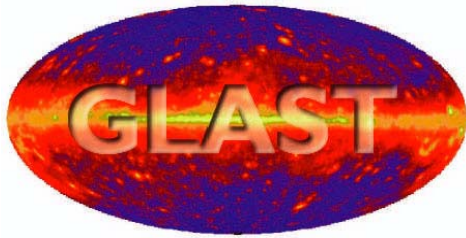


LAT Flight Software

ZLIB User's Manual

Type:	User’s Manual
Version:	V2-1-0
Author:	D.L. Wood
Created:	26 July 2002
Updated:	26 May 2004
Printed:	26 May 2004

A description of how to employ the ZLIB data compression package as it exists in the LAT flight software repository. User interface functions and unit tests are discussed.

Contents

0 Introduction 1

1 Compression 2

1.0 Libraries 2

1.0.0 Memory Compression Libraries 2

1.0.1 File Compression Libraries 3

1.1 Compression Executables 5

2 Inflation 6

2.0 Libraries 6

2.0.0 Memory Inflation Libraries 6

2.0.1 File Inflation Libraries 7

2.1 Inflation Executables 9

3 Error Reporting 10

4 Utilities 12

5 Unit Testing 13

5.0 Unit Test Coverage 13

5.0.0 Constituent: zlib_inflate 13

5.0.1 Constituent: zlib_compress 13

5.0.2 Constituent: zlib_file_inflate 13

5.0.3 Constituent: zlib_file_compress 13

5.1 Running the Unit Test 14

Tables

Table 1 – ZLIB Core MSG Codes 10

Table 2 - ZLIB File MSG Codes 11

Figures

Figure 1 - ZLIB File Compression Format 4

0 Introduction

The GNU ZLIB libraries are provided for use with the LAT flight software. The libraries, source code, and related executables and documentation reside in the CMX package *ZLIB*. The compression of file data is extremely helpful for the LAT in two important ways.

1. Compressed files reduce the amount of uplink bandwidth, which is a scarce commodity, required to send and update files on board for LAT flight operations.
2. Compressed files reduce the amount of non-volatile memory, which is also a scarce commodity, required to store files on board for LAT flight operations.

These uses do not preclude other potential uses that may be discovered in the future.

The C source code for the compression and inflation libraries has not been altered, and is taken from the GNU tagged version 1.3.3. The *makefile* that was included with the GNU distribution has been translated into a CMX *requirements* file. Also, the organization of the GNU library has been altered somewhat both to fit into the CMX scheme and to ensure that the compression and inflation code is separated. This last point is important since only the inflation side of the algorithm is foreseen to be of use on board the LAT during flight. A set of utility executables is provided which not only compress and inflate files in conventional environments but also provide good examples on how to interface to the ZLIB libraries.

The ZLIB compression and inflation libraries taken from GNU operate at near the lowest level, corresponding to the *deflate()/inflate()* interfaces. These functions require the initialization of a *z_stream* structure and only operate from memory to memory. To facilitate operations involving files from a file system, a higher level interface has been provided. This file level interface is particularly useful for operations from file to file.

1 Compression

1.0 Libraries

Two levels of compression libraries are offered, memory based and file based.

1.0.0 Memory Compression Libraries

The ZLIB memory based compression library is provided as the CMX constituent *zlib_compress*. The public entry points to the library are listed below.

```
int deflateInit(z_streamp strm, int level)

int deflate(z_streamp strm, int flush)

int deflateEnd(z_streamp strm)
```

The compression functions all take a pointer to a *z_stream* structure to manage the user parameters as well as maintain the internal state of the compressor. The *next_in* and *avail_in* members contain the address and size of the input data buffer to compress. The *next_out* member contains the address of the compressed data output buffer. The *avail_out* member is the maximum size of output data allowed to be placed in the output buffer. An instance of a *z_stream* structure should be allocated and these members initialized before the call to *deflateInit()*. Upon successful return, the *total_out* member contains the size of the compressed output data. The user is responsible for remembering this value if a one step inflation is to be performed at a later time. The fragment below shows the basic operation.

```
#include "ZLIB/zlib.h"

z_stream stream;
int status;

stream.next_in = my_in_buf;
stream.avail_in = my_in_size;
stream.next_out = my_out_buf;
stream.avail_out = my_out_size_max;
stream.zalloc = NULL;
stream.zfree = NULL;

status = deflateInit(&stream, Z_DEFAULT_COMPRESSION);
if(status != Z_OK)
{
 /* error handler */
}

status = deflate(&stream, Z_FINISH);
if(status != Z_STREAM_END)
{
 deflateEnd(&stream);

 /* error handler */
}

my_out_size = stream.total_out;
deflateEnd(&stream);
```

In the above example, the *zalloc* and *zfree* members of the stream structure are set to NULL to indicate that the default standard library *calloc()* and *free()* functions should be used for the compressor's internal memory management needs. These defaults may be overridden when the standard libraries are not available or when memory heap fragmentation is a concern.

1.0.1 File Compression Libraries

The ZLIB file based compression library is provided as CMX constituent *zlib_file_compress*. The public entry points are listed below.

```
unsigned int ZLIB_fileCompressFromFile(int in, int out unsigned int
    bufSize)

unsigned int ZLIB_fileCompressFromMem(void *inBuf, int out, unsigned
    int inSize, unsigned int bufSize)
```

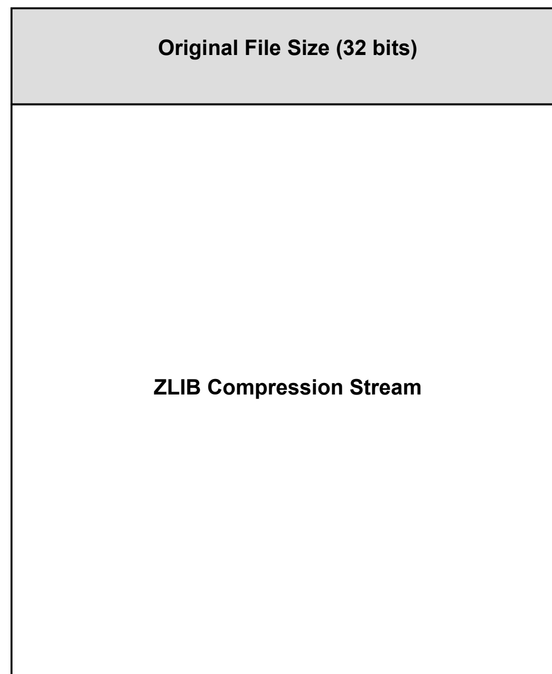
The *ZLIB_fileCompressFromFile()* function takes already open file handles as the *in* and *out* parameters. Upon successful completion, the *in* file will be compressed into the *out* file. Both files handles will remain open after the function returns.

The *ZLIB_fileCompressFromMem()* function takes an already open file handle as *out* parameters. Upon successful completion, the data in the *inBuf* user memory buffer will be compressed into the *out* file. Both files handles will remain open after the function returns.

All of the *zlib_file_compress* constituent functions take a *bufSize* paramter. This is the size in bytes the functions will use when allocating internal temporary stream buffers. Using a larger value will generally result in better performance at the cost of more memory consumption.

In order to facilitate later inflation, the original size of the *in* file is stored as a single 32-bit, big-endian value at the start of the *out* file. Otherwise, the output format is simply the output of the ZLIB compressor, which follows the 32-bit header word.

Figure 1 - ZLIB File Compression Format



The example fragment below shows the use of the file to file compression function.

```
#include "ZLIB/ZLIB_file_compress.h"
#include “MSG/MSG_pubdefs.h”

int in, out;
unsigned int status
char *inName, *outName

in = open(inName, O_RDONLY, 0);
if(in == -1)
{
  /* error handler */
}

out = creat(outName, 0666);
if(out == -1)
{
  /* error handler */
}

status = ZLIB_fileCompressFromFile(in, out, 0x1000);
if(_msg_success(status) == 0)
{
  /* error handler */
}

close(in);
close(out);
```

1.1 Compression Executables

A file compression command line executable which uses the ZLIB compressor is provided as the CMX constituent *zcompress*. The command line format is show below.

```
zcompress in_file out_file
```

The *zcompress* utility compresses the data in *in_file* and writes the output to *out_file*.

2 Inflation

2.0 Libraries

Two levels of inflation libraries are offered, memory based and file based.

2.0.0 Memory Inflation Libraries

The ZLIB memory based inflation library is provided as the CMX constituent *zlib_inflate*. This corresponds to the original interface provided directly by the GNU software. The public entry points to the library are listed below.

```
int inflateInit(z_streamp strm)

int inflate(z_streamp strm, int flush)

int inflateEnd(z_streamp strm)
```

The inflations functions all take a pointer to a *z_stream* structure to manage the user parameters as well as maintain the internal state of the inflator. The *next_in* and *avail_in* members contain the address and size of the input data buffer to inflate. The *next_out* member contains the address of the inflated data output buffer. The *avail_out* member is the maximum size of output data allowed to be placed in the output buffer. This value should be at least equal to the size of the original file before compression. This value must be maintained by the user independently of the ZLIB facilities. An instance of a *z_stream* structure should be allocated and these members initialized before the call to *inflateInit()*. The fragment below shows the basic operation.

```
#include "ZLIB/zlib.h"

z_stream stream;
int status;

stream.next_in = my_in_buf;
stream.avail_in = my_in_size;
stream.next_out = my_out_buf;
stream.avail_out = orig_file_size;
stream.zalloc = NULL;
stream.zfree = NULL;

status = inflateInit(&stream);
if(status != Z_OK)
{
  /* error handler */
}

status = inflate(&stream, Z_FINISH);
if(status != Z_STREAM_END)
{
  inflateEnd(&stream);

  /* error handler */
}

inflateEnd(&stream);
```

In the above example, the *zalloc* and *zfree* members of the stream structure are set to NULL to indicate that the default standard library *calloc()* and *free()* functions should be used for the inflator's internal memory management needs. These defaults may be overridden when the standard libraries are not available or when memory heap fragmentation is a concern.

2.0.1 File Inflation Libraries

The ZLIB file based inflation library is provided as the CMX constituent *zlib_file_inflate*. The public entry points to the library are listed below.

```
unsigned int ZLIB_fileInflateToFile(int in, int out, unsigned int
     bufSize)

unsigned int ZLIB_fileInflateToMem(int in, void *out, unsigned int
     outSize, unsigned int bufSize)

unsigned int ZLIB_fileInflateSizeof(int file, unsigned int *size)
```

The *ZLIB_fileInflateToFile()* function takes already open file handles as the *in* and *out* parameters. Upon successful completion, the *in* file will be inflated to the *out* file. Both files handles will remain open after the function returns.

The *ZLIB_fileInflateToMem()* function takes an already open file handle as the *in* parameter. The *out* parameter should point to a buffer where the file inflation output data will be written. The *outSize* parameter gives the size limit of the output buffer to prevent overflows. Upon successful completion, the *in* file will be inflated to the *out* buffer. The input file handle will remain open after the function returns. The *ZLIB_fileInflateSizeof()* function allows users to query the size of the required output buffer needed to inflate a file.

All of the *zlib_file_inflate* constituent functions take a *bufSize* paramter. This is the size in bytes the functions will use when allocating internal temporary stream buffers. Using a larger value will generally result in better performance at the cost of more memory consumption.

The file inflation libraries expect to find the original size of the file stored as a single 32-bit, big-endian value at the start of the input file. This is the format of the compressed output produced by the compression tools (see Section 1).

The example fragment below shows the use of the file to file inflation function.

```
#include "ZLIB/ZLIB_file_inflate.h"
#include “MSG/MSG_pubdefs.h”

int in, out;
char *inName, *outName;
unsigned int status;

in = open(inName, O_RDONLY, 0);
if(in == -1)
{
  /* error handler */
}

out = creat(outName, 0666);
if(out == -1)
{
  /* error handler */
}

status = ZLIB_fileInflateToFile(in, out, 0x1000);
if(_msg_success(status) == 0)
{
  /* error handler */
}

close(in);
close(out);
```

2.1 Inflation Executables

A file inflation command line executable which uses the ZLIB file based inflator is provided as the CMX constituent *zinflate*. The command line format is show below.

```
zinflate in_file out_file
```

The *zinflate* utility inflates the data in *in_file* and writes the output to *out_file*. The executable expects to find the original size of the file stored as a single 32-bit, big-endian value at the start of *in_file*. Otherwise, the input format is simply the input format expected by ZLIB inflator, which follows the 32-bit header word.

3 Error Reporting

The *ZLIB* file compression and file inflation libraries supports the *MSG* status reporting system. The core error messages are translations of the ZLIB native error codes produced by the functions in *zlib_inflate* and *zlib_compress*.

Table 1 – ZLIB Core MSG Codes

MSG Facility	MSG Code	Description
ZLIB	ZLIB_SUCCESS	Success.
	ZLIB_ELIBDATA	ZLIB data error. Corresponds to native error code Z_DATA_ERROR.
	ZLIB_ELIBSTRM	ZLIB stream error. Corresponds to native error code Z_STREAM_ERROR.
	ZLIB_ELIBMEMA	ZLIB memory error. Corresponds to native error code ZLIB_MEM_ERROR.
	ZLIB_ELIBBUFF	ZLIB buffer error. Corresponds to native error code ZLIB_BUF_ERROR.
	ZLIB_ELIBVERS	ZLIB version error. Corresponds to native error code ZLIB_VERSION_ERROR.

Note that the constituents *zlib_inflate* and *zlib_compress* themselves do not return or signal the MSG codes. Each MSG report will also print out the ZLIB internal error message string as a paramter.

The constituents *zlib_file_compress* and *zlib_file_inflate* produces additional error codes which describe errors at the file operation level.

Table 2 - ZLIB File MSG Codes

MSG Facility	MSG Code	Description	Parameter
ZLIB	ZLIB_EFILPARM	A function parameter value is out of range.	The name of the function parameter.
	ZLIB_EFILMEMA	Memory allocation error.	The size in byte of the allocation request
	ZLIB_EFILFILR	File read error.	The system call *errno* value.
	ZLIB_EFILFILW	File write error.	The system call *errno* value.
	ZLIB_EFILFILS	File seek error.	The system call *errno* value.
	ZLIB_EFILFILT	Error obtaining file status.	The system call *errno* value.

The *zlib_file_inflate* and *zlib_file_compress* constituents signal all error messages at run time.

4 Utilities

The *ZLIB* package exports the checksum function *adler32(),* which it uses internally to verify the inflated data set against the original data set. The checksum function may be used to provide a general purpose 32-bit checksum algorithm.

```
uLong adler32(uLong alder, const Bytef *buf, uInt len)
```

An initial call to *adler32()* with NULL for the *buf* parameter returns the seed checksum value. This value is passed back in as the *adler* parameter, with the address and size of the memory region to checksum.

```
#include ZLIB/zlib.h”

unsigned int chksum = adler32(0, NULL, 0);
chksum = adler32(chksum, my_data_ptr, my_data_size);
```

Beware that the adler32() function is sensitive to the ordering of the byte stream when calculating the checksum value. When transferring and validating a checksum and associated data set between different machines, make sure that the function is presented with the data in exactly the same byte order on both machines; otherwise, they will produce different checksum values.

5 Unit Testing

The *ZLIB* package provides a unit test which test the libraries both for successful baseline functionality as well as some error coverage paths.

5.0 Unit Test Coverage

5.0.0 Constituent: zlib_inflate

The fault handling tests ensure that all of the library functions reject out of range parameters with the proper error codes. In addition, the data inflation libraries are run against a special test which ensures that they can recover from a corrupted compressed data set. The basic functionality tests ensure that all of the library functions can properly manipulate the ZLIB compressed data format. The functionality tests use some sample compressed data files to provide a reference.

5.0.1 Constituent: zlib_compress

The fault handling tests ensure that all of the library functions reject out of range parameters with the proper error codes. The basic functionality tests ensure that all of the library functions can properly generate the ZLIB compressed data format. The functionality tests use some sample compressed data files to provide a reference.

5.0.2 Constituent: zlib_file_inflate

The fault handling tests ensure that all of the library functions reject out of range parameters with the proper error codes. The basic functionality tests ensure that all of the library functions can properly manipulate the ZLIB compressed data format when reading and writing files. The functionality tests use some sample compressed data files to provide a reference.

5.0.3 Constituent: zlib_file_compress

The fault handling tests ensure that all of the library functions reject out of range parameters with the proper error codes. The basic functionality tests ensure that all of the library functions can properly manipulate the ZLIB compressed data format when reading and writing files. The functionality tests use some sample compressed data files to provide a reference.

5.1 Running the Unit Test

The *ZLIB* package unit test may be run directly on UNIX hosts by typing *zlib_unit_test* at the command shell. If the proper modules are loaded, then the unit test may be run on VxWorks hosts by typing *zlib_unit_test* at the VxWorks shell prompt. The preferred method, however, is to use LTX. As part of the *ZLIB* package, a LTX script is provided which defines a test called *zlib_unit_test*.