pointed to by *encoder*, storing the results in *buffer*.

The return value is the number of bytes actually read. This might be less than *size*; for example, if there aren't that many bytes left in the stream or if more bytes have to be yet written with the '`LZ_compress_write`' function. Note that reading less than *size* bytes is not an error.

`int LZ_compress_write ( struct LZ_Encoder * const encoder, uint8_t * const buffer, const int size )` [Function]

The '`LZ_compress_write`' function writes up to *size* bytes from *buffer* to the stream pointed to by *encoder*.

The return value is the number of bytes actually written. This might be less than *size*. Note that writing less than *size* bytes is not an error.

`int LZ_compress_write_size ( struct LZ_Encoder * const encoder )` [Function]

The '`LZ_compress_write_size`' function returns the maximum number of bytes that can be immediately written through the '`LZ_compress_write`' function.

It is guaranteed that an immediate call to '`LZ_compress_write`' will accept a *size* up to the returned number of bytes.

`enum LZ_Errno LZ_compress_errno ( struct LZ_Encoder * const encoder )` [Function]

Returns the current error code for *encoder* (see Chapter 7 [Error codes], page 13).

`int LZ_compress_finished ( struct LZ_Encoder * const encoder )` [Function]

Returns 1 if all the data have been read and '`LZ_compress_close`' can be safely called. Otherwise it returns 0.

`int LZ_compress_member_finished ( struct LZ_Encoder * const encoder )` [Function]

Returns 1 if the current member, in a multimember data stream, has been fully read and '`LZ_compress_restart_member`' can be safely called. Otherwise it returns 0.

`unsigned long long LZ_compress_data_position ( struct LZ_Encoder * const encoder )` [Function]

Returns the number of input bytes already compressed in the current member.

`unsigned long long LZ_compress_member_position` ( *struct LZ_Encoder * const* `encoder` ) [Function]

Returns the number of compressed bytes already produced, but perhaps not yet read, in the current member.

`unsigned long long LZ_compress_total_in_size` ( *struct LZ_Encoder * const* `encoder` ) [Function]

Returns the total number of input bytes already compressed.

`unsigned long long LZ_compress_total_out_size` ( *struct LZ_Encoder * const* `encoder` ) [Function]

Returns the total number of compressed bytes already produced, but perhaps not yet read.

# 6 Decompression functions

These are the functions used to decompress data. In case of error, all of them return -1 or 0, for signed and unsigned return values respectively, except '`LZ_decompress_open`' whose return value must be verified by calling '`LZ_decompress_errno`' before using it.

`struct LZ_Decoder * LZ_decompress_open` ( *void* ) [Function]
: Initializes the internal stream state for decompression and returns a pointer that can only be used as the *decoder* argument for the other LZ_decompress functions, or a null pointer if the decoder could not be allocated.

  The returned pointer must be verified by calling '`LZ_decompress_errno`' before using it. If '`LZ_decompress_errno`' does not return '`LZ_ok`', the returned pointer must not be used and should be freed with '`LZ_decompress_close`' to avoid memory leaks.

`int LZ_decompress_close` ( *struct LZ_Decoder * const* `decoder` ) [Function]
: Frees all dynamically allocated data structures for this stream. This function discards any unprocessed input and does not flush any pending output. After a call to '`LZ_decompress_close`', *decoder* can no more be used as an argument to any LZ_decompress function.

`int LZ_decompress_finish` ( *struct LZ_Decoder * const* `decoder` ) [Function]
: Use this function to tell '`lzlib`' that all the data for this stream have already been written (with the '`LZ_decompress_write`' function).

`int LZ_decompress_reset` ( *struct LZ_Decoder * const* `decoder` ) [Function]
: Resets the internal state of *decoder* as it was just after opening it with the '`LZ_decompress_open`' function. Data stored in the internal buffers is discarded. Position counters are set to 0.

`int LZ_decompress_sync_to_member` ( *struct LZ_Decoder * const* `decoder` ) [Function]
: Resets the error state of *decoder* and enters a search state that lasts until a new member header (or the end of the stream) is found. After a successful call to '`LZ_decompress_sync_to_member`', data written with '`LZ_decompress_write`' will be consumed and '`LZ_decompress_read`' will return 0 until a header is found.

  This function is useful to discard any data preceding the first member, or to discard the rest of the current member, for example in case of a data error. If the decoder is already at the beginning of a member, this function does nothing.

`int LZ_decompress_read` ( *struct LZ_Decoder * const* `decoder`, *uint8_t * const* `buffer`, *const int* `size` ) [Function]
: The '`LZ_decompress_read`' function reads up to *size* bytes from the stream pointed to by *decoder*, storing the results in *buffer*.

  The return value is the number of bytes actually read. This might be less than *size*; for example, if there aren't that many bytes left in the stream or if more bytes have to be yet written with the '`LZ_decompress_write`' function. Note that reading less than *size* bytes is not an error.

`int LZ_decompress_write` ( *struct LZ_Decoder * const* `decoder`, *uint8_t * const* `buffer`, *const int* `size` ) [Function]

The '`LZ_decompress_write`' function writes up to *size* bytes from *buffer* to the stream pointed to by *decoder*.

The return value is the number of bytes actually written. This might be less than *size*. Note that writing less than *size* bytes is not an error.

`int LZ_decompress_write_size` ( *struct LZ_Decoder * const* `decoder` ) [Function]

The '`LZ_decompress_write_size`' function returns the maximum number of bytes that can be immediately written through the '`LZ_decompress_write`' function.

It is guaranteed that an immediate call to '`LZ_decompress_write`' will accept a *size* up to the returned number of bytes.

`enum LZ_Errno LZ_decompress_errno` ( *struct LZ_Decoder * const* `decoder` ) [Function]

Returns the current error code for *decoder* (see Chapter 7 [Error codes], page 13).

`int LZ_decompress_finished` ( *struct LZ_Decoder * const* `decoder` ) [Function]

Returns 1 if all the data have been read and '`LZ_decompress_close`' can be safely called. Otherwise it returns 0.

`int LZ_decompress_member_finished` ( *struct LZ_Decoder * const* `decoder` ) [Function]

Returns 1 if the previous call to '`LZ_decompress_read`' finished reading the current member, indicating that final values for member are available through '`LZ_decompress_data_crc`', '`LZ_decompress_data_position`', and '`LZ_decompress_member_position`'. Otherwise it returns 0.

`int LZ_decompress_member_version` ( *struct LZ_Decoder * const* `decoder` ) [Function]

Returns the version of current member from member header.

`int LZ_decompress_dictionary_size` ( *struct LZ_Decoder * const* `decoder` ) [Function]

Returns the dictionary size of current member from member header.

`unsigned LZ_decompress_data_crc` ( *struct LZ_Decoder * const* `decoder` ) [Function]

Returns the 32 bit Cyclic Redundancy Check of the data decompressed from the current member. The returned value is valid only when '`LZ_decompress_member_finished`' returns 1.

`unsigned long long LZ_decompress_data_position` ( *struct LZ_Decoder * const* `decoder` ) [Function]

Returns the number of decompressed bytes already produced, but perhaps not yet read, in the current member.

`unsigned long long LZ_decompress_member_position` ( *struct LZ_Decoder * const* `decoder` ) [Function]

Returns the number of input bytes already decompressed in the current member.

`unsigned long long LZ_decompress_total_in_size` ( *struct LZ_Decoder * const* `decoder` ) [Function]

Returns the total number of input bytes already decompressed.

`unsigned long long LZ_decompress_total_out_size` ( *struct LZ_Decoder * const* `decoder` ) [Function]

Returns the total number of decompressed bytes already produced, but perhaps not yet read.

# 7 Error codes

Most library functions return -1 to indicate that they have failed. But this return value only tells you that an error has occurred. To find out what kind of error it was, you need to verify the error code by calling '`LZ_(de)compress_errno`'.

Library functions don't change the value returned by '`LZ_(de)compress_errno`' when they succeed; thus, the value returned by '`LZ_(de)compress_errno`' after a successful call is not necessarily LZ_ok, and you should not use '`LZ_(de)compress_errno`' to determine whether a call failed. If the call failed, then you can examine '`LZ_(de)compress_errno`'.

The error codes are defined in the header file '`lzlib.h`'.

`enum LZ_Errno LZ_ok` [Constant]
: The value of this constant is 0 and is used to indicate that there is no error.

`enum LZ_Errno LZ_bad_argument` [Constant]
: At least one of the arguments passed to the library function was invalid.

`enum LZ_Errno LZ_mem_error` [Constant]
: No memory available. The system cannot allocate more virtual memory because its capacity is full.

`enum LZ_Errno LZ_sequence_error` [Constant]
: A library function was called in the wrong order. For example '`LZ_compress_restart_member`' was called before '`LZ_compress_member_finished`' indicates that the current member is finished.

`enum LZ_Errno LZ_header_error` [Constant]
: An invalid member header (one with the wrong magic bytes) was read. If this happens at the end of the data stream it may indicate trailing data.

`enum LZ_Errno LZ_unexpected_eof` [Constant]
: The end of the data stream was reached in the middle of a member.

`enum LZ_Errno LZ_data_error` [Constant]
: The data stream is corrupt.

`enum LZ_Errno LZ_library_error` [Constant]
: A bug was detected in the library. Please, report it (see Chapter 11 [Problems], page 20).

# 8 Error messages

`const char * LZ_strerror` ( *const enum LZ_Errno* `lz_errno` ) [Function]

Returns the standard error message for a given error code. The messages are fairly short; there are no multi-line messages or embedded newlines. This function makes it easy for your program to report informative error messages about the failure of a library call.

The value of *lz_errno* normally comes from a call to '`LZ_(de)compress_errno`'.

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

Each member has the following structure:

```
+--+--+--+--+----+----+=============+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| ID string | VN | DS | LZMA stream | 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 substracting 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 substract from the base size to obtain the dictionary size.
Example: 0xD3 = 2^19 - 6 * 2^15 = 512 KiB - 6 * 32 KiB = 320 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 "Stream format" in `lzip`, for a complete description.
Lzip only uses the LZMA marker '`2`' ("End Of Stream" marker). Lzlib also uses the LZMA marker '`3`' ("Sync Flush" marker).

‘`CRC32 (4 bytes)`’
: 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.

# 10 A small tutorial with examples

This chapter shows the order in which the library functions should be called depending on what kind of data stream you want to compress or decompress. See the file '`bbexample.c`' in the source distribution for an example of how buffer-to-buffer compression/decompression can be implemented using lzlib.

Note that lzlib's interface is symmetrical. That is, the code for normal compression and decompression is identical except because one calls LZ_compress* functions while the other calls LZ_decompress* functions.

Example 1: Normal compression (*member_size* > total output).

```
1) LZ_compress_open
2) LZ_compress_write
3) LZ_compress_read
4) go back to step 2 until all input data have been written
5) LZ_compress_finish
6) LZ_compress_read
7) go back to step 6 until LZ_compress_finished returns 1
8) LZ_compress_close
```

Example 2: Normal compression using LZ_compress_write_size.

```
1) LZ_compress_open
2) go to step 5 if LZ_compress_write_size returns 0
3) LZ_compress_write
4) if no more data to write, call LZ_compress_finish
5) LZ_compress_read
6) go back to step 2 until LZ_compress_finished returns 1
7) LZ_compress_close
```

Example 3: Decompression.

```
1) LZ_decompress_open
2) LZ_decompress_write
3) LZ_decompress_read
4) go back to step 2 until all input data have been written
5) LZ_decompress_finish
6) LZ_decompress_read
7) go back to step 6 until LZ_decompress_finished returns 1
8) LZ_decompress_close
```

Example 4: Decompression using LZ_decompress_write_size.

```
1) LZ_decompress_open
2) go to step 5 if LZ_decompress_write_size returns 0
3) LZ_decompress_write
4) if no more data to write, call LZ_decompress_finish
5) LZ_decompress_read
```

```
5a) optionally, if LZ_decompress_member_finished returns 1, read
    final values for member with LZ_decompress_data_crc, etc.
6) go back to step 2 until LZ_decompress_finished returns 1
7) LZ_decompress_close
```

Example 5: Multimember compression (*member_size* < total output).

```
 1) LZ_compress_open
 2) go to step 5 if LZ_compress_write_size returns 0
 3) LZ_compress_write
 4) if no more data to write, call LZ_compress_finish
 5) LZ_compress_read
 6) go back to step 2 until LZ_compress_member_finished returns 1
 7) go to step 10 if LZ_compress_finished() returns 1
 8) LZ_compress_restart_member
 9) go back to step 2
10) LZ_compress_close
```

Example 6: Multimember compression (user-restarted members).

```
 1) LZ_compress_open
 2) LZ_compress_write
 3) LZ_compress_read
 4) go back to step 2 until member termination is desired
 5) LZ_compress_finish
 6) LZ_compress_read
 7) go back to step 6 until LZ_compress_member_finished returns 1
 8) verify that LZ_compress_finished returns 1
 9) go to step 12 if all input data have been written
10) LZ_compress_restart_member
11) go back to step 2
12) LZ_compress_close
```

Example 7: Decompression with automatic removal of leading data.

```
1) LZ_decompress_open
2) LZ_decompress_sync_to_member
3) go to step 6 if LZ_decompress_write_size returns 0
4) LZ_decompress_write
5) if no more data to write, call LZ_decompress_finish
6) LZ_decompress_read
7) go back to step 3 until LZ_decompress_finished returns 1
8) LZ_decompress_close
```

Example 8: Streamed decompression with automatic resynchronization to next member in case of data error.

```
1) LZ_decompress_open
2) go to step 5 if LZ_decompress_write_size returns 0
```

```
3) LZ_decompress_write
4) if no more data to write, call LZ_decompress_finish
5) if LZ_decompress_read produces LZ_header_error or LZ_data_error,
   call LZ_decompress_sync_to_member
6) go back to step 2 until LZ_decompress_finished returns 1
7) LZ_decompress_close
```

# 11 Reporting bugs

There are probably bugs in lzlib. There are certainly errors and omissions in this manual. If you report them, they will get fixed. If you don't, no one will ever know about them and they will remain unfixed for all eternity, if not longer.

If you find a bug in lzlib, please send electronic mail to `lzip-bug@nongnu.org`. Include the version number, which you can find by running `minilzip --version` or in '`LZ_version_string`' from '`lzlib.h`'.