Lzlib

Compression library for the lzip format
for Lzlib version 1.12, 2 January 2021

by Antonio Diaz Diaz

Table of Contents

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 program lziprecover 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 source 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 '`bbexample.c`', '`ffexample.c`', and '`main.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 are 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 largest dictionary size that does not exceed neither the data size nor the limit given to '`LZ_compress_open`'. 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.

Lzlib is able to compress and decompress streams of unlimited size by automatically creating multimember output. The members so created are large, about 2 PiB each.

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).

LANGUAGE NOTE: Uncompressed = not compressed = plain data; it may never have been compressed. Decompressed is used to refer to data which have undergone the process of decompression.

2 Library version

One goal of lzlib is to keep perfect backward compatibility with older versions of itself down to 1.0. Any application working with an older lzlib should work with a newer lzlib. Installing a newer lzlib should not break anything. This chapter describes the constants and functions that the application can use to discover the version of the library being used.

`LZ_API_VERSION` [Constant]

This constant is defined in '`lzlib.h`' and works as a version test macro. The application should verify at compile time that LZ_API_VERSION is greater than or equal to the version required by the application:

```
#if !defined LZ_API_VERSION || LZ_API_VERSION < 1012
#error "lzlib 1.12 or newer needed."
#endif
```

Before version 1.8, lzlib didn't define LZ_API_VERSION.
LZ_API_VERSION was first defined in lzlib 1.8 to 1.
Since lzlib 1.12, LZ_API_VERSION is defined as (major * 1000 + minor).

NOTE: Version test macros are the library's way of announcing functionality to the application. They should not be confused with feature test macros, which allow the application to announce to the library its desire to have certain symbols and prototypes exposed.

`int LZ_api_version` (*void*) [Function]

If LZ_API_VERSION >= 1012, this function is declared in '`lzlib.h`' (else it doesn't exist). It returns the LZ_API_VERSION of the library object code being used. The application should verify at run time that the value returned by `LZ_api_version` is greater than or equal to the version required by the application. An application may be dinamically linked at run time with a different version of lzlib than the one it was compiled for, and this should not break the program as long as the library used provides the functionality required by the application.

```
#if defined LZ_API_VERSION && LZ_API_VERSION >= 1012
  if( LZ_api_version() < 1012 )
    show_error( "lzlib 1.12 or newer needed." );
#endif
```

`const char * LZ_version_string` [Constant]

This string constant is defined in the header file '`lzlib.h`' and represents the version of the library being used at compile time.

`const char * LZ_version` (*void*) [Function]

This function returns a string representing the version of the library being used at run time.

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 function '`LZ_compress_write`'. For the normal variant of LZMA, its size is two times the dictionary size set with the function '`LZ_compress_open`' or 64 KiB, whichever is larger. For the fast variant, its size is 1 MiB.
- Output compression buffer. Read from by the function '`LZ_compress_read`'. Its size is 64 KiB.
- Input decompression buffer. Written to by the function '`LZ_decompress_write`'. Its size is 64 KiB.
- Output decompression buffer. Read from by the function '`LZ_decompress_read`'. 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 size specified 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. Valid values range from 4 KiB to 2 PiB. A 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. Values larger than 2 PiB will be reduced to 2 PiB to prevent the uncompressed size of the member from overflowing.

`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 longer be used as an argument to any LZ_compress function. It is safe to call '`LZ_compress_close`' with a null argument.

`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 function '`LZ_compress_write`'). It is safe to call '`LZ_compress_finish`' as many times as needed. After all the 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 mem-

ber has been fully read (with the function '`LZ_compress_read`'). See [member_size], page 6, for a description of *member_size*.

`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 function '`LZ_compress_write`'. First call '`LZ_compress_sync_flush`'. Then call '`LZ_compress_read`' until it returns 0.

This function writes a LZMA marker '`3`' ("Sync Flush" marker) to the compressed output. Note that the sync flush marker is not allowed in lzip files; it is a device for interactive communication between applications using lzlib, but is useless and wasteful in a file, and is excluded from the media type '`application/lzip`'. The LZMA marker '`2`' ("End Of Stream" marker) is the only marker allowed in lzip files. See Chapter 10 [Data format], page 18.

Repeated use of '`LZ_compress_sync_flush`' may degrade compression ratio, so use it only when needed. If the interval between calls to '`LZ_compress_sync_flush`' is large (comparable to dictionary size), creating a multimember data stream with '`LZ_compress_restart_member`' may be an alternative.

Combining multimember stream creation with flushing may be tricky. If there are more bytes available than those needed to complete *member_size*, '`LZ_compress_restart_member`' needs to be called when '`LZ_compress_member_finished`' returns 1, followed by a new call to '`LZ_compress_sync_flush`'.

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

The function '`LZ_compress_read`' reads up to *size* bytes from the stream pointed to by *encoder*, storing the results in *buffer*. If LZ_API_VERSION >= 1012, *buffer* may be a null pointer, in which case the bytes read are discarded.

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 function '`LZ_compress_write`'. 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 function '`LZ_compress_write`' 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 function '`LZ_compress_write_size`' returns the maximum number of bytes that can be immediately written through '`LZ_compress_write`'. For efficiency reasons, once the input buffer is full and '`LZ_compress_write_size`' returns 0, almost all the buffer must be compressed before a size greater than 0 is returned again. (This is

done to minimize the amount of data that must be copied to the beginning of the buffer before new data can be accepted).

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 12. It is safe to call '`LZ_compress_errno`' with a null argument, in which case it returns '`LZ_bad_argument`'.

`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. '`LZ_compress_finished`' implies '`LZ_compress_member_finished`'.

`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 longer be used as an argument to any LZ_decompress function. It is safe to call '`LZ_decompress_close`' with a null argument.

`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 function '`LZ_decompress_write`'). It is safe to call '`LZ_decompress_finish`' as many times as needed. It is not required to call '`LZ_decompress_finish`' if the input stream only contains whole members, but not calling it prevents lzlib from detecting a truncated member.

`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 function '`LZ_decompress_open`'. 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 function '`LZ_decompress_read`' reads up to *size* bytes from the stream pointed to by *decoder*, storing the results in *buffer*. If LZ_API_VERSION >= 1012, *buffer* may be a null pointer, in which case the bytes read are discarded.

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 function '`LZ_decompress_write`'. Note that reading less than *size* bytes is not an error.

'`LZ_decompress_read`' returns at least once per member so that '`LZ_decompress_member_finished`' can be called (and trailer data retrieved) for each member, even for empty members. Therefore, '`LZ_decompress_read`' returning 0 does not mean that the end of the stream has been reached. The increase in the value returned by '`LZ_decompress_total_in_size`' can be used to tell the end of the stream from an empty member.

In case of decompression error caused by corrupt or truncated data, '`LZ_decompress_read`' does not signal the error immediately to the application, but waits until all the bytes decoded have been read. This allows tools like tarlz to recover as much data as possible from each damaged member. See `tarlz`.

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

The function '`LZ_decompress_write`' 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 function '`LZ_decompress_write_size`' returns the maximum number of bytes that can be immediately written through '`LZ_decompress_write`'. This number varies smoothly; each compressed byte consumed may be overwritten immediately, increasing by 1 the value returned.

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 12. It is safe to call '`LZ_decompress_errno`' with a null argument, in which case it returns '`LZ_bad_argument`'.

`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. '`LZ_decompress_finished`' does not imply '`LZ_decompress_member_finished`'.

`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 the current member, read from the 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. If '`LZ_decompress_member_position`' is 6 or less, it indicates either a format version not supported, an invalid dictionary size, a corrupt header in a multimember data stream, or trailing data too similar to a valid lzip header. Lziprecover can be used to remove conflicting trailing data from a file.

`enum LZ_Errno LZ_library_error` [Constant]
: A bug was detected in the library. Please, report it. See Chapter 12 [Problems], page 26.

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 Invoking minilzip

Minilzip is a test program for the compression library lzlib, fully compatible with lzip 1.4 or newer.

Lzip is a lossless data compressor with a user interface similar to the one of gzip or bzip2. Lzip uses a simplified form of the 'Lempel-Ziv-Markov chain-Algorithm' (LZMA) stream format, chosen to maximize safety and interoperability. Lzip can compress about as fast as gzip (lzip -0) or compress most files more than bzip2 (lzip -9). Decompression speed is intermediate between gzip and bzip2. Lzip is better than gzip and bzip2 from a data recovery perspective. Lzip has been designed, written, and tested with great care to replace gzip and bzip2 as the standard general-purpose compressed format for unix-like systems.

The format for running minilzip is:

```
minilzip [options] [files]
```

If no file names are specified, minilzip compresses (or decompresses) from standard input to standard output. A hyphen '`-`' used as a *file* argument means standard input. It can be mixed with other *files* and is read just once, the first time it appears in the command line.

minilzip supports the following options: See Section "Argument syntax" in `arg_parser`.

`-h`
`--help` Print an informative help message describing the options and exit.

`-V`
`--version`
Print the version number of minilzip on the standard output and exit. This version number should be included in all bug reports.

`-a`
`--trailing-error`
Exit with error status 2 if any remaining input is detected after decompressing the last member. Such remaining input is usually trailing garbage that can be safely ignored.

`-b` *bytes*
`--member-size=`*bytes*
When compressing, set the member size limit to *bytes*. It is advisable to keep members smaller than RAM size so that they can be repaired with lziprecover in case of corruption. A small member size may degrade compression ratio, so use it only when needed. Valid values range from 100 kB to 2 PiB. Defaults to 2 PiB.

`-c`
`--stdout` Compress or decompress to standard output; keep input files unchanged. If compressing several files, each file is compressed independently. (The output consists of a sequence of independently compressed members). This option (or '`-o`') is needed when reading from a named pipe (fifo) or from a device. Use it also to recover as much of the decompressed data as possible when decompressing a corrupt file. '`-c`' overrides '`-o`' and '`-S`'. '`-c`' has no effect when testing or listing.

`-d`
`--decompress`
: Decompress the files specified. If a file does not exist or can't be opened, minilzip continues decompressing the rest of the files. If a file fails to decompress, or is a terminal, minilzip exits immediately without decompressing the rest of the files.

`-f`
`--force`
: Force overwrite of output files.

`-F`
`--recompress`
: When compressing, force re-compression of files whose name already has the '`.lz`' or '`.tlz`' suffix.

`-k`
`--keep`
: Keep (don't delete) input files during compression or decompression.

`-m` *bytes*
`--match-length=`*bytes*
: When compressing, set the match length limit in bytes. After a match this long is found, the search is finished. Valid values range from 5 to 273. Larger values usually give better compression ratios but longer compression times.

`-o` *file*
`--output=`*file*
: If '`-c`' has not been also specified, write the (de)compressed output to *file*; keep input files unchanged. If compressing several files, each file is compressed independently. (The output consists of a sequence of independently compressed members). This option (or '`-c`') is needed when reading from a named pipe (fifo) or from a device. '`-o -`' is equivalent to '`-c`'. '`-o`' has no effect when testing or listing.

  When compressing and splitting the output in volumes, *file* is used as a prefix, and several files named '*file*`00001.lz`', '*file*`00002.lz`', etc, are created. In this case, only one input file is allowed.

`-q`
`--quiet`
: Quiet operation. Suppress all messages.

`-s` *bytes*
`--dictionary-size=`*bytes*
: When compressing, set the dictionary size limit in bytes. Minilzip will use for each file the largest dictionary size that does not exceed neither the file size nor this limit. Valid values range from 4 KiB to 512 MiB. Values 12 to 29 are interpreted as powers of two, meaning 2^12 to 2^29 bytes. Dictionary sizes are quantized so that they can be coded in just one byte (see [coded-dict-size], page 18). If the size specified does not match one of the valid sizes, it will be rounded upwards by adding up to (*bytes* / 8) to it.

  For maximum compression you should use a dictionary size limit as large as possible, but keep in mind that the decompression memory requirement is affected at compression time by the choice of dictionary size limit.

`-S` *bytes*
`--volume-size=`*bytes*
: When compressing, and '`-c`' has not been also specified, split the compressed output into several volume files with names '`original_name00001.lz`', '`original_name00002.lz`', etc, and set the volume size limit to *bytes*. Input files are kept unchanged. Each volume is a complete, maybe multimember, lzip file. A small volume size may degrade compression ratio, so use it only when needed. Valid values range from 100 kB to 4 EiB.

`-t`
`--test`
: Check integrity of the files specified, but don't decompress them. This really performs a trial decompression and throws away the result. Use it together with '`-v`' to see information about the files. If a file fails the test, does not exist, can't be opened, or is a terminal, minilzip continues checking the rest of the files. A final diagnostic is shown at verbosity level 1 or higher if any file fails the test when testing multiple files.

`-v`
`--verbose`
: Verbose mode.
When compressing, show the compression ratio and size for each file processed. When decompressing or testing, further -v's (up to 4) increase the verbosity level, showing status, compression ratio, dictionary size, and trailer contents (CRC, data size, member size).

`-0 .. -9`
: Compression level. Set the compression parameters (dictionary size and match length limit) as shown in the table below. The default compression level is '`-6`', equivalent to '`-s8MiB -m36`'. Note that '`-9`' can be much slower than '`-0`'. These options have no effect when decompressing or testing.

  The bidimensional parameter space of LZMA can't be mapped to a linear scale optimal for all files. If your files are large, very repetitive, etc, you may need to use the options '`--dictionary-size`' and '`--match-length`' directly to achieve optimal performance.

  If several compression levels or '`-s`' or '`-m`' options are given, the last setting is used. For example '`-9 -s64MiB`' is equivalent to '`-s64MiB -m273`'

Level	Dictionary size (-s)	Match length limit (-m)
-0	64 KiB	16 bytes
-1	1 MiB	5 bytes
-2	1.5 MiB	6 bytes
-3	2 MiB	8 bytes
-4	3 MiB	12 bytes
-5	4 MiB	20 bytes
-6	8 MiB	36 bytes
-7	16 MiB	68 bytes
-8	24 MiB	132 bytes
-9	32 MiB	273 bytes

`--fast`
`--best`
: Aliases for GNU gzip compatibility.

`--loose-trailing`
: When decompressing or testing, allow trailing data whose first bytes are so similar to the magic bytes of a lzip header that they can be confused with a corrupt header. Use this option if a file triggers a "corrupt header" error and the cause is not indeed a corrupt header.

`--check-lib`
: Compare the version of lzlib used to compile minilzip with the version actually being used and exit. Report any differences found. Exit with error status 1 if differences are found. A mismatch may indicate that lzlib is not correctly installed or that a different version of lzlib has been installed after compiling the shared version of minilzip. '`minilzip -v --check-lib`' shows the version of lzlib being used and the value of '`LZ_API_VERSION`' (if defined). See Chapter 2 [Library version], page 3.

Numbers given as arguments to options may be followed by a multiplier and an optional '`B`' for "byte".

Table of SI and binary prefixes (unit multipliers):

Prefix	Value	Prefix	Value
k	kilobyte (10^3 = 1000)	Ki	kibibyte (2^10 = 1024)
M	megabyte (10^6)	Mi	mebibyte (2^20)
G	gigabyte (10^9)	Gi	gibibyte (2^30)
T	terabyte (10^12)	Ti	tebibyte (2^40)
P	petabyte (10^15)	Pi	pebibyte (2^50)
E	exabyte (10^18)	Ei	exbibyte (2^60)
Z	zettabyte (10^21)	Zi	zebibyte (2^70)
Y	yottabyte (10^24)	Yi	yobibyte (2^80)

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, 3 for an internal consistency error (eg, bug) which caused minilzip to panic.

10 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 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 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). See [sync_flush], page 7.

‘`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.

11 A small tutorial with examples

This chapter provides real code examples for the most common uses of the library. See these examples in context in the files '`bbexample.c`' and '`ffexample.c`' from the source distribution of lzlib.

Note that the interface of lzlib 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.

11.1 Buffer compression

Buffer-to-buffer single-member compression (*member_size* > total output).

```
/* Compresses 'insize' bytes from 'inbuf' to 'outbuf'.
   Returns the size of the compressed data in '*outlenp'.
   In case of error, or if 'outsize' is too small, returns false and does
   not modify '*outlenp'.
*/
bool bbcompress( const uint8_t * const inbuf, const int insize,
                 const int dictionary_size, const int match_len_limit,
                 uint8_t * const outbuf, const int outsize,
                 int * const outlenp )
  {
  int inpos = 0, outpos = 0;
  bool error = false;
  struct LZ_Encoder * const encoder =
    LZ_compress_open( dictionary_size, match_len_limit, INT64_MAX );
  if( !encoder || LZ_compress_errno( encoder ) != LZ_ok )
    { LZ_compress_close( encoder ); return false; }

  while( true )
    {
    int ret = LZ_compress_write( encoder, inbuf + inpos, insize - inpos );
    if( ret < 0 ) { error = true; break; }
    inpos += ret;
    if( inpos >= insize ) LZ_compress_finish( encoder );
    ret = LZ_compress_read( encoder, outbuf + outpos, outsize - outpos );
    if( ret < 0 ) { error = true; break; }
    outpos += ret;
    if( LZ_compress_finished( encoder ) == 1 ) break;
    if( outpos >= outsize ) { error = true; break; }
    }

  if( LZ_compress_close( encoder ) < 0 ) error = true;
  if( error ) return false;
  *outlenp = outpos;
  return true;
  }
```

11.2 Buffer decompression

Buffer-to-buffer decompression.

```
/* Decompresses 'insize' bytes from 'inbuf' to 'outbuf'.
   Returns the size of the decompressed data in '*outlenp'.
   In case of error, or if 'outsize' is too small, returns false and does
   not modify '*outlenp'.
*/
bool bbdecompress( const uint8_t * const inbuf, const int insize,
                   uint8_t * const outbuf, const int outsize,
                   int * const outlenp )
  {
  int inpos = 0, outpos = 0;
  bool error = false;
  struct LZ_Decoder * const decoder = LZ_decompress_open();
  if( !decoder || LZ_decompress_errno( decoder ) != LZ_ok )
    { LZ_decompress_close( decoder ); return false; }

  while( true )
    {
    int ret = LZ_decompress_write( decoder, inbuf + inpos, insize - inpos );
    if( ret < 0 ) { error = true; break; }
    inpos += ret;
    if( inpos >= insize ) LZ_decompress_finish( decoder );
    ret = LZ_decompress_read( decoder, outbuf + outpos, outsize - outpos );
    if( ret < 0 ) { error = true; break; }
    outpos += ret;
    if( LZ_decompress_finished( decoder ) == 1 ) break;
    if( outpos >= outsize ) { error = true; break; }
    }

  if( LZ_decompress_close( decoder ) < 0 ) error = true;
  if( error ) return false;
  *outlenp = outpos;
  return true;
  }
```

11.3 File compression

File-to-file compression using LZ_compress_write_size.

```
int ffcompress( struct LZ_Encoder * const encoder,
                FILE * const infile, FILE * const outfile )
  {
  enum { buffer_size = 16384 };
  uint8_t buffer[buffer_size];
  while( true )
    {
```

```
    int len, ret;
    int size = min( buffer_size, LZ_compress_write_size( encoder ) );
    if( size > 0 )
      {
      len = fread( buffer, 1, size, infile );
      ret = LZ_compress_write( encoder, buffer, len );
      if( ret < 0 || ferror( infile ) ) break;
      if( feof( infile ) ) LZ_compress_finish( encoder );
      }
    ret = LZ_compress_read( encoder, buffer, buffer_size );
    if( ret < 0 ) break;
    len = fwrite( buffer, 1, ret, outfile );
    if( len < ret ) break;
    if( LZ_compress_finished( encoder ) == 1 ) return 0;
    }
  return 1;
  }
```

11.4 File decompression

File-to-file decompression using LZ_decompress_write_size.

```
int ffdecompress( struct LZ_Decoder * const decoder,
                  FILE * const infile, FILE * const outfile )
  {
  enum { buffer_size = 16384 };
  uint8_t buffer[buffer_size];
  while( true )
    {
    int len, ret;
    int size = min( buffer_size, LZ_decompress_write_size( decoder ) );
    if( size > 0 )
      {
      len = fread( buffer, 1, size, infile );
      ret = LZ_decompress_write( decoder, buffer, len );
      if( ret < 0 || ferror( infile ) ) break;
      if( feof( infile ) ) LZ_decompress_finish( decoder );
      }
    ret = LZ_decompress_read( decoder, buffer, buffer_size );
    if( ret < 0 ) break;
    len = fwrite( buffer, 1, ret, outfile );
    if( len < ret ) break;
    if( LZ_decompress_finished( decoder ) == 1 ) return 0;
    }
  return 1;
  }
```

11.5 File-to-file multimember compression

Example 1: Multimember compression with members of fixed size (*member_size* < total output).

```
int ffmmcompress( FILE * const infile, FILE * const outfile )
  {
  enum { buffer_size = 16384, member_size = 4096 };
  uint8_t buffer[buffer_size];
  bool done = false;
  struct LZ_Encoder * const encoder =
    LZ_compress_open( 65535, 16, member_size );
  if( !encoder || LZ_compress_errno( encoder ) != LZ_ok  )
    { fputs( "ffexample: Not enough memory.\n", stderr );
      LZ_compress_close( encoder ); return 1; }
  while( true )
    {
    int len, ret;
    int size = min( buffer_size, LZ_compress_write_size( encoder ) );
    if( size > 0 )
      {
      len = fread( buffer, 1, size, infile );
      ret = LZ_compress_write( encoder, buffer, len );
      if( ret < 0 || ferror( infile ) ) break;
      if( feof( infile ) ) LZ_compress_finish( encoder );
      }
    ret = LZ_compress_read( encoder, buffer, buffer_size );
    if( ret < 0 ) break;
    len = fwrite( buffer, 1, ret, outfile );
    if( len < ret ) break;
    if( LZ_compress_member_finished( encoder ) == 1 )
      {
      if( LZ_compress_finished( encoder ) == 1 ) { done = true; break; }
      if( LZ_compress_restart_member( encoder, member_size ) < 0 ) break;
      }
    }
  if( LZ_compress_close( encoder ) < 0 ) done = false;
  return done;
  }
```

Example 2: Multimember compression (user-restarted members). (Call LZ_compress_open with *member_size* > largest member).

```
/* Compresses 'infile' to 'outfile' as a multimember stream with one member
   for each line of text terminated by a newline character or by EOF.
   Returns 0 if success, 1 if error.
*/
int fflfcompress( struct LZ_Encoder * const encoder,
                  FILE * const infile, FILE * const outfile )
  {
```

```
  enum { buffer_size = 16384 };
  uint8_t buffer[buffer_size];
  while( true )
    {
    int len, ret;
    int size = min( buffer_size, LZ_compress_write_size( encoder ) );
    if( size > 0 )
      {
      for( len = 0; len < size; )
        {
        int ch = getc( infile );
        if( ch == EOF || ( buffer[len++] = ch ) == '\n' ) break;
        }
      /* avoid writing an empty member to outfile */
      if( len == 0 && LZ_compress_data_position( encoder ) == 0 ) return 0;
      ret = LZ_compress_write( encoder, buffer, len );
      if( ret < 0 || ferror( infile ) ) break;
      if( feof( infile ) || buffer[len-1] == '\n' )
        LZ_compress_finish( encoder );
      }
    ret = LZ_compress_read( encoder, buffer, buffer_size );
    if( ret < 0 ) break;
    len = fwrite( buffer, 1, ret, outfile );
    if( len < ret ) break;
    if( LZ_compress_member_finished( encoder ) == 1 )
      {
      if( feof( infile ) && LZ_compress_finished( encoder ) == 1 ) return 0;
      if( LZ_compress_restart_member( encoder, INT64_MAX ) < 0 ) break;
      }
    }
  return 1;
  }
```

11.6 Skipping data errors

```
/* Decompresses 'infile' to 'outfile' with automatic resynchronization to
   next member in case of data error, including the automatic removal of
   leading garbage.
*/
int ffrsdecompress( struct LZ_Decoder * const decoder,
                    FILE * const infile, FILE * const outfile )
  {
  enum { buffer_size = 16384 };
  uint8_t buffer[buffer_size];
  while( true )
    {
    int len, ret;
```

```
  int size = min( buffer_size, LZ_decompress_write_size( decoder ) );
  if( size > 0 )
    {
    len = fread( buffer, 1, size, infile );
    ret = LZ_decompress_write( decoder, buffer, len );
    if( ret < 0 || ferror( infile ) ) break;
    if( feof( infile ) ) LZ_decompress_finish( decoder );
    }
  ret = LZ_decompress_read( decoder, buffer, buffer_size );
  if( ret < 0 )
    {
    if( LZ_decompress_errno( decoder ) == LZ_header_error ||
        LZ_decompress_errno( decoder ) == LZ_data_error )
      { LZ_decompress_sync_to_member( decoder ); continue; }
    else break;
    }
  len = fwrite( buffer, 1, ret, outfile );
  if( len < ret ) break;
  if( LZ_decompress_finished( decoder ) == 1 ) return 0;
  }
return 1;
}
```

12 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`'.

Concept index