NAME

Compress::Raw::Zlib - Low-Level Interface to zlib compression library

SYNOPSIS

```
use Compress::Raw::Zlib ;

($d, $status) = new Compress::Raw::Zlib::Deflate( [OPT] ) ;
$status = $d->deflate($input, $output) ;
$status = $d->flush($output [, $flush_type]) ;
$d->deflateReset() ;
$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) = new Compress::Raw::Zlib::Inflate( [OPT] ) ;
$status = $i->inflate($input, $output [, $eof]) ;
$status = $i->inflateSync($input) ;
$i->inflateReset() ;
$i->dict_adler() ;
$d->crc32() ;
$d->adler32() ;
$i->total_in() ;
$i->total_out() ;
$i->msg() ;
$d->get_BufSize();

$crc = adler32($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();
my $flags = Compress::Raw::Zlib::zlibCompileFlags();
```

DESCRIPTION

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

Compress::Raw::Zlib::Deflate

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) = new Compress::Raw::Zlib::Deflate([OPT])

Initialises a deflation object.

If you are familiar with the *zlib* library, it combines the features of the *zlib* functions `deflateInit`, `deflateInit2` and `deflateSetDictionary`.

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

If not successful, the returned deflation object, `$d`, will be *undef* and `$status` will hold the a *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.

Below 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 compress an RFC 1950 data stream, set `WindowBits` to a positive number between 8 and 15.

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

To compress an RFC 1952 data stream (i.e. gzip), set `WindowBits` to `WANT_GZIP`.

For a 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`, `Z_RLE`, `Z_FIXED` and `Z_HUFFMAN_ONLY`.

The default is `Z_DEFAULT_STRATEGY`.

-Dictionary

When a dictionary is specified *Compress::Raw::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 output buffer used by the `$d->deflate` and `$d->flush` methods. If the buffer has to be reallocated to increase the size, it will grow in increments of `Bufsize`.

The default buffer size is 4096.

-AppendOutput

This option controls how data is written to the output buffer by the `$d->deflate` and `$d->flush` methods.

If the `AppendOutput` option is set to false, the output buffers in the `$d->deflate` and `$d->flush` methods will be truncated before uncompressed data is written to them.

If the option is set to true, uncompressed data will be appended to the output buffer in the `$d->deflate` and `$d->flush` methods.

This option defaults to false.

-CRC32

If set to true, a crc32 checksum of the uncompressed data will be calculated. Use the `$d->crc32` method to retrieve this value.

This option defaults to false.

-ADLER32

If set to true, an adler32 checksum of the uncompressed data will be calculated. Use the `$d->adler32` method to retrieve this value.

This option defaults to false.

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

```
    my $d = new Compress::Raw::Zlib::Deflate ( -Bufsize => 300,
                                               -Level   => Z_BEST_SPEED ) ;
```

$status = $d->deflate($input, $output)

Deflates the contents of `$input` and writes the compressed data to `$output`.

The `$input` and `$output` parameters can be either scalars or scalar references.

When finished, `$input` will be completely processed (assuming there were no errors). If the deflation was successful it writes the deflated data to `$output` and returns a status value of `Z_OK`.

On error, it returns a *zlib* error code.

If the `AppendOutput` option is set to true in the constructor for the `$d` object, the compressed data will be appended to `$output`. If it is false, `$output` will be truncated before any compressed data is written to it.

Note: This method will not necessarily write compressed data to `$output` every time it is called. So do not assume that there has been an error if the contents of `$output` is empty on returning from this method. As long as the return code from the method is `Z_OK`, the deflate has succeeded.

$status = $d->flush($output [, $flush_type])

Typically used to finish the deflation. Any pending output will be written to `$output`.

Returns `Z_OK` if successful.

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.

If the `AppendOutput` option is set to true in the constructor for the `$d` object, the compressed data will be appended to `$output`. If it is false, `$output` will be truncated before any compressed data is written to it.

$status = $d->deflateReset()

This method will reset the deflation object `$d`. It can be used when you are compressing multiple data streams and want to use the same object to compress each of them. It should only be used once the previous data stream has been flushed successfully, i.e. a call to `$d->flush(Z_FINISH)` has returned `Z_OK`.

Returns `Z_OK` if successful.

$status = $d->deflateParams([OPT])

Change settings for the deflate object `$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`.

-BufSize

Sets the initial size for the output buffer used by the `$d->deflate` and `$d->flush` methods. If the buffer has to be reallocated to increase the size, it will grow in increments of `Bufsize`.

$status = $d->deflateTune($good_length, $max_lazy, $nice_length, $max_chain)

Tune the internal settings for the deflate object `$d`. This option is only available if you are running zlib 1.2.2.3 or better.

Refer to the documentation in zlib.h for instructions on how to fly `deflateTune`.

$d->dict_adler()

Returns the adler32 value for the dictionary.

$d->crc32()

Returns the crc32 value for the uncompressed data to date.

If the `CRC32` option is not enabled in the constructor for this object, this method will always return 0;

$d->adler32()

Returns the adler32 value for the uncompressed data to date.

$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.

$d->get_Strategy()

Returns the deflation strategy currently used. Valid values are `Z_DEFAULT_STRATEGY`, `Z_FILTERED` and `Z_HUFFMAN_ONLY`.

$d->get_Level()

Returns the compression level being used.

$d->get_BufSize()

Returns the buffer size used to carry out the compression.

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::Raw::Zlib ;

binmode STDIN;
binmode STDOUT;
my $x = new Compress::Raw::Zlib::Deflate
   or die "Cannot create a deflation stream\n" ;

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

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

    print $output ;
}

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

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

print $output ;
```

Compress::Raw::Zlib::Inflate

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

Here is a definition of the interface:

($i, $status) = new Compress::Raw::Zlib::Inflate([OPT])

Initialises an inflation object.

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

If successful, `$i` will hold the inflation object and `$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.

Here is a list of the valid options:

-WindowBits

To uncompress an RFC 1950 data stream, set `WindowBits` to a positive number between 8 and 15.

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

To uncompress an RFC 1952 data stream (i.e. gzip), set `WindowBits` to `WANT_GZIP`.

To auto-detect and uncompress an RFC 1950 or RFC 1952 data stream (i.e. gzip), set `WindowBits` to `WANT_GZIP_OR_ZLIB`.

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 output buffer used by the `$i->inflate` method. If the output buffer in this method has to be reallocated to increase the size, it will grow in increments of `Bufsize`.

Default is 4096.

-Dictionary

The default is no dictionary.

-AppendOutput

This option controls how data is written to the output buffer by the `$i->inflate` method.

If the option is set to false, the output buffer in the `$i->inflate` method will be truncated before uncompressed data is written to it.

If the option is set to true, uncompressed data will be appended to the output buffer by the `$i->inflate` method.

This option defaults to false.

-CRC32

If set to true, a crc32 checksum of the uncompressed data will be calculated. Use the `$i->crc32` method to retrieve this value.

This option defaults to false.

-ADLER32

If set to true, an adler32 checksum of the uncompressed data will be calculated. Use the `$i->adler32` method to retrieve this value.

This option defaults to false.

-ConsumeInput

If set to true, this option will remove compressed data from the input buffer of the `$i->inflate` method as the inflate progresses.

This option can be useful when you are processing compressed data that is embedded in another file/buffer. In this case the data that immediately follows the compressed stream will be left in the input buffer.

This option defaults to true.

-LimitOutput

The `LimitOutput` option changes the behavior of the `$i->inflate` method so that the amount of memory used by the output buffer can be limited.

When `LimitOutput` is used the size of the output buffer used will either be the value of the `Bufsize` option or the amount of memory already allocated to `$output`, whichever is larger. Predicting the output size available is tricky, so don't rely on getting an exact output buffer size.

When `LimitOutout` is not specified `$i->inflate` will use as much memory as it takes to write all the uncompressed data it creates by uncompressing the input buffer.

If `LimitOutput` is enabled, the `ConsumeInput` option will also be enabled.

This option defaults to false.

See *The LimitOutput option* for a discussion on why `LimitOutput` is needed and how to use it.

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

```
    my ($i, $status) = new Compress::Raw::Zlib::Inflate( -Bufsize => 300 )
;
```

$status = $i->inflate($input, $output [,$eof])

Inflates the complete contents of `$input` and writes the uncompressed data to `$output`. The `$input` and `$output` parameters can either be scalars or scalar references.

Returns `Z_OK` if successful and `Z_STREAM_END` if the end of the compressed data has been successfully reached.

If not successful `$status` will hold the *zlib* error code.

If the `ConsumeInput` option has been set to true when the `Compress::Raw::Zlib::Inflate` object is created, the `$input` parameter is modified by `inflate`. On completion it will contain what remains of the input buffer after inflation. In practice, this means that when the return status is `Z_OK` the `$input` parameter will contain an empty string, and when the return status is `Z_STREAM_END` the `$input` 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) and there is useful data immediately after the deflation stream.

If the `AppendOutput` option is set to true in the constructor for this object, the uncompressed data will be appended to `$output`. If it is false, `$output` will be truncated before any uncompressed data is written to it.

The `$eof` parameter needs a bit of explanation.

Prior to version 1.2.0, zlib assumed that there was at least one trailing byte immediately after the compressed data stream when it was carrying out decompression. This normally isn't a problem

because the majority of zlib applications guarantee that there will be data directly after the compressed data stream. For example, both gzip (RFC 1950) and zip both define trailing data that follows the compressed data stream.

The `$eof` parameter only needs to be used if **all** of the following conditions apply

1 You are either using a copy of zlib that is older than version 1.2.0 or you want your application code to be able to run with as many different versions of zlib as possible.

2 You have set the `WindowBits` parameter to `-MAX_WBITS` in the constructor for this object, i.e. you are uncompressing a raw deflated data stream (RFC 1951).

3 There is no data immediately after the compressed data stream.

If **all** of these are the case, then you need to set the `$eof` parameter to true on the final call (and only the final call) to `$i->inflate`.

If you have built this module with zlib >= 1.2.0, the `$eof` parameter is ignored. You can still set it if you want, but it won't be used behind the scenes.

$status = $i->inflateSync($input)

This method can be used to attempt to recover good data from a compressed data stream that is partially corrupt. It scans `$input` 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 `$input` will be have all data up to the flush point removed. This data can then be passed to the `$i->inflate` method to be uncompressed.

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.

Note *full flush points* are not present by default in compressed data streams. They must have been added explicitly when the data stream was created by calling `Compress::Deflate::flush` with `Z_FULL_FLUSH`.

$status = $i->inflateReset()

This method will reset the inflation object `$i`. It can be used when you are uncompressing multiple data streams and want to use the same object to uncompress each of them.

Returns `Z_OK` if successful.

$i->dict_adler()

Returns the adler32 value for the dictionary.

$i->crc32()

Returns the crc32 value for the uncompressed data to date.

If the `CRC32` option is not enabled in the constructor for this object, this method will always return 0;

$i->adler32()

Returns the adler32 value for the uncompressed data to date.

If the `ADLER32` option is not enabled in the constructor for this object, this method will always return 0;

$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.

$d->get_BufSize()

Returns the buffer size used to carry out the decompression.

Examples

Here is an example of using `inflate`.

```
use strict ;
use warnings ;

use Compress::Raw::Zlib;

my $x = new Compress::Raw::Zlib::Inflate()
   or die "Cannot create a inflation stream\n" ;

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

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

    print $output ;

    last if $status != Z_OK ;
}

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

The next example show how to use the `LimitOutput` option. Notice the use of two nested loops in this case. The outer loop reads the data from the input source - STDIN and the inner loop repeatedly calls `inflate` until `$input` is exhausted, we get an error, or the end of the stream is reached. One point worth remembering is by using the `LimitOutput` option you also get `ConsumeInput` set as well - this makes the code below much simpler.

```
use strict ;
use warnings ;

use Compress::Raw::Zlib;

my $x = new Compress::Raw::Zlib::Inflate(LimitOutput => 1)
   or die "Cannot create a inflation stream\n" ;

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

```
    my ($output, $status) ;

  OUTER:
    while (read(STDIN, $input, 4096))
    {
        do
        {
            $status = $x->inflate($input, $output) ;

            print $output ;

            last OUTER
                unless $status == Z_OK || $status == Z_BUF_ERROR ;
        }
        while ($status == Z_OK && length $input);
    }

    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::Raw::Zlib::zlib_version();

Returns the version of the zlib library.

my $flags = Compress::Raw::Zlib::zlibCompileFlags();

Returns the flags indicating compile-time options that were used to build the zlib library. See the zlib documentation for a description of the flags returned by `zlibCompileFlags`.

Note that when the zlib sources are built along with this module the `sprintf` flags (bits 24, 25 and 26) should be ignored.

If you are using zlib 1.2.0 or older, `zlibCompileFlags` will return 0.

The LimitOutput option.

By default `$i->inflate($input, $output)` will uncompress *all* data in `$input` and write *all* of the uncompressed data it has generated to `$output`. This makes the interface to `inflate` much

simpler - if the method has uncompressed `$input` successfully *all* compressed data in `$input` will have been dealt with. So if you are reading from an input source and uncompressing as you go the code will look something like this

```
use strict ;
use warnings ;

use Compress::Raw::Zlib;

my $x = new Compress::Raw::Zlib::Inflate()
   or die "Cannot create a inflation stream\n" ;

my $input = '' ;

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

    print $output ;

    last if $status != Z_OK ;
}

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

The points to note are

- The main processing loop in the code handles reading of compressed data from STDIN.
- The status code returned from `inflate` will only trigger termination of the main processing loop if it isn't `Z_OK`. When `LimitOutput` has not been used the `Z_OK` status means that the end of the compressed data stream has been reached or there has been an error in uncompression.
- After the call to `inflate` *all* of the uncompressed data in `$input` will have been processed. This means the subsequent call to `read` can overwrite it's contents without any problem.

For most use-cases the behavior described above is acceptable (this module and it's predecessor, `Compress::Zlib`, have used it for over 10 years without an issue), but in a few very specific use-cases the amount of memory required for `$output` can prohibitively large. For example, if the compressed data stream contains the same pattern repeated thousands of times, a relatively small compressed data stream can uncompress into hundreds of megabytes. Remember `inflate` will keep allocating memory until *all* the uncompressed data has been written to the output buffer - the size of `$output` is unbounded.

The `LimitOutput` option is designed to help with this use-case.

The main difference in your code when using `LimitOutput` is having to deal with cases where the `$input` parameter still contains some uncompressed data that `inflate` hasn't processed yet. The status code returned from `inflate` will be `Z_OK` if uncompression took place and `Z_BUF_ERROR` if the output buffer is full.

Below is typical code that shows how to use `LimitOutput`.

```
    use strict ;
    use warnings ;

    use Compress::Raw::Zlib;

    my $x = new Compress::Raw::Zlib::Inflate(LimitOutput => 1)
       or die "Cannot create a inflation stream\n" ;

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

    my ($output, $status) ;

  OUTER:
    while (read(STDIN, $input, 4096))
    {
        do
        {
            $status = $x->inflate($input, $output) ;

            print $output ;

            last OUTER
                unless $status == Z_OK || $status == Z_BUF_ERROR ;
        }
        while ($status == Z_OK && length $input);
    }

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

Points to note this time:

- There are now two nested loops in the code: the outer loop for reading the compressed data from STDIN, as before; and the inner loop to carry out the uncompression.
- There are two exit points from the inner uncompression loop.

  Firstly when `inflate` has returned a status other than `Z_OK` or `Z_BUF_ERROR`. This means that either the end of the compressed data stream has been reached (`Z_STREAM_END`) or there is an error in the compressed data. In either of these cases there is no point in continuing with reading the compressed data, so both loops are terminated.

  The second exit point tests if there is any data left in the input buffer, `$input` - remember that the `ConsumeInput` option is automatically enabled when `LimitOutput` is used. When the input buffer has been exhausted, the outer loop can run again and overwrite a now empty `$input`.

ACCESSING ZIP FILES

Although it is possible (with some effort on your part) to use this module to access .zip files, there are other perl modules available that will do all the hard work for you. Check out `Archive::Zip`, `Archive::Zip::SimpleZip`, `IO::Compress::Zip` and `IO::Uncompress::Unzip`.

FAQ

Compatibility with Unix compress/uncompress.

This module is not compatible with Unix `compress`.

If you have the `uncompress` program available, you can use this to read compressed files

```
open F, "uncompress -c $filename |";
while (<F>)
{
    ...
```

Alternatively, if you have the `gunzip` program available, you can use this to read compressed files

```
open F, "gunzip -c $filename |";
while (<F>)
{
    ...
```

and this to write compress files, if you have the `compress` program available

```
open F, "| compress -c $filename ";
print F "data";
...
close F ;
```

Accessing .tar.Z files

See previous FAQ item.

If the `Archive::Tar` module is installed and either the `uncompress` or `gunzip` programs are available, you can use one of these workarounds to read `.tar.Z` files.

Firstly with `uncompress`

```
use strict;
use warnings;
use Archive::Tar;

open F, "uncompress -c $filename |";
my $tar = Archive::Tar->new(*F);
...
```

and this with `gunzip`

```
use strict;
use warnings;
use Archive::Tar;

open F, "gunzip -c $filename |";
my $tar = Archive::Tar->new(*F);
...
```

Similarly, if the `compress` program is available, you can use this to write a `.tar.Z` file

```
use strict;
use warnings;
use Archive::Tar;
```

```
use IO::File;

my $fh = new IO::File "| compress -c >$filename";
my $tar = Archive::Tar->new();
...
$tar->write($fh);
$fh->close ;
```

Zlib Library Version Support

By default `Compress::Raw::Zlib` will build with a private copy of version 1.2.5 of the zlib library. (See the *README* file for details of how to override this behaviour)

If you decide to use a different version of the zlib library, you need to be aware of the following issues

- First off, you must have zlib 1.0.5 or better.
- You need to have zlib 1.2.1 or better if you want to use the `-Merge` option with `IO::Compress::Gzip`, `IO::Compress::Deflate` and `IO::Compress::RawDeflate`.

CONSTANTS

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

SEE ALSO

Compress::Zlib, *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