NAME

Compress::Zlib - Interface to zlib compression library

SYNOPSIS

```
use Compress::Zlib ;

($d, $status) = deflateInit( [OPT] ) ;
$status = $d->deflate($input, $output) ;
$status = $d->flush([$flush_type]) ;
$d->deflateParams(OPTS) ;
$d->deflateTune(OPTS) ;
$d->dict_adler() ;
$d->crc32() ;
$d->adler32() ;
$d->total_in() ;
$d->total_out() ;
$d->msg() ;
$d->get_Strategy();
$d->get_Level();
$d->get_BufSize();

($i, $status) = inflateInit( [OPT] ) ;
$status = $i->inflate($input, $output [, $eof]) ;
$status = $i->inflateSync($input) ;
$i->dict_adler() ;
$d->crc32() ;
$d->adler32() ;
$i->total_in() ;
$i->total_out() ;
$i->msg() ;
$d->get_BufSize();

$dest = compress($source) ;
$dest = uncompress($source) ;

$gz = gzopen($filename or filehandle, $mode) ;
$bytesread = $gz->gzread($buffer [,$size]) ;
$bytesread = $gz->gzreadline($line) ;
$byteswritten = $gz->gzwrite($buffer) ;
$status = $gz->gzflush($flush) ;
$offset = $gz->gztell() ;
$status = $gz->gzseek($offset, $whence) ;
$status = $gz->gzclose() ;
$status = $gz->gzeof() ;
$status = $gz->gzsetparams($level, $strategy) ;
$errstring = $gz->gzerror() ;
$gzerrno

$dest = Compress::Zlib::memGzip($buffer) ;
$dest = Compress::Zlib::memGunzip($buffer) ;

$crc = adler32($buffer [,$crc]) ;
$crc = crc32($buffer [,$crc]) ;

$crc = crc32_combine($crc1, $crc2, $len2);
```

```
$adler = adler32_combine($adler1, $adler2, $len2);

my $version = Compress::Raw::Zlib::zlib_version();
```

DESCRIPTION

The *Compress::Zlib* module provides a Perl interface to the *zlib* compression library (see "AUTHOR" for details about where to get *zlib*).

The `Compress::Zlib` module can be split into two general areas of functionality, namely a simple read/write interface to *gzip* files and a low-level in-memory compression/decompression interface.

Each of these areas will be discussed in the following sections.

Notes for users of Compress::Zlib version 1

Compress::Zlib version 1" The main change in `Compress::Zlib` version 2.x is that it does not now interface directly to the zlib library. Instead it uses the `IO::Compress::Gzip` and `IO::Uncompress::Gunzip` modules for reading/writing gzip files, and the `Compress::Raw::Zlib` module for some low-level zlib access.

The interface provided by version 2 of this module should be 100% backward compatible with version 1. If you find a difference in the expected behaviour please contact the author (See "AUTHOR"). See "GZIP INTERFACE"

With the creation of the `IO::Compress` and `IO::Uncompress` modules no new features are planned for `Compress::Zlib` - the new modules do everything that `Compress::Zlib` does and then some. Development on `Compress::Zlib` will be limited to bug fixes only.

If you are writing new code, your first port of call should be one of the new `IO::Compress` or `IO::Uncompress` modules.

GZIP INTERFACE

A number of functions are supplied in *zlib* for reading and writing *gzip* files that conform to RFC 1952. This module provides an interface to most of them.

If you have previously used `Compress::Zlib` 1.x, the following enhancements/changes have been made to the `gzopen` interface:

1. If you want to open either STDIN or STDOUT with `gzopen`, you can now optionally use the special filename "`-`" as a synonym for `\*STDIN` and `\*STDOUT`.

2. In `Compress::Zlib` version 1.x, `gzopen` used the zlib library to open the underlying file. This made things especially tricky when a Perl filehandle was passed to `gzopen`. Behind the scenes the numeric C file descriptor had to be extracted from the Perl filehandle and this passed to the zlib library.

   Apart from being non-portable to some operating systems, this made it difficult to use `gzopen` in situations where you wanted to extract/create a gzip data stream that is embedded in a larger file, without having to resort to opening and closing the file multiple times.

   It also made it impossible to pass a perl filehandle that wasn't associated with a real filesystem file, like, say, an `IO::String`

   In `Compress::Zlib` version 2.x, the `gzopen` interface has been completely rewritten to use the IO::Compress::Gzip for writing gzip files and IO::Uncompress::Gunzip for reading gzip files. None of the limitations mentioned above apply.

3. Addition of `gzseek` to provide a restricted `seek` interface.

4. Added `gztell`.

A more complete and flexible interface for reading/writing gzip files/buffers is included with the module `IO-Compress-Zlib`. See IO::Compress::Gzip and IO::Uncompress::Gunzip for more details.

$gz = gzopen($filename, $mode)
$gz = gzopen($filehandle, $mode)

This function opens either the *gzip* file `$filename` for reading or writing or attaches to the opened filehandle, `$filehandle`. It returns an object on success and`undef` on f ailure.

When writing a gzip file this interface will *always* create the smallest possible gzip header (exactly 10 bytes). If you want greater control over what gets stored in the gzip header (like the original filename or a comment) use IO::Compress::Gzip instead. Similarly if you want to read the contents of the gzip header use IO::Uncompress::Gunzip.

The second parameter, `$mode`, is used to specify whether the file is opened for reading or writing and to optionally specify a compression level and compression strategy when writing. The format of the `$mode` parameter is similar to the mode parameter to the 'C' function `fopen`, so ''rb'' is used to open for reading, ''wb'' for writing and ''ab'' for appending (writing at the end of the file).

To specify a compression level when writing, append a digit between 0 and 9 to the mode string — 0 means no compression and 9 means maximum compression. If no compression level is specified Z_DEFAULT_COMPRESSION is used.

To specify the compression strategy when writing, append 'f' for filtered data, 'h' for Huffman only compression, or 'R' for run-length encoding. If no strategy is specified Z_DEFAULT_STRATEGY is used.

So, for example, ''wb9'' means open for writing with the maximum compression using the default strategy and ''wb4R'' means open for writing with compression level 4 and run-length encoding.

Refer to the *zlib* documentation for the exact format of the `$mode` parameter.

$bytesread = $gz->gzread($buffer [, $size]) ;

Reads `$size` bytes from the compressed file into `$buffer`. If `$size` is not specified, it will default to 4096. If the scalar `$buffer` is not large enough, it will be extended automatically.

Returns the number of bytes actually read. On EOF it returns 0 and in the case of an error, -1.

$bytesread = $gz->gzreadline($line) ;

Reads the next line from the compressed file into `$line`.

Returns the number of bytes actually read. On EOF it returns 0 and in the case of an error, -1.

It is legal to intermix calls to `gzread` and `gzreadline`.

To maintain backward compatibility with version 1.x of this module `gzreadline` ignores the `$/` variable - it *always* uses the string `"\n"` as the line delimiter.

If you want to read a gzip file a line at a time and have it respect the `$/` variable (or `$INPUT_RECORD_SEPARATOR`, or `$RS` when `English` is in use) see IO::Uncompress::Gunzip.

$byteswritten = $gz->gzwrite($buffer) ;

Writes the contents of `$buffer` to the compressed file. Returns the number of bytes actually written, or 0 on error.

$status = $gz->gzflush($flush_type) ;

Flushes all pending output into the compressed file.

This method takes an optional parameter, `$flush_type`, that controls how the flushing will be carried out. By default the `$flush_type` used is `Z_FINISH`. Other valid values for `$flush_type` are `Z_NO_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH` and `Z_BLOCK`. It is strongly recommended that you only set the `flush_type` parameter if you fully understand the implications of what it does - overuse of `flush` can seriously degrade the level of compression achieved. See the `zlib` documentation for details.

Returns 0 on success.

$offset = $gz->*gztell()* ;
Returns the uncompressed file offset.

$status = $gz->gzseek($offset, $whence) ;
Provides a sub-set of the `seek` functionality, with the restriction that it is only legal to seek forward in the compressed file. It is a fatal error to attempt to seek backward.

When opened for writing, empty parts of the file will have NULL (0x00) bytes written to them.

The `$whence` parameter should be one of SEEK_SET, SEEK_CUR or SEEK_END.

Returns 1 on success, 0 on failure.

$gz->gzclose
Closes the compressed file. Any pending data is flushed to the file before it is closed.

Returns 0 on success.

$gz->gzsetparams($level, $strategy
Change settings for the deflate stream `$gz`.

The list of the valid options is shown below. Options not specified will remain unchanged.

Note: This method is only available if you are running zlib 1.0.6 or better.

$level
Defines the compression level. Valid values are 0 through 9, `Z_NO_COMPRESSION`, `Z_BEST_SPEED`, `Z_BEST_COMPRESSION`, and `Z_DEFAULT_COMPRESSION`.

$strategy
Defines the strategy used to tune the compression. The valid values are `Z_DEFAULT_STRATEGY`, `Z_FILTERED` and `Z_HUFFMAN_ONLY`.

$gz->gzerror
Returns the *zlib* error message or number for the last operation associated with `$gz`. The return value will be the *zlib* error number when used in a numeric context and the *zlib* error message when used in a string context. The *zlib* error number constants, shown below, are available for use.

```
Z_OK
Z_STREAM_END
Z_ERRNO
Z_STREAM_ERROR
Z_DATA_ERROR
Z_MEM_ERROR
Z_BUF_ERROR
```

$gzerrno
The `$gzerrno` scalar holds the error code associated with the most recent *gzip* routine. Note that unlike `gzerror()`, the error is *not* associated with a particular file.

As with `gzerror()` it returns an error number in numeric context and an error message in string context. Unlike `gzerror()` though, the error message will correspond to the *zlib* message when the error is associated with *zlib* itself, or the UNIX error message when it is not (i.e. *zlib* returned `Z_ERRORNO`).

As there is an overlap between the error numbers used by *zlib* and UNIX, `$gzerrno` should only be used to check for the presence of *an* error in numeric context. Use `gzerror()` to check for specific *zlib* errors. The *gzcat* example below shows how the variable can be used safely.

Examples
Here is an example script which uses the interface. It implements a *gzcat* function.

```
use strict ;
use warnings ;

use Compress::Zlib ;

# use stdin if no files supplied
@ARGV = '-' unless @ARGV ;

foreach my $file (@ARGV) {
my $buffer ;

my $gz = gzopen($file, "rb")
or die "Cannot open $file: $gzerrno\n" ;

print $buffer while $gz->gzread($buffer) > 0 ;

die "Error reading from $file: $gzerrno" . ($gzerrno+0) . "\n"
if $gzerrno != Z_STREAM_END ;

$gz->gzclose() ;
}
```

Below is a script which makes use of `gzreadline`. It implements a very simple *grep* like script.

```
use strict ;
use warnings ;

use Compress::Zlib ;

die "Usage: gzgrep pattern [file...]\n"
unless @ARGV >= 1;

my $pattern = shift ;

# use stdin if no files supplied
@ARGV = '-' unless @ARGV ;

foreach my $file (@ARGV) {
my $gz = gzopen($file, "rb")
or die "Cannot open $file: $gzerrno\n" ;

while ($gz->gzreadline($_) > 0) {
print if /$pattern/ ;
}

die "Error reading from $file: $gzerrno\n"
if $gzerrno != Z_STREAM_END ;

$gz->gzclose() ;
}
```

This script, *gzstream*, does the opposite of the *gzcat* script above. It reads from standard input and writes a gzip data stream to standard output.

```
use strict ;
use warnings ;
```

```
use Compress::Zlib ;

binmode STDOUT; # gzopen only sets it on the fd

my $gz = gzopen(\*STDOUT, "wb")
or die "Cannot open stdout: $gzerrno\n" ;

while (<>) {
$gz->gzwrite($_)
or die "error writing: $gzerrno\n" ;
}

$gz->gzclose ;
```

Compress::Zlib::memGzip

This function is used to create an in-memory gzip file with the minimum possible gzip header (exactly 10 bytes).

```
$dest = Compress::Zlib::memGzip($buffer)
or die "Cannot compress: $gzerrno\n";
```

If successful, it returns the in-memory gzip file. Otherwise it returns `undef` and the `$gzerrno` variable will store the zlib error code.

The `$buffer` parameter can either be a scalar or a scalar reference.

See IO::Compress::Gzip for an alternative way to carry out in-memory gzip compression.

Compress::Zlib::memGunzip

This function is used to uncompress an in-memory gzip file.

```
$dest = Compress::Zlib::memGunzip($buffer)
or die "Cannot uncompress: $gzerrno\n";
```

If successful, it returns the uncompressed gzip file. Otherwise it returns `undef` and the `$gzerrno` variable will store the zlib error code.

The `$buffer` parameter can either be a scalar or a scalar reference. The contents of the `$buffer` parameter are destroyed after calling this function.

If `$buffer` consists of multiple concatenated gzip data streams only the first will be uncompressed. Use `gunzip` with the `MultiStream` option in the `IO::Uncompress::Gunzip` module if you need to deal with concatenated data streams.

See IO::Uncompress::Gunzip for an alternative way to carry out in-memory gzip uncompression.

COMPRESS/UNCOMPRESS

Two functions are provided to perform in-memory compression/uncompression of RFC 1950 data streams. They are called `compress` and `uncompress`.

`$dest` = compress($source [, `$level`]) ;

Compresses `$source`. If successful it returns the compressed data. Otherwise it returns *undef*.

The source buffer, `$source`, can either be a scalar or a scalar reference.

The `$level` parameter defines the compression level. Valid values are 0 through 9, `Z_NO_COMPRESSION`, `Z_BEST_SPEED`, `Z_BEST_COMPRESSION`, and `Z_DEFAULT_COMPRESSION`. If`$level` is not specified `Z_DEFAULT_COMPRESSION` will be used.

`$dest` = uncompress($source) ;

Uncompresses `$source`. If successful it returns the uncompressed data. Otherwise it returns *undef*.

The source buffer can either be a scalar or a scalar reference.

Please note: the two functions defined above are *not* compatible with the Unix commands of the same name.

See IO::Deflate and IO::Inflate included with this distribution for an alternative interface for reading/writing RFC 1950 files/buffers.

Deflate Interface

This section defines an interface that allows in-memory compression using the *deflate* interface provided by zlib.

Here is a definition of the interface available:

($d, $status) = deflateInit([OPT])

Initialises a deflation stream.

It combines the features of the *zlib* functions `deflateInit`, `deflateInit2` and `deflateSetDictionary`.

If successful, it will return the initialised deflation stream, `$d` and `$status` of `Z_OK` in a list context. In scalar context it returns the deflation stream, `$d`, only.

If not successful, the returned deflation stream (`$d`) will be *undef* and `$status` will hold the exact *zlib* error code.

The function optionally takes a number of named options specified as `-Name=>value` pairs. This allows individual options to be tailored without having to specify them all in the parameter list.

For backward compatibility, it is also possible to pass the parameters as a reference to a hash containing the name=>value pairs.

The function takes one optional parameter, a reference to a hash. The contents of the hash allow the deflation interface to be tailored.

Here is a list of the valid options:

-Level

Defines the compression level. Valid values are 0 through 9, `Z_NO_COMPRESSION`, `Z_BEST_SPEED`, `Z_BEST_COMPRESSION`, and `Z_DEFAULT_COMPRESSION`.

The default is Z_DEFAULT_COMPRESSION.

-Method

Defines the compression method. The only valid value at present (and the default) is Z_DEFLATED.

-WindowBits

To create an RFC 1950 data stream, set `WindowBits` to a positive number.

To create an RFC 1951 data stream, set `WindowBits` to `-MAX_WBITS`.

For a full definition of the meaning and valid values for `WindowBits` refer to the *zlib* documentation for *deflateInit2*.

Defaults to MAX_WBITS.

-MemLevel

For a definition of the meaning and valid values for `MemLevel` refer to the *zlib* documentation for *deflateInit2*.

Defaults to MAX_MEM_LEVEL.

-Strategy

Defines the strategy used to tune the compression. The valid values are `Z_DEFAULT_STRATEGY`, `Z_FILTERED` and `Z_HUFFMAN_ONLY`.

The default is Z_DEFAULT_STRATEGY.

-Dictionary
When a dictionary is specified *Compress::Zlib* will automatically call `deflateSetDictionary` directly after calling `deflateInit`. The Adler32 value for the dictionary can be obtained by calling the method `$d->`*dict_adler()*>.

The default is no dictionary.

-Bufsize
Sets the initial size for the deflation buffer. If the buffer has to be reallocated to increase the size, it will grow in increments of `Bufsize`.

The default is 4096.

Here is an example of using the `deflateInit` optional parameter list to override the default buffer size and compression level. All other options will take their default values.

```
deflateInit( -Bufsize => 300,
-Level => Z_BEST_SPEED ) ;
```

($out, $status) = $d->deflate($buffer)
Deflates the contents of `$buffer`. The buffer can either be a scalar or a scalar reference. When finished, `$buffer` will be completely processed (assuming there were no errors). If the deflation was successful it returns the deflated output, `$out`, and a status value, `$status`, of `Z_OK`.

On error, `$out` will be *undef* and `$status` will contain the *zlib* error code.

In a scalar context `deflate` will return `$out` only.

As with the *deflate* function in *zlib*, it is not necessarily the case that any output will be produced by this method. So don't rely on the fact that `$out` is empty for an error test.

($out, $status) = $d->*flush()* =head2 ($out, $status) = $d->flush($flush_type)
Typically used to finish the deflation. Any pending output will be returned via `$out`. `$status` will have a value `Z_OK` if successful.

In a scalar context `flush` will return `$out` only.

Note that flushing can seriously degrade the compression ratio, so it should only be used to terminate a decompression (using `Z_FINISH`) or when you want to create a *full flush point* (using `Z_FULL_FLUSH`).

By default the `flush_type` used is `Z_FINISH`. Other valid values for `flush_type` are `Z_NO_FLUSH`, `Z_PARTIAL_FLUSH`, `Z_SYNC_FLUSH` and `Z_FULL_FLUSH`. It is strongly recommended that you only set the `flush_type` parameter if you fully understand the implications of what it does. See the `zlib` documentation for details.

$status = $d->deflateParams([OPT])
Change settings for the deflate stream `$d`.

The list of the valid options is shown below. Options not specified will remain unchanged.

-Level
Defines the compression level. Valid values are 0 through 9, `Z_NO_COMPRESSION`, `Z_BEST_SPEED`, `Z_BEST_COMPRESSION`, and `Z_DEFAULT_COMPRESSION`.

-Strategy
Defines the strategy used to tune the compression. The valid values are `Z_DEFAULT_STRATEGY`, `Z_FILTERED` and `Z_HUFFMAN_ONLY`.

$d->*dict_adler()*
Returns the adler32 value for the dictionary.

$d->*msg()*
Returns the last error message generated by zlib.

$d->*total_in()*
Returns the total number of bytes uncompressed bytes input to deflate.

$d->*total_out()*
Returns the total number of compressed bytes output from deflate.

Example
Here is a trivial example of using `deflate`. It simply reads standard input, deflates it and writes it to standard output.

```
use strict ;
use warnings ;

use Compress::Zlib ;

binmode STDIN;
binmode STDOUT;
my $x = deflateInit()
or die "Cannot create a deflation stream\n" ;

my ($output, $status) ;
while (<>)
{
($output, $status) = $x->deflate($_) ;

$status == Z_OK
or die "deflation failed\n" ;

print $output ;
}

($output, $status) = $x->flush() ;

$status == Z_OK
or die "deflation failed\n" ;

print $output ;
```

Inflate Interface

This section defines the interface available that allows in-memory uncompression using the *deflate* interface provided by zlib.

Here is a definition of the interface:

($i, `$status`) = *inflateInit()*
Initialises an inflation stream.

In a list context it returns the inflation stream, `$i`, and the *zlib* status code in `$status`. In a scalar context it returns the inflation stream only.

If successful, `$i` will hold the inflation stream and `$status` will be `Z_OK`.

If not successful, `$i` will be *undef* and `$status` will hold the *zlib* error code.

The function optionally takes a number of named options specified as `-Name=>value` pairs. This allows individual options to be tailored without having to specify them all in the parameter list.

For backward compatibility, it is also possible to pass the parameters as a reference to a hash containing the name=>value pairs.

The function takes one optional parameter, a reference to a hash. The contents of the hash allow the deflation interface to be tailored.

Here is a list of the valid options:

-WindowBits

To uncompress an RFC 1950 data stream, set `WindowBits` to a positive number.

To uncompress an RFC 1951 data stream, set `WindowBits` to `-MAX_WBITS`.

For a full definition of the meaning and valid values for `WindowBits` refer to the *zlib* documentation for *inflateInit2*.

Defaults to MAX_WBITS.

-Bufsize

Sets the initial size for the inflation buffer. If the buffer has to be reallocated to increase the size, it will grow in increments of `Bufsize`.

Default is 4096.

-Dictionary

The default is no dictionary.

Here is an example of using the `inflateInit` optional parameter to override the default buffer size.

```
inflateInit( -Bufsize => 300 ) ;
```

($out, `$status`) = `$i`->inflate($buffer)

Inflates the complete contents of `$buffer`. The buffer can either be a scalar or a scalar reference.

Returns `Z_OK` if successful and `Z_STREAM_END` if the end of the compressed data has been successfully reached. If not successful,`$out` will be *undef* and `$status` will hold the *zlib* error code.

The `$buffer` parameter is modified by `inflate`. On completion it will contain what remains of the input buffer after inflation. This means that `$buffer` will be an empty string when the return status is `Z_OK`. When the return status is `Z_STREAM_END` the `$buffer` parameter will contains what (if anything) was stored in the input buffer after the deflated data stream.

This feature is useful when processing a file format that encapsulates a compressed data stream (e.g. gzip, zip).

`$status` = `$i`->inflateSync($buffer)

Scans `$buffer` until it reaches either a *full flush point* or the end of the buffer.

If a *full flush point* is found, `Z_OK` is returned and `$buffer` will be have all data up to the flush point removed. This can then be passed to the `deflate` method.

Any other return code means that a flush point was not found. If more data is available, `inflateSync` can be called repeatedly with more compressed data until the flush point is found.

`$i`->*dict_adler()*

Returns the adler32 value for the dictionary.

`$i`->*msg()*

Returns the last error message generated by zlib.

`$i`->*total_in()*

Returns the total number of bytes compressed bytes input to inflate.

`$i`->*total_out()*

Returns the total number of uncompressed bytes output from inflate.

Example

Here is an example of using `inflate`.

```
use strict ;
use warnings ;

use Compress::Zlib ;
```

```
my $x = inflateInit()
or die "Cannot create a inflation stream\n" ;

my $input = '' ;
binmode STDIN;
binmode STDOUT;

my ($output, $status) ;
while (read(STDIN, $input, 4096))
{
($output, $status) = $x->inflate(\$input) ;

print $output
if $status == Z_OK or $status == Z_STREAM_END ;

last if $status != Z_OK ;
}

die "inflation failed\n"
unless $status == Z_STREAM_END ;
```

CHECKSUM FUNCTIONS

Two functions are provided by *zlib* to calculate checksums. For the Perl interface, the order of the two parameters in both functions has been reversed. This allows both running checksums and one off calculations to be done.

```
$crc = adler32($buffer [,$crc]) ;
$crc = crc32($buffer [,$crc]) ;
```

The buffer parameters can either be a scalar or a scalar reference.

If the `$crc` parameters is `undef`, the crc value will be reset.

If you have built this module with zlib 1.2.3 or better, two more CRC-related functions are available.

```
$crc = crc32_combine($crc1, $crc2, $len2);
$adler = adler32_combine($adler1, $adler2, $len2);
```

These functions allow checksums to be merged. Refer to the*zlib* documentation for more details.

Misc

my `$version` = *Compress::Zlib::zlib_version()*;

Returns the version of the zlib library.

CONSTANTS

All the *zlib* constants are automatically imported when you make use of *Compress::Zlib*

SEE ALSO

IO::Compress::Gzip, IO::Uncompress::Gunzip, IO::Compress::Deflate, IO::Uncompress::Inflate, IO::Compress::RawDeflate, IO::Uncompress::RawInflate, IO::Compress::Bzip2, IO::Uncompress::Bunzip2, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Uncompress::AnyInflate, IO::Uncompress::AnyUncompress

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, Archive::Tar, IO::Zlib

For RFC 1950, 1951 and 1952 see *http://www.faqs.org/rfcs/rfc1950.html*, *http://www.faqs.org/rfcs/rfc1951.html* and *http://www.faqs.org/rfcs/rfc1952.html*

The *zlib* compression library was written by Jean-loup Gailly *gzip@prep.ai.mit.edu* and Mark Adler

madler@alumni.caltech.edu.

The primary site for the *zlib* compression library is *http://www.zlib.org*.

The primary site for gzip is *http://www.gzip.org*.

AUTHOR

This module was written by Paul Marquess, *pmqs@cpan.org*.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE