emCompress-Pro

Professional compression system

User Guide & Reference Manual

Document: UM17004
Software Version: 2.20.0
Revision: 0
Date: April 16, 2026

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

The information written in this document is assumed to be accurate without guarantee. The information in this manual is subject to change for functional or performance improvements without notice. SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions in this document. SEGGER disclaims any warranties or conditions, express, implied or statutory for the fitness of the product for a particular purpose. It is your sole responsibility to evaluate the fitness of the product for any specific use.

Copyright notice

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of SEGGER. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2015-2026 SEGGER Microcontroller GmbH, Monheim am Rhein / Germany

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

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: ticket_emcompress@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: April 16, 2026

Software	Date	By	Description
2.20	2026-04-16	MHA	Update to latest software version. Added chapter *Supported algorithms*. Chapter *emCompress-Pro API* • Added thorough explanation about signalling of end of data. Chapter *Resource requirements* • Added benchmark results. • Added formulas for estimating dynamic memory requirements. Chapter *Utilities* • Added chapter about usage of `CX_Tool` and `CX_Optimize`.
2.12	2026-01-01	PC	First release.

© 2015-2026 SEGGER Microcontroller GmbH

 © 2015-2026 SEGGER Microcontroller GmbH

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.
SEGGER home page	A hyperlink to an external document or web site.

© 2015-2026 SEGGER Microcontroller GmbH

 © 2015-2026 SEGGER Microcontroller GmbH

Table of contents

1 Introducing emCompress-Pro 9

- 1.1 What is emCompress-Pro? 10
- 1.2 Features 11
- 1.3 Package content 12
- 1.4 Sample applications 13
  - 1.4.1 A note on the samples 13
- 1.5 Recommended project structure 14
  - 1.5.1 Include directories 14

2 Supported algorithms 15

- 2.1 LZPJ 16
- 2.2 DEFLATE 16
- 2.3 LZMA 16
  - 2.3.1 Compatibility with the XZ tool 17
- 2.4 SMASH-2 18

3 Using emCompress-Pro 19

- 3.1 Compressing and decompressing data 20
  - 3.1.1 Providing dynamic memory 20
  - 3.1.2 Preparing for compression 20
  - 3.1.3 Compressing data 22
  - 3.1.4 Decompressing data 23
- 3.2 Compression of program code 25

4 emCompress-Pro API 27

- 4.1 Preprocessor definitions 28
  - 4.1.1 Version number 28
  - 4.1.2 CX_LZSS_ENCODE_LARGE_WINDOW 28
  - 4.1.3 CX_LZSS_ENCODE_MAX_WINDOW_SIZE 28
  - 4.1.4 CX_LZSS_HASH_TABLE_SIZE 28
- 4.2 Data types 30
  - 4.2.1 CX_PARAS 30
  - 4.2.2 CX_STREAM 31
  - 4.2.3 CX_ENCODE_CONTEXT 32
  - 4.2.4 CX_DECODE_CONTEXT 33
- 4.3 Data definitions 34
  - 4.3.1 Compression algorithms 34
  - 4.3.2 Decompression algorithms 35
- 4.4 Compression functions 36

 © 2015-2026 SEGGER Microcontroller GmbH

4.4.1 CX_ENCODE_Init() 37
4.4.2 CX_ENCODE_Process() 38
4.4.3 CX_ENCODE_Exit() 39
4.5 Decompression functions 40
4.5.1 CX_DECODE_Init() 41
4.5.2 CX_DECODE_Process() 42
4.5.3 CX_DECODE_Exit() 43
4.6 Conditioning functions 44
4.6.1 CX_PRECOND_T32_Run() 45
4.6.2 CX_PRECOND_A32_Run() 46
4.6.3 CX_PRECOND_A64_Run() 47
4.6.4 CX_PRECOND_RV32_Run() 48
4.6.5 CX_PRECOND_IA32_Run() 49
4.7 Utility functions 50
4.7.1 CX_PARAS_Clear() 51
4.7.2 CX_GetErrorText() 52
4.7.3 CX_GetCopyrightText() 53
4.7.4 CX_GetVersionText() 54

5 Resource requirements 55

5.1 SMASH-2 algorithm 57
5.2 DEFLATE algorithm 59
5.3 LZPJ algorithm 61
5.4 LZMA algorithm 63

6 Utilities 65

6.1 Compression algorithm comparison 66
6.1.1 Command line and options 66
6.1.2 Using the CX_Optimize tool 66
6.1.2.1 Optimizing SMASH-2 compression 67
6.1.2.2 Optimizing LZMA compression 67
6.2 Compression/decompression utility 69
6.2.1 Command line and options 69
6.2.2 Using the CX_Tool utility 69

7 Glossary 70

8 Indexes 71

8.1 Index of functions 72
8.2 Index of types 73

 © 2015-2026 SEGGER Microcontroller GmbH

Chapter 1

Introducing emCompress-Pro

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

© 2015-2026 SEGGER Microcontroller GmbH

1.1 What is emCompress-Pro?

emCompress-Pro is a compression system that is able to reduce the size of data that must be compressed or decompressed by a small target microcontroller. Typical uses of emCompress-Pro are:

- Compress and decompress communication data over a limited-bandwidth link.
- Decompress firmware images that must be dynamically expanded on device reprogramming.
- Decompress configuration bitstreams to program FPGA and CPLD devices.
- Permanent files for embedded web server static content.

Of course, emCompress-Pro is not limited to these applications, it can be used whenever it’s beneficial to reduce the size of dynamic or static data.

 © 2015-2026 SEGGER Microcontroller GmbH

1.2 Features

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

- Clean ISO/ANSI C source code.
- Small decompressor ROM footprint.
- Essentially zero-RAM compression and decompression.
- Easy-to-understand and simple-to-use API.
- Simple configuration.

© 2015-2026 SEGGER Microcontroller GmbH

1.3 Package content

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

Files	Description
Application	Sample application source code.
Config	Configuration header files.
Doc	emCompress-Pro documentation.
COMPRESS	emCompress-Pro source code.
SEGGER	SEGGER software component source code used in emCompress-Pro.
Tool	Supporting applications in binary form.

 © 2015-2026 SEGGER Microcontroller GmbH

1.4 Sample applications

emCompress-Pro ships with a number of sample applications that show how to integrate compression capability into your application.

The sample applications are:

Application	Description
`CX_Compress.c`	Compress file using DEFLATE algorithm.
`CX_Decompress.c`	Decompress file compressed with DEFLATE algorithm.

1.4.1 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.

© 2015-2026 SEGGER Microcontroller GmbH

1.5 Recommended project structure

We recommend keeping emCompress-Pro 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-Pro by simply replacing the COMPRESS and SEGGER directories. Your application files can be stored anywhere.

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

1.5.1 Include directories

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

- `Config`
- `COMPRESS`
- `SEGGER`

> **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-Pro if you have old files included and therefore mix different versions. If you keep emCompress-Pro 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.

 © 2015-2026 SEGGER Microcontroller GmbH

Chapter 2

Supported algorithms

emCompress-Pro supports four compression algorithms: SMASH-2, DEFLATE, LZPJ and LZ-MA. These algorithms differ in complexity, achievable compression rate, speed, and resource requirements. The algorithms are composed of several layers, which process data which has passed through other layers. During decompression, data is fed through the layers in reverse order.

The LZSS base layer

The first layer, on which the four algorithms are based, is the LZSS layer. The LZSS (Lempel–Ziv–Storer–Szymanski) algorithm is an extension of the well-known LZ77 (Lempel–Ziv-1). The LZSS algorithm scans the data for repeated blocks. When it finds a repetition of a block, it inserts a reference to the block, containing the offset and the length of the block. Block references are only inserted if the net length of the reference is smaller than the uncompressed equivalent. For example, to insert a reference to a two character long block at a large distance may require more space than simply inserting the two characters as literals. The LZSS layer consumes the raw data and at each step, reports to the next layer whether literal data or a reference to a previously seen block should be inserted.

The LZSS algorithm can be controlled by several parameters, whose exact ranges depend on the layers on top of the LZSS algorithm which handle the encoding of the LZSS tokens into a stream.

- Window size: The window size determines the context size which can be used to look for matching blocks which can be replaced by references.
- Optimization level: The LZSS algorithm curates a hash based with linked listed which it uses to find blocks which it can reference. The optimization parameter determines how far back through the linked lists the algorithm shall search for a suitable match.
- Hash table size: The hash table defines the entry point to the linked list of matches with the same hash. The size of the hash table provides a trade-off between speed and memory requirements: When the hash table is small, collisions are more likely and therefore the linked lists behind each hash table entry will become longer, and the search for a match will take longer. At low optimization levels and large window sizes, the lists will only be partially searched. When the table is larger, it consumes more memory. However, matches can potentially be spread more evenly, resulting in a shorter average linked list which needs to be searched for each hit in the hash map.
- Minimum and maximum match length: The minimum length a matching block needs to have before it is considered to be inserted as a reference is defined by the algorithm running on top of LZSS. The lower limit is usually determined by the expected size of encoding of the reference versus literal data. The upper limit is determined by the maximum match length which the bitstream can encode.

Parameter	LZPJ	DEFLATE	LZMA	SMASH-2
Optimization	0 - 10	0 - 10	0 - 10	0 - 10
Window size	16 - 32768	16 - 32768	1 - 16777216	256 - 16384
Minimum match length	3 - 258	3 - 258	2 - 273	2 - 258
Maximum match length	18 - 258	18 - 258	2 - 273	2 - 258

Additional requirements:

- LZPJ, DEFLATE, SMASH-2: The minimum match length must be smaller than the maximum match length.
- LZMA: The minimum match length must be smaller or equal to the maximum match length.
- SMASH-2: The window size must be a power of two.
- SMASH-2, LZPJ: The window size must be larger than the minimum match length.

The size of the hash map is configured at compile time using the `CX_LZSS_HASH_TABLE_SIZE` configuration flag. For window sizes larger than 65260 bytes, the `CX_LZSS_ENCODE_LARGE_WINDOW` flag has to be set to 1.

2.1 LZPJ

The LZPJ (Lempel-Ziv "Plain Jane") algorithm takes the output of the LZSS algorithm and turns it into a simple bitstream which only contains the literals and the references. For this reason, implementations of LZPJ are very simple and fast, at the cost of compression efficiency.

2.2 DEFLATE

The DEFLATE algorithm is also based on the LZSS algorithm. It employs entropy encoding using Huffman tables to efficiently represent the tokens delivered by the LZSS algorithm. emCompress-Pro determines the entries of the Huffman tables dynamically during compression, but it can also read data compressed with static Huffman tables by other compression utilities.

In order to be able to adapt the Huffman table to changes in the input data, the DEFLATE algorithm's output is block based. Each block has its own Huffman table. The maximum block size is a parameter which must be provided to the algorithm. Usual values for the block length are 16384 to 32768.

2.3 LZMA

The LZMA (Lempel-Ziv-Markov chain algorithm) also uses LZSS as its foundation. It supports very large dictionary sizes for inserting references and a sophisticated tracker to optimize references to previous blocks. It employs a range encoder instead of a Huffman encoder for even more dense storage of data.

A range encoder is based on the probability distribution of the characters in a stream of data. The range encoder used in LZMA does not treat the data uniformly, but tracks the probabilities of characters based on certain contexts:

- Literal context (LC): Number of most significant bits of the previous byte to use for context determination when encoding a literal.
- Literal position (LP): Number of least significant position bits to use for context determination when encoding literals.
- Position bits (PB): Number of least significant position bits to use for context determination for general encoding.

The range of valid values for these parameters depends on whether compression and decompression are only performed with emCompress-Pro or whether other tools, like the `xz`

 © 2015-2026 SEGGER Microcontroller GmbH

tool must also be able to work with the data. See section *Optimizing LZMA compression* on page 67 for details on how to determine the best values of these parameters. The values of LC, LP and PB contribute to the dynamic memory requirements for compression and decompression, see sub-section *"LZMA algorithm"* on page 63 in the section about memory requirements.

Parameter	Default	emCompress-Pro	XZ
LC	3	0 - 8	0 - 4
LP	0	0 - 4	0 - 4
PB	2	0 - 4	0 - 4
			LC + LP ≤ 4

2.3.1 Compatibility with the XZ tool

emCompress-Pro can produce and consume raw LZMA streams which are called "LZMA1" in the terminology of the XZ tool. emCompress-Pro's LZMA compressor has been designed to provide a reasonable compression ratio within the memory limitations of embedded systems. Therefore, its compression efficiency may be lower than what can be achieved by the XZ tool, which features a highly optimized compressor.

Compression with the XZ tool

To compress data with the XZ tool for consumption by emCompress-Pro, the following parameters need to be specified:

```
xz -z -k --suffix=.lzma -T 1 --format=raw
    --lzma1=preset=<OPT>,lc=<LC>,lp=<LP>,pb=<PB>,dict=<WS>
    <input-file>
```

- **-z**: Compress data
- **-k**: Keep the input file
- **--suffix=.lzma**: Write output to a file with the same name as the input file with the added suffix **.lzma**. The name of the output file can not be specified directly, only via the **--suffix** parameter.
- **-T 1**: Use a single thread only.
- `<OPT>`: Specifies the optimization level, which ranges from 0 to 9.
- `<LC>`, `<LP>` and `<PB>`: Specifies the values for LC, LP and PB.
- `<WS>`: Specifies the window size.

For example, to compress a file `Firmware.input` with a window size of 32768, maximum optimization (9), LC=1, LP=2 and PB=2, the following command can be used to generate a compressed file `Firmware.input.lzma`:

```
xz -z -k --suffix=.lzma -T 1 --format=raw
    --lzma1=preset=9,lc=1,lp=2,pb=2,dict=32768
    Firmware.input
```

This file can be decompressed using emCompress-Pro's `XZ_Tool` (section *compression/decompression utility* on page 69) like this:

```
CX_Tool -d lzma --lc=1 --lp=2 --pb=2 --ws=32768
    Firmware.input.lzma Firmware.output
```

Decompression with the XZ tool

```
xz -d -k --suffix=.lzma -T 1 --format=raw
    --lzma1=preset=9,lc=<LC>,lp=<LP>,pb=<PB>,dict=<WS>
    <input-file>
```

- **-d**: Decompress data
- **-k**: Keep the input file
- **--suffix=.lzma**: Write output to a file with the name as the input file, but with the suffix **.lzma** removed. The name of the output file can not be specified explicitly, the XZ tool can only use the same filename without the specified suffix.

 © 2015-2026 SEGGER Microcontroller GmbH

- **-T 1**: Use a single thread only.
- `<LC>`, `<LP>` and `<PB>`: Specifies the values for LC, LP and PB.
- `<WS>`: Specifies the window size.

For example, when data is compressed using emCompress-Pro's `XZ_Tool` (section *compression/decompression utility* on page 69) like this:

```
CX_Tool -c lzma --lc=1 --lp=2 --pb=2 --ws=32768 --opt=10
    Firmware.input Firmware.output.lzma
```

It can be decompressed using the XZ tool, resulting in an uncompressed file named `Firmware.output`:

```
xz -d -k --suffix=.lzma -T 1 --format=raw
    --lzma1=preset=9,lc=1,lp=2,pb=2,dict=32768
    Firmware.output.lzma
```

2.4 SMASH-2

SMASH-2 uses LZSS combined with variable length encoding of references to matches. The SMASH-2 implementation in emCompress-Pro features an optimization for compressing firmware with 16-bit or 32-bit instruction width, which allows more efficient encoding of distances to matches.

The optimization for different instruction sets is enabled using the ISA parameter:

Instruction set	ISA-parameter
Variable length instructions (e.g. IA-32), no optimization	0
16-bit instructions	1
32-bit instructions	2

 © 2015-2026 SEGGER Microcontroller GmbH

Chapter 3

Using emCompress-Pro

emCompress-Pro divides into two parts:

- A compressor that is responsible for compressing data using a selected compression algorithm and parameters, and
- A decompressor that is responsible for decompressing data compressed with a selected compression algorithm and parameters.

© 2015-2026 SEGGER Microcontroller GmbH

3.1 Compressing and decompressing data

Compressing a stream of data comprises three steps:

- Initialization
- Repeatedly presenting uncompressed data and extracting compressed data
- Finalization

3.1.1 Providing dynamic memory

The amount of memory required for compression and decompression depends on the algorithm and the compression parameters. For this reason, dynamic memory is used instead of static memory. The dynamic memory management system can be chosen to match the capabilities of the system on which emCompress-Pro is used.

emCompress-Pro accesses the memory management system via a context, which is initialized to use the chosen memory management system. The available memory management systems are described below.

System heap

On PCs and on some embedded systems, a heap is available using the standard `malloc` and `free` functions. This heap can be be used by emCompress-Pro by initializing the context as follows:

```
#include "CX.h"

static SEGGER_MEM_CONTEXT   MemCtx;

SEGGER_MEM_SYSTEM_HEAP_Init(&MemCtx);
```

Stack-like buffer implementation

On systems which do not provide a heap system, the stack-like buffer implementation is the simplest option for memory management. It needs a static buffer which is large enough to accomodate for the maximum dynamic memory requirements of emCompress-Pro. Assuming that the maximum memory requirement has been determined to be 64 KB (using section *Resource requirements* on page 55), the following code can be used:

```
#include "CX.h"

#define HEAP_SIZE 65536

static SEGGER_MEM_CONTEXT     MemCtx;
static SEGGER_MEM_SBUFFER     MemSBuffer;
static U8                     MemBuffer[HEAP_SIZE];

SEGGER_MEM_SBUFFER_Init(&MemCtx, &MemSBuffer, &MemBuffer, HEAP_SIZE);
```

3.1.2 Preparing for compression

A compression operation is prepared by calling `CX_ENCODE_Init()`, passing in:

- An encoding context to initialize
- A compression algorithm to use
- A set of parameters that configure the compression algorithm
- A memory context to allocate working storage for compression

For instance:

```
static CX_ENCODE_CONTEXT    EncCtx;    ❶
static CX_PARAS             Paras;    ❷
static SEGGER_MEM_CONTEXT   MemCtx;    ❸
```


© 2015-2026 SEGGER Microcontroller GmbH

```
SEGGER_MEM_SYSTEM_HEAP_Init(&MemCtx);

CX_PARAS_Clear(&Paras);    ❹
Paras.WindowSize = 32768;    ❺
Paras.MinLen     = 3;    ❻
Paras.MaxLen     = 258;
Paras.Optimize   = 10;       ❼
Paras.BlockLen   = 32768;    ❽

Status = CX_ENCODE_Init(&EncCtx,    ❾
                        &CX_DEFLATE_Encode,
                        &Paras,
                        pMemCtx);
if (Status == 0) {
  printf("Successfully initialized encoder\n");
} else {
  printf("Initializatio of encoder failed: %s\n",
         CX_GetErrorText(Status));
}
```

❶ **Declare encoding context**

The encoding context is defined by the type `CX_ENCODE_CONTEXT`. It contains the encoding state of data presented for compression. The fields within this structure are private and must not be modified. SEGGER makes no guarantee that these fields have the the same name, are present, or have the same interpretation between releases.

❷ **Declare encoding parameters**

The encoding parameters are defined within the structure `CX_PARAS`. SEGGER guarantees the presence and interpretation of these fields between releases. These fields are initialized by the user and are described below.

❸ **Declare memory context**

A memory context is declared and initialized to use the system's heap implementation to provide dynamic memory to emCompress-Pro.

❹ **Initialize compression parameters**

Before setting up compression algorithm parameters, they must be cleared to zero using `CX_PARAS_Clear()`.

❺ **Initialize window size**

This example initializes the compression parameters used by the DEFLATE compressor. The window size specifies the number of characters to look backwards over to find a match and to replace the characters at the cursor with a reference to the match, thereby compressing the incoming data. Larger window sizes provide more data and therefore a larger probability of finding a match and producing a better-compressed output.

In this case the window is initialized to 32768 characters, the largest window that the DEFLATE algorithm supports.

❻ **Initialize match lengths**

This example initializes the match size to between 3 and 258 characters inclusive, the widest possible range for DEFLATE. Adjusting these parameters can improve compression and reduce the size of the compressed output (using longer match lengths), or can improve the speed of compression (by using shorter match lengths).

❼ **Initialize the optimization**

The optimization controls how much effort the compressor puts into finding the best match. In this case, the best possible compression is requested by setting the parameter to 10.

 © 2015-2026 SEGGER Microcontroller GmbH

❽ Initialize block length

DEFLATE is a block-based algorithm rather than a streaming algorithm. Data are collected into a block for compression, compressed, and emitted. Blocks are chained together such that compression can adapt to the changing characteristics of the input stream. In this case, the block size is set to 32768 characters, the maximum for DEFLATE.

❾ Initialize encoder

The encoding process is initialized by calling `CX_ENCODE_Init()`, which return a status code indicating success or failure.

3.1.3 Compressing data

Data are compressed using `CX_ENCODE_Process()`. The general form of compression uses a `CX_STREAM` structure that provides the data to compress and the buffer to place compressed data into. For instance, to read a file, compress the data, and write the output, where the files are already open in binary mode:

```
static int _CompressFile(FILE *pFileIn, FILE *pFileOut) {
  CX_STREAM Stream;
  U8        aIn [1024];
  U8        aOut[1024];
  int       Status;
  int       Eof;
  size_t    NumBytesToWrite;
  size_t    NumBytes;
  //
  memset(&Stream, 0, sizeof(Stream));
  Eof = CX_FLUSH_NONE;
  //
  for (;;) {
    if (Stream.AvailIn == 0) {
      NumBytes = fread(aIn, 1, sizeof(aIn), pFileIn);
      if (feof(pFileIn) != 0) { // end of file
        Eof = CX_FLUSH_END;
      } else if (ferror(pFileIn) != 0) {
        Status = CX_STATUS_GENERAL_ERROR;
        break;
      }
      // Note: Stream.pIn always has to be set to point to the
      //       start of the buffer, since it is moved through
      //       the buffer while data is processed.
      Stream.pIn     = aIn;
      Stream.AvailIn = NumBytes;
    }
    // Note: Stream.pOut always has to be set to point to the
    //       start of the buffer, since it is moved through
    //       the buffer while data is written into it.
    Stream.pOut      = aOut;
    Stream.AvailOut = sizeof(aOut);
    Status = CX_ENCODE_Process(&EncCtx, &Stream, Eof);
    if (Status < 0) {
      break;
    }

    NumBytesToWrite = sizeof(aOut) - Stream.AvailOut;
    NumBytes = fwrite(aOut, 1, NumBytesToWrite, pFileOut);
    if (NumBytes != NumBytesToWrite) {
      Status = CX_STATUS_GENERAL_ERROR;
      break;
    }

    if (Status == CX_STATUS_DONE) {
      // Compressor is done consuming the input data and has
      // flushed its output.
```

 © 2015-2026 SEGGER Microcontroller GmbH

```
      break;
    }
  }
  //
  return Status;
}
```

3.1.4 Decompressing data

Decompressing data has identical form to compressing data, except that whenever “Encode” is used to encode (compress) data, “Decode” is used to decode (decompress) data. Care has to be taken to call the function `CX_DECODE_Process()` until it returns `CX_STATUS_DONE`.

```
static int _DecompressFile(FILE *pFileIn, FILE *pFileOut) {
  CX_STREAM Stream;
  U8        aIn [1024];
  U8        aOut[1024];
  int       Status;
  int       Eof;
  size_t    NumBytesToWrite;
  size_t    NumBytes;
  //
  memset(&Stream, 0, sizeof(Stream));
  Eof = CX_FLUSH_NONE;
  //
  for (;;) {
    if (Stream.AvailIn == 0) {
      NumBytes = fread(aIn, 1, sizeof(aIn), pFileIn);
      if (feof(pFileIn) != 0) { // end of file
        Eof = CX_FLUSH_END;
      } else if (ferror(pFileIn) != 0) {
        Status = CX_STATUS_GENERAL_ERROR;
        break;
      }

      // Note: Stream.pIn always has to be set to point to the start
      //       of the buffer, since it is moved through the buffer
      //       while data is processed.
      Stream.pIn     = aIn;
      Stream.AvailIn = NumBytes;
    }

    // Note: Stream.pOut always has to be set to point to the start
    //       of the buffer, since it is moved through the buffer
    //       while data is written into it.
    Stream.pOut      = aOut;
    Stream.AvailOut = sizeof(aOut);
    Status = CX_DECODE_Process(&DecCtx, &Stream, Eof);
    if (Status < 0) {
      break;
    }

    NumBytesToWrite = sizeof(aOut) - Stream.AvailOut;
    NumBytes = fwrite(aOut, 1, NumBytesToWrite, pFileOut);
    if (NumBytes != NumBytesToWrite) {
      return CX_STATUS_GENERAL_ERROR;
    }

    // Did the stream end?
    if (Status == CX_STATUS_DONE) {
      if (Eof == CX_FLUSH_END) {
        // Stream ended and Eof, done with success.
        return CX_STATUS_DONE;
      }
```

 © 2015-2026 SEGGER Microcontroller GmbH

```
      if (Stream.AvailIn != 0) {
        // Stream ended, but not all input data has been consumed
        return CX_STATUS_GENERAL_ERROR;
      }

      // Eof has not been signalled yet. Read from the file
      // one more time to check whether Eof is returned then
      // or more data is returned (remember: Eof is only returned
      // if one reads beyond the last byte in the file).
    }
  }
  //
  return Status;
}
```

 © 2015-2026 SEGGER Microcontroller GmbH

3.2 Compression of program code

If the content of the data to be compressed is known in advance, it can generally be better compressed by applying a *conditioner* to the data before compression.

Conditioners are available for the following instruction sets:

- Arm T32 (Thumb-2)
- Arm A32 (AArch32)
- Arm A64 (AArch64)
- RISC-V RV32
- Intel IA-32

The following shows how conditioning, for instance, improves the data file `Firmware.input` for all compressors. The utility is described in *Compression algorithm comparison* on page 66.

Without conditioning

```
C:> CX_Util.exe Firmware.input

Input size:    218824 bytes
Conditioner:   None
SMASH-2 flags: None

Window  SMASH2-PRO        LZPJ     DEFLATE        LZMA
------  ----------  ----------  ----------  ----------
   256      104648      131504      100288       83938
   512       94988      120095       93314       78642
  1024       89468      113406       88948       75197
  2048       86497      109440       86270       73499
  4096       84572      106652       84440       72072
  8192       83197      104235       83080       70798
 16384       82755      102896       82516       70009
------  ----------  ----------  ----------  ----------
 Total      626125      788228      618856      524155
------  ----------  ----------  ----------  ----------

C:> _
```

With Arm A32 conditioning

```
C:> CX_Util.exe --a32 Firmware.input

Input size:    218824 bytes
Conditioner:   Arm A32
SMASH-2 flags: None

Window  SMASH2-PRO        LZPJ     DEFLATE        LZMA
------  ----------  ----------  ----------  ----------
   256       99749      123481       94516       79490
   512       89202      111026       86614       73482
  1024       82980      103486       81566       69366
  2048       79484       98985       78402       67153
  4096       77008       95648       76114       65223
  8192       75215       93025       74478       63588
 16384       74437       91731       73816       62483
------  ----------  ----------  ----------  ----------
 Total      578075      717382      565506      480785
------  ----------  ----------  ----------  ----------

C:> _
```

 © 2015-2026 SEGGER Microcontroller GmbH

Additional conditioning

In addition, the SMASH-2 compressor can usually better compress code for any 16-bit or 32-bit instruction set:

```
C:> CX_Util.exe --a32 --fw32 Firmware.input

Input size:    218824 bytes
Conditioner:   Arm A32
SMASH-2 flags: 32-bit instruction set

Window  SMASH2-PRO        LZPJ     DEFLATE        LZMA
------  ----------  ----------  ----------  ----------
   256       96539      123481       94516       79490
   512       86583      111026       86614       73482
  1024       80842      103486       81566       69366
  2048       77730       98985       78402       67153
  4096       75531       95648       76114       65223
  8192       73858       93025       74478       63588
 16384       73345       91731       73816       62483
------  ----------  ----------  ----------  ----------
 Total      564428      717382      565506      480785
------  ----------  ----------  ----------  ----------

C:> _
```

In this case, the SMASH-2 compressor, with a window size of 512 bytes or more compresses better than DEFLATE.

 © 2015-2026 SEGGER Microcontroller GmbH

Chapter 4

emCompress-Pro API

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

© 2015-2026 SEGGER Microcontroller GmbH

4.1 Preprocessor definitions

4.1.1 Version number

Description

Symbol expands to a number that identifies the specific emCompress-Pro release.

Definition

```
#define CX_VERSION    22000
```

Symbols

Definition	Description
CX_VERSION	Internal use.

4.1.2 CX_LZSS_ENCODE_LARGE_WINDOW

Description

Configures LZSS for large window sizes.

This is a helper configuration flag which can be used without specifying an exact window size as is required for CX_LZSS_ENCODE_MAX_WINDOW_SIZE.

Definition

```
#define CX_LZSS_ENCODE_LARGE_WINDOW    0
```

4.1.3 CX_LZSS_ENCODE_MAX_WINDOW_SIZE

Description

Controls the maximum window size supported by the LZSS algorithm.

- ≤ 65260 bytes: In this case, U16 is used to store references to to previous matches in the hash table.
- > 65260 bytes. In this case, U32 is used in the hash table.

Definition

```
#define CX_LZSS_ENCODE_MAX_WINDOW_SIZE    CX_LZSS_MAX_U16_WINDOW_SIZE
```

4.1.4 CX_LZSS_HASH_TABLE_SIZE

Description

Controls the size of the hash table used by the LZSS algorithm.

Depending on the CX_LZSS_ENCODE_MAX_WINDOW_SIZE parameter, each entry in the hash table is either 16-bit or 32-bit large.

For usual window sizes on target hardware, a size of 256 entries has been chosen as a good middle ground between memory consumption and speed.

Allowed values:

- 0 -- Small, 256 entries.
- 1 -- Medium, 1,048,576 entries.
- 2 -- Large, 16,777,216 entries.

© 2015-2026 SEGGER Microcontroller GmbH

Definition

```
#define CX_LZSS_HASH_TABLE_SIZE     0
```

 © 2015-2026 SEGGER Microcontroller GmbH

4.2 Data types

4.2.1 CX_PARAS

Description

Compression parameters.

Type definition

```
typedef struct {
  CX_PARA  WindowSize;
  CX_PARA  MinLen;
  CX_PARA  MaxLen;
  CX_PARA  P1;
  CX_PARA  P2;
  CX_PARA  P3;
  CX_PARA  BlockLen;
  CX_PARA  Optimize;
} CX_PARAS;
```

Structure members

Member	Description
WindowSize	Number of octets in matching window.
MinLen	Minimum length of match.
MaxLen	Maximum length of match.
P1	LC for LZMA, ISA width for SMASH-2.
P2	LP for LZMA.
P3	PB for LZMA.
BlockLen	Number of bytes for one block (DEFLATE only).
Optimize	Optimization level, [0..10], higher values produce better matches but take more time.

 © 2015-2026 SEGGER Microcontroller GmbH

4.2.2 CX_STREAM

Description

Streaming interface.

Type definition

```
typedef struct {
  U32           AvailIn;
  const U8 * pIn;
  U32           AvailOut;
  U8          * pOut;
} CX_STREAM;
```

Structure members

Member	Description
AvailIn	Number of elements available as input octets
pIn	Pointer to available input octets
AvailOut	Number of elements available for output octets
pOut	Pointer to output octets

 © 2015-2026 SEGGER Microcontroller GmbH

4.2.3 CX_ENCODE_CONTEXT

Description

Private encoding context.

Type definition

```
typedef struct {
  void                    * pWork;
  const CX_ENCODE_API     * pAPI;
  SEGGER_MEM_CONTEXT      * pMem;
  CX_BIT_ENCODE_CONTEXT     Bitstream;
  CX_BUFFER                 Block;
  CX_ENCODE_STATE           State;
} CX_ENCODE_CONTEXT;
```

Structure members

Member	Description
pWork	Internal use.
pAPI	Internal use.
pMem	Internal use.
Bitstream	Internal use.
Block	Internal use.
State	Internal use.

 © 2015-2026 SEGGER Microcontroller GmbH

4.2.4 CX_DECODE_CONTEXT

Description

Private decoding context.

Type definition

```
typedef struct {
  void                     * pWork;
  const CX_DECODE_API      * pAPI;
  SEGGER_MEM_CONTEXT       * pMem;
  CX_BIT_DECODE_CONTEXT      Bitstream;
  CX_DECODE_STATE            State;
} CX_DECODE_CONTEXT;
```

Structure members

Member	Description
pWork	Internal use.
pAPI	Internal use.
pMem	Internal use.
Bitstream	Internal use.
State	Internal use.

 © 2015-2026 SEGGER Microcontroller GmbH

4.3 Data definitions

4.3.1 Compression algorithms

Definition

```
extern const CX_ENCODE_API CX_SMASH2_Encode;
extern const CX_ENCODE_API CX_LZPJ_Encode;
extern const CX_ENCODE_API CX_LZMA_Encode;
extern const CX_ENCODE_API CX_DEFLATE_Encode;
```

Description

Each encoding API is passed to `CX_ENCODE_Init()` in order to select the appropriate compression algorithm.

 © 2015-2026 SEGGER Microcontroller GmbH

4.3.2 Decompression algorithms

Definition

```
extern const CX_DECODE_API CX_SMASH2_Decode;
extern const CX_DECODE_API CX_LZPJ_Decode;
extern const CX_DECODE_API CX_LZMA_Decode;
extern const CX_DECODE_API CX_DEFLATE_Decode;
```

Description

Each encoding API is passed to `CX_DECODE_Init()` in order to select the appropriate decompression algorithm.

 © 2015-2026 SEGGER Microcontroller GmbH

4.4 Compression functions

emCompress-Pro defines the following compression functions:

Function	Description
CX_ENCODE_Init()	Initialize encoder.
CX_ENCODE_Process()	Run encoder coroutine.
CX_ENCODE_Exit()	Finalize encoder and deallocate memory.

 © 2015-2026 SEGGER Microcontroller GmbH

4.4.1 CX_ENCODE_Init()

Description

Initialize encoder. The encoder parameters are specific to the encoder being selected.

Prototype

```
int CX_ENCODE_Init(      CX_ENCODE_CONTEXT  * pSelf,
                   const CX_ENCODE_API      * pAPI,
                   const CX_PARAS           * pParas,
                         SEGGER_MEM_CONTEXT * pMem);
```

Parameters

Parameter	Description
pSelf	Pointer to encoder context.
pAPI	Pointer to encoder API.
pParas	Pointer to encoder parameters.
pMem	Pointer to memory allocation context.

Return value

≥ 0 Success.
< 0 Error (no memory, invalid parameters).

 © 2015-2026 SEGGER Microcontroller GmbH

4.4.2 CX_ENCODE_Process()

Description

Run encoder coroutine.

Reads data from the stream's input buffer and writes compressed data into the stream's output buffer. Pass flags = CX_FLUSH_NONE when more input data is still available after the data which is currently in the stream's input buffer. Pass flags = CX_FLUSH_EOF when no more data is available after the data which is left in the buffer.

The function will return CX_STATUS_SUCCESS when the input data was successfully written into the stream's output buffer and more data may be available at the next call to this function.

The function must be called until it returns CX_STATUS_DONE, even if no more input data is available. When CX_STATUS_DONE is returned, the compression stream has been successfully finalized and written to the stream's output buffer.

The pointers to the input/output buffers of the streams are moved through those buffers as the function reads from them and writes to them. This means that when the input buffers is refilled or the data from the output buffer has been consumed by the caller, care must be taken to reset those pointers to the start of the respective buffers.

Prototype

```
int CX_ENCODE_Process(CX_ENCODE_CONTEXT * pSelf,
                      CX_STREAM         * pStream,
                      unsigned            Flags);
```

Parameters

Parameter	Description
pSelf	Pointer to encoder context.
pStream	Pointer to I/O stream.
Flags	Encoding flags, either CX_FLUSH_NONE or CX_FLUSH_EOF when the stream ends.

Return value

= 1 Success encoding data, encode is complete (CX_STATUS_DONE).
= 0 Success encoding data, waiting for more data (CX_STATUS_SUCCESS).
< 0 Error during encoding, current error status (CX_STATUS_...).

 © 2015-2026 SEGGER Microcontroller GmbH

4.4.3 CX_ENCODE_Exit()

Description

Finalize encoder and deallocate memory.

Calling this function multiple times during an exit sequence is allowed.

Prototype

```
void CX_ENCODE_Exit(CX_ENCODE_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to encoder context.

 © 2015-2026 SEGGER Microcontroller GmbH

4.5 Decompression functions

emCompress-Pro defines the following decompression functions:

Function	Description
CX_DECODE_Init()	Initialize decoder.
CX_DECODE_Process()	Run decoder coroutine.
CX_DECODE_Exit()	Finalize decoder and deallocate memory.

 © 2015-2026 SEGGER Microcontroller GmbH

4.5.1 CX_DECODE_Init()

Description

Initialize decoder.

Prototype

```
int CX_DECODE_Init(       CX_DECODE_CONTEXT  * pSelf,
                    const CX_DECODE_API      * pAPI,
                    const CX_PARAS           * pParas,
                          SEGGER_MEM_CONTEXT * pMem);
```

Parameters

Parameter	Description
pSelf	Pointer to decoder context.
pAPI	Pointer to decoder API.
pParas	Pointer to decoder parameters.
pMem	Pointer to memory allocation context.

Return value

≥ 0 Success.
< 0 Error (no memory, invalid parameters).

 © 2015-2026 SEGGER Microcontroller GmbH

4.5.2 CX_DECODE_Process()

Description

Run decoder coroutine.

Reads data from the stream's input buffer and writes decompressed data into the stream's output buffer. Pass flags = CX_FLUSH_NONE when more input data is still available after the data which is currently in the stream's input buffer. Pass flags = CX_FLUSH_EOF when no more data is available after the data which is left in the buffer.

The function will return CX_STATUS_SUCCESS when the input data was successfully written into the stream's output buffer and more data may be available at the next call to this function.

The function must be called until it returns CX_STATUS_DONE, even if no more input data is available. When CX_STATUS_DONE is returned, the compression stream has been successfully consumed and all output has been written to the stream's output buffer.

If there is data in the input buffer after the end of the compressed stream, this function will not consume it, but return CX_STATUS_DONE. It is the responsibility of the caller to decide whether data after the end of the stream constitutes an error condition or whether it can be discarded or processed by other means.

The pointers to the input/output buffers of the streams are moved through those buffers as the function reads from them and writes to them. This means that when the input buffers is refilled or the data from the output buffer has been consumed by the caller, care must be taken to reset those pointers to the start of the respective buffers.

Prototype

```
int CX_DECODE_Process(CX_DECODE_CONTEXT * pSelf,
                      CX_STREAM         * pStream,
                      unsigned            Flags);
```

Parameters

Parameter	Description
pSelf	Pointer to decoder context.
pStream	Pointer to I/O stream.
Flags	Encoding flags, either CX_FLUSH_NONE or CX_FLUSH_EOF when the stream ends.

Return value

= 1 Success decoding bitstream, decode is complete (CX_STATUS_DONE).
= 0 Success decoding bitstream, waiting for more data (CX_STATUS_SUCCESS).
< 0 Decoder is in error, current error status (CX_STATUS_...).

 © 2015-2026 SEGGER Microcontroller GmbH

4.5.3 CX_DECODE_Exit()

Description

Finalize decoder and deallocate memory.

Prototype

```
void CX_DECODE_Exit(CX_DECODE_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to decoder context.

© 2015-2026 SEGGER Microcontroller GmbH

4.6 Conditioning functions

emCompress-Pro defines the following conditioning functions:

Function	Description
CX_PRECOND_T32_Run()	Condition for Arm T32 ISA.
CX_PRECOND_A32_Run()	Condition for Arm A32 ISA.
CX_PRECOND_A64_Run()	Condition for Arm A64 ISA.
CX_PRECOND_RV32_Run()	Condition for RISC-V RV32I ISA.
CX_PRECOND_IA32_Run()	Condition for Intel IA-32 ISA.

© 2015-2026 SEGGER Microcontroller GmbH

4.6.1 CX_PRECOND_T32_Run()

Description

Condition for Arm T32 ISA.

Prototype

```
void CX_PRECOND_T32_Run(      U8   * pOutput,
                        const U8   * pInput,
                              U32    Len,
                              int    Encode);
```

Parameters

Parameter	Description
pOutput	Pointer to object that receives the conditioned output.
pInput	Pointer to object to condition.
Len	Octet length of input and output objects.
Encode	Nonzero to encode, zero to decode.

Additional information

In-place conditioning is supported when pInput is equal to pOutput. Other instances where the input and output overlap result in undefined behavior.

 © 2015-2026 SEGGER Microcontroller GmbH

4.6.2 CX_PRECOND_A32_Run()

Description

Condition for Arm A32 ISA.

Prototype

```
void CX_PRECOND_A32_Run(      U8   * pOutput,
                        const U8   * pInput,
                              U32    Len,
                              int    Encode);
```

Parameters

Parameter	Description
pOutput	Pointer to object that receives the conditioned output.
pInput	Pointer to object to condition.
Len	Octet length of input and output objects.
Encode	Nonzero to encode, zero to decode.

Additional information

In-place conditioning is supported when pInput is equal to pOutput. Other instances where the input and output overlap result in undefined behavior.

 © 2015-2026 SEGGER Microcontroller GmbH

4.6.3 CX_PRECOND_A64_Run()

Description

Condition for Arm A64 ISA.

Prototype

```
void CX_PRECOND_A64_Run(       U8   * pOutput,
                         const U8   * pInput,
                               U32    Len,
                               int    Encode);
```

Parameters

Parameter	Description
pOutput	Pointer to object that receives the conditioned output.
pInput	Pointer to object to condition.
Len	Octet length of input and output objects.
Encode	Nonzero to encode, zero to decode.

Additional information

In-place conditioning is supported when pInput is equal to pOutput. Other instances where the input and output overlap result in undefined behavior.

 © 2015-2026 SEGGER Microcontroller GmbH

4.6.4 CX_PRECOND_RV32_Run()

Description

Condition for RISC-V RV32I ISA.

Prototype

```
void CX_PRECOND_RV32_Run(      U8   * pOutput,
                         const U8   * pInput,
                               U32    Len,
                               int    Encode);
```

Parameters

Parameter	Description
pOutput	Pointer to object that receives the conditioned output.
pInput	Pointer to object to condition.
Len	Octet length of input and output objects.
Encode	Nonzero to encode, zero to decode.

Additional information

In-place conditioning is supported when pInput is equal to pOutput. Other instances where the input and output overlap result in undefined behavior.

 © 2015-2026 SEGGER Microcontroller GmbH

4.6.5 CX_PRECOND_IA32_Run()

Description

Condition for Intel IA-32 ISA.

Prototype

```
void CX_PRECOND_IA32_Run(       U8   * pOutput,
                          const U8   * pInput,
                                U32    Len,
                                int    Encode);
```

Parameters

Parameter	Description
pOutput	Pointer to object that receives the conditioned output.
pInput	Pointer to object to condition.
Len	Octet length of input and output objects.
Encode	Nonzero to encode, zero to decode.

Additional information

In-place conditioning is supported when pInput is equal to pOutput. Other instances where the input and output overlap result in undefined behavior.

 © 2015-2026 SEGGER Microcontroller GmbH

4.7 Utility functions

emCompress-Pro defines the following utility functions:

Function	Description
CX_PARAS_Clear()	Clear all parameters to zero.
CX_GetErrorText()	Get error status as printable string.
CX_GetCopyrightText()	Get copyright as printable string.
CX_GetVersionText()	Get version as printable string.

 © 2015-2026 SEGGER Microcontroller GmbH

4.7.1 CX_PARAS_Clear()

Description

Clear all parameters to zero.

Prototype

```
void CX_PARAS_Clear(CX_PARAS * pParas);
```

Parameters

Parameter	Description
pParas	Pointer to parameters.

 © 2015-2026 SEGGER Microcontroller GmbH

4.7.2 CX_GetErrorText()

Description

Get error status as printable string.

Prototype

```
char *CX_GetErrorText(int Status);
```

Return value

Zero-terminated error string.

 © 2015-2026 SEGGER Microcontroller GmbH

4.7.3 CX_GetCopyrightText()

Description

Get copyright as printable string.

Prototype

```
char *CX_GetCopyrightText(void);
```

Return value

Zero-terminated copyright string.

 © 2015-2026 SEGGER Microcontroller GmbH

4.7.4 CX_GetVersionText()

Description

Get version as printable string.

Prototype

```
char *CX_GetVersionText(void);
```

Return value

Zero-terminated version string.

 © 2015-2026 SEGGER Microcontroller GmbH

Chapter 5

Resource requirements

The memory requirements to compress and decompress data depend entirely on the algorithm chosen and the parameters that the algorithm is instantiated with. Algorithms that take more memory and more time compress better than those that use less memory and less time.

This section details the general memory requirements for compression and decompression.

Memory requirements can be divided into static memory requirements and dynamic memory requirements.

Static memory requirements

Static memory requirements are known at compile time and are required by the emCompress-Pro library irregardless of the selected compression algorithm. The structures listed below require about 150 bytes of static RAM.

Purpose	Structures
Encode/decode contexts	`CX_ENCODE_CONTEXT`, `CX_DECODE_CONTEXT`
Compression parameters	`CX_PARAS`
Algorithm specific API references	`CX_..._ENCODE` / `CX_..._DECODE`
Memory context	`SEGGER_MEM_CONTEXT`, `SEGGER_MEM_SBUFFER`

Dynamic memory requirements

Dynamic memory is allocated by the compression/decompression algorithms at run time using a suitable dynamic memory allocator. See section *Providing dynamic memory* on page 20 for more information on how to provide dynamic memory.

Since the memory requirements depend on many parameters, formulas are provided to calculate estimates for the memory requirements. An Excel sheet, which can calculate the memory requirements based on the compression parameters, is also provided in the shipping package of emCompress-Pro. The actual memory requirements may differ due to word sizes, alignment and memory management overhead. The Excel table takes assumes an alignment of 4 bytes for each block of dynamic memory, while the formulas below do not include alignment for simplicity.

The dynamic memory requirements of the algorithms is the sum of the memory requirements of the LZSS algorithm and the algorithm specific memory requirements. The memory

© 2015-2026 SEGGER Microcontroller GmbH

requirements of the LZSS algorithm for encoding are presented below. The requirements of the algorithms which use the LZSS algorithm are presented in the following sections, including the accumulated requirements for decoding.

LZSS requirements for encoding

The LZSS dynamic memory requirements depend on these parameters:

- Window size (WS): Size of the window in which the algorithm looks for matches.
- Hash table size (HTS): Default: 256. Number of entries in the hash matching tables. Configured via the `CX_LZSS_HASH_TABLE_SIZE` configuration flag.
- Window index size (WIS): Default: 2. Size of the datatype used for indexing the window in bytes. Configured via the `CX_LZSS_ENCODE_MAX_WINDOW_SIZE` configuration flag.
- Maximum match length (MML): Depends on the algorithm, valid ranges are specified in the sections about the respective algorithms.

Description	Formula
LZSS: Window	WS + MML + 2
LZSS: Matcher chain pointers	(WS + MML + 2) * WIS
LZSS: Matcher hash table	HTS * WIS
LZSS Total	HTS * WIS + (WS + MML + 2) * (WIS + 1)

Example: WS = 32768, MML = 258 (for LZPJ), HTS = 256, WIS = 2:

Memory consumption = 256 * 2 + (32768 + 258 + 2) * (2 + 1) = 99596

 © 2015-2026 SEGGER Microcontroller GmbH

5.1 SMASH-2 algorithm

Encoding sizes

The SMASH-2 encoder requires approximately 3.2 kB of ROM when using Arm Thumb-2 instructions.

The following are the dynamic RAM requirements for the SMASH-2 encoder:

Parameter	Allowed range
Window size (WS)	256 to 16384
Maximum match length (MML)	2 to 258

Description	Formula
LZSS total	HTS * WIS + (WS + MML + 2) * (WIS + 1)
SMASH-2 context	116
Total	116 + HTS * WIS + (WS + MML + 2) * (WIS + 1)

Example: WS = 16384, MML = 258, HTS = 256, WIS = 2:

Memory consumption = 116 + 256 * 2 + (16384 + 258 + 2) * (2 + 1) = 50560

Decoding sizes

The SMASH-2 decoder requires approximately 1.3 kB of ROM when using Arm Thumb-2 instructions.

The following are the dynamic RAM requirements for the SMASH-2 decoder:

Description	Formula
LZSS window	(WS + 1)
SMASH-2 context	48
Total	48 + (WS + 1)

Example: WS = 16384

Memory consumption = 48 + (16384 + 1) = 16433

Processing speed

The tables below list typical compression/decompression values reachable with SMASH-2. Tests were done on a Cortex-M7 MCU running at 200 MHz, using internal RAM. The input/output buffer sizes were each 64 KB.

Compression on the target was performed with `CX_LZSS_HASH_TABLE_SIZE`=0, `CX_LZSS_ENCODE_LARGE_WINDOW`=0. Instruction set optimization was off (0). *Ratio* denotes the compression rate, *rate* denotes the achieved input data rate, *memory* denotes the amount of dynamic memory required.

SMASH-2 compression on the target				
Window size	Optimization = 0 Ratio / Rate [kB/s]	Optimization = 2 Ratio / Rate [kB/s]	Optimization = 10 Ratio / Rate [kB/s]	Memory [kB]
256	82.9% / 523.2	70.1% / 523.2	62.9% / 350.5	2.2
512	80.4% / 527.7	61.3% / 476.0	57.9% / 272.4	3.0
1024	79.2% / 531.8	56.3% / 410.6	54.1% / 190.5	4.5
2048	73.1% / 479.0	52.6% / 320.3	51.0% / 121.0	7.6
4096	65.6% / 401.7	49.8% / 225.4	48.5% / 70.7	13.7
8192	61.3% / 306.9	47.8% / 147.9	46.7% / 38.5	26.0
16384	58.7% / 214.4	46.6% / 94.2	45.8% / 29.4	51.0

 © 2015-2026 SEGGER Microcontroller GmbH

For the decompression tests, compressed data was prepared on the PC with `CX_LZSS_HASH_TABLE_SIZE` = 1, `CX_LZSS_ENCODE_LARGE_WINDOW` = 1. Parameters: Instruction set optimization = off, Optimization = 10, default values for minimum and maximum match lengths.

SMASH-2 decompression on the target				
Window size	Ratio	Input rate [kB/s]	Output rate [kB/s]	Dynamic memory [kB]
256	62.9%	904.9	1438.9	0.3
512	57.9%	885.9	1530.2	0.6
1024	54.1%	875.9	1619.7	1.1
2048	51.0%	870.5	1708.2	2.1
4096	48.5%	866.3	1787.8	4.2
8192	46.7%	865.2	1852.7	8.3
16384	45.8%	874.8	1911.3	16.5

 © 2015-2026 SEGGER Microcontroller GmbH

5.2 DEFLATE algorithm

Encoding sizes

The DEFLATE encoder requires approximately 6.2 kB of ROM when using Arm Thumb-2 instructions.

The following are the dynamic RAM requirements for the DEFLATE encoder:

Parameter	Allowed range
Window size (WS)	16 to 32768
Maximum match length (MML)	18 to 258
Block length (BL)	> 0, usually 16384 - 32768

Description	Formula
LZSS total	HTS * WIS + (WS + MML + 2) * (WIS + 1)
DEFLATE context	6124
Block buffer	BL
Total	6124 + BL + HTS * WIS + (WS + MML + 2) * (WIS + 1)

Example: BL = 16384, WS = 16384, MML = 258, HTS = 256, WIS = 2

Memory consumption = 6124 + 16384 + 256 * 2 + (16384 + 258 + 2) * (2 + 1) = 72952

Decoding sizes

The DEFLATE decoder requires approximately 2.5 kB of ROM when using Arm Thumb-2 instructions.

The following are the dynamic RAM requirements for the DEFLATE decoder:

Description	Formula
LZSS window	WS
DEFLATE context	416
Overflow buffer	260
Dynamic Huffman table	2124
Total	2800 + WS

Example: WS = 16384

Memory consumption = 2800 + 16384 = 19184

Processing speed

The tables below list typical compression/decompression values reachable with DEFLATE. Tests were done on a Cortex-M7 MCU running at 200 MHz, using internal RAM. The input/output buffer sizes were each 64 KB.

Compression on the target was performed with `CX_LZSS_HASH_TABLE_SIZE` = 0, `CX_LZSS_ENCODE_LARGE_WINDOW` = 0. Blocksize was set to 16384. *Ratio* denotes the compression rate, *rate* denotes the achieved input data rate, *memory* denotes the amount of dynamic memory required.

DEFLATE compression on the target				
Window size	Optimization = 0 Ratio / Rate [kB/s]	Optimization = 2 Ratio / Rate [kB/s]	Optimization = 10 Ratio / Rate [kB/s]	Memory [kB]
256	57.3% / 323.3	57.3% / 323.3	54.3% / 227.6	24.6
512	56.1% / 324.1	53.4% / 297.4	51.7% / 182.3	25.4
1024	55.3% / 325.6	50.7% / 257.7	49.5% / 129.7	26.9

 © 2015-2026 SEGGER Microcontroller GmbH

2048	51.3% / 295.6	48.5% / 205.2	47.6% / 85.3	30.0
4096	48.8% / 250.7	46.7% / 151.2	46.0% / 53.9	36.1
8192	47.1% / 198.2	45.6% / 108.8	45.2% / 38.4	48.4
16384	46.1% / 149.2	45.2% / 85.0	45.0% / 38.2	73.0
32768	45.5% / 110.8	45.1% / 68.8	45.0% / 38.2	122.1

For the decompression tests, compressed data was prepared on the PC with `CX_LZSS_HASH_TABLE_SIZE` = 1, `CX_LZSS_ENCODE_LARGE_WINDOW` = 1. Parameters: Block length = 32768, optimization = 10, default values for minimum and maximum match lengths.

DEFLATE decompression on the target				
Window size	Ratio	Input rate [kB/s]	Output rate [kB/s]	Dynamic memory [kB]
256	54.3%	353.9	652.3	3.0
512	51.5%	359.6	697.6	3.3
1024	49.3%	366.4	743.9	3.8
2048	47.2%	373.3	791.1	4.8
4096	45.4%	379.7	835.6	6.9
8192	44.3%	385.0	869.4	11.0
16384	43.8%	390.2	891.6	19.2
32768	43.6%	393.4	901.5	35.6

 © 2015-2026 SEGGER Microcontroller GmbH

5.3 LZPJ algorithm

Encoding sizes

The LZPJ encoder requires approximately 2.5 kB of ROM when using Arm Thumb-2 instructions.

The following are the RAM requirements for the LZPJ encoder:

Parameter	Allowed range
Window size (WS)	16 to 32768
Maximum match length (MML)	18 to 258

Description	Formula
LZSS total	HTS * WIS + (WS + MML + 2) * (WIS + 1)
LZPJ context	108
Total	108 + HTS * WIS + (WS + MML + 2) * (WIS + 1)

Example: WS = 32768, MML = 258, HTS = 256, WIS = 2

Memory consumption = 108 + 256 * 2 + (32768 + 258 + 2) * (2 + 1) = 99704

Decoding sizes

The LZPJ decoder requires approximately 1.0 kB of ROM when using Arm Thumb-2 instructions.

The following are the RAM requirements for the LZPJ decoder:

Description	Formula
LZSS window	(WS + 1)
LZPJ context	40
Total	40 + (WS + 1)

Example: WS = 32768

Memory consumption = 40 + (32768 + 1) = 32809

Processing speed

The tables below list typical compression/decompression values reachable with LZPJ. Tests were done on a Cortex-M7 MCU running at 200 MHz, using internal RAM. The input/output buffer sizes were each 64 KB.

Compression on the target was performed with `CX_LZSS_HASH_TABLE_SIZE` = 0, `CX_LZSS_ENCODE_LARGE_WINDOW` = 0. *Ratio* denotes the compression rate, *rate* denotes the achieved input data rate, *memory* denotes the amount of dynamic memory required.

LZPJ compression on the target				
Window size	Optimization = 0 Ratio / Rate [kB/s]	Optimization = 2 Ratio / Rate [kB/s]	Optimization = 10 Ratio / Rate [kB/s]	Memory [kB]
256	82.9% / 594.2	82.9% / 594.2	74.8% / 434.9	2.2
512	80.4% / 603.6	75.8% / 586.1	69.0% / 353.6	3.0
1024	79.2% / 609.8	68.2% / 519.8	64.4% / 250.2	4.5
2048	73.1% / 596.1	63.1% / 412.9	60.5% / 159.0	7.6
4096	65.6% / 512.4	59.3% / 291.7	57.4% / 90.1	13.7
8192	61.3% / 394.2	56.6% / 190.3	55.1% / 47.7	26.0
16384	58.7% / 274.0	55.0% / 118.9	53.9% / 36.9	51.0

 © 2015-2026 SEGGER Microcontroller GmbH

32768	56.8% / 178.9	53.9% / 75.6	53.0% / 24.4	99.8

For the decompression tests, compressed data was prepared on the PC with `CX_LZSS_HASH_TABLE_SIZE` = 1, `CX_LZSS_ENCODE_LARGE_WINDOW` = 1. Parameters: Optimization = 10, default values for minimum and maximum match lengths.

LZPJ decompression on the target				
Window size	Ratio	Input rate [kB/s]	Output rate [kB/s]	Dynamic memory [kB]
256	74.8%	945.9	1265.2	0.3
512	69.0%	942.3	1366.0	0.6
1024	64.4%	943.8	1466.4	1.1
2048	60.5%	948.0	1567.3	2.1
4096	57.4%	954.3	1663.0	4.2
8192	55.1%	962.0	1746.3	8.3
16384	53.9%	973.6	1807.3	16.5
32768	53.0%	989.1	1867.2	32.9

 © 2015-2026 SEGGER Microcontroller GmbH

5.4 LZMA algorithm

Encoding sizes

The LZMA encoder requires approximately 4.2 kB of ROM when using Arm Thumb-2 instructions.

The following are the RAM requirements for the LZMA encoder:

Parameter	Allowed range
Window size (WS)	1 to 16777216
Maximum match length (MML)	2 to 273
Literal context (LC)	0 to 8
Literal position (LP)	0 to 4
Position bits (PB)	0 to 4

Description	Formula
LZSS total	HTS * WIS + (WS + MML + 2) * (WIS + 1)
LZMA context	2608
Range encoder	1536 * 2^(LC + LP) + 24 * 2^PB
Total	2608 + 1536 * 2^(LC + LP) + 24 * 2^PB + HTS * WIS + (WS + MML + 2) * (WIS + 1)

Example: WS = 32768, MML = 273, HTS = 256, WIS = 2, LC = 3, LP = 0, PB = 2

Memory consumption = 2608 + 1536 * 2^(3 + 0) + 24 * 2^2 + 256 * 2 + (32768 + 273 + 2) * (2 + 1) = 114636

Decoding sizes

The LZMA decoder requires approximately 2.5 kB of ROM when using Arm Thumb-2 instructions.

The following are the RAM requirements for the LZMA decoder:

Description	Formula
LZSS window	(WS + 1)
LZMA context	2248
LZMA LC + LP probabilities	1536 * 2^(LC + LP)
LZMA PB probabilities	160 * 2^PB
Total	2248 + 1536 * 2^(LC + LP) + 160 * 2^PB + (WS + 1)

Example: WS = 32768, LC = 3, LP = 0, PB = 2

Memory consumption = 2248 + 1536 * 2^(3 + 0) + 160 * 2^2 + (32768 + 1) = 47945

Processing speed

The tables below list typical compression/decompression values reachable with LZPJ. Tests were done on a Cortex-M7 MCU running at 200 MHz, using internal RAM. The input/output buffer sizes were each 64 KB.

Compression on the target was performed with `CX_LZSS_HASH_TABLE_SIZE` = 0, `CX_LZSS_ENCODE_LARGE_WINDOW` = 0. Further parameters: LC = 3, LP = 0, PB = 2. *Ratio* denotes the compression rate, *rate* denotes the achieved input data rate, *memory* denotes the amount of dynamic memory required.

LZMA compression on the target

Window size	Optimization = 0 Ratio / Rate [kB/s]	Optimization = 2 Ratio / Rate [kB/s]	Optimization = 10 Ratio / Rate [kB/s]	Memory [kB]
256	53.9% / 365.6	53.9% / 365.6	51.4% / 270.8	17.1
512	52.9% / 367.3	50.7% / 340.9	49.1% / 217.9	17.9
1024	52.2% / 368.7	48.2% / 300.7	47.0% / 155.7	19.5
2048	48.6% / 340.9	45.9% / 245.7	45.0% / 101.3	22.5
4096	46.0% / 296.6	43.9% / 181.4	43.2% / 59.0	28.7
8192	44.0% / 237.0	42.3% / 122.0	41.7% / 31.3	41.0
16384	42.6% / 171.6	41.2% / 77.0	40.8% / 21.4	65.5
32768	41.4% / 114.9	40.3% / 49.0	39.9% / 22.5	114.7

For the decompression tests, compressed data was prepared on the PC with `CX_LZSS_HASH_TABLE_SIZE` = 1, `CX_LZSS_ENCODE_LARGE_WINDOW` = 1. Parameters: Optimization = 10, LC = 3, LP = 0, PB = 2, default values for minimum and maximum match lengths.

LZMA decompression on the target				
Window size	Ratio	Input rate [kB/s]	Output rate [kB/s]	Dynamic memory [kB]
256	51.4%	440.7	857.7	15.5
512	49.1%	449.2	915.2	15.7
1024	47.0%	457.6	973.2	16.3
2048	45.0%	463.3	1029.4	17.3
4096	43.2%	468.4	1085.0	19.3
8192	41.7%	474.5	1137.6	23.4
16384	40.8%	480.5	1178.2	31.6
32768	39.9%	485.6	1215.5	48.0
65536	39.5%	489.4	1240.1	80.8

 © 2015-2026 SEGGER Microcontroller GmbH

Chapter 6

Utilities

This section presents the utilities shipped with emCompress-Pro:

- `CX_Optimize`, a utility to benchmark the compression achieved by algorithms for a particular input.
- `CX_Tool`, a utility to compress and decompress files.

© 2015-2026 SEGGER Microcontroller GmbH

6.1 Compression algorithm comparison

emCompress-Pro ships with an application which compares the compression performance of included algorithms, `CX_Optimize`.

6.1.1 Command line and options

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

The command line syntax is:

```
CX_Optimize [options] <inputfile>
```

Options

Option	Description
`--help`	Displays the help message.
Common options	
`--algo=<algo>`	Determine optimum for `<algo>`. Default: `all`, other allowed values: `deflate`, `lmza`, `lzpj`, `smash2`.
`--ws=<win-size>`	Use a window size of <win-size> bytes
`--wsmin=<min-size>`	Start iteration over window sizes at <min-size>.
`--wsmax=<max-size>`	Stop iteration over window sizes at <max-size>.
SMASH-2 options	
`--fw=auto`	Find the optimum option for the instruction set.
`--fw=16`	Assume program code using 16-bit instruction set.
`--fw=32`	Assume program code using 32-bit instruction set.
DEFLATE options	
`--block-size=<block-size>`	Set the block size. Default: 32768
LZMA options	
`--lcmin=<lc-min>`	Set the minimum value of LC to try.
`--lcmax=<lc-max>`	Set the maximum value of LC to try.
`--lpmin=<lp-min>`	Set the minimum value of LP to try.
`--lpmax=<lp-max>`	Set the maximum value of LP to try.
`--pbmin=<pb-min>`	Set the minimum value of PB to try.
`--pbmax=<pb-max>`	Set the maximum value of PB to try.
`--xz-compat`	Set XZ-tool compatibility mode (LC + LP ≤ 4).
Conditioning options	
`--t32`	Use Arm T32 conditioner.
`--a32`	Use Arm A32 conditioner.
`--a64`	Use Arm A64 conditioner.
`--rv32`	Use RISC-V RV32 conditioner.
`--ia32`	Use Intel IA-32 conditioner.

6.1.2 Using the CX_Optimize tool

The `CX_Optimize` tool, when called with only a filename as parameter, lists the compression efficiencies for all algorithms. It iterates over a range of common window sizes. Results are only listed for those algorithms for which the window size is valid.

 © 2015-2026 SEGGER Microcontroller GmbH

```
File:           .\Firmware.input
Input size:     218824 bytes
Conditioner:    None
SMASH-2 flags:  None
DEFLATE flags:  Block size: 32768
LZMA flags:     LC: 3, LP: 0, PB: 2

Window    SMASH2                LZPJ              DEFLATE           LZMA
--------  --------------------  ----------------  ----------------  -------------------------------
     256      104648  47.8% I=0      131504  60.1%      100288  45.8%       83938  38.4% LC=3,LP=0,PB=2
     512       94988  43.4% I=0      120095  54.9%       93314  42.6%       78642  35.9% LC=3,LP=0,PB=2
    1024       89468  40.9% I=0      113406  51.8%       88948  40.6%       75197  34.4% LC=3,LP=0,PB=2
    2048       86497  39.5% I=0      109440  50.0%       86270  39.4%       73499  33.6% LC=3,LP=0,PB=2
    4096       84572  38.6% I=0      106652  48.7%       84440  38.6%       72072  32.9% LC=3,LP=0,PB=2
    8192       83197  38.0% I=0      104235  47.6%       83080  38.0%       70798  32.4% LC=3,LP=0,PB=2
   16384       82755  37.8% I=0      102896  47.0%       82516  37.7%       70009  32.0% LC=3,LP=0,PB=2
   32768                     -       102401  46.8%       82371  37.6%       69460  31.7% LC=3,LP=0,PB=2
   65260                     -                   -                   -       69072  31.6% LC=3,LP=0,PB=2
```

6.1.2.1 Optimizing SMASH-2 compression

The `CX_Optimize` tool can optimize compression using the SMASH-2 algorithm by using an optimization which is specific to the width of the employed instruction set. By specifying **--fw=auto**, the tool will determine the optimum setting automatically. Below, an invocation with **--fw=auto** determines that the file is best compressed with an optimization for 32-bit wide instructions:

```
CX_Optimize --algo=smash2 --fw=auto ./Firmware.input
```

```
File:           .\Firmware.input
Input size:     218824 bytes
Conditioner:    None
SMASH-2 flags:  Try all (I=0: none, I=1: 16-bit, I=2: 32-bit)

Window    SMASH2
--------  --------------------
     256      101407  46.3% I=2
     512       92305  42.2% I=2
    1024       87250  39.9% I=2
    2048       84655  38.7% I=2
    4096       82984  37.9% I=2
    8192       81701  37.3% I=2
   16384       81466  37.2% I=2
   32768                     -
   65260                     -
```

6.1.2.2 Optimizing LZMA compression

LZMA compression can be optimized by adjusting the LC, LP and PB parameters. The default value of LC = 3 is well suited for text, since it chooses the probability trackers based on the upper three bits of the character. This means that the probability trackers can adapt to different probabilities of characters following lower- and upper-case characters. For firmware code, setting LP = 2 can lead to better results, since the probability trackers will then be based on the position of the currect byte instead.

If compatibility with the XZ tool of the LZMA suite is desired, the **--xz-compat** parameter should be specified to restrict the sum of LC and LP to the range allowed by the XZ tool.

Invocing the `CX_Optimize` tool on the 32-bit instruction set firmware image used above, the best setting for the LP parameter is determined to be 2:

```
CX_Optimize --algo=lzma --lcmin=0 --lcmax=4 --lpmin=0 --lpmax=4 --xz-compat --pbmin=0 --pbmax=4 --wsmin=8192 --wsmax=32768 ./Firmware.input
```

 © 2015-2026 SEGGER Microcontroller GmbH

```
File:           .\Firmware.input
Input size:     218824 bytes
Conditioner:    None
LZMA flags:     LC: 0 - 4, LP: 0 - 4, PB: 0 - 4

Window    LZMA
--------  --------------------------------
    8192       68724  31.4% LC=1,LP=2,PB=2
   16384       67966  31.1% LC=1,LP=2,PB=2
   32768       67435  30.8% LC=1,LP=2,PB=2
```

When running `CX_Optimize` on a 16-bit firmware image, the best setting for LP is determined to be 1:

```
File:           .\FirmwareThumb2.input
Input size:     104768 bytes
Conditioner:    None
LZMA flags:     LC: 0 - 4, LP: 0 - 4, PB: 0 - 4

Window    LZMA
--------  --------------------------------
    8192       71014  67.8% LC=3,LP=1,PB=1
   16384       70514  67.3% LC=3,LP=1,PB=1
   32768       69923  66.7% LC=3,LP=1,PB=1
```

Applying the same command to an HTML file determines the following optimum settings:

```
File:           .\HTML.input
Input size:     324751 bytes
Conditioner:    None
LZMA flags:     LC: 0 - 4, LP: 0 - 4, PB: 0 - 4

Window    LZMA
--------  --------------------------------
    8192       40007  12.3% LC=4,LP=0,PB=0
   16384       38819  12.0% LC=4,LP=0,PB=0
   32768       37796  11.6% LC=4,LP=0,PB=0
```

 © 2015-2026 SEGGER Microcontroller GmbH

6.2 Compression/decompression utility

This section describes the compression/decompression utility which ships with emCompress-Pro, `CX_Tool`.

6.2.1 Command line and options

The command line syntax is:

`CX_Tool [options] inputfile outputfile`

Options

Option	Description
--help	Displays the help message.
-c <algo>	Compress using <algo>: deflate, lzma, lzpj or smash2.
-d <algo>	Decompress using <algo>: deflate, lzma, lzpj or smash2.
Common options	
--ws=<win-size>	Use a window size of <win-size> bytes.
--opt=<opt>	Set optimization level (0-10), default: 10.
--minlen=<min-len>	Set minimum match length.
--maxlen=<max-len>	Set maximum match length.
SMASH-2 options	
--isa=auto	Find the optimum option for the instruction set.
--isa=1	Assume program code using 16-bit instruction set.
--isa=2	Assume program code using 32-bit instruction set.
DEFLATE options	
--block-size=<block-size>	Set the block size. Default: 32768
LZMA options	
--lc=<lc>	Set the value of LC.
--lp=<lp>	Set the value of LP.
--pb=<pb>	Set the value of PB.

6.2.2 Using the CX_Tool utility

This section provides examples of usage of the `CX_Tool` utility.

The following command compresses the file `Firmware.input` using the LZMA algorithm and stores it in the file `Firmware.lzma`:

```
CX_Tool -c lzma --lc=1 --lp=2 --pb=2 --ws=32768 --opt=10
    Firmware.input Firmware.lzma
```

The following command decompresses the file which was compressed above:

```
CX_Tool -d lzma --lc=1 --lp=2 --pb=2 --ws=32768
    Firmware.lzma Firmware.output
```

 © 2015-2026 SEGGER Microcontroller GmbH

Chapter 7

Glossary

Bitstream
A sequence of bits read on bit-by-bit basis.

Codec
Coder-decoder. A device or algorithm capable of coding or decoding a digital data stream. A lossless compressor and decompressor combination constitutes a codec.

Compressor
An algorithm that attempts to find redundancy in data and remove that redundancy thereby compressing the data to use fewer bits.

Decompressor
An algorithm that reverses the effect of compression and recovers the original data from the encoded bitstream.

ISA
Instruction Set Architecture. Defines the instruction set, registers and memory access models which an MCU supports.

KB
Kilobyte. Defined as either 1,024 or 1,000 bytes by context. In the microcontroller world and this manual it is understood to be 1,024 bytes and is routinely shortened further to “K” when describing microcontroller RAM or flash sizes.

LZMA
Lempel-Ziv-Markov chain algorithm. A compression algorithm which combines LZSS with a sophisticated range encoder.

LZPJ
Lempel-Ziv “Plain Jane” algorithm. An algorithm which provides a bitstream to encode LZSS compressed data.

LZSS
Lempel-Ziv-Storer-Szymanski. A compression scheme that is based on LZ77.

SMASH
Small Microcontroller Advanced Super-High format. SEGGER’s proprietary format for compressing data.

XZ tool
A tool which is part of the XZ Utils which can handle LZMA compression/decompression.
Website: *https://tukaani.org/xz/*

© 2015-2026 SEGGER Microcontroller GmbH

Chapter 8

Indexes

© 2015-2026 SEGGER Microcontroller GmbH

8.1 Index of functions

CX_DECODE_Exit, **43**
CX_DECODE_Init, **41**
CX_DECODE_Process, 23, **42**
CX_ENCODE_Exit, **39**
CX_ENCODE_Init, 21, **37**
CX_ENCODE_Process, 22, **38**
CX_GetCopyrightText, **53**
CX_GetErrorText, 21, **52**
CX_GetVersionText, **54**
CX_PARAS_Clear, 21, **51**
CX_PRECOND_A32_Run, **46**
CX_PRECOND_A64_Run, **47**
CX_PRECOND_IA32_Run, **49**
CX_PRECOND_RV32_Run, **48**
CX_PRECOND_T32_Run, **45**

 © 2015-2026 SEGGER Microcontroller GmbH

8.2 Index of types

CX_DECODE_CONTEXT, **33**
CX_ENCODE_CONTEXT, 20, **32**
CX_PARAS, 20, **30**
CX_STREAM, 22, 23, **31**

 © 2015-2026 SEGGER Microcontroller GmbH