emCompress-LZMA

LZMA compression system

User Guide & Reference Manual

Document: UM17002
Software Version: 2.40.0
Revision: 0
Date: June 6, 2024

SEGGER

A product of SEGGER Microcontroller GmbH

www.segger.com

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel.	+49 2173-99312-0
Fax.	+49 2173-99312-28
E-mail:	support@segger.com[*]
Internet:	*www.segger.com*

[*]By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: June 6, 2024

Software	Revision	Date	By	Description
2.40.0	0	2024-04-10	RH	Added functions `COMPRESS_LZMA_DECODE_T2_Run()` and `COMPRESS_LZMA_ENCODE_T2_Run()`
2.30.0	0	2022-12-19	RH	Update data type configuration.
2.20b	0	2016-11-04	JL	Added data types. Broken links fixed.
2.20a	0	2016-10-28	JL	Version update, no changes.
2.20	0	2016-10-17	PC	Release.
1.00	0	2016-09-27	PC	Internal version.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler).
- The C programming language.
- The target processor.
- DOS command line.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
`User Input`	Text entered at the keyboard by a user in a session transcript.
`Secret Input`	Text entered at the keyboard by a user, but not echoed (e.g. password entry), in a session transcript.
Reference	Reference to chapters, sections, tables and figures or other documents.
Emphasis	Very important sections.

Table of contents

Chapter 1

Introduction to emCompress-LZMA

This section presents an overview of emCompress-LZMA, its structure, and its capabilities.

1.1 What is emCompress-LZMA?

emCompress-LZMA is a compression system that is able to reduce the storage requirements of data that must be embedded into an application. Typical uses of emCompress-LZMA are:

- Firmware images that must be dynamically expanded on device reprogramming.
- Configuration bitstreams to program FPGA and CPLD devices.
- Permanent files for embedded web server static content.

Of course, emCompress-LZMA is not limited to these applications, it can be used whenever it’s beneficial to reduce the size of stored content.

1.2 Features

emCompress-LZMA is written in standard ANSI C and can run on virtually any CPU. Here’s a list summarizing the main features of emCompress-LZMA:

- Clean ISO/ANSI C source code.
- Small decompressor ROM footprint.
- Completely tunable decompressor RAM footprint.
- Easy-to-understand and simple-to-use API.
- Simple configuration.
- Royalty free.

1.3 Recommended project structure

We recommend keeping emCompress-LZMA separate from your application files. It is good practice to keep all the program files (including the header files) together in the `COMPRESS` subdirectory of your project's root directory. This practice has the advantage of being very easy to update to newer versions of emCompress-LZMA by simply replacing the `COMPRESS` and `SEGGER` directories. Your application files can be stored anywhere.

Warning

When updating to a newer emCompress-LZMA version: as files may have been added, moved or deleted, the project directories may need to be updated accordingly.

1.4 Package content

emCompress is provided in source code and contains everything needed. The following table shows the content of the emCompress Package:

Directory	Description
`Config`	Configuration header files.
`Doc`	emCompress-LZMA documentation.
`COMPRESS`	emCompress-LZMA decompressor and optional compressor source code.
`Application`	Supporting and sample applications in source code.
`Windows`	Supporting applications and compressor library in binary form.

1.4.1 Compressor include directories

You should make sure that the include path contains the following directories (the order of inclusion is of no importance):

- `Config`
- `COMPRESS`

Warning

Always make sure that you have only one version of each file!

It is frequently a major problem when updating to a new version of emCompress-LZMA if you have old files included and therefore mix different versions. If you keep emCompress-LZMA in the directories as suggested (and only in these), this type of problem cannot occur. When updating to a newer version, you should be able to keep your configuration files and leave them unchanged. For safety reasons, we recommend backing up (or at least renaming) the `COMPRESS` directories before updating.

Chapter 2

Compressor

This section describes the emCompress-LZMA compressor interface.

2.1 Distribution types

emCompress-LZMA is shipped in two forms, as source code with a configurable compressor, and as a corresponding library where the compressor is preconfigured with a fixed in memory footprint.

Source code distribution

Configuration of the compressor is through the configuration file `COMPRESS_LZMA_ENCODE_Conf.h` where the compressor can be tuned for memory use. This is applicable should there be a requirement for on-target compression where memory is limited.

Library distribution

Configuration of the compressor with the library is not possible: it is preconfigured to allow compression with all configuration parameters up to their maximum value, and with dictionary (window) sizes up to one megabyte. The file `COMPRESS_LZMA_ENCODE_Conf.h` reflects the build configuration of the library *and must not be modified*.

2.2 Sample applications

emCompress-LZMA ships with a number of sample applications that show how to integrate shell capability into your application. Each sample application adds an additional capability to the shell and is a small incremental step from the previous version.

The sample applications are:

Application	Description
`COMPRESS_LZMA_OneShot.c`	Single-shot compression.
`COMPRESS_LZMA_Compress.c`	Stream data into and out of the compressor.
`COMPRESS_LZMA_Optimize.c`	Search for optimal compressor parameters.

2.2.1 A note on the samples

Each sample that we present in this section is written in a style that makes it easy to describe and that fits comfortably within the margins of printed paper. Therefore, it may well be that you would rewrite the sample to have a slightly different structure that fits better, but please keep in mind that these examples are written with clarity as the prime objective, and to that end we sacrifice some brevity and efficiency.

2.2.2 Where to find the sample code

All samples are included in the `Application` directory of the emCompress-LZMA distribution.

2.3 Single-shot compression

The first example, `COMPRESS_LZMA_OneShot.c`, provides a simple way to compress a single block of data without streaming. This is the smallest working example but is not not how you would typically compress data in your application.

For a complete listing of this application, see *`COMPRESS_LZMA_OneShot.c` complete listing* on page 19.

2.3.1 Include files

The emCompress-LZMA encoding API is exposed by including the file `COMPRESS_LZMA_ENCODE.h`:

```
#include "COMPRESS_LZMA_ENCODE.h"
```

2.3.2 Setting up the compressor

Application entry starts to set up compression parameters.

```
void main(int argc, char **argv) {
  COMPRESS_LZMA_STREAM Stream;
  COMPRESS_LZMA_PARAS  Paras;
  int                  Status;
  //
  Paras.LC         = 0;    ❶
  Paras.LP         = 0;
  Paras.PB         = 0;
  Paras.WindowSize = 1024;    ❷
  Paras.MinLen     = 3;    ❸
  Paras.MaxLen     = 273;
  Paras.Optimize   = 5;    ❹
  //
  COMPRESS_LZMA_ENCODE_Init(&_Encoder, &Paras);    ❺
```

Step 1: Configure the LZMA probabilty model

The three parameters LC, LP, and PB configure the LZMA probability model and how the LZMA encoder performs. For better compression, these parameters are usually nonzero but as they grow more memory is needed to hold the encoder state. This example uses the minimum amount of working store possible by setting these parameters to zero.

Step 2: Configure the window size

The window size is the maximum number of octets that the decoder will need to maintain as context in order to copy a reference to previously-decoded content. As the window size increases, the compressed output usually decreases as there are more opportunities to find matching substrings within the window.

Step 3: Configure match lengths

The match length range defined by LZMA is between 2 and 273 octets. However, a minimum match of two octets produces compressed images that are larger than images compressed with a minimum match of three bytes, so a good default is a minimum match length of three octets.

Step 4: Configure optimization

The optimization level, between `0` and `9`, configures the trade-off between compression efficiency and encoding time. Lower optimization levels will compress faster but will result in larger compressed images, whereas higher optimization will compress slower and will result in smaller compressed images. A default value of 5 is a good trade-off.

Step 5: Initialize compressor

Once the compression parameters are set up, the compressor context is initialized for compression using `COMPRESS_LZMA_ENCODE_Init`.

2.3.3 Running the compressor

Once the compressor has been configured, data can be passed through it for compression. This example compresses some fixed, static data:

```
static const U8 _aMessage[] = {
  "With increasing complexity of today's devices, "
  "customers expect firmware updates over the life "
  "of a device. It's important to be able to replace "
  "both firmware images and FPGA configuration bitstreams "
  "in the field. Typically, firmware upgrades and other "
  "static content only grow in size over the lifetime of "
  "a device: features are added to software, not taken "
  "away, it's just what happens. Using quickly-written "
  "ad-hoc utilities to convert content directly to compilable "
  "code (possibly with some form of simple compression) is "
  "usual, but it's towards the end of a project when you "
  "realize that the static content just won't fit into your "
  "device with the baggage of new firmware features."
};
```

And proceed to the code that prepares the compressor to accept it:

```
  //
  Stream.pIn      = &_aMessage[0];    ❶
  Stream.AvailIn  = sizeof(_aMessage);
  Stream.pOut     = &_aOutput[0];    ❷
  Stream.AvailOut = sizeof(_aOutput);
  //
  Status = COMPRESS_LZMA_ENCODE_Run(&_Encoder, &Stream, 1);    ❸
  //
  if (Status < 0) {  ❹
    printf("Error encoding data\n");
  } else if (Status == 0) {
    printf("Did not encode completely, output buffer too small\n");
  } else {
    printf("%d bytes compressed to %u bytes\n",
           sizeof(_aMessage),
           sizeof(_aOutput) - Stream.AvailOut);    ❺
  }
}
```

Step 1: Set up data to compress

The input stream to the compressor is defined by the `pIn` and `AvailIn` members of the `COMPRESS_LZMA_STREAM` structure. The `pIn` member points to the first octet of data to be compressed and `AvailIn` defines the number of octets of input that are available to the compressor.

Step 2: Set up bitstream output buffer

In a similar manner to the input, the output stream is defined by the `pOut` and `AvailOut` members of the `COMPRESS_LZMA_STREAM` structure. The `pOut` member points to the object that receives the octets output by the compressor and `AvailOut` defines the number of octets that buffer can hold.

Step 3: Run the compressor

Once the input and output are set up, the compression is performed by calling `COMPRESS_LZMA_ENCODE_Run`. This function read data from the input stream, compresses it, and writes it to the output stream. We provide the encoding context, the stream object that specifies the data areas, and a final non-zero "flush" value that indicates to the compressor that this is all the input data it must deal with.

Step 4: Decode the processing status

When the compressor has worked its way through the compression algorithm, it returns a status code indicating its processing status. There are three different states that the compressor will return:

- Values less than zero indicate a processing error. The specific return value indicates the type of error detected.
- A zero status indicates that the compressor is still in its data compression phase and must be called again to continue compressing.
- Stats values greater than zero indicates that the compressor has finished processing and all input data is consumed and all output data provided.

In this case we expect the compressor to immediately finish as the output buffer that holds the compressed bitstream is much larger than the text input to the compressor. Whilst this will not always be the case, the text provided to the compressor in this case does indeed have redundancy and so compresses to a smaller size—running the application reveals how much.

Step 5: Extract the compressed data

When the compressor returns with a nonnegative return code, the compressed data can be read from the buffer given to the compressor. On return, the `AvailOut` member is decreased by one and the `pIn` pointer increased by one for each octet written to the buffer. The number of available octets, therefore, is equal to the number of octets remaining in the buffer on return subtracted from the size of the buffer on entry.

The above application generates the following output:

```
C:> COMPRESS_LZMA_OneShot.exe

687 bytes compressed to 428 bytes

C:> _
```

2.3.4 COMPRESS_LZMA_OneShot.c complete listing

```
/*********************************************************************
*                (c) SEGGER Microcontroller GmbH & Co. KG            *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : COMPRESS_LZMA_OneShotC.c
Purpose     : Example emCompress-LZMA one-shot compression.

*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/
```

```
#include "COMPRESS_LZMA_ENCODE.h"
#include <stdio.h>

/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/

static const U8 _aMessage[] = {
  "With increasing complexity of today's devices, "
  "customers expect firmware updates over the life "
  "of a device. It's important to be able to replace "
  "both firmware images and FPGA configuration bitstreams "
  "in the field. Typically, firmware upgrades and other "
  "static content only grow in size over the lifetime of "
  "a device: features are added to software, not taken "
  "away, it's just what happens. Using quickly-written "
  "ad-hoc utilities to convert content directly to compilable "
  "code (possibly with some form of simple compression) is "
  "usual, but it's towards the end of a project when you "
  "realize that the static content just won't fit into your "
  "device with the baggage of new firmware features."
};

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static COMPRESS_LZMA_ENCODE_CONTEXT _Encoder;
static U8                           _aOutput[512];

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       main()
*
*  Function description
*    Application entry point.
*
*  Parameters
*    argc - Argument count.
*    argv - Argument vector.
*/
int main(int argc, char **argv) {
  COMPRESS_LZMA_STREAM Stream;
  COMPRESS_LZMA_PARAS  Paras;
  int                  Status;
  //
  Paras.LC         = 0;
  Paras.LP         = 0;
  Paras.PB         = 0;
  Paras.WindowSize = 1024;
  Paras.MinLen     = 3;
  Paras.MaxLen     = 273;
  Paras.Optimize   = 5;
  //
  COMPRESS_LZMA_ENCODE_Init(&_Encoder, &Paras);
  //
  Stream.pIn      = &_aMessage[0];
  Stream.AvailIn  = sizeof(_aMessage);
  Stream.pOut     = &_aOutput[0];
  Stream.AvailOut = sizeof(_aOutput);
  //
  Status = COMPRESS_LZMA_ENCODE_Run(&_Encoder, &Stream, 1);
  //
  if (Status < 0) {
    printf("Error encoding data\n");
  } else if (Status == 0) {
    printf("Did not encode completely, output buffer too small\n");
  } else {
    printf("%u bytes compressed to %u bytes\n",
           (unsigned)sizeof(_aMessage),
           (unsigned)(sizeof(_aOutput) - Stream.AvailOut));
  }
  return 0;
}

/*************************** End of file ****************************/
```

2.4 Compressing a file

This example, `COMPRESS_LZMA_Compress.c`, expands on the previous example to compress a file, block by block, writing the output to a compressed file with a correct header.

For a complete listing of this application, see *`COMPRESS_LZMA_Compress.c` complete listing* on page 23.

2.4.1 Incremental compression

Rather than describe the boilerplate code of processing a command line, opening files, and setting up the encoder as previously, we present the code that incrementally compresses a file:

```
Status = 0;    ❶
while (Status == 0) {
  Stream.pIn     = _aInBuffer;    ❷
  Stream.AvailIn = fread(_aInBuffer, 1, sizeof(_aInBuffer), pInFile);
  //
  Flush = Stream.AvailIn != sizeof(_aInBuffer);    ❸
  //
  while (Status == 0 && (Stream.AvailIn > 0 || Flush)) {    ❹
    //
    Stream.pOut     = &_aOutBuffer[0];    ❺
    Stream.AvailOut = sizeof(_aOutBuffer);
    //
    Status = COMPRESS_LZMA_ENCODE_Run(&_Encoder, &Stream, Flush);    ❻
    if (Status >= 0) {
      fwrite(_aOutBuffer, 1, sizeof(_aOutBuffer) - Stream.AvailOut, pOutFile);
    }
  }
}
```

Step 1: Set up main compress loop

The compressor maintains the current compression status in the variable `Status`; when the status is nonzero, compression has completed either successfully or with an error.

Step 2: Set up input

There are no surprises here, simply set up the block to be passed to the compressor by reading the source file.

Step 3: Set up flush flag

The compressor needs to be told that there will be no more input and finish up compression: that happens when we have not completely filled the input buffer from the source file.

Step 4: Compress until block consumed

The inner loop will continually call the compressor to consume the input buffer and deliver data to the output buffer. The `while` loop ensures that we call the compressor when we have more data or we need to flush. When the status indicates that compression is complete, the loop terminates.

Step 5: Set up output buffer

The output buffer can be any size—larger buffers may lead to faster compression, but the overhead is not very significant.

Step 6: Compress and write output buffer

This is the same as the previous application, only this time we write the compressed output to the file, and this concludes the compression loop.

2.4.2 Encapsulating the bitstream

The compression loop generates a compressed bitstream, but the header which the decompressor requires is missing. The header cannot be constructed until the entire bitstream is compressed, so there additional code is required that deals with this.

The header is 13 bytes in size:

```
U8 aHeader[13];
```

Before writing the compressed bitstream in the loop above the file pointer is advanced to reserve space for a header:

```
fseek(pOutFile, sizeof(aHeader), SEEK_SET);
```

Once the compressor has competed compression, the header can be constructed and written to the start of the file:

```
COMPRESS_LZMA_ENCODE_WrHeader(&_Encoder, &aHeader[0]);
fseek(pOutFile, 0, SEEK_SET);
fwrite(aHeader, 1, sizeof(aHeader), pOutFile);
```

This concludes the example.

2.4.3 COMPRESS_LZMA_Compress.c complete listing

```
/*********************************************************************
*                (c) SEGGER Microcontroller GmbH & Co. KG            *
*                         The Embedded Experts                       *
*                            www.segger.com                          *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : COMPRESS_LZMA_Compress.c
Purpose     : Example emCompress-LZMA one-shot compression.

*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/

#include "COMPRESS_LZMA_ENCODE.h"
#include <stdio.h>
#include <stdlib.h>

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/

#define BUFFER_IN_SIZE    (32*1024)
#define BUFFER_OUT_SIZE   (32*1024)

/*********************************************************************
*
*       Static data
*
```

```
**********************************************************************
*/

static COMPRESS_LZMA_ENCODE_CONTEXT _Encoder;
static U8                           _aInBuffer [BUFFER_IN_SIZE];
static U8                           _aOutBuffer[BUFFER_OUT_SIZE];

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _GetStatusText()
*
*  Function description
*    Decode emCompress-LZMA status code.
*
*  Parameters
*    Status - Status code.
*
*  Return value
*    Non-zero pointer to status description.
*/
static const char * _GetStatusText(int Status) {
  if (Status >= 0) {
    return "OK";
  }
  switch (Status) {
  case COMPRESS_LZMA_STATUS_PARAMETER_ERROR: return "Parameter error";
  case COMPRESS_LZMA_STATUS_BITSTREAM_ERROR: return "Bitstream error";
  case COMPRESS_LZMA_STATUS_USAGE_ERROR:     return "Usage error";
  default:                                   return "Unknown error";
  }
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       main()
*
*  Function description
*    Application entry point.
*
*  Parameters
*    argc - Argument count.
*    argv - Argument vector.
*/
int main(int argc, char **argv) {
  COMPRESS_LZMA_STREAM   Stream;
  COMPRESS_LZMA_PARAS    Paras;
  FILE                 * pInFile;
  FILE                 * pOutFile;
  U8                     aHeader[13];
  int                    Flush;
  int                    Status;
  //
  if (argc != 3) {
    fprintf(stderr, "Usage: %s <input-file> <output-file>\n", argv[0]);
    exit(100);
  }
  //
  pInFile = fopen(argv[1], "rb");
  if (pInFile == 0) {
    printf("%s: can't open %s for reading\n", argv[0], argv[1]);
    exit(100);
  }
  //
  pOutFile = fopen(argv[2], "wb");
  if (pOutFile == 0) {
    printf("%s: can't open %s for writing\n", argv[0], argv[2]);
    fclose(pInFile);
    exit(100);
  }
  //
  Paras.LC         = 0;
  Paras.LP         = 0;
  Paras.PB         = 0;
  Paras.WindowSize = 1024;
  Paras.MinLen     = 3;
  Paras.MaxLen     = 273;
  Paras.Optimize   = 5;
  //
  COMPRESS_LZMA_ENCODE_Init(&_Encoder, &Paras);
  //
  // Reserve space for an LZMA header in the output file.
  //
  fseek(pOutFile, sizeof(aHeader), SEEK_SET);
  //
  Status = 0;
  while (Status == 0) {
    //
    // Read next chunk and set up coder buffers.
    //
    Stream.pIn     = _aInBuffer;
    Stream.AvailIn = fread(_aInBuffer, 1, sizeof(_aInBuffer), pInFile);
    //
    // Flush compressor on end of input file.
    //
    Flush = Stream.AvailIn != sizeof(_aInBuffer);
    //
    // Encode input data, writing encoded data to file.
    //
    while (Status == 0 && (Stream.AvailIn > 0 || Flush)) {
      //
      Stream.pOut     = &_aOutBuffer[0];
      Stream.AvailOut = sizeof(_aOutBuffer);
      //
      Status = COMPRESS_LZMA_ENCODE_Run(&_Encoder, &Stream, Flush);
      if (Status >= 0) {
        fwrite(_aOutBuffer, 1, sizeof(_aOutBuffer) - Stream.AvailOut, pOutFile);
      }
    }
  }
  //
  // Rewind file and write correct header.
  //
  COMPRESS_LZMA_ENCODE_WrHeader(&_Encoder, &aHeader[0]);
  fseek(pOutFile, 0, SEEK_SET);
  fwrite(aHeader, 1, sizeof(aHeader), pOutFile);
  //
  fclose(pInFile);
  fclose(pOutFile);
  //
  if (Status < 0) {
    printf("FATAL: %s\n", _GetStatusText(Status));
    exit(100);
  }
  return 0;
}

/*************************** End of file ****************************/
```

2.5 Optimizing compression parameters

If you require the best compression possible, you can compress a file with varying parameters and search for the best. It's not possible to find optimal compression parameters, in general, but it is possible to change the context parameters and search for those that provide the best compression.

The optimizer runs over the input file multiple times, varying the probability model's parameters:

```
for (Paras.LC = 0; Paras.LC <= 8; ++Paras.LC) {
  for (Paras.LP = 0; Paras.LP <= 4; ++Paras.LP) {
    for (Paras.PB = 0; Paras.PB <= 4; ++Paras.PB) {
```

The compressor is run inside the innermost loop, using the selected parameters, and the result printed. The output from the application shows how the search progresses:

```
C:> COMPRESS_LZMA_Optimize.exe K66_SEGGER_emPower.bin

LC=0, LP=0, PB=0: 150842
LC=0, LP=0, PB=1: 149306
LC=0, LP=1, PB=0: 147939
LC=0, LP=1, PB=1: 146402
LC=1, LP=1, PB=1: 145641
LC=2, LP=1, PB=1: 145139
LC=3, LP=1, PB=1: 144967

C:> _
```

The application itself is very straightforward and is not explained further.

2.5.1 COMPRESS_LZMA_Optimize.c complete listing

```
/*********************************************************************
*                (c) SEGGER Microcontroller GmbH & Co. KG            *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : COMPRESS_LZMA_Optimize.c
Purpose     : Find optimal compression parameters for a file.

*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/

#include "COMPRESS_LZMA_ENCODE.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/
```

```
#define BUFFER_IN_SIZE    (32*1024)
#define BUFFER_OUT_SIZE   (32*1024)

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static union {
  COMPRESS_LZMA_ENCODE_CONTEXT    Encoder;
  COMPRESS_LZMA_ENCODE_T2_CONTEXT EncoderT2;
} _u;
static U8                           _aInBuffer [BUFFER_IN_SIZE];
static U8                           _aOutBuffer[BUFFER_OUT_SIZE];

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       main()
*
*  Function description
*    Application entry point.
*
*  Parameters
*    argc - Argument count.
*    argv - Argument vector.
*/
int main(int argc, char **argv) {
  COMPRESS_LZMA_STREAM   Stream;
  COMPRESS_LZMA_PARAS    Paras;
  const char           * sInputPathname;
  FILE                 * pInFile;
  U64                    MinSize;
  U64                    EncodedSize;
  int                    Flush;
  int                    Status;
  int                    i;
  int                    ThumbMode;
  unsigned               MaxLP;
  unsigned               MaxLC;
  unsigned               MaxPB;
  //
  Paras.WindowSize = COMPRESS_LZMA_CONFIG_ENCODE_WINDOW_SIZE_MAX;
  Paras.MinLen     = 3;
  Paras.MaxLen     = 273;
  Paras.Optimize   = 5;
  MaxLC            = COMPRESS_LZMA_CONFIG_ENCODE_LC_MAX;
  MaxLP            = COMPRESS_LZMA_CONFIG_ENCODE_LP_MAX;
  MaxPB            = COMPRESS_LZMA_CONFIG_ENCODE_PB_MAX;
  //
  sInputPathname = 0;
  //
  for (i = 1; i < argc; ++i) {
    const char *sArg = argv[i];
    if (strncmp(sArg, "-lc=", 4) == 0) {
      MaxLC = COMPRESS_LZMA_MIN((unsigned)atoi(&sArg[4]), MaxLC);
    } else if (strncmp(sArg, "-lp=", 4) == 0) {
      MaxLP = COMPRESS_LZMA_MIN((unsigned)atoi(&sArg[4]), MaxLP);
    } else if (strncmp(sArg, "-pb=", 4) == 0) {
      MaxPB = COMPRESS_LZMA_MIN((unsigned)atoi(&sArg[4]), MaxPB);
    } else if (strncmp(sArg, "-ws=", 4) == 0) {
      Paras.WindowSize = atoi(&sArg[4]);
    } else if (strncmp(sArg, "-O", 2) == 0) {
      Paras.Optimize = atoi(&sArg[2]);
    } else if (strncmp(sArg, "-k", 2) == 0) {
      Paras.LC = 3;
      Paras.LP = 0;
      Paras.PB = 2;
    } else if (strncmp(sArg, "-minlen=", 8) == 0) {
      Paras.MinLen = atoi(&sArg[8]);
    } else if (strncmp(sArg, "-maxlen=", 8) == 0) {
      Paras.MaxLen = atoi(&sArg[8]);
    } else if (sArg[0] == '-') {
      printf("%s: unknown option %s\n", argv[0], sArg);
      exit(100);
    } else if (sInputPathname == 0) {
      sInputPathname = sArg;
    } else {
      printf("%s: too many filenames\n", argv[0]);
      exit(100);
    }
  }
  //
  if (sInputPathname == 0) {
    printf("%s: require input filename\n", argv[0]);
    exit(100);
  }
  //
  pInFile = fopen(sInputPathname, "rb");
  if (pInFile == 0) {
    printf("%s: can't open %s for reading\n", argv[0], argv[1]);
    exit(100);
  }
  MinSize          = ~0ULL;
  //
  for (Paras.LC = 0; Paras.LC <= MaxLC; ++Paras.LC) {
    for (Paras.LP = 0; Paras.LP <= MaxLP; ++Paras.LP) {
      for (Paras.PB = 0; Paras.PB <= MaxPB; ++Paras.PB) {
        for (ThumbMode = 0; ThumbMode < 2; ++ThumbMode) {
          //
          if (ThumbMode) {
            Status = COMPRESS_LZMA_ENCODE_T2_Init(&_u.EncoderT2, &Paras);
          } else {
            Status = COMPRESS_LZMA_ENCODE_Init(&_u.Encoder, &Paras);
          }
          if (Status < 0) {
            continue;
          }
          //
          fseek(pInFile, 0, SEEK_SET);
          //
          EncodedSize = 0;
          Status = 0;
          while (Status == 0) {
            //
            // Read next chunk and set up coder buffers.
            //
            Stream.pIn     = _aInBuffer;
            Stream.AvailIn = fread(_aInBuffer, 1, sizeof(_aInBuffer), pInFile);
            //
            // Flush compressor on end of input file.
            //
            Flush = Stream.AvailIn != sizeof(_aInBuffer);
            //
            // Encode input data.
            //
            while (Status == 0 && (Stream.AvailIn > 0 || Flush)) {
              //
              Stream.pOut     = &_aOutBuffer[0];
              Stream.AvailOut = sizeof(_aOutBuffer);
              //
              if (ThumbMode) {

  Status = COMPRESS_LZMA_ENCODE_T2_Run(&_u.EncoderT2, &Stream, Flush);
              } else {
                Status = COMPRESS_LZMA_ENCODE_Run(&_u.Encoder, &Stream, Flush);
              }
              EncodedSize += sizeof(_aOutBuffer) - Stream.AvailOut;
            }
          }
          //
          printf("\rLC=%d, LP=%d, PB=%d %s: %llu            \r",
                Paras.LC, Paras.LP, Paras.PB, (ThumbMode != 0) ? "T2" : "
 ", EncodedSize);
          if (EncodedSize < MinSize) {
            MinSize = EncodedSize;
            printf("\n");
          }
        }
      }
    }
  }
  //
  printf("\r                              \r");
  return 0;
}

/*************************** End of file ****************************/
```

2.6 Prefiltering

It is possible to improve the effectiveness of compression when the compressor knows the structure of the data and can use a domain-specific filter. In particular, for firmware upgrades, preprocessing the input to the LZMA compressor to provide improve the probability of matches detected by the LZSS engine is particularly helpful.

emCompress-LZMA contains compressor and decompressor functions with an inbuilt filter using architecture-specific transformations for the Thumb-2 instruction set, which further reduce the compressed size of firmware. See *COMPRESS_LZMA_DECODE_T2_Run()* and *COMPRESS_LZMA_ENCODE_T2_Run()*.

2.7 Configuring the compressor

If you have a source distribution of emCompress-LZMA's compressor, you can configure and rebuild the compressor.

The following definitions must be configured for emCompress-LZMA's compressor. The user configuration file for this `COMPRESS_LZMA_ENCODE_Conf.h`. The configuration options are described in the following sections.

2.7.1 Compressor data types

Default

There is no default as the required emCompress-LZMA data types are configured by using a type definition rather than a preprocessor symbol.

Recommended configuration

```
typedef unsigned long long U64;
typedef unsigned int       U32;
typedef unsigned short     U16;
typedef unsigned char       U8;
typedef unsigned int  ULEAST16;
```

Description

Defines the types that the emCompress-LZMA compressor will in its data structures when compressing.

The types above are typical for a 32-bit machine, but the compressor will work with any type definition for each as long as each type can store the prescribed number of bits without loss. For instance, it may well be that the compilation system provides a `<stdint.h>` header file, in which case the following is acceptable:

```
typedef uint64_t        U64;
typedef uint32_t        U32;
typedef uint16_t        U16;
typedef uint8_t         U8;
typedef uint_least16_t  ULEAST16;
```

2.7.2 Maximum number of literal context bits

Default

```
#define COMPRESS_LZMA_CONFIG_ENCODE_LC_MAX     8
```

Description

Configures the maximum number of literal context bits that the compressor will support. Valid configuration values for this symbol are 0 through 8. Smaller values will reduce the memory required for the encoder context but will also reduce the effectiveness of the compressor.

It is a (diagnosed) error to pass a value greater than the configured value as the `LC` member of the `COMPRESS_LZMA_STATE` structure when initializing the compressor. Conversely, it is possible to supply smaller values in order to optimize compressor output or to ensure that the `LC` value is acceptable to the target decompressor.

2.7.3 Maximum number of literal position context bits

Default

```
#define COMPRESS_LZMA_CONFIG_ENCODE_LP_MAX     4
```

Description

Configures the maximum number of literal position context bits that the compressor will support. Valid configuration values for this symbol are 0 through 4. Smaller values will reduce the memory required for the encoder context but will also reduce the effectiveness of the compressor.

It is a (diagnosed) error to pass a value greater than the configured value as the `LP` member of the `COMPRESS_LZMA_STATE` structure when initializing the compressor. Conversely, it is possible to supply smaller values in order to optimize compressor output or to ensure that the `LP` value is acceptable to the target decompressor.

2.7.4 Maximum number of position bits

Default

```
#define COMPRESS_LZMA_CONFIG_EBCODE_PB_MAX     4
```

Description

Configures the maximum number of position context bits that the compressor will support. Valid configuration values for this symbol are 0 through 4. Smaller values will reduce the memory required for the encoder context but will also reduce the effectiveness of the compressor.

It is a (diagnosed) error to pass a value greater than the configured value as the `PB` member of the `COMPRESS_LZMA_STATE` structure when initializing the compressor. Conversely, it is possible to supply smaller values in order to optimize compressor output or to ensure that the `PB` value is acceptable to the target decompressor.

2.7.5 Maximum window size

Default

```
#define COMPRESS_LZMA_CONFIG_ENCODE_WINDOW_SIZE_MAX  (1024*1024)
```

Description

Configures the maximum window size that the compressor will support. Valid configuration values for this symbol are 1 through `0xFFFFFFFF`. Smaller values will reduce the memory required for the encoder context but will also reduce the effectiveness of the compressor. It is highly recommended that the window size remain at 1 megabyte or smaller for reasonable compression times.

It is a (diagnosed) error to pass a value greater than the configured value as the `WindowSize` member of the `COMPRESS_LZMA_STATE` structure when initializing the compressor. Conversely, it is possible to supply smaller values in order to optimize compressor output or to ensure that the `WindowSize` value is acceptable to the target decompressor.

2.7.6 Match hash table size

Default

```
#define COMPRESS_LZMA_CONFIG_HASH_TABLE_SIZE   (256*256*256)
```

Description

Configures the LZSS hash table size for fast matching. Valid configuration values for this symbol are 1 through 16777216. Smaller values will reduce the memory required for the encoder context but will also reduce the effectiveness of the compressor as collisions in the hash table are more likely and therefore potential matches will be missed.

2.7.7 Shipped compressor configuration

This is the shipped configuration file, `COMPRESS_LZMA_ENCODE_Conf.h`, which is also the configuration that matches the library version of the compressor.

```
/*********************************************************************
*                (c) SEGGER Microcontroller GmbH & Co. KG            *
*                         The Embedded Experts                       *
*                            www.segger.com                          *
**********************************************************************
----------------------------------------------------------------------
Purpose     : Configuration settings for emCompress-LZMA.
-------------------------- END-OF-HEADER -----------------------------
*/

#ifndef COMPRESS_LZMA_ENCODE_CONF_H
#define COMPRESS_LZMA_ENCODE_CONF_H

#ifdef __cplusplus
extern "C" {
#endif

//
// Configuration of types required by emCompress-LZMA
//
typedef unsigned long long U64;   // An unsigned integer with 64 bits
typedef unsigned int       U32;   // An unsigned integer with 32 bits
typedef unsigned short     U16;   // An unsigned integer with 16 bits
typedef unsigned char       U8;   // A byte as unsigned value
typedef unsigned int  ULEAST16;   // unsigned integer with a range of at least
 0 to (2^16)-1

// Maximum number of literal context (LC) bits the encoder will accept  [0, 8].
#define COMPRESS_LZMA_CONFIG_ENCODE_LC_MAX           8u

// Maximum number of literal position (LP) bits the encoder will accept [0, 4].
#define COMPRESS_LZMA_CONFIG_ENCODE_LP_MAX           4u

// Maximum number of position bits (PB) bits the encoder will accept    [0, 4].
#define COMPRESS_LZMA_CONFIG_ENCODE_PB_MAX           4u

// Hash table size for encoding: 16m entries, 4 bytes/entry, 64MB hash table
#define COMPRESS_LZMA_CONFIG_HASH_TABLE_SIZE         (256uL*256uL*256uL)

// Maximum size of LZMA window
#define COMPRESS_LZMA_CONFIG_ENCODE_WINDOW_SIZE_MAX  (1024uL*1024uL)

// Buffer size for Thumb precondition operation
#define COMPRESS_LZMA_CONFIG_ENCODE_T2_PRECONDITION_BUFF_SIZE   2048uL
  // must be even and >= 8

#ifdef __cplusplus
}
#endif

#endif

/*************************** End of file ****************************/
```

2.8 Avoiding run-time failures

In order to avoid run-time failures, the application has to observe basic rules when using the `COMPRESS_LZMA` compressor API.

The API makes use of two data structures that must be handled by the application:

COMPRESS_LZMA_ENCODE_CONTEXT

This data structure is private the encoder. The calling application has to make sure that:

- This structure resides in a valid memory area.
- Is properly initialized using `COMPRESS_LZMA_ENCODE_Init()`.
- Is never modified between calls of the compressor functions.

The compressor is not able to check the validity of this data structure or detect modifications. If the conditions above are not met, the behavior of the compressor function is undefined.

COMPRESS_LZMA_STREAM

The application has to properly initialize this structure before any call to the compressor function. It provides input and output buffers for the compressor via pointer and size information. While `NULL`-pointers are checked, the compressor is not able to check for valid memory areas of this buffers. If the provided buffers do not reside in valid memory, the behavior of the compressor function is undefined.

Chapter 3

Decompressor

This section describes the emCompress-LZMA decompressor interface.

3.1 Delivery format

The emCompress-LZMA decompressor is delivered as source code for integration into your target system.

3.2 Streaming decompression

The first and only example, `COMPRESS_LZMA_Decompress.c`, is an application that will decompress the output file from the previous compressor.

For a complete listing of this application, see *`COMPRESS_LZMA_Decompress.c` complete listing* on page 39.

3.2.1 Decompressor include files

The emCompress-LZMA decoding API is exposed by including the file `COMPRESS_LZMA_DECODE.h`:

3.2.2 Decompressor loop

The decompressor uses the same stream interface object as the compressor, `COMPRESS_LZMA_STREAM`. And the decompressor “run” code uses an identical set of parameters and return codes which makes it very simple to use.

In fact, the decompressor loop is identical in form to the compressor loop:

```
Status = 0;
while (Status == 0) {
  //
  Stream.pIn      = _aInBuffer;
  Stream.AvailIn = fread(_aInBuffer, 1, sizeof(_aInBuffer), pInFile);
  EncodedLen     += Stream.AvailIn;
  //
  Flush = Stream.AvailIn != _BlockInSize;
  //
  while (Status == 0 && (Stream.AvailIn > 0 || Flush)) {
    //
    Stream.pOut      = &_aOutBuffer[0];
    Stream.AvailOut = sizeof(_aOutBuffer);
    //
    Status = COMPRESS_LZMA_DECODE_Run(&_Decoder, &Stream, Flush);
    if (Status >= 0) {
      fwrite(_aOutBuffer, 1, sizeof(_aOutBuffer) - Stream.AvailOut, pOutFile);
    }
  }
}
```

3.2.3 COMPRESS_LZMA_Decompress.c complete listing

```
/*********************************************************************
*               (c) SEGGER Microcontroller GmbH & Co. KG             *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : COMPRESS_LZMA_Decompress.c
Purpose     : Example emCompress-LZMA one-shot compression.

*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/
```

```
#include "COMPRESS_LZMA_DECODE.h"
#include <stdio.h>
#include <stdlib.h>

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/

#define BUFFER_IN_SIZE    (32*1024)
#define BUFFER_OUT_SIZE   (32*1024)

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static COMPRESS_LZMA_DECODE_CONTEXT _Decoder;
static U8                           _aInBuffer [BUFFER_IN_SIZE];
static U8                           _aOutBuffer[BUFFER_OUT_SIZE];

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _GetStatusText()
*
*  Function description
*    Decode emCompress-LZMA status code.
*
*  Parameters
*    Status - Status code.
*
*  Return value
*    Non-zero pointer to status description.
*/
static const char * _GetStatusText(int Status) {
  if (Status >= 0) {
    return "OK";
  }
  switch (Status) {
  case COMPRESS_LZMA_STATUS_PARAMETER_ERROR: return "Parameter error";
  case COMPRESS_LZMA_STATUS_BITSTREAM_ERROR: return "Bitstream error";
  case COMPRESS_LZMA_STATUS_USAGE_ERROR:     return "Usage error";
  default:                                   return "Unknown error";
  }
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       main()
*
```

```
*  Function description
*    Application entry point.
*
*  Parameters
*    argc - Argument count.
*    argv - Argument vector.
*/
int main(int argc, char **argv) {
  COMPRESS_LZMA_STREAM   Stream;
  FILE                 * pInFile;
  FILE                 * pOutFile;
  int                    Flush;
  int                    Status;
  //
  if (argc != 3) {
    fprintf(stderr, "Usage: %s <input-file> <output-file>\n", argv[0]);
    exit(100);
  }
  //
  pInFile = fopen(argv[1], "rb");
  if (pInFile == 0) {
    printf("%s: can't open %s for reading\n", argv[0], argv[1]);
    exit(100);
  }
  //
  pOutFile = fopen(argv[2], "wb");
  if (pOutFile == 0) {
    printf("%s: can't open %s for writing\n", argv[0], argv[2]);
    fclose(pInFile);
    exit(100);
  }
  //
  COMPRESS_LZMA_DECODE_Init(&_Decoder);
  //
  Status = 0;
  while (Status == 0) {
    //
    // Read next chunk and set up coder buffers.
    //
    Stream.pIn     = _aInBuffer;
    Stream.AvailIn = fread(_aInBuffer, 1, sizeof(_aInBuffer), pInFile);
    //
    // Flush compressor on end of input file.
    //
    Flush = Stream.AvailIn != sizeof(_aInBuffer);
    //
    // Encode input data, writing encoded data to file.
    //
    while (Status == 0 && (Stream.AvailIn > 0 || Flush)) {
      //
      Stream.pOut     = &_aOutBuffer[0];
      Stream.AvailOut = sizeof(_aOutBuffer);
      //
      Status = COMPRESS_LZMA_DECODE_Run(&_Decoder, &Stream, Flush);
      if (Status >= 0) {
        fwrite(_aOutBuffer, 1, sizeof(_aOutBuffer) - Stream.AvailOut, pOutFile);
      }
    }
  }
  //
  fclose(pInFile);
  fclose(pOutFile);
  //
  if (Status < 0) {
    printf("FATAL: %s\n", _GetStatusText(Status));
    exit(100);
  }
  return 0;
}

/*************************** End of file ****************************/
```

3.3 Configuring the decompressor

The following definitions must be configured for emCompress-LZMA’s decompressor. The user configuration file for this is `COMPRESS_LZMA_DECODE_Conf.h`. The configuration options are described in the following sections.

3.3.1 Decompressor data types

Default

There is no default as the required emCompress-lZMA data types are configured by using a type definition rather than a preprocessor symbol.

Recommended configuration

```
typedef unsigned long long U64;
typedef unsigned int       U32;
typedef unsigned short     U16;
typedef unsigned char       U8;
typedef unsigned int  ULEAST16;
typedef int           ILEAST16;
```

Description

Defines the types that the emCompress-LZMA decompressor will in its data structures when compressing.

The types above are typical for a 32-bit machine, but the decompressor will work with any type definition for each as long as each type can store the prescribed number of bits without loss. For instance, it may well be that the compilation system provides a `<stdint.h>` header file, in which case the following is acceptable:

```
typedef uint64_t             U64;
typedef uint32_t             U32;
typedef uint16_t             U16;
typedef uint8_t               U8;
typedef int_least16_t   ILEAST16;
typedef uint_least16_t  ULEAST16;
```

3.3.2 Maximum accepted literal context bits

Default

```
#define COMPRESS_LZMA_CONFIG_DECODE_LC_MAX     8
```

Description

Configures the maximum number of literal context bits that the decompressor will support. Valid configuration values for this symbol are 0 through 8. Smaller values will reduce the memory required for the decoder context but will also limit the capability of the compressor when encoding a bitstream acceptable to the decompressor.

It is a (diagnosed) error to attempt decoding a bitstream which is encoded with an `LC` value greater than the configured maximum.

3.3.3 Maximum accepted literal position context bits

Default

```
#define COMPRESS_LZMA_CONFIG_DECODE_LP_MAX     4
```

Description

Configures the maximum number of literal position context bits that the decompressor will support. Valid configuration values for this symbol are 0 through 4. Smaller values will reduce the memory required for the decoder context but will also limit the capability of the compressor when encoding a bitstream acceptable to the decompressor.

It is a (diagnosed) error to attempt decoding a bitstream which is encoded with an LP value greater than the configured maximum.

3.3.4 Maximum accepted position bits

Default

```
#define COMPRESS_LZMA_CONFIG_DECODE_PB_MAX     4
```

Description

Configures the maximum number of position context bits that the decompressor will support. Valid configuration values for this symbol are 0 through 4. Smaller values will reduce the memory required for the decoder context but will also limit the capability of the compressor when encoding a bitstream acceptable to the decompressor.

It is a (diagnosed) error to attempt decoding a bitstream which is encoded with an PB value greater than the configured maximum.

3.3.5 Maximum accepted window size

Default

```
#define COMPRESS_LZMA_CONFIG_DECODE_WINDOW_SIZE_MAX  (1024*1024)
```

Description

Configures the maximum window size that the decompressor will support. Valid configuration values for this symbol are 1 through 0xFFFFFFFF. Smaller values will reduce the memory required for the decoder context but will also limit the capability of the compressor when encoding a bitstream acceptable to the decompressor.

It is a (diagnosed) error to attempt decoding a bitstream which is encoded with a window size greater than the configured maximum decoder window size.

3.4 Shipped decompressor configuration

This is the shipped configuration file, `COMPRESS_LZMA_DECODE_Conf.h`.

```
/*********************************************************************
*                (c) SEGGER Microcontroller GmbH & Co. KG            *
*                         The Embedded Experts                       *
*                            www.segger.com                          *
**********************************************************************
----------------------------------------------------------------------
Purpose     : Configuration settings for emCompress-LZMA decoder.
-------------------------- END-OF-HEADER -----------------------------
*/

#ifndef COMPRESS_LZMA_DECODE_CONF_H
#define COMPRESS_LZMA_DECODE_CONF_H

#ifdef __cplusplus
extern "C" {
#endif

//
// Configuration of types required by emCompress-LZMA
//
typedef unsigned long long U64;   // An unsigned integer with 64 bits
typedef unsigned int       U32;   // An unsigned integer with 32 bits
typedef unsigned short     U16;   // An unsigned integer with 16 bits
typedef unsigned char       U8;   // A byte as unsigned value
typedef unsigned int  ULEAST16;   // unsigned integer with a range of at least
 0 to (2^16)-1
typedef int           ILEAST16;   // signed integer with a range of at least -
(2^15) to (2^15)-1

// Maximum number of literal context (LC) bits the decoder will accept  [0, 8].
#define COMPRESS_LZMA_CONFIG_DECODE_LC_MAX     8u

// Maximum number of literal position (LP) bits the decoder will accept [0, 4].
#define COMPRESS_LZMA_CONFIG_DECODE_LP_MAX     4u

// Maximum number of position bits (PB) bits the decoder will accept    [0, 4].
#define COMPRESS_LZMA_CONFIG_DECODE_PB_MAX     4u

// Maximum size of LZMA window
#define COMPRESS_LZMA_CONFIG_DECODE_WINDOW_SIZE_MAX  (1024uL*1024uL)

// Buffer size for Thumb precondition operation
#define COMPRESS_LZMA_CONFIG_DECODE_T2_PRECONDITION_BUFF_SIZE   128uL
  // must be even and >= 8

#ifdef __cplusplus
}
#endif

#endif

/*************************** End of file ****************************/
```

3.5 Avoiding run-time failures

In order to avoid run-time failures, the application has to observe basic rules when using the `COMPRESS_LZMA` decompressor API.

The API makes use of two data structures that must be handled by the application:

COMPRESS_LZMA_DECODE_CONTEXT

This data structure is private the decoder. The calling application has to make sure that:

- This structure resides in a valid memory area.
- Is properly initialized using `COMPRESS_LZMA_DECODE_Init()`.
- Is never modified between calls of the decompressor functions.

The decompressor is not able to check the validity of this data structure or detect modifications. If the conditions above are not met, the behavior of the decompressor function is undefined.

COMPRESS_LZMA_STREAM

The application has to properly initialize this structure before any call to the decompressor function. It provides input and output buffers for the decompressor via pointer and size information. While `NULL`-pointers are checked, the decompressor is not able to check for valid memory areas of this buffers. If the provided buffers do not reside in valid memory, the behavior of the decompressor function is undefined.

Invalid input data (data that does not result from a valid compression) may be detected by the decompressor, but may also result in invalid output data. To verify data consistency, the use of any CRC of HASH mechanism by the application is recommended. Invalid input data never leads to undefined behavior, especially the decompressor will never access memory outside the provided buffers.

3.6 Decompressor memory use

3.6.1 RAM usage

The decompressor has no static data requirement, the only memory required to decompress a stream is provided provided to the decompressor by the client in a `COMPRESS_LZMA_DECODE_CONTEXT` object.

The decoder context size varies according to selected emCompress-LZMA decoder configuration and the specific compilation options selected to compile the decompressor.

Because the target processor’s type sizes and alignments are not known to the generic compressor, and cannot easily be modeled without significant probability of error, the sizes presented here are for a 32-bit Cortex-M processor which should generally translate well to many 32-bit architectures including 32-bit “x86” targets. For nonstandard architectures, such as pure DSPs which are typically word-addressed machines, the memory requirement in words can be derived by configuring the compiler and using the `sizeof` operator to query the memory size in units of `char` rather than byte units.

The context size of the decoder is modeled using the four parameters `LC`, `LP`, `PB`, and `WindowSize`. The application `COMPRESS_LZMA_Sizes.c` will estimate the memory required for a decompressor context.

It must be stressed that this computation is an estimate and the exact memory requirement can be computed with the `sizeof` operator applied to the `COMPRESS_LZMA_DECODE_CONTEXT`. However, the exact size of the decoding context is generally not required, only a sense of its magnitude.

The following table displays the decompressor context size for each combination of LC. LP, and PB. Each size is measured in kilobytes *with a zero window size*, for a typical 32-bit byte-addressed machine.

LP	PB	LC=0	LC=1	LC=2	LC=3	LC=4	LC=5	LC=6	LC=7	LC=8
0	0	3.8	5.3	8.3	14.3	26.3	50.3	98.3	194.3	386.3
0	1	4.0	5.5	8.5	14.5	26.5	50.5	98.5	194.5	386.5
0	2	4.3	5.8	8.8	14.8	26.8	50.8	98.8	194.8	386.8
0	3	4.9	6.4	9.4	15.4	27.4	51.4	99.4	195.4	387.4
0	4	6.2	7.7	10.7	16.7	28.7	52.7	100.7	196.7	388.7
1	0	5.3	8.3	14.3	26.3	50.3	98.3	194.3	386.3	770.3
1	1	5.5	8.5	14.5	26.5	50.5	98.5	194.5	386.5	770.5
1	2	5.8	8.8	14.8	26.8	50.8	98.8	194.8	386.8	770.8
1	3	6.4	9.4	15.4	27.4	51.4	99.4	195.4	387.4	771.4
1	4	7.7	10.7	16.7	28.7	52.7	100.7	196.7	388.7	772.7
2	0	8.3	14.3	26.3	50.3	98.3	194.3	386.3	770.3	1538.3
2	1	8.5	14.5	26.5	50.5	98.5	194.5	386.5	770.5	1538.5
2	2	8.8	14.8	26.8	50.8	98.8	194.8	386.8	770.8	1538.8
2	3	9.4	15.4	27.4	51.4	99.4	195.4	387.4	771.4	1539.4
2	4	10.7	16.7	28.7	52.7	100.7	196.7	388.7	772.7	1540.7
3	0	14.3	26.3	50.3	98.3	194.3	386.3	770.3	1538.3	3074.3
3	1	14.5	26.5	50.5	98.5	194.5	386.5	770.5	1538.5	3074.5
3	2	14.8	26.8	50.8	98.8	194.8	386.8	770.8	1538.8	3074.8
3	3	15.4	27.4	51.4	99.4	195.4	387.4	771.4	1539.4	3075.4
3	4	16.7	28.7	52.7	100.7	196.7	388.7	772.7	1540.7	3076.7
4	0	26.3	50.3	98.3	194.3	386.3	770.3	1538.3	3074.3	6146.3

LP	PB	LC=0	LC=1	LC=2	LC=3	LC=4	LC=5	LC=6	LC=7	LC=8
4	1	26.5	50.5	98.5	194.5	386.5	770.5	1538.5	3074.5	6146.5
4	2	26.8	50.8	98.8	194.8	386.8	770.8	1538.8	3074.8	6146.8
4	3	27.4	51.4	99.4	195.4	387.4	771.4	1539.4	3075.4	6147.4
4	4	28.7	52.7	100.7	196.7	388.7	772.7	1540.7	3076.7	6148.7

3.6.2 ROM usage

The size of code and read-only data of the decompressor if compiled for a Cortex-M4 CPU with the SEGGER compiler version 15.0.2 and balanced optimization is:

```
2274 Bytes
```

3.7 Code coverage application

The file `COMPRESS_LZMA_Coverage` has test vectors that will test both execution paths of all conditional statements when correctly configured. This will demonstrate code coverage with a suitable setup and test tool.

Please note that MISRA requires all switch clauses to have a ’default’ case, which may never be executed. This situations results in a code / branch coverage of less than 100%.

3.7.1 Specific setup

The application must be compiled with the following decode configuration:

```
#define COMPRESS_LZMA_CONFIG_DECODE_LC_MAX                3
#define COMPRESS_LZMA_CONFIG_DECODE_LP_MAX                0
#define COMPRESS_LZMA_CONFIG_DECODE_PB_MAX                2
#define COMPRESS_LZMA_CONFIG_DECODE_WINDOW_SIZE_MAX    1024
```

3.7.2 COMPRESS_LZMA_Coverage.c complete listing

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : COMPRESS_LZMA_Coverage.c
Purpose     : Ensure code coverage over all execution paths.

Notes       : This application must be compiled with the following
              configuration:

              COMPRESS_LZMA_CONFIG_DECODE_LC_MAX          = 3
              COMPRESS_LZMA_CONFIG_DECODE_LP_MAX          = 0
              COMPRESS_LZMA_CONFIG_DECODE_PB_MAX          = 2
              COMPRESS_LZMA_CONFIG_DECODE_WINDOW_SIZE_MAX = 1024
*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/

#include "COMPRESS_LZMA_DECODE.h"
#include <stdio.h>
#include <string.h>

/*********************************************************************
*
*       Configuration check
*
**********************************************************************
*/

#if COMPRESS_LZMA_CONFIG_DECODE_LC_MAX != 3
  #error COMPRESS_LZMA_CONFIG_DECODE_LC_MAX must be #defined to 3
#endif

#if COMPRESS_LZMA_CONFIG_DECODE_LP_MAX != 0
  #error COMPRESS_LZMA_CONFIG_DECODE_LP_MAX must be #defined to 0
#endif

#if COMPRESS_LZMA_CONFIG_DECODE_PB_MAX != 2
  #error COMPRESS_LZMA_CONFIG_DECODE_PB_MAX must be #defined to 2
#endif
```

```
#if COMPRESS_LZMA_CONFIG_DECODE_WINDOW_SIZE_MAX != 1024
  #error COMPRESS_LZMA_CONFIG_DECODE_WINDOW_SIZE_MAX must be #defined to 1024
#endif

#if !defined(WIN32) && !defined(linux)
  int MainTask(void);
  #define main MainTask
#endif

/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/

// Valid, complex bitstream.
static const U8 _aTest1[] = {
  0x5D, 0x00, 0x04, 0x00, 0x00, 0x47, 0x01, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x26, 0x16,
  0x85, 0xBC, 0x45, 0xF0, 0xDF, 0xFF, 0xD2, 0xE8,
  0x41, 0xF5, 0xCE, 0xE5, 0x90, 0xE1, 0xC8, 0x20,
  0xEA, 0xC6, 0x37, 0xBE, 0x2B, 0xD1, 0xF4, 0xC3,
  0x34, 0x6F, 0x2F, 0x83, 0xC2, 0xA6, 0x7C, 0x6F,
  0x3D, 0x88, 0xA0, 0x58, 0x22, 0x1F, 0x3A, 0xBA,
  0x7B, 0xC6, 0xDD, 0x66, 0xFE, 0xF8, 0x92, 0xE4,
  0xCB, 0x1C, 0xC4, 0x19, 0x0A, 0x0C, 0x8B, 0x2E,
  0x39, 0xB8, 0xB8, 0x03, 0xCD, 0x5A, 0x9E, 0x10,
  0x3A, 0x4F, 0x65, 0xFA, 0x41, 0xCB, 0xF2, 0x79,
  0x65, 0xD7, 0xF1, 0x9F, 0xAB, 0x70, 0x1D, 0x6F,
  0xF7, 0xB6, 0x79, 0xCC, 0x8A, 0x7D, 0xCE, 0xDB,
  0xF8, 0xF6, 0x9E, 0xC9, 0x12, 0x9F, 0xAA, 0xBF,
  0x89, 0xFE, 0x05, 0x36, 0x80, 0x00
};

// Always-invalid configuration byte
static const U8 _aTest2[] = {
  0xFF, 0x00, 0x04, 0x00, 0x00, 0x47, 0x01, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00
};

// Bad range coder start byte
static const U8 _aTest3[] = {
  0x00, 0x00, 0x04, 0x00, 0x00, 0x47, 0x01, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00,
  0x00, 0x00
};

// Bad range coder start code
static const U8 _aTest4[] = {
  0x00, 0x00, 0x04, 0x00, 0x00, 0x47, 0x01, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF,
  0xFF, 0xFF
};

// LC == 4, error as this must be compiled with COMPRESS_LZMA_CONFIG_DECODE_LC_MAX == 3
static const U8 _aTest5[] = {
  0x04, 0x00, 0x04, 0x00, 0x00, 0x47, 0x01, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF,
  0xFF, 0xFF
};

// LP == 1, error as this must be compiled with COMPRESS_LZMA_CONFIG_DECODE_LP_MAX == 0
static const U8 _aTest6[] = {
  0x09, 0x00, 0x04, 0x00, 0x00, 0x47, 0x01, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF,
  0xFF, 0xFF
};

// PB == 3, error as this must be compiled with COMPRESS_LZMA_CONFIG_DECODE_PB_MAX == 2
static const U8 _aTest7[] = {
  0x87, 0x00, 0x04, 0x00, 0x00, 0x47, 0x01, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF,
  0xFF, 0xFF
};
```

```
// WindowSize ==
 16MB, error as this must be compiled with COMPRESS_LZMA_CONFIG_DECODE_WINDOW_SIZE_MAX ==
 1024
static const U8 _aTest8[] = {
  0x00, 0x00, 0x00, 0x10, 0x00, 0x47, 0x01, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF,
  0xFF, 0xFF
};

// Bitstream with reference beyond dictionary.
static const U8 _aTest9[] = {
  0x5D, 0x20, 0x00, 0x00, 0x00, 0x47, 0x01, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x26, 0x16,
  0x85, 0xBC, 0x45, 0xF0, 0xDF, 0xFF, 0xD2, 0xE8,
  0x41, 0xF5, 0xCE, 0xE5, 0x90, 0xE1, 0xC8, 0x20,
  0xEA, 0xC6, 0x37, 0xBE, 0x2B, 0xD1, 0xF4, 0xC3,
  0x34, 0x6F, 0x2F, 0x83, 0xC2, 0xA6, 0x7C, 0x6F,
  0x3D, 0x88, 0xA0, 0x58, 0x22, 0x1F, 0x3A, 0xBA,
  0x7B, 0xC6, 0xDD, 0x66, 0xFE, 0xF8, 0x92, 0xE4,
  0xCB, 0x1C, 0xC4, 0x19, 0x0A, 0x0C, 0x8B, 0x2E,
  0x39, 0xB8, 0xB8, 0x03, 0xCD, 0x5A, 0x9E, 0x10,
  0x3A, 0x4F, 0x65, 0xFA, 0x41, 0xCB, 0xF2, 0x79,
  0x65, 0xD7, 0xF1, 0x9F, 0xAB, 0x70, 0x1D, 0x6F,
  0xF7, 0xB6, 0x79, 0xCC, 0x8A, 0x7D, 0xCE, 0xDB,
  0xF8, 0xF6, 0x9E, 0xC9, 0x12, 0x9F, 0xAA, 0xBF,
  0x89, 0xFE, 0x05, 0x36, 0x80, 0x00
};

// Bitstream with unexpected EOS.
static const U8 _aTest10[] = {
  0x5D, 0x00, 0x02, 0x00, 0x00, 0x00, 0x02, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x26, 0x16,
  0x85, 0xBC, 0x45, 0xF0, 0xDF, 0xFF, 0xD2, 0xE8,
  0x41, 0xF5, 0xCE, 0xE5, 0x90, 0xE1, 0xC8, 0x20,
  0xEA, 0xC6, 0x37, 0xBE, 0x2B, 0xD1, 0xF4, 0xC3,
  0x34, 0x6F, 0x2F, 0x83, 0xC2, 0xA6, 0x7C, 0x6F,
  0x3D, 0x88, 0xA0, 0x58, 0x22, 0x1F, 0x3A, 0xBA,
  0x7B, 0xC6, 0xDD, 0x66, 0xFE, 0xF8, 0x92, 0xE4,
  0xCB, 0x1C, 0xC4, 0x19, 0x0A, 0x0C, 0x8B, 0x2E,
  0x39, 0xB8, 0xB8, 0x03, 0xCD, 0x5A, 0x9E, 0x10,
  0x3A, 0x4F, 0x65, 0xFA, 0x41, 0xCB, 0xF2, 0x79,
  0x65, 0xD7, 0xF1, 0x9F, 0xAB, 0x70, 0x1D, 0x6F,
  0xF7, 0xB6, 0x79, 0xCC, 0x8A, 0x7D, 0xCE, 0xDB,
  0xF8, 0xF6, 0x9E, 0xC9, 0x12, 0x9F, 0xAA, 0xBF,
  0x8A, 0x08, 0xF5, 0x99, 0x8D, 0x7F, 0xFA, 0x18,
  0x0A, 0x52, 0x00
};

// Longer bitstream forcing buffer wraparound.
static const unsigned char _aTest11[] = {
  0x00, 0x00, 0x04, 0x00, 0x00, 0x41, 0x04, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x25, 0x19,
  0xD6, 0x94, 0xF9, 0xC9, 0x49, 0x9E, 0x37, 0xF7,
  0x30, 0xAB, 0x3E, 0x19, 0x3C, 0xE3, 0xAF, 0xDE,
  0xF1, 0x36, 0xDA, 0xCB, 0xF8, 0xDF, 0x86, 0x36,
  0xB1, 0x76, 0x13, 0x4C, 0xEA, 0x72, 0x6E, 0x17,
  0xC9, 0x21, 0x39, 0x10, 0x67, 0xEA, 0x6E, 0xDE,
  0xBC, 0xDF, 0xD6, 0x82, 0xE9, 0x6A, 0xE4, 0x4F,
  0xF7, 0x9D, 0x44, 0x93, 0x2D, 0x18, 0x55, 0x7F,
  0x85, 0xFA, 0x7F, 0xFB, 0x1F, 0xA7, 0x56, 0xEB,
  0xAE, 0x8A, 0xAA, 0x35, 0xAB, 0x6A, 0x13, 0x98,
  0xC1, 0x0A, 0x5E, 0x6E, 0x1F, 0xE7, 0x3C, 0x2E,
  0x60, 0x14, 0x05, 0x7F, 0x76, 0x43, 0x15, 0x93,
  0x68, 0x98, 0xE3, 0x4F, 0x9E, 0x9C, 0x58, 0xC3,
  0x0C, 0xCB, 0x14, 0x23, 0xCE, 0x97, 0x39, 0x9B,
  0x09, 0xFD, 0xD6, 0xC0, 0x14, 0xE2, 0xC4, 0x32,
  0xA4, 0x1E, 0xE8, 0x6A, 0xA8, 0x90, 0xA4, 0x7D,
  0x38, 0xCE, 0x50, 0x0D, 0x72, 0xCB, 0xD1, 0x8F,
  0x67, 0x24, 0x79, 0x9B, 0xE0, 0x32, 0x7D, 0x5B,
  0xF4, 0xEC, 0x93, 0xA3, 0x9C, 0xCC, 0x82, 0x00,
  0xE9, 0xD0, 0x0F, 0xFC, 0x0C, 0x7A, 0x46, 0xAC,
  0x0A, 0x11, 0x31, 0x12, 0x65, 0x4B, 0xC2, 0x19,
  0x08, 0x96, 0x9A, 0x46, 0xC3, 0xBB, 0x46, 0xEE,
  0xC4, 0x7E, 0x1A, 0x69, 0xBB, 0x97, 0x03, 0x2D,
  0x0F, 0x71, 0xE7, 0x9E, 0x3F, 0xC4, 0x14, 0x19,
  0x81, 0xB3, 0x1A, 0xF9, 0x50, 0xDF, 0xF1, 0x49,
  0xEA, 0xCE, 0xB6, 0xF3, 0xF8, 0xBB, 0x2F, 0xC4,
  0xC6, 0xB2, 0x71, 0xB6, 0x5E, 0x85, 0xE1, 0xB8,
  0x95, 0xB0, 0xEF, 0x58, 0x34, 0xC2, 0x65, 0xEB,
  0xE8, 0x16, 0xD0, 0x82, 0x84, 0x8B, 0xC2, 0x36,
  0x62, 0x45, 0xE5, 0x3D, 0x9E, 0xE2, 0xCE, 0xBA,
  0xB5, 0xA5, 0x71, 0xC4, 0x17, 0x63, 0x96, 0x0B,
  0x86, 0xE2, 0xF9, 0x34, 0x8F, 0x16, 0x53, 0x00,
  0x0F, 0x7A, 0xCA, 0x38, 0xA0, 0x3C, 0x24, 0x2C,
  0x39, 0xE9, 0xFA, 0x2E, 0x66, 0xAF, 0x7E, 0xF2,
  0x29, 0x11, 0x95, 0xE0, 0xCB, 0xFF, 0x04, 0xA5,
  0x89, 0xDB, 0xC6, 0xB9, 0x7B, 0x84, 0xF2, 0xEA,
  0x24, 0xC3, 0x5F, 0x5D, 0x2B, 0x03, 0x97, 0xB7,
  0xFF, 0x5F, 0x7F, 0x26, 0xF1, 0x74, 0x89, 0x26,
  0x51, 0x72, 0x78, 0x01, 0xF0, 0x3B, 0x32, 0xCA,
  0x11, 0x5D, 0xC2, 0x2D, 0x83, 0xF7, 0x84, 0xD1,
  0x70, 0x9B, 0x11, 0x67, 0x99, 0xF5, 0xA5, 0xA6,
  0xFC, 0xCF, 0xAB, 0x71, 0x09, 0x85, 0xD7, 0x16,
  0x0D, 0xE3, 0x12, 0x66, 0x44, 0xC6, 0x52, 0x5D,
  0x8E, 0xE8, 0x60, 0xCF, 0x7F, 0x50, 0xDE, 0xE0,
  0xF6, 0x22, 0x39, 0x10, 0x59, 0x75, 0x33, 0x02,
  0x64, 0xA3, 0x27, 0xE3, 0x3F, 0x4F, 0xB8, 0x84,
  0xDE, 0x06, 0x89, 0x31, 0x24, 0x1E, 0x27, 0x03,
  0xB8, 0xD0, 0xDD, 0x5B, 0xCD, 0x7C, 0xF2, 0xE3,
  0x0A, 0x10, 0x7B, 0x54, 0x92, 0x3E, 0xD1, 0x2C,
  0xFA, 0xB5, 0xCC, 0xAE, 0x6A, 0x6D, 0x9E, 0xC7,
  0x64, 0x5E, 0xA7, 0xC4, 0x34, 0x2F, 0x69, 0x37,
  0xA9, 0x48, 0x6C, 0xF0, 0xB4, 0xCE, 0xEC, 0xFA,
  0xC0, 0xB2, 0x0C, 0xF9, 0x6A, 0x1C, 0xB5, 0xE5,
  0x79, 0x8B, 0x20, 0xBE, 0x1F, 0xD6, 0x2F, 0xAA,
  0xAC, 0xC4, 0xE1, 0x5D, 0x12, 0xF5, 0x70, 0x9A,
  0x43, 0xAF, 0x3C, 0x18, 0x58, 0x21, 0x8C, 0x55,
  0xB7, 0x06, 0xB4, 0xC5, 0xA0, 0x0B, 0x93, 0xDD,
  0xB3, 0xF1, 0xF3, 0x55, 0x2F, 0xB2, 0x04, 0xB8,
  0xA8, 0xDC, 0xEB, 0x35, 0x6D, 0x55, 0x9D, 0xA3,
  0x68, 0x96, 0xD6, 0x20, 0x7C, 0xC6, 0x79, 0x5B,
  0x1F, 0xE8, 0xD6, 0x0E, 0x72, 0xB2, 0xA2, 0xEF,
  0xFB, 0xD3, 0xE9, 0xE0, 0xD8, 0x1E, 0xB0, 0x07,
  0x7C, 0x68, 0x06, 0x0E, 0xD5, 0xDF, 0x31, 0xCE,
  0xAA, 0x4B, 0x66, 0xD4, 0x7B, 0x25, 0x88, 0x05,
  0x8A, 0xFA, 0x10, 0x60, 0xC4, 0x17, 0xC5, 0x97,
  0xC3, 0xE9, 0xBE, 0x14, 0xC3, 0xAF, 0x23, 0x00,
  0xC8, 0x1D, 0x1F, 0xBB, 0xFF, 0x96, 0x5A, 0xAC,
  0xF3, 0x02, 0xC3, 0xD1, 0x40, 0x3D, 0x58, 0x37,
  0xC3, 0x7A, 0xE9, 0xD9, 0x3F, 0x5F, 0xCC, 0x70,
  0x80, 0x9D, 0xAE, 0xD8, 0x1F, 0xEA, 0xAE, 0x9A,
  0x38, 0xD3, 0x7B, 0xDC, 0x84, 0x9B, 0x3E, 0x51,
  0xD7, 0x3B, 0x1A, 0xAB, 0x94, 0x64, 0x4E, 0xD9,
  0x31, 0x8F, 0xFA, 0x45, 0x03, 0x14, 0xB5, 0x0C,
  0x9F, 0x3E, 0x71, 0xC4, 0x41, 0x5E, 0x22, 0x2B,
  0x8F, 0xCB, 0xB0, 0x21, 0x45, 0xD3, 0x8A, 0x01,
  0xB4, 0x46, 0x63, 0x73, 0x46, 0x26, 0xA0, 0xA5,
  0x4B, 0xEE, 0x7B, 0xFC, 0xD0, 0xE2, 0x06, 0xFD,
  0xFF, 0x18, 0xBB, 0x87, 0xD3, 0xD7, 0xC1
};

static const U8 _aTest1Original[] = {
  0x4C, 0x5A, 0x4D, 0x41, 0x20, 0x64, 0x65, 0x63,
  0x6F, 0x64, 0x65, 0x72, 0x20, 0x74, 0x65, 0x73,
  0x74, 0x20, 0x65, 0x78, 0x61, 0x6D, 0x70, 0x6C,
  0x65, 0x0D, 0x0A, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x0D, 0x0A, 0x21, 0x20,
  0x4C, 0x5A, 0x4D, 0x41, 0x20, 0x21, 0x20, 0x44,
  0x65, 0x63, 0x6F, 0x64, 0x65, 0x72, 0x20, 0x21,
  0x20, 0x54, 0x45, 0x53, 0x54, 0x20, 0x21, 0x0D,
  0x0A, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x0D, 0x0A, 0x21, 0x20, 0x54, 0x45,
  0x53, 0x54, 0x20, 0x21, 0x20, 0x4C, 0x5A, 0x4D,
  0x41, 0x20, 0x21, 0x20, 0x44, 0x65, 0x63, 0x6F,
  0x64, 0x65, 0x72, 0x20, 0x21, 0x0D, 0x0A, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x0D, 0x0A, 0x2D, 0x2D, 0x2D, 0x2D, 0x20, 0x54,
  0x65, 0x73, 0x74, 0x20, 0x4C, 0x69, 0x6E, 0x65,
  0x20, 0x31, 0x20, 0x2D, 0x2D, 0x2D, 0x2D, 0x2D,
  0x2D, 0x2D, 0x2D, 0x20, 0x0D, 0x0A, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x0D,
  0x0A, 0x2D, 0x2D, 0x2D, 0x2D, 0x20, 0x54, 0x65,
  0x73, 0x74, 0x20, 0x4C, 0x69, 0x6E, 0x65, 0x20,
  0x32, 0x20, 0x2D, 0x2D, 0x2D, 0x2D, 0x2D, 0x2D,
  0x2D, 0x2D, 0x20, 0x0D, 0x0A, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x0D, 0x0A,
  0x3D, 0x3D, 0x3D, 0x20, 0x45, 0x6E, 0x64, 0x20,
  0x6F, 0x66, 0x20, 0x74, 0x65, 0x73, 0x74, 0x20,
  0x66, 0x69, 0x6C, 0x65, 0x20, 0x3D, 0x3D, 0x3D,
  0x3D, 0x20, 0x0D, 0x0A, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x3D,
  0x3D, 0x3D, 0x3D, 0x3D, 0x3D, 0x0D, 0x0A
};

static const U8 _aTest11Original[] = {
  0x4A, 0x61, 0x62, 0x62, 0x65, 0x72, 0x77, 0x6F,
  0x63, 0x6B, 0x79, 0x0D, 0x0A, 0x20, 0x20, 0x42,
  0x59, 0x20, 0x4C, 0x45, 0x57, 0x49, 0x53, 0x20,
  0x43, 0x41, 0x52, 0x52, 0x4F, 0x4C, 0x4C, 0x0D,
  0x0A, 0x0D, 0x0A, 0x0D, 0x0A, 0x27, 0x54, 0x77,
  0x61, 0x73, 0x20, 0x62, 0x72, 0x69, 0x6C, 0x6C,
  0x69, 0x67, 0x2C, 0x20, 0x61, 0x6E, 0x64, 0x20,
  0x74, 0x68, 0x65, 0x20, 0x73, 0x6C, 0x69, 0x74,
  0x68, 0x79, 0x20, 0x74, 0x6F, 0x76, 0x65, 0x73,
  0x0D, 0x0A, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
  0x44, 0x69, 0x64, 0x20, 0x67, 0x79, 0x72, 0x65,
  0x20, 0x61, 0x6E, 0x64, 0x20, 0x67, 0x69, 0x6D,
  0x62, 0x6C, 0x65, 0x20, 0x69, 0x6E, 0x20, 0x74,
  0x68, 0x65, 0x20, 0x77, 0x61, 0x62, 0x65, 0x3A,
  0x0D, 0x0A, 0x41, 0x6C, 0x6C, 0x20, 0x6D, 0x69,
  0x6D, 0x73, 0x79, 0x20, 0x77, 0x65, 0x72, 0x65,
  0x20, 0x74, 0x68, 0x65, 0x20, 0x62, 0x6F, 0x72,
  0x6F, 0x67, 0x6F, 0x76, 0x65, 0x73, 0x2C, 0x0D,
  0x0A, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x41,
  0x6E, 0x64, 0x20, 0x74, 0x68, 0x65, 0x20, 0x6D,
  0x6F, 0x6D, 0x65, 0x20, 0x72, 0x61, 0x74, 0x68,
  0x73, 0x20, 0x6F, 0x75, 0x74, 0x67, 0x72, 0x61,
  0x62, 0x65, 0x2E, 0x0D, 0x0A, 0x0D, 0x0A, 0x22,
  0x42, 0x65, 0x77, 0x61, 0x72, 0x65, 0x20, 0x74,
  0x68, 0x65, 0x20, 0x4A, 0x61, 0x62, 0x62, 0x65,
  0x72, 0x77, 0x6F, 0x63, 0x6B, 0x2C, 0x20, 0x6D,
  0x79, 0x20, 0x73, 0x6F, 0x6E, 0x21, 0x0D, 0x0A,
  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x54, 0x68,
  0x65, 0x20, 0x6A, 0x61, 0x77, 0x73, 0x20, 0x74,
  0x68, 0x61, 0x74, 0x20, 0x62, 0x69, 0x74, 0x65,
  0x2C, 0x20, 0x74, 0x68, 0x65, 0x20, 0x63, 0x6C,
  0x61, 0x77, 0x73, 0x20, 0x74, 0x68, 0x61, 0x74,
  0x20, 0x63, 0x61, 0x74, 0x63, 0x68, 0x21, 0x0D,
  0x0A, 0x42, 0x65, 0x77, 0x61, 0x72, 0x65, 0x20,
  0x74, 0x68, 0x65, 0x20, 0x4A, 0x75, 0x62, 0x6A,
  0x75, 0x62, 0x20, 0x62, 0x69, 0x72, 0x64, 0x2C,
  0x20, 0x61, 0x6E, 0x64, 0x20, 0x73, 0x68, 0x75,
  0x6E, 0x0D, 0x0A, 0x20, 0x20, 0x20, 0x20, 0x20,
  0x20, 0x54, 0x68, 0x65, 0x20, 0x66, 0x72, 0x75,
  0x6D, 0x69, 0x6F, 0x75, 0x73, 0x20, 0x42, 0x61,
  0x6E, 0x64, 0x65, 0x72, 0x73, 0x6E, 0x61, 0x74,
  0x63, 0x68, 0x21, 0x22, 0x0D, 0x0A, 0x0D, 0x0A,
  0x48, 0x65, 0x20, 0x74, 0x6F, 0x6F, 0x6B, 0x20,
  0x68, 0x69, 0x73, 0x20, 0x76, 0x6F, 0x72, 0x70,
  0x61, 0x6C, 0x20, 0x73, 0x77, 0x6F, 0x72, 0x64,
  0x20, 0x69, 0x6E, 0x20, 0x68, 0x61, 0x6E, 0x64,
  0x3B, 0x0D, 0x0A, 0x20, 0x20, 0x20, 0x20, 0x20,
  0x20, 0x4C, 0x6F, 0x6E, 0x67, 0x20, 0x74, 0x69,
  0x6D, 0x65, 0x20, 0x74, 0x68, 0x65, 0x20, 0x6D,
  0x61, 0x6E, 0x78, 0x6F, 0x6D, 0x65, 0x20, 0x66,
  0x6F, 0x65, 0x20, 0x68, 0x65, 0x20, 0x73, 0x6F,
  0x75, 0x67, 0x68, 0x74, 0x97, 0x0D, 0x0A, 0x53,
  0x6F, 0x20, 0x72, 0x65, 0x73, 0x74, 0x65, 0x64,
  0x20, 0x68, 0x65, 0x20, 0x62, 0x79, 0x20, 0x74,
  0x68, 0x65, 0x20, 0x54, 0x75, 0x6D, 0x74, 0x75,
  0x6D, 0x20, 0x74, 0x72, 0x65, 0x65, 0x0D, 0x0A,
  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x41, 0x6E,
  0x64, 0x20, 0x73, 0x74, 0x6F, 0x6F, 0x64, 0x20,
  0x61, 0x77, 0x68, 0x69, 0x6C, 0x65, 0x20, 0x69,
  0x6E, 0x20, 0x74, 0x68, 0x6F, 0x75, 0x67, 0x68,
  0x74, 0x2E, 0x0D, 0x0A, 0x0D, 0x0A, 0x41, 0x6E,
  0x64, 0x2C, 0x20, 0x61, 0x73, 0x20, 0x69, 0x6E,
  0x20, 0x75, 0x66, 0x66, 0x69, 0x73, 0x68, 0x20,
  0x74, 0x68, 0x6F, 0x75, 0x67, 0x68, 0x74, 0x20,
  0x68, 0x65, 0x20, 0x73, 0x74, 0x6F, 0x6F, 0x64,
  0x2C, 0x0D, 0x0A, 0x20, 0x20, 0x20, 0x20, 0x20,
  0x20, 0x54, 0x68, 0x65, 0x20, 0x4A, 0x61, 0x62,
  0x62, 0x65, 0x72, 0x77, 0x6F, 0x63, 0x6B, 0x2C,
  0x20, 0x77, 0x69, 0x74, 0x68, 0x20, 0x65, 0x79,
  0x65, 0x73, 0x20, 0x6F, 0x66, 0x20, 0x66, 0x6C,
  0x61, 0x6D, 0x65, 0x2C, 0x0D, 0x0A, 0x43, 0x61,
  0x6D, 0x65, 0x20, 0x77, 0x68, 0x69, 0x66, 0x66,
  0x6C, 0x69, 0x6E, 0x67, 0x20, 0x74, 0x68, 0x72,
  0x6F, 0x75, 0x67, 0x68, 0x20, 0x74, 0x68, 0x65,
  0x20, 0x74, 0x75, 0x6C, 0x67, 0x65, 0x79, 0x20,
  0x77, 0x6F, 0x6F, 0x64, 0x2C, 0x0D, 0x0A, 0x20,
  0x20, 0x20, 0x20, 0x20, 0x20, 0x41, 0x6E, 0x64,
  0x20, 0x62, 0x75, 0x72, 0x62, 0x6C, 0x65, 0x64,
  0x20, 0x61, 0x73, 0x20, 0x69, 0x74, 0x20, 0x63,
  0x61, 0x6D, 0x65, 0x21, 0x0D, 0x0A, 0x0D, 0x0A,
  0x4F, 0x6E, 0x65, 0x2C, 0x20, 0x74, 0x77, 0x6F,
  0x21, 0x20, 0x4F, 0x6E, 0x65, 0x2C, 0x20, 0x74,
  0x77, 0x6F, 0x21, 0x20, 0x41, 0x6E, 0x64, 0x20,
  0x74, 0x68, 0x72, 0x6F, 0x75, 0x67, 0x68, 0x20,
  0x61, 0x6E, 0x64, 0x20, 0x74, 0x68, 0x72, 0x6F,
  0x75, 0x67, 0x68, 0x0D, 0x0A, 0x20, 0x20, 0x20,
  0x20, 0x20, 0x20, 0x54, 0x68, 0x65, 0x20, 0x76,
  0x6F, 0x72, 0x70, 0x61, 0x6C, 0x20, 0x62, 0x6C,
  0x61, 0x64, 0x65, 0x20, 0x77, 0x65, 0x6E, 0x74,
  0x20, 0x73, 0x6E, 0x69, 0x63, 0x6B, 0x65, 0x72,
  0x2D, 0x73, 0x6E, 0x61, 0x63, 0x6B, 0x21, 0x0D,
  0x0A, 0x48, 0x65, 0x20, 0x6C, 0x65, 0x66, 0x74,
  0x20, 0x69, 0x74, 0x20, 0x64, 0x65, 0x61, 0x64,
  0x2C, 0x20, 0x61, 0x6E, 0x64, 0x20, 0x77, 0x69,
  0x74, 0x68, 0x20, 0x69, 0x74, 0x73, 0x20, 0x68,
  0x65, 0x61, 0x64, 0x0D, 0x0A, 0x20, 0x20, 0x20,
  0x20, 0x20, 0x20, 0x48, 0x65, 0x20, 0x77, 0x65,
  0x6E, 0x74, 0x20, 0x67, 0x61, 0x6C, 0x75, 0x6D,
  0x70, 0x68, 0x69, 0x6E, 0x67, 0x20, 0x62, 0x61,
  0x63, 0x6B, 0x2E, 0x0D, 0x0A, 0x0D, 0x0A, 0x22,
  0x41, 0x6E, 0x64, 0x20, 0x68, 0x61, 0x73, 0x74,
  0x20, 0x74, 0x68, 0x6F, 0x75, 0x20, 0x73, 0x6C,
  0x61, 0x69, 0x6E, 0x20, 0x74, 0x68, 0x65, 0x20,
  0x4A, 0x61, 0x62, 0x62, 0x65, 0x72, 0x77, 0x6F,
  0x63, 0x6B, 0x3F, 0x0D, 0x0A, 0x20, 0x20, 0x20,
  0x20, 0x20, 0x20, 0x43, 0x6F, 0x6D, 0x65, 0x20,
  0x74, 0x6F, 0x20, 0x6D, 0x79, 0x20, 0x61, 0x72,
  0x6D, 0x73, 0x2C, 0x20, 0x6D, 0x79, 0x20, 0x62,
  0x65, 0x61, 0x6D, 0x69, 0x73, 0x68, 0x20, 0x62,
  0x6F, 0x79, 0x21, 0x0D, 0x0A, 0x4F, 0x20, 0x66,
  0x72, 0x61, 0x62, 0x6A, 0x6F, 0x75, 0x73, 0x20,
  0x64, 0x61, 0x79, 0x21, 0x20, 0x43, 0x61, 0x6C,
  0x6C, 0x6F, 0x6F, 0x68, 0x21, 0x20, 0x43, 0x61,
  0x6C, 0x6C, 0x61, 0x79, 0x21, 0x22, 0x0D, 0x0A,
  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x48, 0x65,
  0x20, 0x63, 0x68, 0x6F, 0x72, 0x74, 0x6C, 0x65,
  0x64, 0x20, 0x69, 0x6E, 0x20, 0x68, 0x69, 0x73,
  0x20, 0x6A, 0x6F, 0x79, 0x2E, 0x0D, 0x0A, 0x0D,
  0x0A, 0x27, 0x54, 0x77, 0x61, 0x73, 0x20, 0x62,
  0x72, 0x69, 0x6C, 0x6C, 0x69, 0x67, 0x2C, 0x20,
  0x61, 0x6E, 0x64, 0x20, 0x74, 0x68, 0x65, 0x20,
  0x73, 0x6C, 0x69, 0x74, 0x68, 0x79, 0x20, 0x74,
  0x6F, 0x76, 0x65, 0x73, 0x0D, 0x0A, 0x20, 0x20,
  0x20, 0x20, 0x20, 0x20, 0x44, 0x69, 0x64, 0x20,
  0x67, 0x79, 0x72, 0x65, 0x20, 0x61, 0x6E, 0x64,
  0x20, 0x67, 0x69, 0x6D, 0x62, 0x6C, 0x65, 0x20,
  0x69, 0x6E, 0x20, 0x74, 0x68, 0x65, 0x20, 0x77,
  0x61, 0x62, 0x65, 0x3A, 0x0D, 0x0A, 0x41, 0x6C,
  0x6C, 0x20, 0x6D, 0x69, 0x6D, 0x73, 0x79, 0x20,
  0x77, 0x65, 0x72, 0x65, 0x20, 0x74, 0x68, 0x65,
  0x20, 0x62, 0x6F, 0x72, 0x6F, 0x67, 0x6F, 0x76,
  0x65, 0x73, 0x2C, 0x0D, 0x0A, 0x20, 0x20, 0x20,
  0x20, 0x20, 0x20, 0x41, 0x6E, 0x64, 0x20, 0x74,
  0x68, 0x65, 0x20, 0x6D, 0x6F, 0x6D, 0x65, 0x20,
  0x72, 0x61, 0x74, 0x68, 0x73, 0x20, 0x6F, 0x75,
  0x74, 0x67, 0x72, 0x61, 0x62, 0x65, 0x2E, 0x0D,
  0x0A
};

// Wrong DecodedSize
static const U8 _aTest12[] = {
  0x5D, 0x00, 0x04, 0x00, 0x00, 0x47, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x26, 0x16,
  0x85, 0xBC, 0x45, 0xF0, 0xDF, 0xFF, 0xD2, 0xE8,
  0x41, 0xF5, 0xCE, 0xE5, 0x90, 0xE1, 0xC8, 0x20,
  0xEA, 0xC6, 0x37, 0xBE, 0x2B, 0xD1, 0xF4, 0xC3,
  0x34, 0x6F, 0x2F, 0x83, 0xC2, 0xA6, 0x7C, 0x6F,
  0x3D, 0x88, 0xA0, 0x58, 0x22, 0x1F, 0x3A, 0xBA,
  0x7B, 0xC6, 0xDD, 0x66, 0xFE, 0xF8, 0x92, 0xE4,
  0xCB, 0x1C, 0xC4, 0x19, 0x0A, 0x0C, 0x8B, 0x2E,
  0x39, 0xB8, 0xB8, 0x03, 0xCD, 0x5A, 0x9E, 0x10,
  0x3A, 0x4F, 0x65, 0xFA, 0x41, 0xCB, 0xF2, 0x79,
  0x65, 0xD7, 0xF1, 0x9F, 0xAB, 0x70, 0x1D, 0x6F,
  0xF7, 0xB6, 0x79, 0xCC, 0x8A, 0x7D, 0xCE, 0xDB,
  0xF8, 0xF6, 0x9E, 0xC9, 0x12, 0x9F, 0xAA, 0xBF,
  0x89, 0xFE, 0x05, 0x36, 0x80, 0x00
};

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static COMPRESS_LZMA_DECODE_CONTEXT _Decoder;
static U8                           _aOutBuffer[1100];

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _DecodeOnce()
*
*  Function description
*    Decode bitstream in a single call.
*
*  Parameters
*    pData   - Pointer to compressed bitstream octet string.
*    DataLen - Octet length of compressed bitstream octet string.
*
*  Return value
*    < 0  - Error in one or more parameters.
*    >= 0 - Success.
*/
static int _DecodeOnce(const U8 *pData, unsigned DataLen) {
  COMPRESS_LZMA_STREAM Stream;
  int                  Status;
  //
  Stream.AvailIn  = DataLen;
  Stream.pIn      = pData;
  Stream.AvailOut = sizeof(_aOutBuffer);
  Stream.pOut     = _aOutBuffer;
  //
  COMPRESS_LZMA_DECODE_Init(&_Decoder);
  Status = COMPRESS_LZMA_DECODE_Run(&_Decoder, &Stream, 1);
  //
  return Status;
```

```
}

/*********************************************************************
*
*       _DecodeVerify()
*
*  Function description
*    Decode bitstream and verify.
*
*  Parameters
*    pData   - Pointer to compressed bitstream octet string.
*    DataLen - Octet length of compressed bitstream octet string.
*    pOrig   - Pointer to expected original octet string.
*    OrigLen - Octet length of original octet string.
*
*  Return value
*    < 0  - Error in one or more parameters.
*    >= 0 - Success.
*/
static int _DecodeVerify(const U8 *pData, unsigned DataLen, const U8 *pOrig, unsigned OrigLen) {
  COMPRESS_LZMA_STREAM Stream;
  int                  Status;
  //
  Stream.AvailIn  = DataLen;
  Stream.pIn      = pData;
  Stream.AvailOut = sizeof(_aOutBuffer);
  Stream.pOut     = _aOutBuffer;
  //
  COMPRESS_LZMA_DECODE_Init(&_Decoder);
  Status = COMPRESS_LZMA_DECODE_Run(&_Decoder, &Stream, 1);
  //
  if (Status < 0) {
    return Status;
  }
  if (OrigLen != sizeof(_aOutBuffer) - Stream.AvailOut) {
    return -1;
  }
  if (memcmp(_aOutBuffer, pOrig, OrigLen) != 0) {
    return -1;
  }
  return 0;
}

/*********************************************************************
*
*       _DecodeStream()
*
*  Function description
*    Decode bitstream using streaming interface.
*
*  Parameters
*    pData   - Pointer to compressed bitstream octet string.
*    DataLen - Octet length of compressed bitstream octet string.
*
*  Return value
*    < 0  - Error in one or more parameters.
*    >= 0 - Success.
*/
static int _DecodeStream(const U8 *pData, unsigned DataLen) {
  COMPRESS_LZMA_STREAM Stream;
  int                  Status;
  int                  N;
  //
  COMPRESS_LZMA_DECODE_Init(&_Decoder);
  N = DataLen;
  Status = 0;
  Stream.pIn = pData;
  while (Status == 0 && N >= 0) {
    Stream.AvailIn = 1;
    while (Status == 0 && (Stream.AvailIn > 0 || N == 0)) {
      Stream.pOut     = &_aOutBuffer[0];
      Stream.AvailOut = 1;
      Status = COMPRESS_LZMA_DECODE_Run(&_Decoder, &Stream, N == 0);
    }
    --N;
  }
  //
  return Status;
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       MainTask()
*
*  Function description
*    Application entry point.
*/
int main(void) {
  COMPRESS_LZMA_STREAM Stream;
  int                  Passed;
  //
  Passed  = 1;
  //
  // Valid output, more input required but none available without flush.
  //
  COMPRESS_LZMA_DECODE_Init(&_Decoder);
  Stream.pIn      = &_aTest1[0];
  Stream.AvailIn  = 0;
  Stream.pOut     = &_aOutBuffer[0];
  Stream.AvailOut = sizeof(_aOutBuffer);
  COMPRESS_LZMA_DECODE_Init(&_Decoder);
  Passed &= COMPRESS_LZMA_DECODE_Run(&_Decoder, &Stream, 0) == COMPRESS_LZMA_STATUS_USAGE_ERROR;
  //
  // Valid output, more input required but null buffer.
  //
  COMPRESS_LZMA_DECODE_Init(&_Decoder);
  Stream.pIn      = 0;
  Stream.AvailIn  = 1;
  Stream.pOut     = &_aOutBuffer[0];
  Stream.AvailOut = sizeof(_aOutBuffer);
  COMPRESS_LZMA_DECODE_Init(&_Decoder);
  Passed &= COMPRESS_LZMA_DECODE_Run(&_Decoder, &Stream, 0) == COMPRESS_LZMA_STATUS_USAGE_ERROR;
  //
  // Valid input, but no output space
  //
  Stream.pIn      = &_aTest1[0];
  Stream.AvailIn  = sizeof(_aTest1);
  Stream.pOut     = &_aOutBuffer[0];
  Stream.AvailOut = 0;
  COMPRESS_LZMA_DECODE_Init(&_Decoder);
  Passed &= COMPRESS_LZMA_DECODE_Run(&_Decoder, &Stream, 0) == COMPRESS_LZMA_STATUS_USAGE_ERROR;
  //
  // Valid input, but null output buffer
  //
  Stream.pIn      = &_aTest1[0];
  Stream.AvailIn  = sizeof(_aTest1);
  Stream.pOut     = 0;
  Stream.AvailOut = sizeof(_aOutBuffer);
  COMPRESS_LZMA_DECODE_Init(&_Decoder);
  Passed &= COMPRESS_LZMA_DECODE_Run(&_Decoder, &Stream, 0) == COMPRESS_LZMA_STATUS_USAGE_ERROR;
  //
  Passed &= _DecodeOnce  (_aTest1,  sizeof(_aTest1))  == 1;
  Passed &= _DecodeOnce  (_aTest2,  sizeof(_aTest2))
  == COMPRESS_LZMA_STATUS_PARAMETER_ERROR;
  Passed &= _DecodeStream(_aTest1,  sizeof(_aTest1))  == 1;
  Passed &= _DecodeStream(_aTest1,  1)
  == COMPRESS_LZMA_STATUS_BITSTREAM_ERROR;  // Ensure truncated header detected
  Passed &= _DecodeStream(_aTest1,  12)
  == COMPRESS_LZMA_STATUS_BITSTREAM_ERROR;
  Passed &= _DecodeStream(_aTest1,  13)
  == COMPRESS_LZMA_STATUS_BITSTREAM_ERROR;
  Passed &= _DecodeStream(_aTest1,  14)
  == COMPRESS_LZMA_STATUS_BITSTREAM_ERROR;
  Passed &= _DecodeStream(_aTest1,  18)
  == COMPRESS_LZMA_STATUS_BITSTREAM_ERROR;
  Passed &= _DecodeStream(_aTest1,  19)
  == COMPRESS_LZMA_STATUS_BITSTREAM_ERROR;
  Passed &= _DecodeOnce  (_aTest1,  64)
  == COMPRESS_LZMA_STATUS_BITSTREAM_ERROR;  // Trigger exhaustion of input whilst decoding
  Passed &= _DecodeStream(_aTest2,  sizeof(_aTest2))
  == COMPRESS_LZMA_STATUS_PARAMETER_ERROR;
  Passed &= _DecodeStream(_aTest3,  sizeof(_aTest3))
  == COMPRESS_LZMA_STATUS_BITSTREAM_ERROR;
  Passed &= _DecodeStream(_aTest4,  sizeof(_aTest4))
  == COMPRESS_LZMA_STATUS_BITSTREAM_ERROR;
  Passed &= _DecodeStream(_aTest5,  sizeof(_aTest5))
  == COMPRESS_LZMA_STATUS_PARAMETER_ERROR;
  Passed &= _DecodeStream(_aTest6,  sizeof(_aTest6))
  == COMPRESS_LZMA_STATUS_PARAMETER_ERROR;
  Passed &= _DecodeStream(_aTest7,  sizeof(_aTest7))
  == COMPRESS_LZMA_STATUS_PARAMETER_ERROR;
  Passed &= _DecodeStream(_aTest8,  sizeof(_aTest8))
  == COMPRESS_LZMA_STATUS_PARAMETER_ERROR;
  Passed &= _DecodeStream(_aTest9,  sizeof(_aTest9))
  == COMPRESS_LZMA_STATUS_BITSTREAM_ERROR;
  Passed &= _DecodeStream(_aTest10, sizeof(_aTest10)) == COMPRESS_LZMA_STATUS_BITSTREAM_ERROR;
  Passed &= _DecodeStream(_aTest11, sizeof(_aTest11)) == 1;
  Passed &= _DecodeVerify(_aTest1,
  sizeof(_aTest1), _aTest1Original, sizeof(_aTest1Original))  == 0;
  // Verify decoded output is correct.
  Passed &= _DecodeVerify(_aTest11, sizeof(_aTest11), _aTest11Original, sizeof(_aTest11Original))
  == 0;  // Verify decoded output is correct.
  Passed &= _DecodeStream(_aTest12, sizeof(_aTest12)) == COMPRESS_LZMA_STATUS_BITSTREAM_ERROR;
  //
  if (Passed) {
    printf("All test vectors passed\n");
    return 0;
  }
  printf("One or more test vectors failed\n");
  return 1;
}

/*************************** End of file ****************************/
```

3.8 Compressor-decompressor verification application

The file `COMPRESS_LZMA_Verify` will compress and decompress multiple files using all valid compression configurations for LC, LP, and PB. The command line options are similar to the sample full-feature application and allow setting the window sizes and match lengths.

3.8.1 COMPRESS_LZMA_Verify.c complete listing

```
/*********************************************************************
*               (c) SEGGER Microcontroller GmbH & Co. KG             *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : COMPRESS_LZMA_Verify.c
Purpose     : Run compress-decompress cycle and validate output.

*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/

#include "COMPRESS_LZMA_ENCODE.h"
#include "COMPRESS_LZMA_DECODE.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/

#define BUFFER_SIZE         (101*1024*1024)  // 101 MB

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static COMPRESS_LZMA_ENCODE_CONTEXT _Encoder;
static COMPRESS_LZMA_DECODE_CONTEXT _Decoder;
static U8                           _aFileBuf   [BUFFER_SIZE];
static U8                           _aEncodedBuf[BUFFER_SIZE];
static U8                           _aDecodedBuf[BUFFER_SIZE];

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _GetStatusText()
*
*  Function description
*    Decode emCompress-LZMA status code.
*
*  Parameters
*    Status - Status code.
*
*  Return value
*    Non-zero pointer to status description.
*/
static const char * _GetStatusText(int Status) {
  switch (Status) {
  case 0:                                  return "Still processing";
  case 1:                                  return "Complete";
  case COMPRESS_LZMA_STATUS_PARAMETER_ERROR: return "Parameter error";
  case COMPRESS_LZMA_STATUS_BITSTREAM_ERROR: return "Bitstream error";
  case COMPRESS_LZMA_STATUS_USAGE_ERROR:     return "Usage error";
  default:                                 return "Unknown error";
  }
}

/*********************************************************************
*
*       _Verify()
*
*  Function description
*    Verify a compress-decompress cycle works.
*
*  Parameters
*    pParas  - Pointer to encoder parameters.
*    pInFile - Pointer to input file at position 0.
*
*  Return value
*    0 - test passed.
*    1 - test failed.
*/
static int _Verify(const COMPRESS_LZMA_PARAS *pParas, FILE *pInFile) {
  COMPRESS_LZMA_STREAM EncoderStream;
  COMPRESS_LZMA_STREAM DecoderStream;
  int                  EncoderStatus;
  int                  DecoderStatus;
  U64                  OriginalSize;
  U64                  EncodedSize;
  U64                  DecodedSize;
  int                  Res;
  //
  Res = 0;
  printf("LC=%d, LP=%d, PB=%d: ", pParas->LC, pParas->LP, pParas->PB);
  //
  EncoderStatus = COMPRESS_LZMA_ENCODE_Init(&_Encoder, pParas);
  if (EncoderStatus < 0) {
    printf("Skipped, encoder will not initialize\n");
    return 1;
  }
  COMPRESS_LZMA_DECODE_Init(&_Decoder);
  //
  EncodedSize   = 0;
  DecodedSize   = 0;
  EncoderStatus = 0;
  DecoderStatus = 0;
  //
  EncoderStream.pIn      = _aFileBuf;
  EncoderStream.AvailIn  = fread(_aFileBuf, 1, sizeof(_aFileBuf), pInFile);
  EncoderStream.pOut     = &_aEncodedBuf[13];
  EncoderStream.AvailOut = sizeof(_aEncodedBuf) - 13;
  OriginalSize           = EncoderStream.AvailIn;
  //
  EncoderStatus = COMPRESS_LZMA_ENCODE_Run(&_Encoder, &EncoderStream, 1);
  if (EncoderStatus <= 0) {
    printf("Encoder error: %s\n", _GetStatusText(EncoderStatus));
    return 1;
  }
  EncodedSize = sizeof(_aEncodedBuf) - 13 - EncoderStream.AvailOut;
  COMPRESS_LZMA_ENCODE_WrHeader(&_Encoder, &_aEncodedBuf[0]);
  //
  DecoderStream.pIn      = &_aEncodedBuf[0];
  DecoderStream.AvailIn  = sizeof(_aEncodedBuf) - EncoderStream.AvailOut + 13;
  DecoderStream.pOut     = &_aDecodedBuf[0];
  DecoderStream.AvailOut = sizeof(_aDecodedBuf);
  //
  DecoderStatus = COMPRESS_LZMA_DECODE_Run(&_Decoder, &DecoderStream, 1);
  if (DecoderStatus <= 0) {
    printf("Decoder error: %s\n", _GetStatusText(DecoderStatus));
    Res = 1;
  }
  DecodedSize = sizeof(_aDecodedBuf) - DecoderStream.AvailOut;
  //
  if (OriginalSize != DecodedSize) {
    printf("Compress-decompress size mismatch\n");
    Res = 1;
  } else if (memcmp(_aFileBuf, _aDecodedBuf, (unsigned)DecodedSize) != 0) {
    printf("Compress-decompress content mismatch\n");
    Res = 1;
  } else {
    printf("OK   %5.2fx   %lld -> %lld -> %lld
\n", (float)DecodedSize / EncodedSize, OriginalSize, EncodedSize, DecodedSize);
  }
  return Res;
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       main()
*
*  Function description
*    Application entry point.
*
*  Parameters
*    argc - Argument count.
*    argv - Argument vector.
*/
int main(int argc, char **argv) {
  COMPRESS_LZMA_PARAS    Paras;
  FILE                 * pInFile;
  const char           * sArg;
  int                    i;
  unsigned               MaxLP;
  unsigned               MaxLC;
  unsigned               MaxPB;
  int                    Result;
  //
  Paras.WindowSize = COMPRESS_LZMA_CONFIG_ENCODE_WINDOW_SIZE_MAX;
  Paras.MinLen     = 3;
  Paras.MaxLen     = 273;
  Paras.Optimize   = 5;
  MaxLC            = COMPRESS_LZMA_CONFIG_ENCODE_LC_MAX;
  MaxLP            = COMPRESS_LZMA_CONFIG_ENCODE_LP_MAX;
  MaxPB            = COMPRESS_LZMA_CONFIG_ENCODE_PB_MAX;
  //
  for (i = 1; i < argc; ++i) {
    sArg = argv[i];
    if (strncmp(sArg, "-lc=", 4) == 0) {
      MaxLC = COMPRESS_LZMA_MIN((unsigned)atoi(&sArg[4]), MaxLC);
    } else if (strncmp(sArg, "-lp=", 4) == 0) {
      MaxLP = COMPRESS_LZMA_MIN((unsigned)atoi(&sArg[4]), MaxLP);
    } else if (strncmp(sArg, "-pb=", 4) == 0) {
      MaxPB = COMPRESS_LZMA_MIN((unsigned)atoi(&sArg[4]), MaxPB);
    } else if (strncmp(sArg, "-ws=", 4) == 0) {
      Paras.WindowSize = atoi(&sArg[4]);
    } else if (strncmp(sArg, "-O", 2) == 0) {
      Paras.Optimize = atoi(&sArg[2]);
    } else if (strncmp(sArg, "-k", 2) == 0) {
      Paras.LC = 3;
      Paras.LP = 0;
      Paras.PB = 2;
    } else if (strncmp(sArg, "-minlen=", 8) == 0) {
      Paras.MinLen = atoi(&sArg[8]);
    } else if (strncmp(sArg, "-maxlen=", 8) == 0) {
      Paras.MaxLen = atoi(&sArg[8]);
    } else if (sArg[0] == '-') {
      printf("%s: unknown option %s\n", argv[0], sArg);
      exit(100);
    }
  }
  //
  Result = 0;
  for (i = 1; i < argc; ++i) {
    sArg = argv[i];
    if (sArg[0] != '-') {
      pInFile = fopen(sArg, "rb");
      if (pInFile == 0) {
        printf("%s: can't open %s for reading\n", argv[0], argv[1]);
        exit(100);
      }
      //
      printf("\nVerification of %s:\n\n", sArg);
      //
      for (Paras.LC = 0; Paras.LC <= MaxLC; ++Paras.LC) {
        for (Paras.LP = 0; Paras.LP <= MaxLP; ++Paras.LP) {
          for (Paras.PB = 0; Paras.PB <= MaxPB; ++Paras.PB) {
            fseek(pInFile, 0, SEEK_SET);
            Result |= _Verify(&Paras, pInFile);
          }
        }
      }
      fclose(pInFile);
    }
  }
  if (Result) {
    printf("some tests *** FAILED ***\n");
  }
  return Result;
}

/**************************** End of file ****************************/
```

Chapter 4

Sample full-feature application

emCompress-LZMA ships with an application, in source form, which compresses files into LZMA-alone format. These files can be fed into the emCompress-LZMA decompressor, byte for byte, and be decompressed on target hardware.

4.1 Command line and options

The emCompress-LZMA example compression application accepts the command line options described in the following sections.

The command line syntax is:

`COMPRESS_LZMA_Util` [*options*] *inputfile* [*outputfile*]

The output file is optional: if given, the compressed LZMA bitstream is written to it.

The example application displays some information relating to the compression process and selected parameters.

Example

```
C:> COMPRESS_LZMA_Util.exe -O9 -ws=1024 ptt5

(c) 2016 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress-LZMA V2.20 compiled Oct 17 2016 12:13:11

Encoding parameters:
  Coding:          LC=0, LP=0, PB=0
  Window size:     1024 bytes
  Match length:    3 to 273 bytes
  Match distance:  1 to 1024 bytes

Coder performance:
  Max. length:     273 bytes
  Max. distance:   750 bytes

Statistics:
  Decoded size:    513216 bytes
  Encoded size:    49403 bytes (including header)
  Ratio:           10.39x
  Space savings:   90.38% (of original)
  Standardized:    0.77 bits/byte

C:> _
```

4.1.1 Set literal context bits (-lc)

Syntax

`-lc=`*number*

Description

Set the number of literal context bits to use when compressing. Acceptable values are 0 through 8 inclusive.

The default is zero.

4.1.2 Set literal position bits (-lp)

Syntax

`-lp=`*number*

Description

Set the number of literal position bits to use when compressing. Acceptable values are 0 through 4 inclusive.

The default is zero.

4.1.3 Set literal position bits (-pb)

Syntax

`-pb=`*number*

Description

Set the number of position bits to use when compressing. Acceptable values are 0 through 4 inclusive.

The default is zero.

4.1.4 Set window size (-ws)

`-ws=`*number*

Description

Set the window size to when compressing. The window size is the naming convention used by emCompress and emCompress-LZMA to describe the maximum match distance and, therefore, the maximum number of octets that must be stored to satisfy references made by the decompressor. The LZMA SDK refers to this as the "dictionary size" and both terms should be considered equivalent.

Increasing the window size will usually increase compression ratios and reduce the size of the compressed bitstream at the expense of requiring extra RAM during decompression.

The default is `131,072`.

4.1.5 Use prefiltering for thumb-2 instructions (-t2)

`-t2`

Description

Use prefiltering of the data before compression / after decompression to achieve better compression rates for ARM thumb firmware code.

4.1.6 Set minimum match length (-minlen)

`-minlen=`*number*

Description

Set the minimum match length when compressing. Acceptable values are 2 through 273, with the default being 3. We highly recommend that this value is left unchanged.

Although the LZMA compression scheme supports match lengths of two, this usually leads to worse compressor performance in terms of both speed and compressed bitstream size.

4.1.7 Set maximum match length (-maxlen)

Syntax

`-maxlen=`*number*

Description

Set the maximum match length when compressing. Acceptable values are 2 through 273, with the default being 273. We highly recommend that this value is left unchanged.

The window size and the maximum match length are not independent. If the window size is reduced, emCompress-LZMA will also reduce the maximum match length so that it does not exceed the capacity of the window to store the matched sequence. The emCompress-LZMA example application does this outside of the emCompress-LZMA library as the library will diagnose incorrect compression parameters with an error.

4.1.8 Set block input size (-bi)

Syntax

`-bi=`*number*

Description

Set the block input size, the maximum amount of data sent to the compressor in a single call. Acceptable values are 1 through 65536, with the default being 65536.

This has no effect on the compressed output and is present to enable the compressor to be tested, if necessary, with a known input and known block sizes.

4.1.9 Set block output size (-bo)

Syntax

`-bo=`*number*

Description

Set the block output size, the maximum amount of output space available to the compressor in a single call. Acceptable values are 1 through 65536, with the default being 65536.

This has no effect on the compressed output and is present to enable the compressor to be tested, if necessary, with a known input and known block sizes.

4.1.10 Set optimization level (-O)

Syntax

`-O`*number*

Description

Set the optimization level to use when compression. Acceptable values are 0 (faster, good compression) to 9 (slower, best compression).

4.2 Utility listing

```
/*********************************************************************
*                (c) SEGGER Microcontroller GmbH & Co. KG            *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : COMPRESS_LZMA_Util.c
Purpose     : Example emCompress-LZMA application.

*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/

#include "COMPRESS_LZMA_ENCODE.h"
#include "COMPRESS_LZMA_DECODE.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static union {
  COMPRESS_LZMA_ENCODE_CONTEXT    Encoder;
  COMPRESS_LZMA_DECODE_CONTEXT    Decoder;
  COMPRESS_LZMA_ENCODE_T2_CONTEXT EncoderT2;
  COMPRESS_LZMA_DECODE_T2_CONTEXT DecoderT2;
} _u;
static unsigned                    _BlockInSize;
static unsigned                    _BlockOutSize;
static U8                          _aInBuffer [64*1024];
static U8                          _aOutBuffer[64*1024];

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _GetStatusText()
*
*  Function description
*    Decode emCompress-LZMA status code.
*
*  Parameters
*    Status - Status code.
*
*  Return value
*    Non-zero pointer to status description.
*/
static const char * _GetStatusText(int Status) {
  if (Status >= 0) {
    return "OK";
  }
  switch (Status) {
  case COMPRESS_LZMA_STATUS_PARAMETER_ERROR: return "Parameter error";
  case COMPRESS_LZMA_STATUS_BITSTREAM_ERROR: return "Bitstream error";
  case COMPRESS_LZMA_STATUS_USAGE_ERROR:     return "Usage error";
  default:                                   return "Unknown error";
  }
}

/*********************************************************************
*
*       _Encode()
*
*  Function description
*    Encode a file.
*
*  Parameters
*    pInFile  - Pointer to file to encode.
*    pOutFile - Pointer to file that recieves the encoded bitstream.
*    pParas   - Pointer to LZMA encoding parameters.
*    Verbose  - Flag indicating verbose output.
*/
static void _Encode(FILE *pInFile, FILE *pOutFile, COMPRESS_LZMA_PARAS *pParas, int Verbose, int ThumbMode)
  COMPRESS_LZMA_STREAM                Stream;
  int                                 Status;
  int                                 Flush;
  unsigned                            EncodedLen;
  unsigned                            DecodedLen;
  U8                                  aHeader[13];
  //
  // Initialize encoder.
  //
  if (ThumbMode) {
    COMPRESS_LZMA_ENCODE_T2_Init(&_u.EncoderT2, pParas);
  } else {
    COMPRESS_LZMA_ENCODE_Init(&_u.Encoder, pParas);
  }
  //
  // Reserve space for an LZMA header in the output file.
  //
  if (pOutFile) {
    fseek(pOutFile, 13, SEEK_SET);
  }
  //
  // Encode file.
  //
  EncodedLen = 0;
  DecodedLen = 0;
  Status     = 0;
  while (Status == 0) {
    //
    // Read next chunk and set up coder buffers.
    //
    Stream.pIn     = _aInBuffer;
    Stream.AvailIn = fread(_aInBuffer, 1, _BlockInSize, pInFile);
    DecodedLen    += Stream.AvailIn;
    //
    // Flush compressor on end of input file.
    //
    Flush = Stream.AvailIn != _BlockInSize;
    //
    // Encode input data, writing encoded data to file.
    //
    while (Status == 0 && (Stream.AvailIn > 0 || Flush)) {
      //
      Stream.pOut     = &_aOutBuffer[0];
      Stream.AvailOut = _BlockOutSize;
      //
      if (ThumbMode) {
        Status = COMPRESS_LZMA_ENCODE_T2_Run(&_u.EncoderT2, &Stream, Flush);
      } else {
        Status = COMPRESS_LZMA_ENCODE_Run(&_u.Encoder, &Stream, Flush);
      }
      //
      if (Status >= 0 && pOutFile != NULL) {
        fwrite(_aOutBuffer, 1, _BlockOutSize - Stream.AvailOut, pOutFile);
      }
      EncodedLen += _BlockOutSize - Stream.AvailOut;
    }
  }
  //
  // Rewind file and write correct header.
  //
  if (pOutFile) {
    if (ThumbMode) {
      COMPRESS_LZMA_ENCODE_T2_WrHeader(&_u.EncoderT2, &aHeader[0]);
    } else {
      COMPRESS_LZMA_ENCODE_WrHeader(&_u.Encoder, &aHeader[0]);
    }
    fseek(pOutFile, 0, SEEK_SET);
    fwrite(aHeader, 1, sizeof(aHeader), pOutFile);
  }
  //
  if (Status < 0) {
    fprintf(stderr, "FATAL: %s\n", _GetStatusText(Status));
    exit(1);
  } else {
    if (Verbose) {
      fprintf(stderr, "Encoding parameters:\n");
      fprintf(stderr, "  Coding:         LC=%u, LP=%u, PB=%u%s\n",
  _u.Encoder.LC, _u.Encoder.LP, _u.Encoder.PB, (ThumbMode != 0) ? ", T2" : "");
      fprintf(stderr, "  Window size:    %u bytes\n",             _u.Encoder.WindowSize);
      fprintf(stderr, "  Match length:   %d to %d bytes\n",
  _u.Encoder.LZSSCoder.MinLength, _u.Encoder.LZSSCoder.MaxLength);
      fprintf(stderr, "  Match distance: 1 to %d bytes\n",
  _u.Encoder.LZSSCoder.MaxDistance);
      fprintf(stderr, "\n");
      fprintf(stderr, "Coder performance:\n");
      fprintf(stderr, "  Max. length:    %d bytes\n",
  _u.Encoder.LZSSCoder.MaxUsedLength);
      fprintf(stderr, "  Max. distance:  %d bytes\n",
  _u.Encoder.LZSSCoder.MaxUsedDistance);
      fprintf(stderr, "\n");
      fprintf(stderr, "Statistics:\n");
      fprintf(stderr, "  Decoded size:   %u bytes\n",             DecodedLen);
      fprintf(stderr, "  Encoded size:   %u bytes (including header)\n", EncodedLen + 13);
      fprintf(stderr, "  Ratio:          %.2fx\n",
  (float)DecodedLen / EncodedLen);
      if (DecodedLen == 0) {
        fprintf(stderr, "  Space savings:  -inf%% (of original)\n");
        fprintf(stderr, "  Standardized:   inf bits/byte\n");
      } else {
        fprintf(stderr, "  Space savings:  %.2f%% (of
 original)\n", 100.0f - EncodedLen * 100.0f / DecodedLen);
        fprintf(stderr, "  Standardized:   %.2f bits/byte\n",
  EncodedLen * 8.0f / DecodedLen);
      }
    }
  }
}

/*********************************************************************
*
*       _Decode()
*
*  Function description
*    Decode a file.
*
*  Parameters
*    pInFile  - Pointer to file containing the encoded bitstream.
*    pOutFile - Pointer to file that recieves the decoded bitstream.
*    Verbose  - Flag indicating verbose output.
*/
static void _Decode(FILE *pInFile, FILE *pOutFile, int Verbose, int ThumbMode) {
  COMPRESS_LZMA_STREAM                Stream;
  int                                 Status;
  int                                 Flush;
  unsigned                            EncodedLen;
  unsigned                            DecodedLen;
  //
  // Initialize decoder.
  //
  if (ThumbMode) {
    COMPRESS_LZMA_DECODE_T2_Init(&_u.DecoderT2);
  } else {
    COMPRESS_LZMA_DECODE_Init(&_u.Decoder);
  }
  //
  DecodedLen = 0;
  EncodedLen = 0;
  Status     = 0;
  while (Status == 0) {
    //
    // Read next chunk and set up coder buffers.
    //
    Stream.pIn     = _aInBuffer;
    Stream.AvailIn = fread(_aInBuffer, 1, _BlockInSize, pInFile);
    EncodedLen    += Stream.AvailIn;
    //
    // Exit on end of file.
    //
    Flush = Stream.AvailIn != _BlockInSize;
    //
    // Decode input data, writing decoded data to file.
    //
    while (Status == 0 && (Stream.AvailIn > 0 || Flush)) {
      //
      Stream.pOut     = &_aOutBuffer[0];
      Stream.AvailOut = _BlockOutSize;
      //
      if (ThumbMode) {
        Status = COMPRESS_LZMA_DECODE_T2_Run(&_u.DecoderT2, &Stream, Flush);
      } else {
        Status = COMPRESS_LZMA_DECODE_Run(&_u.Decoder, &Stream, Flush);
      }
      if (Status >= 0 && pOutFile) {
        fwrite(_aOutBuffer, 1, _BlockOutSize - Stream.AvailOut, pOutFile);
      }
      DecodedLen += _BlockOutSize - Stream.AvailOut;
    }
  }
  //
  if (Status < 0) {
    fprintf(stderr, "FATAL: %s\n", _GetStatusText(Status));
    exit(1);
  } else {
    if (Verbose) {
      fprintf(stderr, "Decoding parameters:\n");
      fprintf(stderr, "  Coding:         LC=%u, LP=%u, PB=%u%s\n",
  _u.Decoder.LC, _u.Decoder.LP, _u.Decoder.PB, (ThumbMode != 0) ? ", T2" : "");
      fprintf(stderr, "  Window size:    %u bytes\n",             _u.Decoder.WindowSize);
      fprintf(stderr, "\n");
      fprintf(stderr, "Statistics:\n");
      fprintf(stderr, "  Encoded size:   %u bytes\n",             EncodedLen);
      fprintf(stderr, "  Decoded size:   %u bytes \n",            DecodedLen);
      fprintf(stderr, "  Ratio:          %.2fx\n",
  (float)DecodedLen / EncodedLen);
      fprintf(stderr, "  Space savings:  %.2f%% (of
 original)\n", 100.0f - EncodedLen * 100.0f / DecodedLen);
      fprintf(stderr, "  Standardized:   %.2f bits/byte\n",
  EncodedLen * 8.0f / DecodedLen);
    }
  }
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       main()
*
*  Function description
*    Application entry point.
*
*  Parameters
*    argc - Argument count.
*    argv - Argument vector.
*/
int main(int argc, char **argv) {
  COMPRESS_LZMA_PARAS Paras;
  int                 i;
  int                 Verbose;
  int                 Action;
  int                 ThumbMode;
  const char        * sInputPathname;
  const char        * sOutputPathname;
  FILE              * pInFile;
  FILE              * pOutFile;
  //
  fprintf(stderr, "\n");
  fprintf(stderr, "(c) 2015-2022 SEGGER Microcontroller GmbH    www.segger.com\n");
  fprintf(stderr, "emCompress-LZMA V%d.%02d.%02d
", COMPRESS_LZMA_VERSION / 10000, COMPRESS_LZMA_VERSION / 100 % 100, COMPRESS_LZMA_VERSION % 100);
  fprintf(stderr, "compiled " __DATE__ " " __TIME__ "\n");
  fprintf(stderr, "\n");
  //
  Paras.LC         = 0;
  Paras.LP         = 0;
  Paras.PB         = 0;
  Paras.WindowSize = COMPRESS_LZMA_MIN(COMPRESS_LZMA_CONFIG_ENCODE_WINDOW_SIZE_MAX,
                                       COMPRESS_LZMA_CONFIG_DECODE_WINDOW_SIZE_MAX);
  Paras.MinLen     = 3;
  Paras.MaxLen     = 273;
  Paras.Optimize   = 5;
  //
  sInputPathname   = 0;
  sOutputPathname  = 0;
  Action           = -1;  // Compress
  ThumbMode        = 0;
  Verbose          = 1;
  _BlockInSize     = sizeof(_aInBuffer);
  _BlockOutSize    = sizeof(_aOutBuffer);
  //
  for (i = 1; i < argc; ++i) {
    const char *sArg = argv[i];
    if (strncmp(sArg, "-lc=", 4) == 0) {
      Paras.LC = atoi(&sArg[4]);
    } else if (strncmp(sArg, "-lp=", 4) == 0) {
      Paras.LP = atoi(&sArg[4]);
    } else if (strncmp(sArg, "-pb=", 4) == 0) {
      Paras.PB = atoi(&sArg[4]);
    } else if (strncmp(sArg, "-ws=", 4) == 0) {
      Paras.WindowSize = atoi(&sArg[4]);
    } else if (strncmp(sArg, "-O", 2) == 0) {
      Paras.Optimize = atoi(&sArg[2]);
    } else if (strncmp(sArg, "-k", 2) == 0) {
      Paras.LC = 3;
      Paras.LP = 0;
      Paras.PB = 2;
    } else if (strncmp(sArg, "-bi=", 4) == 0) {

  _BlockInSize = COMPRESS_LZMA_MAX(1, COMPRESS_LZMA_MIN(atoi(&sArg[4]), sizeof(_aInBuffer)));
    } else if (strncmp(sArg, "-bo=", 4) == 0) {

  _BlockOutSize = COMPRESS_LZMA_MAX(1, COMPRESS_LZMA_MIN(atoi(&sArg[4]), sizeof(_aOutBuffer)));
    } else if (strncmp(sArg, "-minlen=", 8) == 0) {
      Paras.MinLen = atoi(&sArg[8]);
    } else if (strncmp(sArg, "-maxlen=", 8) == 0) {
      Paras.MaxLen = atoi(&sArg[8]);
    } else if (strcmp(sArg, "-d") == 0) {
      Action = +1;
    } else if (strcmp(sArg, "-c") == 0) {
      Action = -1;
    } else if (strcmp(sArg, "-v") == 0) {
      Verbose = 1;
    } else if (strcmp(sArg, "-q") == 0) {
      Verbose = 0;
    } else if (strcmp(sArg, "-t2") == 0) {
      ThumbMode = 1;
    } else if (sArg[0] == '-') {
      printf("%s: unknown option %s\n", argv[0], sArg);
      exit(100);
    } else if (sInputPathname == 0) {
      sInputPathname = sArg;
    } else if (sOutputPathname == 0) {
      sOutputPathname = sArg;
    } else {
      printf("%s: too many filenames\n", argv[0]);
      exit(100);
    }
  }
  //
  if (sInputPathname == 0) {
    printf("%s: require input filename\n", argv[0]);
    exit(100);
  }
  //
  pInFile = fopen(sInputPathname, "rb");
  if (pInFile == 0) {
    printf("%s: can't open %s for reading\n", argv[0], sInputPathname);
    exit(100);
  }
  //
  if (Action < 0 && COMPRESS_LZMA_ENCODE_Init(&_u.Encoder, &Paras) < 0) {
    printf("%s: inconsistent or invalid compression parameters\n", argv[0]);
    exit(100);
  }
  //
  pOutFile = 0;
  if (sOutputPathname != 0) {
    pOutFile = fopen(sOutputPathname, "wb");
    if (pOutFile == 0) {
      printf("%s: can't open %s for writing\n", argv[0], sInputPathname);
      fclose(pInFile);
      exit(100);
    }
  }
  //
  if (Action < 0) {
    _Encode(pInFile, pOutFile, &Paras, Verbose, ThumbMode);
  } else {
    _Decode(pInFile, pOutFile, Verbose, ThumbMode);
  }
  //
  if (pInFile) {
    fclose(pInFile);
  }
  if (pOutFile) {
    fclose(pOutFile);
  }
  return 0;
}

/*************************** End of file ****************************/
```

Chapter 5

API reference

This section describes the public API for emCompress-LZMA. Any functions or data structures that are not described here but are exposed through inclusion of either `COMPRESS_LZMA_ENCODE.h` or `COMPRESS_LZMA_DECODE.h` must be considered private and subject to change.

5.1 Compressor functions

Function	Description
`COMPRESS_LZMA_ENCODE_Init()`	Initialize LZMA encoder.
`COMPRESS_LZMA_ENCODE_Run()`	Run LZMA encoder.
`COMPRESS_LZMA_ENCODE_WrHeader()`	Write LZMA-alone header.
`COMPRESS_LZMA_ENCODE_T2_Init()`	Initialize LZMA encoder with thumb pre-conditioner.
`COMPRESS_LZMA_ENCODE_T2_Run()`	Run LZMA encoder with thumb pre-conditioner.
`COMPRESS_LZMA_ENCODE_T2_WrHeader()`	Write LZMA-alone header.

5.1.1 COMPRESS_LZMA_ENCODE_Init()

Description

Initialize LZMA encoder.

Prototype

```
int COMPRESS_LZMA_ENCODE_Init(      COMPRESS_LZMA_ENCODE_CONTEXT * pSelf,
                              const COMPRESS_LZMA_PARAS          * pParas);
```

Parameters

Parameter	Description
pSelf	Pointer to encoder context to be initialized.
pParas	Pointer to encoder parameters.

Return value

< 0	Error in one or more parameters.
≥ 0	Success.

5.1.2 COMPRESS_LZMA_ENCODE_Run()

Description

Run LZMA encoder.

Prototype

```
int COMPRESS_LZMA_ENCODE_Run(COMPRESS_LZMA_ENCODE_CONTEXT * pSelf,
                             COMPRESS_LZMA_STREAM         * pStream,
                             int                            Flush);
```

Parameters

Parameter	Description
`pSelf`	Pointer to LZMA encoder context. The context must be created by a call to `COMPRESS_LZMA_ENCODE_Init()` or originate from a previous call to `COMPRESS_LZMA_ENCODE_Run()`. The application should never modify the context in any way, otherwise the behavior of this function in undefined.
`pStream`	Pointer to stream communication object.
`Flush`	A value ≠ 0 indicates the end of the input stream flag.

Return value

≤ 0	Processing error.
= 0	OK, continue encoding.
> 0	Encoding complete.

5.1.3 COMPRESS_LZMA_ENCODE_WrHeader()

Description

Write LZMA-alone header.

Prototype

```
void COMPRESS_LZMA_ENCODE_WrHeader(const COMPRESS_LZMA_ENCODE_CONTEXT * pSelf,
                                         U8                           * pHeader);
```

Parameters

Parameter	Description
pSelf	Pointer to LZMA encoder context.
pHeader	Pointer to object that receives the 13-octet LZMA-alone header.

5.1.4 COMPRESS_LZMA_ENCODE_T2_Init()

Description

Initialize LZMA encoder with thumb pre-conditioner.

Prototype

```
int COMPRESS_LZMA_ENCODE_T2_Init(      COMPRESS_LZMA_ENCODE_T2_CONTEXT * pSelf,
                                 const COMPRESS_LZMA_PARAS             * pParas);
```

Parameters

Parameter	Description
pSelf	Pointer to encoder context to be initialized.
pParas	Pointer to encoder parameters.

Return value

< 0	Error in one or more parameters.
≥ 0	Success.

5.1.5 COMPRESS_LZMA_ENCODE_T2_Run()

Description

Run LZMA encoder with thumb pre-conditioner.

Prototype

```
int COMPRESS_LZMA_ENCODE_T2_Run(COMPRESS_LZMA_ENCODE_T2_CONTEXT * pSelf,
                                COMPRESS_LZMA_STREAM            * pStream,
                                int                               Flush);
```

Parameters

Parameter	Description
pSelf	Pointer to LZMA encoder context. The context must be created by a call to COMPRESS_LZMA_ENCODE_T2_Init() or originate from a previous call to COMPRESS_LZMA_ENCODE_T2_Run(). The application should never modify the context in any way, otherwise the behavior of this function in undefined.
pStream	Pointer to stream communication object.
Flush	A value ≠ 0 indicates the end of the input stream flag.

Return value

≤ 0	Processing error.
= 0	OK, continue encoding.
> 0	Encoding complete.

5.1.6 COMPRESS_LZMA_ENCODE_T2_WrHeader()

Description

Write LZMA-alone header.

Prototype

```
void COMPRESS_LZMA_ENCODE_T2_WrHeader
                                 (const COMPRESS_LZMA_ENCODE_T2_CONTEXT * pSelf,
                                        U8                              * pHeader);
```

Parameters

Parameter	Description
pSelf	Pointer to LZMA encoder context.
pHeader	Pointer to object that receives the 13-octet LZMA-alone header.

5.2 Decompressor functions

Function	Description
COMPRESS_LZMA_DECODE_Init()	Initialize LZMA decoder.
COMPRESS_LZMA_DECODE_Run()	Run LZMA decoder.
COMPRESS_LZMA_DECODE_T2_Init()	Initialize LZMA decoder with thumb pre-conditioner.
COMPRESS_LZMA_DECODE_T2_Run()	Run LZMA decoder.

5.2.1 COMPRESS_LZMA_DECODE_Init()

Description

Initialize LZMA decoder.

Prototype

```
void COMPRESS_LZMA_DECODE_Init(COMPRESS_LZMA_DECODE_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to decoder context to be initialized.

5.2.2 COMPRESS_LZMA_DECODE_Run()

Description

Run LZMA decoder.

Prototype

```
ILEAST16 COMPRESS_LZMA_DECODE_Run(COMPRESS_LZMA_DECODE_CONTEXT * pSelf,
                                  COMPRESS_LZMA_STREAM         * pStream,
                                  ILEAST16                       Flush);
```

Parameters

Parameter	Description
`pSelf`	Pointer to decoder context. The context must be created by a call to `COMPRESS_LZMA_DECODE_Init()` or originate from a previous call to `COMPRESS_LZMA_DECODE_Run()`. The application should never modify the context in any way, otherwise the behavior of this function in undefined.
`pStream`	Pointer to stream context.
`Flush`	A value ≠ 0 indicates the end of the input stream flag.

Return value

< 0	Processing error or bitstream error
= 0	Call again to continue processing
> 0	Decompression complete

5.2.3 COMPRESS_LZMA_DECODE_T2_Init()

Description

Initialize LZMA decoder with thumb pre-conditioner.

Prototype

```
void COMPRESS_LZMA_DECODE_T2_Init(COMPRESS_LZMA_DECODE_T2_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to decoder context to be initialized.

5.2.4 COMPRESS_LZMA_DECODE_T2_Run()

Description

Run LZMA decoder.

Prototype

```
ILEAST16 COMPRESS_LZMA_DECODE_T2_Run(COMPRESS_LZMA_DECODE_T2_CONTEXT * pSelf,
                                     COMPRESS_LZMA_STREAM            * pStream,
                                     ILEAST16                          Flush);
```

Parameters

Parameter	Description
pSelf	Pointer to decoder context. The context must be created by a call to COMPRESS_LZMA_DECODE_T2_Init() or originate from a previous call to COMPRESS_LZMA_DECODE_T2_Run(). The application should never modify the context in any way, otherwise the behavior of this function in undefined.
pStream	Pointer to stream context.
Flush	A value ≠ 0 indicates the end of the input stream flag.

Return value

< 0	Processing error or bitstream error
= 0	Call again to continue processing
> 0	Decompression complete

Chapter 6

Indexes

6.1 Index of functions