***zlib* 1.2.11 Manual**

**Contents**

1. [Prologue](https://www.zlib.net/manual.html#Prologue)
2. [Version](https://www.zlib.net/manual.html#Version)
3. [Introduction](https://www.zlib.net/manual.html#Introduction)
4. [Stream Data Structures](https://www.zlib.net/manual.html#Stream)
5. [Structures Usage](https://www.zlib.net/manual.html#Usage)
6. [Constants](https://www.zlib.net/manual.html#Constants)
7. [Basic Functions](https://www.zlib.net/manual.html#Basic)
8. [Advanced Functions](https://www.zlib.net/manual.html#Advanced)
9. [Utility Functions](https://www.zlib.net/manual.html#Utility)
10. [*gzip* File Access Functions](https://www.zlib.net/manual.html#Gzip)
11. [Checksum Functions](https://www.zlib.net/manual.html#Checksum)
12. [Undocumented Functions](https://www.zlib.net/manual.html#Undocumented)

**Prologue**

*zlib* general purpose compression library  
version 1.2.11, January 15th, 2017

Copyright (C) 1995-2017 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

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](http://tools.ietf.org/html/rfc1950) (*zlib* format), [rfc1951](http://tools.ietf.org/html/rfc1951) (*deflate* format) and [rfc1952](http://tools.ietf.org/html/rfc1952) (*gzip* format).

**Version**

**#define ZLIB\_VERSION "1.2.11"**

**#define ZLIB\_VERNUM 0x12b0**

**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* and raw *deflate* 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 the 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 {**

**z\_const Bytef \*next\_in; /\* next input byte \*/**

**uInt avail\_in; /\* number of bytes available at next\_in \*/**

**uLong total\_in; /\* total number of input bytes read so far \*/**

**Bytef \*next\_out; /\* next output byte will go here \*/**

**uInt avail\_out; /\* remaining free space at next\_out \*/**

**uLong total\_out; /\* total number of bytes output so far \*/**

**z\_const 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**

**for deflate, or the decoding state for inflate \*/**

**uLong adler; /\* Adler-32 or CRC-32 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. In that case, *zlib* is thread-safe. When zalloc and zfree are Z\_NULL on entry to the initialization function, they are set to internal routines that use the standard library functions malloc() and free().

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 by 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 for deflate().

**#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().
* Generate 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. Some output may be provided even if flush is zero.

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 buffer because there might be more output pending. See deflatePending(), which can be used if desired to determine whether or not there is more ouput in that case.

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 codes 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 or Z\_BUF\_ERROR, 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 in the first deflate call after deflateInit if all the compression is to be done in a single step. In order to complete in one call, avail\_out must be at least the value returned by deflateBound (see below). Then deflate is guaranteed to return Z\_STREAM\_END. If not enough output space is provided, deflate will not return Z\_STREAM\_END, and it must be called again as described above.

deflate() sets strm->adler to the Adler-32 checksum of all input read so far (that is, total\_in bytes). If a *gzip* stream is being generated, then strm->adler will be the CRC-32 checksum of the input read so far. (See deflateInit2 below.)

deflate() may update strm->data\_type if it can make a good guess about the input data type (Z\_BINARY or Z\_TEXT). If 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 Z\_NULL or the state was inadvertently written over by the application), or 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. In the current version of inflate, the provided input is not read or consumed. The allocation of a sliding window will be deferred to the first call of inflate (if the decompression does not complete on the first call). 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. Actual decompression will be done by inflate(). So next\_in, and avail\_in, 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), then next\_in and avail\_in are updated accordingly, and processing will resume at this point for the next call of inflate().
* Generate 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. If the caller of inflate() does not provide both available input and available output space, it is possible that there will be no progress made. 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. To assist in this, on return inflate() always sets 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 of the uncompressed data for the operation to complete. (The size of the uncompressed data may have been saved by the compressor for this purpose.) The use of Z\_FINISH is not required to perform an inflation in one step. However it may be used to inform inflate that a faster approach can be used for the single inflate() call. Z\_FINISH also informs inflate to not maintain a sliding window if the stream completes, which reduces inflate's memory footprint. If the stream does not complete, either because not all of the stream is provided or not enough output space is provided, then a sliding window will be allocated and inflate() can be called again to continue the operation as if Z\_NO\_FLUSH had been used.

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 effects of the flush parameter in this implementation are on the return value of inflate() as noted below, when inflate() returns early when Z\_BLOCK or Z\_TREES is used, and when inflate() avoids the allocation of memory for a sliding window when Z\_FINISH is used.

If a preset dictionary is needed after this call (see inflateSetDictionary below), inflate sets strm->adler to the Adler-32 checksum of the dictionary chosen by the compressor and returns Z\_NEED\_DICT; otherwise it sets strm->adler to the Adler-32 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 Adler-32 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 unless inflateGetHeader() is used. When processing *gzip*-wrapped *deflate* data, strm->adler32 is set to the CRC-32 of the output produced so far. The CRC-32 is checked against the *gzip* trailer, as is the uncompressed length, modulo 2^32.

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, in which case strm->msg points to a string with a more specific error), Z\_STREAM\_ERROR if the stream structure was inconsistent (for example next\_in or next\_out was Z\_NULL, or the state was inadvertently written over by the application), Z\_MEM\_ERROR if there was not enough memory, Z\_BUF\_ERROR if no progress was 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 to be attempted.

**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, or Z\_STREAM\_ERROR if the stream state was inconsistent.

**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.

For the current implementation of deflate(), a windowBits value of 8 (a window size of 256 bytes) is not supported. As a result, a request for 8 will result in 9 (a 512-byte window). In that case, providing 8 to inflateInit2() will result in an error when the *zlib* header with 9 is checked against the initialization of inflate(). The remedy is to not use 8 with deflateInit2() with this initialization, or at least in that case use 9 with inflateInit2().

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 a 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 the appropriate value, if the operating system was determined at compile time. If a *gzip* stream is being written, strm->adler is a CRC-32 instead of an Adler-32.

For raw *deflate* or *gzip* encoding, a request for a 256-byte window is rejected as invalid, since only the *zlib* header provides a means of transmitting the window size to the decompressor.

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). without producing any compressed output. When using the *zlib* format, this function must be called immediately after deflateInit, deflateInit2 or deflateReset, and before any call of deflate. When doing raw *deflate*, this function must be called either before any call of deflate, or immediately after the completion of a *deflate* block, i.e. after all input has been consumed and all output has been delivered when using any of the flush options Z\_BLOCK, Z\_PARTIAL\_FLUSH, Z\_SYNC\_FLUSH, or Z\_FULL\_FLUSH. 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 deflateInit or deflateInit2. 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 Adler-32 value of the dictionary; the decompressor may later use this value to determine which dictionary has been used by the compressor. (The Adler-32 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 Adler-32 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 not at a block boundary for raw *deflate*). deflateSetDictionary does not perform any compression: this will be done by deflate().

**ZEXTERN int ZEXPORT deflateGetDictionary OF((z\_streamp strm,**

**Bytef \*dictionary,**

**uInt \*dictLength));**

Returns the sliding dictionary being maintained by deflate. dictLength is set to the number of bytes in the dictionary, and that many bytes are copied to dictionary. dictionary must have enough space, where 32768 bytes is always enough. If deflateGetDictionary() is called with dictionary equal to Z\_NULL, then only the dictionary length is returned, and nothing is copied. Similary, if dictLength is Z\_NULL, then it is not set.

deflateGetDictionary() may return a length less than the window size, even when more than the window size in input has been provided. It may return up to 258 bytes less in that case, due to how *zlib*'s implementation of *deflate* manages the sliding window and lookahead for matches, where matches can be up to 258 bytes long. If the application needs the last window-size bytes of input, then that would need to be saved by the application outside of *zlib*.

deflateGetDictionary returns Z\_OK on success, or Z\_STREAM\_ERROR if the stream state is inconsistent.

**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 the internal compression state. The stream will leave the compression level and any other attributes that may have been set unchanged.

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 approach (which is a function of the level) or the strategy is changed, and if any input has been consumed in a previous deflate() call, then the input available so far is compressed with the old level and strategy using deflate(strm, Z\_BLOCK). There are three approaches for the compression levels 0, 1..3, and 4..9 respectively. The new level and strategy will take effect at the next call of deflate().

If a deflate(strm, Z\_BLOCK) is performed by deflateParams(), and it does not have enough output space to complete, then the parameter change will not take effect. In this case, deflateParams() can be called again with the same parameters and more output space to try again.

In order to assure a change in the parameters on the first try, the *deflate* stream should be flushed using deflate() with Z\_BLOCK or other flush request until strm.avail\_out is not zero, before calling deflateParams(). Then no more input data should be provided before the deflateParams() call. If this is done, the old level and strategy will be applied to the data compressed before deflateParams(), and the new level and strategy will be applied to the the data compressed after deflateParams().

deflateParams returns Z\_OK on success, Z\_STREAM\_ERROR if the source stream state was inconsistent or if a parameter was invalid, or Z\_BUF\_ERROR if there was not enough output space to complete the compression of the available input data before a change in the strategy or approach. Note that in the case of a Z\_BUF\_ERROR, the parameters are not changed. A return value of Z\_BUF\_ERROR is not fatal, in which case deflateParams() can be retried with more output space.

**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(). If that first deflate() call is provided the sourceLen input bytes, an output buffer allocated to the size returned by deflateBound(), and the flush value Z\_FINISH, then deflate() is guaranteed to return Z\_STREAM\_END. Note that it is possible for the compressed size to be larger than the value returned by deflateBound() if flush options other than Z\_FINISH or Z\_NO\_FLUSH are used.

**ZEXTERN int ZEXPORT deflatePending OF((z\_streamp strm,**

**unsigned \*pending,**

**int \*bits));**

deflatePending() returns the number of bytes and bits of output that have been generated, but not yet provided in the available output. The bytes not provided would be due to the available output space having being consumed. The number of bits of output not provided are between 0 and 7, where they await more bits to join them in order to fill out a full byte. If pending or bits are Z\_NULL, then those values are not set.

deflatePending returns Z\_OK if success, or Z\_STREAM\_ERROR if the source stream state was inconsistent.

**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, Z\_BUF\_ERROR if there was not enough room in the internal buffer to insert the bits, 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 Adler-32 or a CRC-32 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 CRC-32 instead of an Adler-32. Unlike the gunzip utility and gzread() (see below), inflate() will not automatically decode concatenated *gzip* streams. inflate() will return Z\_STREAM\_END at the end of the *gzip* stream. The state would need to be reset to continue decoding a subsequent *gzip* stream.

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 Adler-32 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 at any time to set the dictionary. If the provided dictionary is smaller than the window and there is already data in the window, then the provided dictionary will amend what's there. 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 Adler-32 value). inflateSetDictionary does not perform any decompression: this will be done by subsequent calls of inflate().

**ZEXTERN int ZEXPORT inflateGetDictionary OF((z\_streamp strm,**

**Bytef \*dictionary,**

**uInt \*dictLength));**

Returns the sliding dictionary being maintained by inflate. dictLength is set to the number of bytes in the dictionary, and that many bytes are copied to dictionary. dictionary must have enough space, where 32768 bytes is always enough. If inflateGetDictionary() is called with dictionary equal to Z\_NULL, then only the dictionary length is returned, and nothing is copied. Similary, if dictLength is Z\_NULL, then it is not set.

inflateGetDictionary returns Z\_OK on success, or Z\_STREAM\_ERROR if the stream state is inconsistent.

**ZEXTERN int ZEXPORT inflateSync OF((z\_streamp strm));**

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

inflateSync searches for a 00 00 FF FF pattern in the compressed data. All full flush points have this pattern, but not all occurrences of this pattern are full flush points.

inflateSync returns Z\_OK if a possible 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 the internal decompression state. The 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. If the window size is changed, then the memory allocated for the window is freed, and the window will be reallocated by inflate() if needed.

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 –65536 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 parameters 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 \*,**

**z\_const 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 potentially 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. inflate() can be faster on modern CPUs when used with large buffers. inflateBack() 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 default behavior of inflate(), which expects a *zlib* 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: ZLIB\_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 data. compress() is equivalent to compress2() with a level parameter of Z\_DEFAULT\_COMPRESSION.

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 data.

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 uncompressed data.

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. In the case where there is not enough room, uncompress() will fill the output buffer with the uncompressed data up to that point.

**ZEXTERN int ZEXPORT uncompress2 OF((Bytef \*dest, uLongf \*destLen,**

**const Bytef \*source, uLong \*sourceLen));**

Same as uncompress, except that sourceLen is a pointer, where the length of the source is \*sourceLen. On return, \*sourceLen is the number of source bytes consumed.

***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 struct gzFile\_s \*gzFile; /\* semi-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.) 'T' will request transparent writing or appending with no compression and not using the *gzip* format.

'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. The addition of 'x' when writing will create the file exclusively, which fails if the file already exists. On systems that support it, the addition of 'e' when reading or writing will set the flag to close the file on an execve() call.

These functions, as well as *gzip*, will read and decode a sequence of *gzip* streams in a file. The append function of gzopen() can be used to create such a file. (Also see gzflush() for another way to do this.) When appending, gzopen does not test whether the file begins with a *gzip* stream, nor does it look for the end of the *gzip* streams to begin appending. gzopen will simply append a *gzip* stream to the existing file.

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. When reading, this will be detected automatically by looking for the magic two-byte *gzip* header.

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. If you are using fileno() to get the file descriptor from a FILE \*, then you will have to use dup() to avoid double-close()ing the file descriptor. Both gzclose() and fclose() will close the associated file descriptor, so they need to have different file descriptors.

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. Three times that size in buffer space is allocated. 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. Previously provided data is flushed before the parameter change.

gzsetparams returns Z\_OK if success, Z\_STREAM\_ERROR if the file was not opened for writing, Z\_ERRNO if there is an error writing the flushed data, or Z\_MEM\_ERROR if there is a memory allocation error.

**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 is not in *gzip* format, gzread copies the given number of bytes into the buffer directly from the file.

After reaching the end of a *gzip* stream in the input, gzread will continue to read, looking for another *gzip* stream. Any number of *gzip* streams may be concatenated in the input file, and will all be decompressed by gzread(). If something other than a *gzip* stream is encountered after a *gzip* stream, that remaining trailing garbage is ignored (and no error is returned).

gzread can be used to read a *gzip* file that is being concurrently written. Upon reaching the end of the input, gzread will return with the available data. If the error code returned by gzerror is Z\_OK or Z\_BUF\_ERROR, then gzclearerr can be used to clear the end of file indicator in order to permit gzread to be tried again. Z\_OK indicates that a *gzip* stream was completed on the last gzread. Z\_BUF\_ERROR indicates that the input file ended in the middle of a *gzip* stream. Note that gzread does not return –1 in the event of an incomplete *gzip* stream. This error is deferred until gzclose(), which will return Z\_BUF\_ERROR if the last gzread ended in the middle of a *gzip* stream. Alternatively, gzerror can be used before gzclose to detect this case.

gzread returns the number of uncompressed bytes actually read, less than len for end of file, or –1 for error. If len is too large to fit in an int, then nothing is read, –1 is returned, and the error state is set to Z\_STREAM\_ERROR.

**ZEXTERN z\_size\_t ZEXPORT gzfread OF((voidp buf, z\_size\_t size, z\_size\_t nitems,**

**gzFile file));**

Read up to nitems items of size size from file to buf, otherwise operating as gzread() does. This duplicates the interface of stdio's fread(), with size\_t request and return types. If the library defines size\_t, then z\_size\_t is identical to size\_t. If not, then z\_size\_t is an unsigned integer type that can contain a pointer.

gzfread() returns the number of full items read of size size, or zero if the end of the file was reached and a full item could not be read, or if there was an error. gzerror() must be consulted if zero is returned in order to determine if there was an error. If the multiplication of size and nitems overflows, i.e. the product does not fit in a z\_size\_t, then nothing is read, zero is returned, and the error state is set to Z\_STREAM\_ERROR.

In the event that the end of file is reached and only a partial item is available at the end, i.e. the remaining uncompressed data length is not a multiple of size, then the final partial item is nevetheless read into buf and the end-of-file flag is set. The length of the partial item read is not provided, but could be inferred from the result of gztell(). This behavior is the same as the behavior of fread() implementations in common libraries, but it prevents the direct use of gzfread() to read a concurrently written file, reseting and retrying on end-of-file, when size is not 1.

**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 z\_size\_t ZEXPORT gzfwrite OF((voidpc buf, z\_size\_t size,**

**z\_size\_t nitems, gzFile file));**

gzfwrite() writes nitems items of size size from buf to file, duplicating the interface of stdio's fwrite(), with size\_t request and return types. If the library defines size\_t, then z\_size\_t is identical to size\_t. If not, then z\_size\_t is an unsigned integer type that can contain a pointer.

gzfwrite() returns the number of full items written of size size, or zero if there was an error. If the multiplication of size and nitems overflows, i.e. the product does not fit in a z\_size\_t, then nothing is written, zero is returned, and the error state is set to Z\_STREAM\_ERROR.

**ZEXTERN int ZEXPORTVA gzprintf Z\_ARG((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 a negative *zlib* error code 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 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. This is implemented as a macro for speed. As such, it does not do all of the checking the other functions do. I.e. it does not check to see if file is NULL, nor whether the structure file points to has been clobbered or not.

**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 concatenated *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.

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().

When writing, gzdirect() returns true (1) if transparent writing was requested ("wT" for the gzopen() mode), or false (0) otherwise. (Note: gzdirect() is not needed when writing. Transparent writing must be explicitly requested, so the application already knows the answer. When linking statically, using gzdirect() will include all of the *zlib* code for *gzip* file reading and decompression, which may not be desired.)

**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, Z\_MEM\_ERROR if out of memory, Z\_BUF\_ERROR if the last read ended in the middle of a *gzip* stream, 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 CRC-32 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\_z OF((uLong adler, const Bytef \*buf,**

**z\_size\_t len));**

Same as adler32(), but with a size\_t length.

**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. Note that the z\_off\_t type (like off\_t) is a signed integer. If len2 is negative, the result has no meaning or utility.

**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 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\_z OF((uLong adler, const Bytef \*buf,**

**z\_size\_t len));**

Same as crc32(), but with a size\_t length.

**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 z\_crc\_t FAR \* ZEXPORT get\_crc\_table OF((void));**

**ZEXTERN int ZEXPORT inflateUndermine OF((z\_streamp, int));**

**ZEXTERN int ZEXPORT inflateValidate OF((z\_streamp, int));**

**ZEXTERN unsigned long ZEXPORT inflateCodesUsed OF ((z\_streamp));**

**ZEXTERN int ZEXPORT inflateResetKeep OF((z\_streamp));**

**ZEXTERN int ZEXPORT deflateResetKeep OF((z\_streamp));**

**#if (defined(\_WIN32) || defined(\_\_CYGWIN\_\_)) && !defined(Z\_SOLO)**

**ZEXTERN gzFile ZEXPORT gzopen\_w OF((const wchar\_t \*path,**

**const char \*mode));**

**#endif**

**#if defined(STDC) || defined(Z\_HAVE\_STDARG\_H)**

**# ifndef Z\_SOLO**

**ZEXTERN int ZEXPORTVA gzvprintf Z\_ARG((gzFile file,**

**const char \*format,**

**va\_list va));**

**# endif**

**#endif**