# *zlib* 1.2.5 Manual

---

## Contents

---

## Prologue

***zlib*** general purpose compression library
version 1.2.5, April 19th, 2010

Jean-loup Gailly        Mark Adler

The data format used by the ***zlib*** library is described by RFCs (Request for Comments) 1950 to 1952 in the files rfc1950.txt (zlib format), rfc1951.txt (deflate format) and rfc1952.txt (gzip format).

## Version

```
#define ZLIB_VERSION "1.2.5"
#define ZLIB_VERNUM 0x1250
```

## Introduction

The ***zlib*** compression library provides in-memory compression and decompression functions, including integrity checks of the uncompressed data. This version of the library supports only one compression method (deflation) but other algorithms will be added later and will have the same stream interface.

Compression can be done in a single step if the buffers are large enough (for example if an input file is mmap'ed), or can be done by repeated calls of the compression function. In the latter case, the application must provide more input and/or consume the output (providing more output space) before each call.

The compressed data format used by default by the in-memory functions is the ***zlib*** format, which is a ***zlib*** wrapper documented in RFC 1950, wrapped around a deflate stream, which is itself documented in RFC 1951.

The library also supports reading and writing files in ***gzip*** (.gz) format with an interface similar to that of stdio using the functions that start with "gz". The ***gzip*** format is different from the ***zlib*** format. ***gzip*** is a ***gzip*** wrapper, documented in RFC 1952, wrapped around a deflate stream.

This library can optionally read and write ***gzip*** streams in memory as well.

The ***zlib*** format was designed to be compact and fast for use in memory and on communications channels. The ***gzip*** format was designed for single- file compression on file systems, has a larger header than ***zlib*** to maintain directory information, and uses a different, slower check method than ***zlib***.

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.

## Stream Data Structures

```
typedef voidpf (*alloc_func) OF((voidpf opaque, uInt items, uInt size));
typedef void   (*free_func)  OF((voidpf opaque, voidpf address));

struct internal_state;

typedef struct z_stream_s {
    Bytef    *next_in;  /* next input byte */
    uInt     avail_in;  /* number of bytes available at next_in */
    uLong    total_in;  /* total nb of input bytes read so far */

    Bytef    *next_out; /* next output byte should be put there */
    uInt     avail_out; /* remaining free space at next_out */
    uLong    total_out; /* total nb of bytes output so far */

    char     *msg;      /* last error message, NULL if no error */
    struct internal_state FAR *state; /* not visible by applications */

    alloc_func zalloc;  /* used to allocate the internal state */
    free_func  zfree;   /* used to free the internal state */
    voidpf     opaque;  /* private data object passed to zalloc and zfree */

    int     data_type;  /* best guess about the data type: binary or text */
    uLong   adler;      /* adler32 value of the uncompressed data */
    uLong   reserved;   /* reserved for future use */
} z_stream;

typedef z_stream FAR *z_streamp;
```

***gzip*** header information passed to and from ***zlib*** routines. See RFC 1952 for more details on the meanings of these fields.

```
typedef struct gz_header_s {
    int     text;       /* true if compressed data believed to be text */
    uLong   time;       /* modification time */
    int     xflags;     /* extra flags (not used when writing a gzip file) */
    int     os;         /* operating system */
    Bytef   *extra;     /* pointer to extra field or Z_NULL if none */
    uInt    extra_len;  /* extra field length (valid if extra != Z_NULL) */
    uInt    extra_max;  /* space at extra (only when reading header) */
    Bytef   *name;      /* pointer to zero-terminated file name or Z_NULL */
    uInt    name_max;   /* space at name (only when reading header) */
    Bytef   *comment;   /* pointer to zero-terminated comment or Z_NULL */
    uInt    comm_max;   /* space at comment (only when reading header) */
    int     hcrc;       /* true if there was or will be a header crc */
    int     done;       /* true when done reading gzip header (not used
                           when writing a gzip file) */
} gz_header;

typedef gz_header FAR *gz_headerp;
```

## Structures Usage

The application must update `next_in` and `avail_in` when `avail_in` has dropped to zero. It must update `next_out` and `avail_out` when `avail_out` has dropped to zero. The application must initialize zalloc, zfree and opaque before calling the init function. All other fields are set by the compression library and must not be updated by the application.

The opaque value provided by the application will be passed as the first parameter for calls of zalloc and zfree. This can be useful for custom memory management. The compression library attaches no meaning to the opaque value.

zalloc must return Z_NULL if there is not enough memory for the object. If ***zlib*** is used in a multi-threaded application, zalloc and zfree must be thread safe.

On 16-bit systems, the functions zalloc and zfree must be able to allocate exactly 65536 bytes, but will not be required to allocate more than this if the symbol MAXSEG_64K is defined (see zconf.h). WARNING: On MSDOS, pointers returned by zalloc for objects of exactly 65536 bytes ***must*** have their offset normalized to zero. The default allocation function provided by this library ensures this (see zutil.c). To reduce memory requirements and avoid any allocation of 64K objects, at the expense of compression ratio, compile the library with -DMAX_WBITS=14 (see zconf.h).

The fields total_in and total_out can be used for statistics or progress reports. After compression, total_in holds the total size of the uncompressed data and may be saved for use in the decompressor (particularly if the decompressor wants to decompress everything in a single step).

## Constants

Allowed flush values; see deflate() and inflate() below for details.

```
#define Z_NO_FLUSH      0
#define Z_PARTIAL_FLUSH 1
#define Z_SYNC_FLUSH    2
#define Z_FULL_FLUSH    3
#define Z_FINISH        4
#define Z_BLOCK         5
#define Z_TREES         6
```

Return codes for the compression/decompression functions. Negative values are errors, positive values are used for special but normal events.

```
#define Z_OK            0
#define Z_STREAM_END    1
#define Z_NEED_DICT     2
#define Z_ERRNO        (-1)
#define Z_STREAM_ERROR (-2)
#define Z_DATA_ERROR   (-3)
#define Z_MEM_ERROR    (-4)
#define Z_BUF_ERROR    (-5)
```

```
#define Z_VERSION_ERROR (-6)
```

Compression levels.

```
#define Z_NO_COMPRESSION         0
#define Z_BEST_SPEED             1
#define Z_BEST_COMPRESSION       9
#define Z_DEFAULT_COMPRESSION  (-1)
```

Compression strategy — see `deflateInit2()` below for details.

```
#define Z_FILTERED            1
#define Z_HUFFMAN_ONLY        2
#define Z_RLE                 3
#define Z_FIXED               4
#define Z_DEFAULT_STRATEGY    0
```

Possible values of the `data_type` field (though see `inflate()`).

```
#define Z_BINARY   0
#define Z_TEXT     1
#define Z_ASCII    Z_TEXT   /* for compatibility with 1.2.2 and earlier */
#define Z_UNKNOWN  2
```

The deflate compression method (the only one supported in this version).

```
#define Z_DEFLATED   8
```

For initializing `zalloc`, `zfree`, `opaque`.

```
#define Z_NULL  0
```

For compatibility with versions < 1.0.2.

```
#define zlib_version zlibVersion()
```

## Basic Functions

```
ZEXTERN const char * ZEXPORT zlibVersion OF((void));
```

The application can compare `zlibVersion` and `ZLIB_VERSION` for consistency. If the first character differs, the library code actually used is not compatible with the zlib.h header file used by the application. This check is automatically made by `deflateInit` and `inflateInit`.

```
ZEXTERN int ZEXPORT deflateInit OF((z_streamp strm, int level));
```

Initializes the internal stream state for compression. The fields `zalloc`, `zfree` and `opaque` must be initialized before by the caller. If `zalloc` and `zfree` are set to `Z_NULL`, `deflateInit` updates them to use default allocation functions.

The compression level must be `Z_DEFAULT_COMPRESSION`, or between 0 and 9: 1 gives best speed, 9 gives best compression, 0 gives no compression at all (the input data is simply copied a block at a time). `Z_DEFAULT_COMPRESSION` requests a default compromise between speed and compression (currently equivalent to level 6).

`deflateInit` returns `Z_OK` if success, `Z_MEM_ERROR` if there was not enough memory, `Z_STREAM_ERROR` if `level` is not a valid compression level, `Z_VERSION_ERROR` if the ***zlib*** library version (`zlib_version`) is incompatible with the version assumed by the caller (`ZLIB_VERSION`). `msg` is set to null if there is no error message. `deflateInit` does not perform any compression: this will be done by deflate().

**`ZEXTERN int ZEXPORT deflate OF((z_streamp strm, int flush));`**

`deflate` compresses as much data as possible, and stops when the input buffer becomes empty or the output buffer becomes full. It may introduce some output latency (reading input without producing any output) except when forced to flush.

The detailed semantics are as follows. `deflate` performs one or both of the following actions:

- Compress more input starting at `next_in` and update `next_in` and `avail_in` accordingly. If not all input can be processed (because there is not enough room in the output buffer), `next_in` and `avail_in` are updated and processing will resume at this point for the next call of `deflate()`.
- Provide more output starting at `next_out` and update `next_out` and `avail_out` accordingly. This action is forced if the parameter `flush` is non zero. Forcing flush frequently degrades the compression ratio, so this parameter should be set only when necessary (in interactive applications). Some output may be provided even if `flush` is not set.

Before the call of `deflate()`, the application should ensure that at least one of the actions is possible, by providing more input and/or consuming more output, and updating `avail_in` or `avail_out` accordingly; `avail_out` should never be zero before the call. The application can consume the compressed output when it wants, for example when the output buffer is full (`avail_out == 0`), or after each call of `deflate()`. If `deflate` returns `Z_OK` and with zero `avail_out`, it must be called again after making room in the output buffer because there might be more output pending.

Normally the parameter `flush` is set to `Z_NO_FLUSH`, which allows `deflate` to decide how much data to accumulate before producing output, in order to maximize compression.

If the parameter `flush` is set to `Z_SYNC_FLUSH`, all pending output is flushed to the output buffer and the output is aligned on a byte boundary, so that the decompressor can get all input data available so far. (In particular `avail_in` is zero after the call if enough output space has been provided before the call.) Flushing may degrade compression for some compression algorithms and so it should be used only when necessary. This completes the current deflate block and follows it with an empty stored block that is three bits plus filler bits to the next byte, followed by four bytes (00 00 ff ff).

If `flush` is set to `Z_PARTIAL_FLUSH`, all pending output is flushed to the output buffer, but the output is not aligned to a byte boundary. All of the input data so far will be available to the decompressor, as for `Z_SYNC_FLUSH`. This completes the current deflate block and follows it with an empty fixed codes block that is 10 bits long. This assures that enough bytes are output in order for the decompressor to finish the block before the empty fixed code block.

If flush is set to `Z_BLOCK`, a deflate block is completed and emitted, as for `Z_SYNC_FLUSH`, but the output is not aligned on a byte boundary, and up to seven bits of the current block are held to be written as the next byte after the next deflate block is completed. In this case, the decompressor may not be provided enough bits at this point in order to complete decompression of the data provided so far to the compressor. It may need to wait for the next block to be emitted. This is for advanced applications that need to control the emission of deflate blocks.

If `flush` is set to `Z_FULL_FLUSH`, all output is flushed as with `Z_SYNC_FLUSH`, and the compression state is reset so that decompression can restart from this point if previous compressed data has been damaged or if random access is desired. Using `Z_FULL_FLUSH` too often can seriously degrade compression.

If `deflate` returns with `avail_out == 0`, this function must be called again with the same value of the `flush` parameter and more output space (updated `avail_out`), until the flush is complete (`deflate` returns with non-zero `avail_out`). In the case of a `Z_FULL_FLUSH` or `Z_SYNC_FLUSH`, make sure that `avail_out` is greater than six to avoid repeated flush markers due to `avail_out == 0` on return.

If the parameter `flush` is set to `Z_FINISH`, pending input is processed, pending output is flushed and `deflate` returns with `Z_STREAM_END` if there was enough output space; if `deflate` returns with `Z_OK`, this function must be called again with `Z_FINISH` and more output space (updated `avail_out`) but no more input data, until it returns with `Z_STREAM_END` or an error. After `deflate` has returned `Z_STREAM_END`, the only possible operations on the stream are `deflateReset` or `deflateEnd`.

`Z_FINISH` can be used immediately after `deflateInit` if all the compression is to be done in a single step. In this case, `avail_out` must be at least the value returned by `deflateBound` (see below). If `deflate` does not return `Z_STREAM_END`, then it must be called again as described above.

`deflate()` sets `strm->adler` to the adler32 checksum of all input read so far (that is, `total_in` bytes).

deflate() may update strm->data_type if it can make a good guess about the input data type (Z_BINARY or Z_TEXT). In doubt, the data is considered binary. This field is only for information purposes and does not affect the compression algorithm in any manner.

deflate() returns Z_OK if some progress has been made (more input processed or more output produced), Z_STREAM_END if all input has been consumed and all output has been produced (only when flush is set to Z_FINISH), Z_STREAM_ERROR if the stream state was inconsistent (for example if next_in or next_out was NULL), Z_BUF_ERROR if no progress is possible (for example avail_in or avail_out was zero). Note that Z_BUF_ERROR is not fatal, and deflate() can be called again with more input and more output space to continue compressing.

**ZEXTERN int ZEXPORT deflateEnd OF((z_streamp strm));**

All dynamically allocated data structures for this stream are freed. This function discards any unprocessed input and does not flush any pending output.

deflateEnd returns Z_OK if success, Z_STREAM_ERROR if the stream state was inconsistent, Z_DATA_ERROR if the stream was freed prematurely (some input or output was discarded). In the error case, msg may be set but then points to a static string (which must not be deallocated).

**ZEXTERN int ZEXPORT inflateInit OF((z_streamp strm));**

Initializes the internal stream state for decompression. The fields next_in, avail_in, zalloc, zfree and opaque must be initialized before by the caller. If next_in is not Z_NULL and avail_in is large enough (the exact value depends on the compression method), inflateInit determines the compression method from the ***zlib*** header and allocates all data structures accordingly; otherwise the allocation will be deferred to the first call of inflate. If zalloc and zfree are set to Z_NULL, inflateInit updates them to use default allocation functions.

inflateInit returns Z_OK if success, Z_MEM_ERROR if there was not enough memory, Z_VERSION_ERROR if the zlib library version is incompatible with the version assumed by the caller, or Z_STREAM_ERROR if the parameters are invalid, such as a null pointer to the structure. msg is set to null if there is no error message. inflateInit does not perform any decompression apart from possibly reading the ***zlib*** header if present: actual decompression will be done by inflate(). (So next_in and avail_in may be modified, but next_out and avail_out are unused and unchanged.) The current implementation of inflateInit() does not process any header information -- that is deferred until inflate() is called.

**ZEXTERN int ZEXPORT inflate OF((z_streamp strm, int flush));**

inflate decompresses as much data as possible, and stops when the input buffer becomes empty or the output buffer becomes full. It may introduce some output latency (reading input without producing any output) except when forced to flush.

The detailed semantics are as follows. inflate performs one or both of the following actions:

- Decompress more input starting at next_in and update next_in and avail_in accordingly. If not all input can be processed (because there is not enough room in the output buffer), next_in is updated and processing will resume at this point for the next call of inflate().
- Provide more output starting at next_out and update next_out and avail_out accordingly. inflate() provides as much output as possible, until there is no more input data or no more space in the output buffer (see below about the flush parameter).

Before the call of inflate(), the application should ensure that at least one of the actions is possible, by providing more input and/or consuming more output, and updating the next_* and avail_* values accordingly. The application can consume the uncompressed output when it wants, for example when the output buffer is full (avail_out == 0), or after each call of inflate(). If inflate returns Z_OK and with zero avail_out, it must be called again after making room in the output buffer because there might be more output pending.

The flush parameter of inflate() can be Z_NO_FLUSH, Z_SYNC_FLUSH, Z_FINISH, Z_BLOCK, or Z_TREES. Z_SYNC_FLUSH requests that inflate() flush as much output as possible to the output buffer. Z_BLOCK requests that inflate() stop if and when it gets to the next deflate block boundary. When decoding the ***zlib*** or ***gzip*** format, this will cause inflate() to return immediately after the header and before the first block. When doing a raw inflate, inflate() will go ahead and process the first block, and will return when it gets to the end of that block, or when it runs out of data.

The Z_BLOCK option assists in appending to or combining deflate streams. Also to assist in this, on return inflate() will set strm->data_type to the number of unused bits in the last byte taken from strm->next_in, plus 64 if inflate() is currently decoding the last block in the deflate stream, plus 128 if inflate() returned immediately after decoding an end-of-block code or decoding the complete header up to just before the first byte of the deflate stream. The end-of-block will not be indicated until all of the uncompressed data from that block has been written to strm->next_out. The number of unused bits may in general be greater than seven, except when bit 7 of data_type is set, in which case the number of unused bits will be less than eight. data_type is set as noted here every time inflate() returns for all flush options, and so can be used to determine the amount of currently consumed input in bits.

The Z_TREES option behaves as Z_BLOCK does, but it also returns when the end of each deflate block header is reached, before any actual data in that block is decoded. This allows the caller to determine the length of the deflate block header for later use in random access within a deflate block. 256 is added to the value of strm->data_type when inflate() returns immediately after reaching the end of the deflate block header.

inflate() should normally be called until it returns Z_STREAM_END or an error. However if all decompression is to be performed in a single step (a single call of inflate), the parameter flush should be set to `Z_FINISH`. In this case all pending input is processed and all pending output is flushed; `avail_out` must be large enough to hold all the uncompressed data. (The size of the uncompressed data may have been saved by the compressor for this purpose.) The next operation on this stream must be `inflateEnd` to deallocate the decompression state. The use of `Z_FINISH` is never required, but can be used to inform `inflate` that a faster approach may be used for the single `inflate()` call.

In this implementation, `inflate()` always flushes as much output as possible to the output buffer, and always uses the faster approach on the first call. So the only effect of the `flush` parameter in this implementation is on the return value of `inflate()`, as noted below, or when it returns early because `Z_BLOCK` or `Z_TREES` is used.

If a preset dictionary is needed after this call (see `inflateSetDictionary` below), `inflate` sets `strm->adler` to the adler32 checksum of the dictionary chosen by the compressor and returns `Z_NEED_DICT`; otherwise it sets `strm->adler` to the adler32 checksum of all output produced so far (that is, `total_out` bytes) and returns `Z_OK`, `Z_STREAM_END` or an error code as described below. At the end of the stream, `inflate()` checks that its computed adler32 checksum is equal to that saved by the compressor and returns `Z_STREAM_END` only if the checksum is correct.

`inflate()` will decompress and check either ***zlib***-wrapped or ***gzip***-wrapped deflate data. The header type is detected automatically, if requested when initializing with `inflateInit2()`. Any information contained in the ***gzip*** header is not retained, so applications that need that information should instead use raw inflate, see `inflateInit2()` below, or `inflateBack()` and perform their own processing of the ***gzip*** header and trailer.

`inflate()` returns `Z_OK` if some progress has been made (more input processed or more output produced), `Z_STREAM_END` if the end of the compressed data has been reached and all uncompressed output has been produced, `Z_NEED_DICT` if a preset dictionary is needed at this point, `Z_DATA_ERROR` if the input data was corrupted (input stream not conforming to the ***zlib*** format or incorrect check value), `Z_STREAM_ERROR` if the stream structure was inconsistent (for example if `next_in` or `next_out` was `NULL`), `Z_MEM_ERROR` if there was not enough memory, `Z_BUF_ERROR` if no progress is possible or if there was not enough room in the output buffer when `Z_FINISH` is used. Note that `Z_BUF_ERROR` is not fatal, and `inflate()` can be called again with more input and more output space to continue decompressing. If `Z_DATA_ERROR` is returned, the application may then call `inflateSync()` to look for a good compression block if a partial recovery of the data is desired.

**`ZEXTERN int ZEXPORT inflateEnd OF((z_streamp strm));`**

All dynamically allocated data structures for this stream are freed. This function discards any unprocessed input and does not flush any pending output.

`inflateEnd` returns `Z_OK` if success, `Z_STREAM_ERROR` if the stream state was inconsistent. In the error case, `msg` may be set but then points to a static string (which must not be deallocated).

## Advanced Functions

The following functions are needed only in some special applications.

```
ZEXTERN int ZEXPORT deflateInit2 OF((z_streamp strm,
                                     int  level,
                                     int  method,
                                     int  windowBits,
                                     int  memLevel,
                                     int  strategy));
```

This is another version of `deflateInit` with more compression options. The fields `next_in`, `zalloc`, `zfree` and `opaque` must be initialized before by the caller.

The `method` parameter is the compression method. It must be `Z_DEFLATED` in this version of the library.

The `windowBits` parameter is the base two logarithm of the window size (the size of the history buffer). It should be in the range 8..15 for this version of the library. Larger values of this parameter result in better compression at the expense of memory usage. The default value is 15 if `deflateInit` is used instead.

`windowBits` can also be –8..–15 for raw deflate. In this case, `-windowBits` determines the window size. `deflate()` will then generate raw deflate data with no ***zlib*** header or trailer, and will not compute an adler32 check value.

`windowBits` can also be greater than 15 for optional ***gzip*** encoding. Add 16 to `windowBits` to write a simple ***gzip*** header and trailer around the compressed data instead of a ***zlib*** wrapper. The ***gzip*** header will have no file name, no extra data, no comment, no modification time (set to zero), no header crc, and the operating system will be set to 255 (unknown). If a ***gzip*** stream is being written, `strm->adler` is a crc32 instead of an adler32.

The `memLevel` parameter specifies how much memory should be allocated for the internal compression state. `memLevel=1` uses minimum memory but is slow and reduces compression ratio; `memLevel=9` uses maximum memory for optimal speed. The default value is 8. See zconf.h for total memory usage as a function of `windowBits` and `memLevel`.

The `strategy` parameter is used to tune the compression algorithm. Use the value `Z_DEFAULT_STRATEGY` for normal data, `Z_FILTERED` for data produced by a filter (or predictor), `Z_HUFFMAN_ONLY` to force Huffman encoding only (no string match), or `Z_RLE` to limit match distances to one (run-length encoding). Filtered data consists mostly of small values with a somewhat random distribution. In this case, the compression algorithm is tuned to compress them better. The effect of `Z_FILTERED` is to force more Huffman coding and less string matching; it is somewhat intermediate between `Z_DEFAULT_STRATEGY` and `Z_HUFFMAN_ONLY`. `Z_RLE` is designed to be almost as fast as

Z_HUFFMAN_ONLY, but give better compression for ***PNG*** image data. The strategy parameter only affects the compression ratio but not the correctness of the compressed output even if it is not set appropriately. Z_FIXED prevents the use of dynamic Huffman codes, allowing for a simpler decoder for special applications.

deflateInit2 returns Z_OK if success, Z_MEM_ERROR if there was not enough memory, Z_STREAM_ERROR if a parameter is invalid (such as an invalid method), or Z_VERSION_ERROR if the ***zlib*** library version (zlib_version) is incompatible with the version assumed by the caller (ZLIB_VERSION). msg is set to null if there is no error message. deflateInit2 does not perform any compression: this will be done by deflate().

```
ZEXTERN int ZEXPORT deflateSetDictionary OF((z_streamp strm,
                                             const Bytef *dictionary,
                                             uInt  dictLength));
```

Initializes the compression dictionary from the given byte sequence without producing any compressed output. This function must be called immediately after deflateInit, deflateInit2 or deflateReset, before any call of deflate. The compressor and decompressor must use exactly the same dictionary (see inflateSetDictionary).

The dictionary should consist of strings (byte sequences) that are likely to be encountered later in the data to be compressed, with the most commonly used strings preferably put towards the end of the dictionary. Using a dictionary is most useful when the data to be compressed is short and can be predicted with good accuracy; the data can then be compressed better than with the default empty dictionary.

Depending on the size of the compression data structures selected by deflateInit or deflateInit2, a part of the dictionary may in effect be discarded, for example if the dictionary is larger than the window size in deflate or deflate2. Thus the strings most likely to be useful should be put at the end of the dictionary, not at the front. In addition, the current implementation of deflate will use at most the window size minus 262 bytes of the provided dictionary.

Upon return of this function, strm->adler is set to the adler32 value of the dictionary; the decompressor may later use this value to determine which dictionary has been used by the compressor. (The adler32 value applies to the whole dictionary even if only a subset of the dictionary is actually used by the compressor.) If a raw deflate was requested, then the adler32 value is not computed and strm->adler is not set.

deflateSetDictionary returns Z_OK if success, or Z_STREAM_ERROR if a parameter is invalid (such as NULL dictionary) or the stream state is inconsistent (for example if deflate has already been called for this stream or if the compression method is bsort). deflateSetDictionary does not perform any compression: this will be done by deflate().

```
ZEXTERN int ZEXPORT deflateCopy OF((z_streamp dest,
                                    z_streamp source));
```

Sets the destination stream as a complete copy of the source stream.

This function can be useful when several compression strategies will be tried, for example when there are several ways of pre-processing the input data with a filter. The streams that will be discarded should then be freed by calling `deflateEnd`. Note that `deflateCopy` duplicates the internal compression state which can be quite large, so this strategy is slow and can consume lots of memory.

`deflateCopy` returns `Z_OK` if success, `Z_MEM_ERROR` if there was not enough memory, `Z_STREAM_ERROR` if the source stream state was inconsistent (such as `zalloc` being `NULL`). `msg` is left unchanged in both source and destination.

```
ZEXTERN int ZEXPORT deflateReset OF((z_streamp strm));
```

This function is equivalent to `deflateEnd` followed by `deflateInit`, but does not free and reallocate all the internal compression state. The stream will keep the same compression level and any other attributes that may have been set by `deflateInit2`.

`deflateReset` returns `Z_OK` if success, or `Z_STREAM_ERROR` if the source stream state was inconsistent (such as `zalloc` or state being `NULL`).

```
ZEXTERN int ZEXPORT deflateParams OF((z_streamp strm,
                                      int level,
                                      int strategy));
```

Dynamically update the compression level and compression strategy. The interpretation of `level` and `strategy` is as in `deflateInit2`. This can be used to switch between compression and straight copy of the input data, or to switch to a different kind of input data requiring a different strategy. If the compression level is changed, the input available so far is compressed with the old level (and may be flushed); the new level will take effect only at the next call of `deflate()`.

Before the call of `deflateParams`, the stream state must be set as for a call of `deflate()`, since the currently available input may have to be compressed and flushed. In particular, `strm->avail_out` must be non-zero.

`deflateParams` returns `Z_OK` if success, `Z_STREAM_ERROR` if the source stream state was inconsistent or if a parameter was invalid, `Z_BUF_ERROR` if `strm->avail_out` was zero.

```
ZEXTERN int ZEXPORT deflateTune OF((z_streamp strm,
                                    int good_length,
                                    int max_lazy,
                                    int nice_length,
                                     int max_chain));
```

Fine tune `deflate`'s internal compression parameters. This should only be used by someone who understands the algorithm used by ***zlib***'s `deflate` for searching for the best matching string, and even then only by the most fanatic optimizer trying to squeeze out the last compressed bit for their specific input data. Read the deflate.c source code for the meaning of the `max_lazy`, `good_length`, `nice_length`, and `max_chain` parameters.

`deflateTune()` can be called after `deflateInit()` or `deflateInit2()`, and returns `Z_OK` on success, or `Z_STREAM_ERROR` for an invalid deflate stream.

```
ZEXTERN uLong ZEXPORT deflateBound OF((z_streamp strm,
                                       uLong sourceLen));
```

`deflateBound()` returns an upper bound on the compressed size after deflation of `sourceLen` bytes. It must be called after `deflateInit()` or `deflateInit2()`. This would be used to allocate an output buffer for deflation in a single pass, and so would be called before `deflate()`.

```
ZEXTERN int ZEXPORT deflatePrime OF((z_streamp strm,
                                     int bits,
                                     int value));
```

`deflatePrime()` inserts bits in the deflate output stream. The intent is that this function is used to start off the deflate output with the bits leftover from a previous deflate stream when appending to it. As such, this function can only be used for raw deflate, and must be used before the first `deflate()` call after a `deflateInit2()` or `deflateReset()`. `bits` must be less than or equal to 16, and that many of the least significant bits of `value` will be inserted in the output.

`deflatePrime` returns `Z_OK` if success, or `Z_STREAM_ERROR` if the source stream state was inconsistent.

```
ZEXTERN int ZEXPORT deflateSetHeader OF((z_streamp strm,
                                         gz_headerp head));
```

`deflateSetHeader()` provides ***gzip*** header information for when a ***gzip*** stream is requested by `deflateInit2()`. `deflateSetHeader()` may be called after `deflateInit2()` or `deflateReset()` and before the first call of `deflate()`. The text, time, os, extra field, name, and comment information in the provided gz_header structure are written to the ***gzip*** header (xflag is ignored — the extra flags are set according to the compression level). The caller must assure that, if not `Z_NULL`, name and comment are terminated with a zero byte, and that if extra is not `Z_NULL`, that extra_len bytes are available there. If hcrc is true, a ***gzip*** header crc is included. Note that the current versions of the command-line version of ***gzip*** (up through version 1.3.x) do not support header crc's, and will report that it is a "multi-part ***gzip*** file" and give up.

If `deflateSetHeader` is not used, the default ***gzip*** header has text false, the time set to zero, and os set to 255, with no extra, name, or comment fields. The ***gzip*** header is returned to the default state by `deflateReset()`.

`deflateSetHeader` returns `Z_OK` if success, or `Z_STREAM_ERROR` if the source stream state was inconsistent.

```
ZEXTERN int ZEXPORT inflateInit2 OF((z_streamp strm,
                                     int  windowBits));
```

This is another version of `inflateInit` with an extra parameter. The fields `next_in`, `avail_in`, `zalloc`, `zfree` and `opaque` must be initialized before by the caller.

The `windowBits` parameter is the base two logarithm of the maximum window size (the size of the history buffer). It should be in the range 8..15 for this version of the library. The default value is 15 if `inflateInit` is used instead. `windowBits` must be greater than or equal to the `windowBits` value provided to `deflateInit2()` while compressing, or it must be equal to 15 if `deflateInit2()` was not used. If a compressed stream with a larger window size is given as input, `inflate()` will return with the error code `Z_DATA_ERROR` instead of trying to allocate a larger window.

`windowBits` can also be zero to request that `inflate` use the window size in the zlib header of the compressed stream.

`windowBits` can also be –8..–15 for raw inflate. In this case, `-windowBits` determines the window size. `inflate()` will then process raw deflate data, not looking for a ***zlib*** or ***gzip*** header, not generating a check value, and not looking for any check values for comparison at the end of the stream. This is for use with other formats that use the deflate compressed data format such as ***zip***. Those formats provide their own check values. If a custom format is developed using the raw deflate format for compressed data, it is recommended that a check value such as an adler32 or a crc32 be applied to the uncompressed data as is done in the ***zlib***, ***gzip***, and ***zip*** formats. For most applications, the ***zlib*** format should be used as is. Note that comments above on the use in `deflateInit2()` applies to the magnitude of `windowBits`.

`windowBits` can also be greater than 15 for optional ***gzip*** decoding. Add 32 to `windowBits` to enable ***zlib*** and ***gzip*** decoding with automatic header detection, or add 16 to decode only the ***gzip*** format (the ***zlib*** format will return a `Z_DATA_ERROR`). If a ***gzip*** stream is being decoded, `strm->adler` is a crc32 instead of an adler32.

`inflateInit2` returns `Z_OK` if success, `Z_MEM_ERROR` if there was not enough memory, `Z_VERSION_ERROR` if the zlib library version is incompatible with the version assumed by the caller, `Z_STREAM_ERROR` if a parameters are invalid, such as a null pointer to the structure. `msg` is set to null if there is no error message. `inflateInit2` does not perform any decompression apart from reading the ***zlib*** header if present: actual decompression be done by `inflate()`. (So `next_in` and `avail_in` may be modified, but `next_out` and `avail_out` are unused and unchanged.) The current implementation of `inflateInit2()` does not process any header information -- that is deferred until `inflate()` is called.

```
ZEXTERN int ZEXPORT inflateSetDictionary OF((z_streamp strm,
                                             const Bytef *dictionary,
                                             uInt  dictLength));
```

Initializes the decompression dictionary from the given uncompressed byte sequence. This function must be called immediately after a call of `inflate`, if that call returned `Z_NEED_DICT`. The dictionary chosen by the compressor can be determined from the adler32 value returned by that call of `inflate`. The compressor and decompressor must use exactly the same dictionary (see `deflateSetDictionary`). For raw inflate, this function can be called immediately after `inflateInit2()` or `inflateReset()` and before any call of `inflate()` to set the dictionary. The application must insure that the dictionary that was used for compression is provided.

`inflateSetDictionary` returns `Z_OK` if success, `Z_STREAM_ERROR` if a parameter is invalid (such as `NULL` dictionary) or the stream state is inconsistent, `Z_DATA_ERROR` if the given dictionary doesn't match the expected one (incorrect adler32 value). `inflateSetDictionary` does not perform any decompression: this will be done by subsequent calls of `inflate()`.

```
ZEXTERN int ZEXPORT inflateSync OF((z_streamp strm));
```

Skips invalid compressed data until a full flush point (see above the description of `deflate` with `Z_FULL_FLUSH`) can be found, or until all available input is skipped. No output is provided.

`inflateSync` returns `Z_OK` if a full flush point has been found, `Z_BUF_ERROR` if no more input was provided, `Z_DATA_ERROR` if no flush point has been found, or `Z_STREAM_ERROR` if the stream structure was inconsistent. In the success case, the application may save the current current value of `total_in` which indicates where valid compressed data was found. In the error case, the application may repeatedly call `inflateSync`, providing more input each time, until success or end of the input data.

```
ZEXTERN int ZEXPORT inflateCopy OF((z_streamp dest,
                                    z_streamp source));
```

Sets the destination stream as a complete copy of the source stream.

This function can be useful when randomly accessing a large stream. The first pass through the stream can periodically record the `inflate` state, allowing restarting `inflate` at those points when randomly accessing the stream.

`inflateCopy` returns `Z_OK` if success, `Z_MEM_ERROR` if there was not enough memory, `Z_STREAM_ERROR` if the source stream state was inconsistent (such as `zalloc` being `NULL`). `msg` is left unchanged in both source and destination.

```
ZEXTERN int ZEXPORT inflateReset OF((z_streamp strm));
```

This function is equivalent to `inflateEnd` followed by `inflateInit`, but does not free and reallocate all the internal decompression state. The stream will keep attributes that may have been set by `inflateInit2`.

`inflateReset` returns `Z_OK` if success, or `Z_STREAM_ERROR` if the source stream state was inconsistent (such as `zalloc` or state being `NULL`).

```
ZEXTERN int ZEXPORT inflateReset2 OF((z_streamp strm,
                                      int windowBits));
```

This function is the same as `inflateReset`, but it also permits changing the wrap and window size requests. The `windowBits` parameter is interpreted the same as it is for `inflateInit2`.

`inflateReset2` returns `Z_OK` if success, or `Z_STREAM_ERROR` if the source stream state was inconsistent (such as `zalloc` or `state` being `Z_NULL`), or if the `windowBits parameter` is invalid.

```
ZEXTERN int ZEXPORT inflatePrime OF((z_streamp strm,
                                     int bits,
                                     int value));
```

This function inserts bits in the `inflate` input stream. The intent is that this function is used to start inflating at a bit position in the middle of a byte. The provided bits will be used before any bytes are used from `next_in`. This function should only be used with raw inflate, and should be used before the first `inflate()` call after `inflateInit2()` or `inflateReset()`. `bits` must be less than or equal to 16, and that many of the least significant bits of `value` will be inserted in the input.

If `bits` is negative, then the input stream bit buffer is emptied. Then `inflatePrime()` can be called again to put bits in the buffer. This is used to clear out bits leftover after feeding inflate a block description prior to feeding inflate codes.

`inflatePrime` returns `Z_OK` if success, or `Z_STREAM_ERROR` if the source stream state was inconsistent.

```
ZEXTERN long ZEXPORT inflateMark OF((z_streamp strm));
```

This function returns two values, one in the lower 16 bits of the return value, and the other in the remaining upper bits, obtained by shifting the return value down 16 bits. If the upper value is −1 and the lower value is zero, then `inflate()` is currently decoding information outside of a block. If the upper value is −1 and the lower value is non-zero, then inflate is in the middle of a stored block, with the lower value equaling the number of bytes from the input remaining to copy. If the upper value is not −1, then it is the number of bits back from the current bit position in the input of the code (literal or length/distance pair) currently being processed. In that case the lower value is the number of bytes already emitted for that code.

A code is being processed if `inflate` is waiting for more input to complete decoding of the code, or if it has completed decoding but is waiting for more output space to write the literal or match data.

`inflateMark()` is used to mark locations in the input data for random access, which may be at bit positions, and to note those cases where the output of a code may span boundaries of random access blocks. The current location in the input stream can be determined from `avail_in` and `data_type` as noted in the description for the `Z_BLOCK` flush parameter for `inflate`.

`inflateMark` returns the value noted above or `-1 << 16` if the provided source stream state was inconsistent.

```
ZEXTERN int ZEXPORT inflateGetHeader OF((z_streamp strm,
                                         gz_headerp head));
```

`inflateGetHeader()` requests that ***gzip*** header information be stored in the provided gz_header structure. `inflateGetHeader()` may be called after `inflateInit2()` or `inflateReset()`, and before the first call of `inflate()`. As `inflate()` processes the ***gzip*** stream, `head->done` is zero until the header is completed, at which time `head->done` is set to one. If a ***zlib*** stream is being decoded, then `head->done` is set to −1 to indicate that there will be no ***gzip*** header information forthcoming. Note that `Z_BLOCK` can be used to force `inflate()` to return immediately after header processing is complete and before any actual data is decompressed.

The text, time, xflags, and os fields are filled in with the ***gzip*** header contents. hcrc is set to true if there is a header CRC. (The header CRC was valid if done is set to one.) If extra is not `Z_NULL`, then extra_max contains the maximum number of bytes to write to extra. Once done is true, extra_len contains the actual extra field length, and extra contains the extra field, or that field truncated if extra_max is less than extra_len. If name is not `Z_NULL`, then up to name_max characters are written there, terminated with a zero unless the length is greater than name_max. If comment is not `Z_NULL`, then up to comm_max characters are written there, terminated with a zero unless the length is greater than comm_max. When any of extra, name, or comment are not `Z_NULL` and the respective field is not present in the header, then that field is set to `Z_NULL` to signal its absence. This allows the use of `deflateSetHeader()` with the returned structure to duplicate the header. However if those fields are set to allocated memory, then the application will need to save those pointers elsewhere so that they can be eventually freed.

If `inflateGetHeader` is not used, then the header information is simply discarded. The header is always checked for validity, including the header CRC if present. `inflateReset()` will reset the process to discard the header information. The application would need to call `inflateGetHeader()` again to retrieve the header from the next ***gzip*** stream.

`inflateGetHeader` returns `Z_OK` if success, or `Z_STREAM_ERROR` if the source stream state was inconsistent.

```
ZEXTERN int ZEXPORT inflateBackInit OF((z_streamp strm, int windowBits,
                                        unsigned char FAR *window));
```

Initialize the internal stream state for decompression using `inflateBack()` calls. The fields `zalloc`, `zfree` and `opaque` in strm must be initialized before the call. If `zalloc` and `zfree` are `Z_NULL`, then the default library- derived memory allocation routines are used. `windowBits` is the base two logarithm of the window size, in the range 8..15. window is a caller supplied buffer of that size. Except for special applications where it is assured that `deflate` was used with small window sizes, `windowBits` must be 15 and a 32K byte window must be supplied to be able to decompress general deflate streams.

See `inflateBack()` for the usage of these routines.

`inflateBackInit` will return `Z_OK` on success, `Z_STREAM_ERROR` if any of the paramaters are invalid, `Z_MEM_ERROR` if the internal state could not be allocated, or `Z_VERSION_ERROR` if the version of the library does not match the version of the header file.

```
typedef unsigned (*in_func) OF((void FAR *, unsigned char FAR * FAR *));
typedef int (*out_func) OF((void FAR *, unsigned char FAR *, unsigned));
```

```
ZEXTERN int ZEXPORT inflateBack OF((z_streamp strm,
                                    in_func in, void FAR *in_desc,
                                    out_func out, void FAR *out_desc));
```

`inflateBack()` does a raw inflate with a single call using a call-back interface for input and output. This is more efficient than `inflate()` for file i/o applications in that it avoids copying between the output and the sliding window by simply making the window itself the output buffer. This function trusts the application to not change the output buffer passed by the output function, at least until `inflateBack()` returns.

`inflateBackInit()` must be called first to allocate the internal state and to initialize the state with the user-provided window buffer. `inflateBack()` may then be used multiple times to `inflate` a complete, raw deflate stream with each call. `inflateBackEnd()` is then called to free the allocated state.

A raw deflate stream is one with no ***zlib*** or ***gzip*** header or trailer. This routine would normally be used in a utility that reads ***zip*** or ***gzip*** files and writes out uncompressed files. The utility would decode the header and process the trailer on its own, hence this routine expects only the raw deflate stream to decompress. This is different from the normal behavior of `inflate()`, which expects either a ***zlib*** or ***gzip*** header and trailer around the deflate stream.

`inflateBack()` uses two subroutines supplied by the caller that are then called by `inflateBack()` for input and output. `inflateBack()` calls those routines until it reads a complete deflate stream and writes out all of the uncompressed data, or until it encounters an error. The function's parameters and return types are defined above in the `in_func` and `out_func` typedefs. `inflateBack()` will call `in (in_desc, &buf)` which should return the number of bytes of provided input, and a pointer to that input in buf. If there is no input available, `in()` must return zero—`buf` is ignored in that case—and `inflateBack()` will return a buffer error. `inflateBack()` will call `out(out_desc, buf, len)` to write the uncompressed data `buf[0..len-1]`. `out()` should return zero on success, or non-zero on failure. If `out()` returns non-zero, `inflateBack()` will return with an error. Neither `in()` nor `out()` are permitted to change the contents of the window provided to `inflateBackInit()`, which is also the buffer that `out()` uses to write from. The length written by `out()` will be at most the window size. Any non-zero amount of input may be provided by `in()`.

For convenience, `inflateBack()` can be provided input on the first call by setting `strm->next_in` and `strm->avail_in`. If that input is exhausted, then `in()` will be called. Therefore `strm->next_in` must be initialized before calling `inflateBack()`. If `strm->next_in` is `Z_NULL`, then `in()` will be called immediately for input. If `strm->next_in` is not `Z_NULL`, then `strm->avail_in` must also be initialized, and then if `strm->avail_in` is not zero, input will initially be taken from `strm->next_in [0 .. strm->avail_in - 1]`.

The `in_desc` and `out_desc` parameters of `inflateBack()` is passed as the first parameter of `in()` and `out()` respectively when they are called. These descriptors can be optionally used to pass any information that the caller- supplied `in()` and `out()` functions need to do their job.

On return, `inflateBack()` will set `strm->next_in` and `strm->avail_in` to pass back any unused input that was provided by the last `in()` call. The return values of `inflateBack()` can be `Z_STREAM_END` on success, `Z_BUF_ERROR` if `in()` or `out()` returned an error, `Z_DATA_ERROR` if there was a format error in the deflate stream (in which case `strm->msg` is set to indicate the nature of the error), or `Z_STREAM_ERROR` if the stream was not properly initialized. In the case of `Z_BUF_ERROR`, an input or output error can be distinguished using `strm->next_in` which will be `Z_NULL` only if `in()` returned an error. If `strm->next_in` is not `Z_NULL`, then the `Z_BUF_ERROR` was due to `out()` returning non-zero. (`in()` will always be called before `out()`, so `strm->next_in` is assured to be defined if `out()` returns non-zero.) Note that `inflateBack()` cannot return `Z_OK`.

**`ZEXTERN int ZEXPORT inflateBackEnd OF((z_streamp strm));`**

All memory allocated by `inflateBackInit()` is freed.

`inflateBackEnd()` returns `Z_OK` on success, or `Z_STREAM_ERROR` if the stream state was inconsistent.

**`ZEXTERN uLong ZEXPORT zlibCompileFlags OF((void));`**

Return flags indicating compile-time options.

Type sizes, two bits each, 00 = 16 bits, 01 = 32, 10 = 64, 11 = other:

- 1.0: size of `uInt`
- 3.2: size of `uLong`
- 5.4: size of `voidpf` (pointer)
- 7.6: size of `z_off_t`

Compiler, assembler, and debug options:

- 8: `DEBUG`
- 9: `ASMV` or `ASMINF` — use ASM code
- 10: `ZLIB_WINAPI` — exported functions use the `WINAPI` calling convention
- 11: 0 (reserved)

One-time table building (smaller code, but not thread-safe if true):

- 12: `BUILDFIXED` — build static block decoding tables when needed
- 13: `DYNAMIC_CRC_TABLE` — build CRC calculation tables when needed
- 14,15: 0 (reserved)

Library content (indicates missing functionality):

- 16: `NO_GZCOMPRESS` — `gz*` functions cannot compress (to avoid linking deflate code when not needed)
- 17: `NO_GZIP` — `deflate` can't write ***gzip*** streams, and `inflate` can't detect and decode ***gzip*** streams (to avoid linking crc code)
- 18-19: 0 (reserved)

Operation variations (changes in library functionality):

- 20: `PKZIP_BUG_WORKAROUND` — slightly more permissive `inflate`
- 21: `FASTEST` — `deflate` algorithm with only one, lowest compression level
- 22,23: 0 (reserved)

The `sprintf` variant used by `gzprintf` (zero is best):

- 24: 0 = `vs*`, 1 = `s*` — 1 means limited to 20 arguments after the format
- 25: 0 = `*nprintf`, 1 = `*printf` — 1 means `gzprintf()` not secure!
- 26: 0 = returns value, 1 = `void` — 1 means inferred string length returned

Remainder:

- 27-31: 0 (reserved)

## Utility Functions

The following utility functions are implemented on top of the basic stream-oriented functions. To simplify the interface, some default options are assumed (compression level and memory usage, standard memory allocation functions). The source code of these utility functions can easily be modified if you need special options.

```
ZEXTERN int ZEXPORT compress OF((Bytef *dest, uLongf *destLen,
                                 const Bytef *source, uLong sourceLen));
```

Compresses the source buffer into the destination buffer. `sourceLen` is the byte length of the source buffer. Upon entry, `destLen` is the total size of the destination buffer, which must be at least the value returned by `compressBound(sourceLen)`. Upon exit, `destLen` is the actual size of the compressed buffer.

compress returns `Z_OK` if success, `Z_MEM_ERROR` if there was not enough memory, `Z_BUF_ERROR` if there was not enough room in the output buffer.

```
ZEXTERN int ZEXPORT compress2 OF((Bytef *dest, uLongf *destLen,
                                  const Bytef *source, uLong sourceLen,
                                  int level));
```

Compresses the source buffer into the destination buffer. The `level` parameter has the same meaning as in `deflateInit`. `sourceLen` is the byte length of the source buffer. Upon entry, `destLen` is the total size of the destination buffer, which must be at least the value returned by `compressBound (sourceLen)`. Upon exit, `destLen` is the actual size of the compressed buffer.

`compress2` returns `Z_OK` if success, `Z_MEM_ERROR` if there was not enough memory, `Z_BUF_ERROR` if there was not enough room in the output buffer, `Z_STREAM_ERROR` if the `level` parameter is invalid.

```
ZEXTERN uLong ZEXPORT compressBound OF((uLong sourceLen));
```

`compressBound()` returns an upper bound on the compressed size after `compress()` or `compress2 ()` on `sourceLen` bytes. It would be used before a `compress()` or `compress2()` call to allocate the destination buffer.

```
ZEXTERN int ZEXPORT uncompress OF((Bytef *dest, uLongf *destLen,
                                   const Bytef *source, uLong sourceLen));
```

Decompresses the source buffer into the destination buffer. `sourceLen` is the byte length of the source buffer. Upon entry, `destLen` is the total size of the destination buffer, which must be large enough to hold the entire uncompressed data. (The size of the uncompressed data must have been saved previously by the compressor and transmitted to the decompressor by some mechanism outside the scope of this compression library.) Upon exit, `destLen` is the actual size of the compressed buffer.

`uncompress` returns `Z_OK` if success, `Z_MEM_ERROR` if there was not enough memory, `Z_BUF_ERROR` if there was not enough room in the output buffer, or `Z_DATA_ERROR` if the input data was corrupted or incomplete.

## gzip File Access Functions

This library supports reading and writing files in ***gzip*** (.gz) format with an interface similar to that of stdio, using the functions that start with "`gz`". The ***gzip*** format is different from the ***zlib*** format. ***gzip*** is a gzip wrapper, documented in RFC 1952, wrapped around a deflate stream.

**`typedef voidp gzFile;       /* opaque gzip file descriptor */`**

**`ZEXTERN gzFile ZEXPORT gzopen OF((const char *path, const char *mode));`**

Opens a ***gzip*** (.gz) file for reading or writing. The `mode` parameter is as in `fopen` (`"rb"` or `"wb"`) but can also include a compression level (`"wb9"`) or a strategy: `'f'` for filtered data as in `"wb6f"`, `'h'` for Huffman-only compression as in `"wb1h"`, `'R'` for run-length encoding as in `"wb1R"`, or 'F' for fixed code compression as in "wb9F". (See the description of `deflateInit2` for more information about the `strategy` parameter.) Also "a" can be used instead of "w" to request that the ***gzip*** stream that will be written be appended to the file. `"+"` will result in an error, since reading and writing to the same gzip file is not supported.

`gzopen` can be used to read a file which is not in ***gzip*** format; in this case `gzread` will directly read from the file without decompression.

`gzopen` returns `NULL` if the file could not be opened, if there was insufficient memory to allocate the `gzFile` state, or if an invalid mode was specified (an 'r', 'w', or 'a' was not provided, or '+' was provided). `errno` can be checked to determine if the reason `gzopen` failed was that the file could not be opened.

**`ZEXTERN gzFile ZEXPORT gzdopen OF((int fd, const char *mode));`**

`gzdopen()` associates a `gzFile` with the file descriptor `fd`. File descriptors are obtained from calls like `open`, `dup`, `creat`, `pipe` or `fileno` (in the file has been previously opened with `fopen`). The mode parameter is as in `gzopen`. The next call of `gzclose` on the returned `gzFile` will also close the file descriptor `fd`, just like `fclose(fdopen(fd), mode)` closes the file descriptor `fd`. If you want to keep `fd` open, use `fd = dup(fd_keep); gz = gzdopen(fd, mode);`. The duplicated descriptor should be saved to avoid a leak, since `gzdopen` does not close `fd` if it fails.

`gzdopen` returns `NULL` if there was insufficient memory to allocate the `gzFile` state, if an invalid `mode` was specified (an 'r', 'w', or 'a' was not provided, or '+' was provided), or if `fd` is `-1`. The file descriptor is not used until the next `gz*` read, write, seek, or close operation, so `gzdopen` will not detect if `fd` is invalid (unless `fd` is −1).

```
ZEXTERN int ZEXPORT gzbuffer OF((gzFile file, unsigned size));
```

Set the internal buffer size used by this library's functions. The default buffer size is 8192 bytes. This function must be called after `gzopen()` or `gzdopen()`, and before any other calls that read or write the file. The buffer memory allocation is always deferred to the first read or write. Two buffers are allocated, either both of the specified size when writing, or one of the specified size and the other twice that size when reading. A larger buffer size of, for example, 64K or 128K bytes will noticeably increase the speed of decompression (reading).

The new buffer size also affects the maximum length for `gzprintf()`.

`gzbuffer()` returns 0 on success, or −1 on failure, such as being called too late.

```
ZEXTERN int ZEXPORT gzsetparams OF((gzFile file, int level, int strategy));
```

Dynamically update the compression `level` or `strategy`. See the description of `deflateInit2` for the meaning of these parameters.

`gzsetparams` returns `Z_OK` if success, or `Z_STREAM_ERROR` if the file was not opened for writing.

```
ZEXTERN int ZEXPORT gzread OF((gzFile file, voidp buf, unsigned len));
```

Reads the given number of uncompressed bytes from the compressed file. If the input file was not in ***gzip*** format, `gzread` copies the given number of bytes into the buffer.

After reaching the end of a ***gzip*** stream in the input, `gzread` will continue to read, looking for another ***gzip*** stream, or failing that, reading the rest of the input file directly without decompression. The entire input file will be read if `gzread` is called until it returns less than the requested `len`.

`gzread` returns the number of uncompressed bytes actually read (0 for end of file, −1 for error).

```
ZEXTERN int ZEXPORT gzwrite OF((gzFile file,
                                voidpc buf, unsigned len));
```

Writes the given number of uncompressed bytes into the compressed file. `gzwrite` returns the number of uncompressed bytes actually written or 0 in case of error.

```
ZEXTERN int ZEXPORTVA gzprintf OF((gzFile file, const char *format, ...));
```

Converts, formats, and writes the arguments to the compressed file under control of the `format` string, as in `fprintf`. `gzprintf` returns the number of uncompressed bytes actually written, or 0 in case of error. The number of uncompressed bytes written is limited to 8191, or one less than the buffer size given to `gzbuffer()`. The caller should assure that this limit is not exceeded. If it is exceeded, then `gzprintf()` will return return an error (0) with nothing written. In this case, there may also be a buffer overflow with unpredictable consequences, which is possible only if ***zlib*** was compiled with the insecure functions `sprintf()` or `vsprintf()` because the secure `snprintf()` or `vsnprintf()` functions were not available. This can be determined using `zlibCompileFlags()`.

**`ZEXTERN int ZEXPORT gzputs OF((gzFile file, const char *s));`**

Writes the given null-terminated string to the compressed file, excluding the terminating null character.

`gzputs` returns the number of characters written, or −1 in case of error.

**`ZEXTERN char * ZEXPORT gzgets OF((gzFile file, char *buf, int len));`**

Reads bytes from the compressed file until `len-1` characters are read, or a newline character is read and transferred to `buf`, or an end-of-file condition is encountered. If any characters are read or if `len == 1`, the string is terminated with a null character. If no characters are read due to an end-of-file or `len < 1`, then the buffer is left untouched.

`gzgets` returns `buf` which is a null-terminated string, or it returns `NULL` for end-of-file or in case of error. If there was an error, the contents at `buf` are indeterminate.

**`ZEXTERN int ZEXPORT gzputc OF((gzFile file, int c));`**

Writes `c`, converted to an `unsigned char`, into the compressed file. `gzputc` returns the value that was written, or −1 in case of error.

**`ZEXTERN int ZEXPORT gzgetc OF((gzFile file));`**

Reads one byte from the compressed file. `gzgetc` returns this byte or −1 in case of end of file or error.

**`ZEXTERN int ZEXPORT gzungetc OF((int c, gzFile file));`**

Push one character back onto the stream to be read as the first character on the next read. At least one character of push-back is allowed. `gzungetc()` returns the character pushed, or −1 on failure. `gzungetc()` will fail if `c` is −1, and may fail if a character has been pushed but not read yet. If `gzungetc` is used immediately after `gzopen` or `gzdopen`, at least the output buffer size of pushed characters is allowed. (See `gzbuffer` above.) The pushed character will be discarded if the stream is repositioned with `gzseek()` or `gzrewind()`.

**`ZEXTERN int ZEXPORT gzflush OF((gzFile file, int flush));`**

Flushes all pending output into the compressed file. The parameter `flush` is as in the `deflate()` function. The return value is the ***zlib*** error number (see function `gzerror` below). `gzflush` is only permitted when writing.

If the flush parameter is `Z_FINISH`, the remaining data is written and the ***gzip*** stream is completed in the output. If `gzwrite()` is called again, a new ***gzip*** stream will be started in the output. `gzread()` is able to read such concatented ***gzip*** streams.

`gzflush` should be called only when strictly necessary because it will degrade compression if called too often.

```
ZEXTERN z_off_t ZEXPORT gzseek OF((gzFile file,
                                   z_off_t offset, int whence));
```

Sets the starting position for the next `gzread` or `gzwrite` on the given compressed file. The offset represents a number of bytes in the uncompressed data stream. The `whence` parameter is defined as in `lseek`(2); the value `SEEK_END` is not supported.

If the file is opened for reading, this function is emulated but can be extremely slow. If the file is opened for writing, only forward seeks are supported; `gzseek` then compresses a sequence of zeroes up to the new starting position.

`gzseek` returns the resulting offset location as measured in bytes from the beginning of the uncompressed stream, or −1 in case of error, in particular if the file is opened for writing and the new starting position would be before the current position.

```
ZEXTERN int ZEXPORT gzrewind OF((gzFile file));
```

Rewinds the given file. This function is supported only for reading.

`gzrewind(file)` is equivalent to `(int)gzseek(file, 0L, SEEK_SET)`

```
ZEXTERN z_off_t ZEXPORT gztell OF((gzFile file));
```

Returns the starting position for the next `gzread` or `gzwrite` on the given compressed file. This position represents a number of bytes in the uncompressed data stream, and is zero when starting, even if appending or reading a ***gzip*** stream from the middle of a file using `gzdopen()`.

`gztell(file)` is equivalent to `gzseek(file, 0L, SEEK_CUR)`

```
ZEXTERN z_off_t ZEXPORT gzoffset OF((gzFile file));
```

Returns the current offset in the file being read or written. This offset includes the count of bytes that precede the ***gzip*** stream, for example when appending or when using ***gzdopen()*** for reading. When reading, the offset does not include as yet unused buffered input. This information can be used for a progress indicator. On error, ***gzoffset()*** returns –1.

**`ZEXTERN int ZEXPORT gzeof OF((gzFile file));`**

Returns true (1) if the end-of-file indicator has been set while reading, false (0) otherwise. Note that the end-of-file indicator is set only if the read tried to go past the end of the input, but came up short. Therefore, just like `feof()`, `gzeof()` may return false even if there is no more data to read, in the event that the last read request was for the exact number of bytes remaining in the input file. This will happen if the input file size is an exact multiple of the buffer size.

If `gzeof()` returns true, then the read functions will return no more data, unless the end-of-file indicator is reset by `gzclearerr()` and the input file has grown since the previous end of file was detected.

**`ZEXTERN int ZEXPORT gzdirect OF((gzFile file));`**

Returns true (1) if file is being copied directly while reading, or false (0) if file is a ***gzip*** stream being decompressed. This state can change from false to true while reading the input file if the end of a gzip stream is reached, but is followed by data that is not another ***gzip*** stream.

If the input file is empty, `gzdirect()` will return true, since the input does not contain a ***gzip*** stream.

If `gzdirect()` is used immediately after `gzopen()` or `gzdopen()` it will cause buffers to be allocated to allow reading the file to determine if it is a ***gzip*** file. Therefore if `gzbuffer()` is used, it should be called before `gzdirect()`.

**`ZEXTERN int ZEXPORT gzclose OF((gzFile file));`**

Flushes all pending output if necessary, closes the compressed file and deallocates the (de)compression state. Note that once file is closed, you cannot call `gzerror` with file, since its structures have been deallocated. `gzclose` must not be called more than once on the same file, just as `free` must not be called more than once on the same allocation.

`gzclose` will return `Z_STREAM_ERROR` if file is not valid, `Z_ERRNO` on a file operation error, or `Z_OK` on success.

**`ZEXTERN int ZEXPORT gzclose_r OF((gzFile file));`**
**`ZEXTERN int ZEXPORT gzclose_w OF((gzFile file));`**

Same as `gzclose()`, but `gzclose_r()` is only for use when reading, and `gzclose_w()` is only for use when writing or appending. The advantage to using these instead of `gzclose()` is that they avoid linking in ***zlib*** compression or decompression code that is not used when only reading or only writing respectively. If `gzclose()` is used, then both compression and decompression code will be included the application when linking to a static ***zlib*** library.

**`ZEXTERN const char * ZEXPORT gzerror OF((gzFile file, int *errnum));`**

Returns the error message for the last error which occurred on the given compressed `file`. `errnum` is set to ***zlib*** error number. If an error occurred in the file system and not in the compression library, `errnum` is set to `Z_ERRNO` and the application may consult `errno` to get the exact error code.

The application must not modify the returned string. Future calls to this function may invalidate the previously returned string. If file is closed, then the string previously returned by `gzerror` will no longer be available.

`gzerror()` should be used to distinguish errors from end-of-file for those functions above that do not distinguish those cases in their return values.

**`ZEXTERN void ZEXPORT gzclearerr OF((gzFile file));`**

Clears the error and end-of-file flags for `file`. This is analogous to the `clearerr()` function in `stdio`. This is useful for continuing to read a ***gzip*** file that is being written concurrently.

## Checksum Functions

These functions are not related to compression but are exported anyway because they might be useful in applications using the compression library.

**`ZEXTERN uLong ZEXPORT adler32 OF((uLong adler, const Bytef *buf, uInt len));`**

Update a running Adler-32 checksum with the bytes `buf[0..len-1]` and return the updated checksum. If `buf` is `NULL`, this function returns the required initial value for the checksum.

An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much faster.

Usage example:

```
     uLong adler = adler32(0L, Z_NULL, 0);

     while (read_buffer(buffer, length) != EOF) {
       adler = adler32(adler, buffer, length);
     }
     if (adler != original_adler) error();
```

```
ZEXTERN uLong ZEXPORT adler32_combine OF((uLong adler1, uLong adler2,
                                          z_off_t len2));
```

Combine two Adler-32 checksums into one. For two sequences of bytes, `seq1` and `seq2` with lengths `len1` and `len2`, Adler-32 checksums were calculated for each, `adler1` and `adler2`. `adler32_combine`

`()` returns the Adler-32 checksum of `seq1` and `seq2` concatenated, requiring only `adler1`, `adler2`, and `len2`.

**`ZEXTERN uLong ZEXPORT crc32 OF((uLong crc, const Bytef *buf, uInt len));`**

Update a running CRC-32 with the bytes `buf[0..len-1]` and return the updated CRC-32. If `buf` is `Z_NULL`, this function returns the required initial value for the for the crc. Pre- and post-conditioning (one's complement) is performed within this function so it shouldn't be done by the application.

Usage example:

```
     uLong crc = crc32(0L, Z_NULL, 0);

     while (read_buffer(buffer, length) != EOF) {
       crc = crc32(crc, buffer, length);
     }
     if (crc != original_crc) error();
```

**`ZEXTERN uLong ZEXPORT crc32_combine OF((uLong crc1, uLong crc2, z_off_t len2));`**

Combine two CRC-32 check values into one. For two sequences of bytes, `seq1` and `seq2` with lengths `len1` and `len2`, CRC-32 check values were calculated for each, `crc1` and `crc2`. `crc32_combine()` returns the CRC-32 check value of `seq1` and `seq2` concatenated, requiring only `crc1`, `crc2`, and `len2`.

## Undocumented Functions

```
ZEXTERN const char   * ZEXPORT zError           OF((int));
ZEXTERN int            ZEXPORT inflateSyncPoint OF((z_streamp z));
ZEXTERN const uLongf * ZEXPORT get_crc_table    OF((void));
ZEXTERN int            ZEXPORT inflateUndermine OF((z_streamp, int));
```