emCompress

Compression system

User Guide & Reference Manual

Document: UM17001
Software Version: 2.12
Revision: 0
Date: July 21, 2015

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER Microcontroller GmbH & Co. KG (SEGGER) assumes no responsibility for any errors or omissions. SEGGER makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. SEGGER specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

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 & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

Tel. +49 2103-2878-0
Fax. +49 2103-2878-28
E-mail: support@segger.com
Internet: www.segger.com

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: July 21, 2015

Software	Revision	Date	By	Description
2.12	0	150721	JL	Updated references to SEGGER_CRC.
2.10	0	150706	PC	First release.
1.00	0	141213	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.

If you feel that your knowledge of C is not sufficient, we recommend *The C Programming Language* by Kernighan and Richie (ISBN 0–13–1103628), which describes the standard in C programming and, in newer editions, also covers the ANSI C standard.

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.
`Keyword`	Text that you enter at the command prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table of contents

1 Introduction to emCompress 9
1.1 What is emCompress? 10
1.2 Features 11
1.3 Recommended project structure 12
1.4 Package content 13
1.4.1 Include directories 13

2 Using emCompress 15
2.1 Running emCompress 16
2.1.1 Selected compressor 16
2.1.2 Compressor efficiency 16
2.1.3 Decompressor memory 16
2.1.4 Integrity checks 16
2.1.5 Compressed output 17
2.2 Single-file walkthrough 18
2.2.1 Compress the file 18
2.2.2 Call the decompressor 18
2.2.3 Process the decompressed output 19
2.2.4 Compile and test 19
2.2.5 Finished example 20
2.3 Decompression into memory 21
2.4 Defensive decompression 22
2.4.1 Using built-in integrity checks 22
2.4.2 Extra-defensive decompression 23
2.5 Limiting decompressor memory 24
2.6 Adjusting compression performance 26
2.6.1 Speeding up compression 26
2.6.2 Increasing the compression ratio 26
2.7 Batch compression 28
2.7.1 Compressing web server content 28
2.7.2 Applying limits to all files 29
2.7.3 Tuning compression 30
2.8 Group compression 32
2.8.1 What is group compression? 32
2.8.2 Compromises with group compression 33
2.8.3 Group compression walkthrough 33
2.9 Dynamic workspace allocation 35
2.10 Command line options 36
2.10.1 Add codec (-A) 36
2.10.2 Group compression (-g) 36

2.10.3 List performance (-l) 36
2.10.4 Decompressor memory limit (-m) 36
2.10.5 Dry run (-n) 36
2.10.6 Optimization level (-O) 37
2.10.7 Summarize (-s) 37
2.10.8 Verbose (-v) 37
2.10.9 Exclude codec (-X) 37
2.10.10 Exit (--exit) 37
2.10.11 Output directory (--outdir) 38

3 API reference 39

3.1 Core functions 40
3.1.1 COMPRESS_DecompressThruFunc() 41
3.1.2 COMPRESS_DecompressToMem() 42
3.1.3 COMPRESS_QueryEncodedData() 43
3.1.4 COMPRESS_QueryEncodedDataCRC() 44
3.1.5 COMPRESS_QueryEncodedDataSize() 45
3.1.6 COMPRESS_QueryDecodedDataCRC() 46
3.1.7 COMPRESS_QueryDecodedDataSize() 47
3.1.8 COMPRESS_QueryWorkspaceSize() 48

4 Resource use 49

4.1 General comments 50
4.2 Memory footprint 51
4.2.1 Target system configuration 51
4.2.2 RAM use 51
4.2.3 ROM use 51
4.2.4 Integrity check overhead 51

5 Frequently asked questions 53

6 Reference 55

6.1 Unabridged sample output 56
6.2 Compression algorithms 58
6.2.1 STORE codec 58
6.2.2 HUFF codec 58
6.2.3 LZW codec 58

7 Glossary 61

Chapter 1

Introduction to emCompress

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

1.1 What is emCompress?

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

- Clean ISO/ANSI C source code.
- Small decompressor ROM footprint.
- Completely tunable decompressor RAM footprint.
- Wide range of codecs to choose from.
- Automatic selection of best codec for given memory footprint.
- Easy-to-understand and simple-to-use API.
- Group mode compression capability boosts small file compression ratios.
- Complete support for built-in data integrity checks.
- Simple configuration.
- Royalty free.

1.3 Recommended project structure

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

WARNING: When updating to a newer emCompress 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:

Files	Description
Config	Configuration header files.
Doc	emCompress documentation.
COMPRESS	emCompress decompressor source code.
SEGGER	SEGGER software component source code used in emCompress.
Tool	Supporting applications in binary form.

1.4.1 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 if you have old files included and therefore mix different versions. If you keep emCompress 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 to updating.

Chapter 2

Using emCompress

emCompress divides into two parts:

- A compressor application that is responsible for choosing an appropriate compressor and decompressor pair (a *codec*), and
- A set of decompressor sources that are integrated into your application along with the output of the compressor application.

emCompress offers a range of options to control compression schemes and decompression resources. This following sections explain how to use emCompress to deploy compressed content in your application by:

- Compressing a single file.
- Compressing multiple files.
- Restricting the decoder to work in a predetermined fixed-size workspace.
- Restricting the compression algorithms to choose from.

2.1 Running emCompress

To compress a single file, run emCompress on that file. Here we compress a very small example FPGA configuration bitstream to demonsrate the savings that emCompress can make in firmware images:

```
C:> emCompress.exe FPGA.rbf

(c) 2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress V2.10 compiled Jun 23 2015 17:16:26

Input File:    FPGA.rbf
Optimization:  Level 5 (Balanced)
Restriction:   None (assume unlimited decompressor RAM)
Codec:         DEFLATE(32k,3,258) chosen from 125 candidates
Decoding:      35248 bytes required for decompression
Compression:   20.6% (79.4% of original removed)
- Sizes        114557 -> 23643 bytes
Output File:   Compressed_FPGA.c
Elapsed time:  3.788 s

C:> _
```

2.1.1 Selected compressor

emCompress runs through its internal list of compressors and chooses the one that gives the best compression for the input file. In this example, it ran through 126 candidate compressors and chose the DEFLATE codec, shown by the line:

```
Codec:         DEFLATE(32k,3,258) chosen from 125 candidates
```

The numbers in parentheses after DEFLATE parameterize the way the DEFLATE codec operates, and primarily tune the memory requirements of the decompressor.

2.1.2 Compressor efficiency

The compression achieved by emCompress is indicated by the lines:

```
Compression:   20.6% (79.4% of original removed)
- Sizes        114557 -> 23643 bytes
```

This shows that the original file compresses to 20.6% of its original size, reducing it from 114,557 bytes to 23,548 bytes and removing 91,009 bytes overall. There will, of course, be an overhead for the decompression code, but typically the decompressor ROM footprint is a few hundred bytes to just over 2 KB, depending on the codec chosen.

2.1.3 Decompressor memory

When decompressing the bitstream on embedded devices, it's essential to know how much memory is required. emCompress calculates the memory needed by the decoder exactly:

```
Decoding:      35248 bytes required for decompression
```

Clearly, some devices won't have 35 KB of memory to devote to decompression, and emCompress has the capability to resolve this—which we discuss later.

2.1.4 Integrity checks

To guarantee the bitstream is correctly decompressed, emCompress includes a CRC of the compressed bitstream and a CRC of the original input which can be (optionally) checked when decompressing. emCompress shows the CRC of the original image followed by the CRC of the compressed image when invoked with the `-v` option:


```
Encoding:      33659168 bytes used during compression
- Speed        507 ns/byte (58089 us total time) yields 1.88 MB/s
- Image CRC    821B9442 (polynomial is 04C11DB7, start with FFFFFFFF)
Decoding:      35232 bytes required for decompression (258 for window)
- Speed        14 ns/byte (1679 us total time) yields 65.07 MB/s
- Image CRC    A501DFC8 (polynomial is 04C11DB7, start with FFFFFFFF)
```

2.1.5 Compressed output

Finally, the output of the compression step results in a source file that you include into your application:

```
Output File:   Compressed_FPGA.c
```

This file contains the compressed bitstream together with a data structure that controls how the file is decompressed.

2.2 Single-file walkthrough

Integrating a compressed file into your application is straightforward:

- Compress the file with emCompress specifying any restrictions on decompressor memory and codecs to use;
- Write a function that processes the decoded file a chunk at a time; and
- Call the decompressor providing the file to decode and some working storage.

In this example we will compress a small text file, `Jabberwocky.txt`, and write an application that prints the output. The example files used here are provided in the `Example` folder.

2.2.1 Compress the file

First, compress the file with emCompress:

```
C:> emCompress.exe Jabberwocky.txt

(c) 2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress V2.10 compiled Jun 23 2015 17:16:26

Input File:    Jabberwocky.txt
Optimization:  Level 5 (Balanced)
Restriction:   None (assume unlimited decompressor RAM)
Codec:         DEFLATE(2k,3,258) chosen from 125 candidates
Decoding:      4472 bytes required for decompression
Compression:   48.0% (52.0% of original removed)
- Sizes        1089 -> 523 bytes
Output File:   Compressed_Jabberwocky.c
Elapsed time:  1.432 s

C:> _
```

Examining the output file `Compressed_Jabberwocky.c` reveals the encoded bitstream:

```
static const U8 _aEncodedData[523] = {
  0xCC, 0x53, 0x4B, 0x8E, 0xDB, 0x30, 0x0C, 0xDD, …
```

And the control structure for decompression:

```
const COMPRESS_ENCODED_FILE Compressed_Jabberwocky = {
  _Jabberwocky__aBitstream,
  523,
  0x5044A8E3,
  &COMPRESS_DEFLATE_Decode,
  0,
  1089,
  0xA33EE9AD,
  4472,
  { 2048, 3, 258 }
};
```

The control structure is all that you need for decompression, the details enclosed within it are for the decompressor.

Note: The complete content of the generated file `Compressed_Jabberwocky.c` is presented in *Unabridged sample output* on page 56.

2.2.2 Call the decompressor

From emCompress's output, we read off that the workspace requirement is 4,456 bytes, so that is allocated as static data. The compressed file control structure, `Compressed_Jabberwocky` is provided to the decompressor together with a function that will receive the decompressed data for processing (`_PrintData`), and the workspace for decompression. Optionally you can provide a user context and a CRC verification function, but we omit that complication here and pass in zeroes to signal their absence:


```
void main(void) {
  static union {
    U8  Bytes[4472];  // Workspace reported by emCompress
    U32 Long;         // Force long alignment of workspace.
  } Workspace;
  int Status;
  //
  Status = COMPRESS_Decompress(&Compressed_Jabberwocky,
                               &Workspace, sizeof(Workspace),
                               _PrintData, 0,
                               0);
  if (Status >= 0) {
    printf("\nDecompressed %d bytes.\n", Status);
  } else {
    printf("\nDecompression error.\n");
  }
}
```

The value returned will be zero or positive to indicate successful decompression.

One interesting point to note is that the workspace must be aligned correctly for the target processor. For ARM and other 32-bit processors this generally means that the workspace must be aligned on a 32-bit boundary. You can ensure this by allocating a workspace using an array of `unsigned` or by using compiler-dependent pragmas or extensions. However, in this case we force alignment of the 4,472 bytes of workspace by combining it an `unsigned` in a union which forces correct alignment of the workspace to a 32-bit boundary and needs no compiler extensions.

2.2.3 Process the decompressed output

When the decoder has filled its local buffer with decompressed data, it passes that data to the user's handling function for further processing. The handling function receives the user context passed through emCompress's API, along with a pointer to the decompressed data and its size.

```
static int _PrintData(void *pUserContext, void *pData, unsigned NumBytesData) {
  return printf("%.*s", NumBytesData, pData);
}
```

The value returned by the handling function must be zero or positive to continue decompressing; negative values returned by the handler indicate a processing error in the handling function and immediately terminate decoding of the bitstream. That value is passed up and returned by the decompression function — `COMPRESS_Decompress` in this instance.

Note: It is important to stress that the handling function will be called multiple times, in general, with decompressed fragments of varying lengths as the decoder works it way through the compressed bitstream.

2.2.4 Compile and test

To build the program, compile `Decompress.c` with the emCompress source code in `COMPRESS` and SEGGER software components in `SEGGER`. After it's compiled and linked without errors, running the example prints the original input:

```
C:> Ex1.exe
Jabberwocky
  BY LEWIS CARROLL


'Twas brillig, and the slithy toves
      Did gyre and gimble in the wabe:
All mimsy were the borogoves,
      And the mome raths outgrabe.

"Beware the Jabberwock, my son!
```



```
      The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
      The frumious Bandersnatch!"


Decompressed 1089 bytes.

C:> _
```

2.2.5 Finished example

Here is the complete example:

```
// File: Ex1.c
// - Decompress encoded file in stream mode.
//

#include "COMPRESS.h"
#include "Compressed_Jabberwocky.c"
#include <stdio.h>

static int _PrintData(void *pUserContext, void *pData, unsigned NumBytesData) {
  return printf("%.*s", NumBytesData, pData);
}

void main(void) {
  static union {
    U8  Bytes[4472];  // Workspace reported by emCompress
    U32 Long;         // Force long alignment of workspace.
  } Workspace;
  int Status;
  //
  Status = COMPRESS_DecompressThruFunc(&Compressed_Jabberwocky,
                                       &Workspace, sizeof(Workspace),
                                       _PrintData, 0,
                                       0, ~0UL,
                                       0);
  if (Status >= 0) {
    printf("\nDecompressed %d bytes.\n", Status);
  } else {
    printf("\nDecompression error.\n");
  }
}
```


2.3 Decompression into memory

If there is enough RAM available to hold all decompressed content, you can use an all-at-once decompression function. Continuing with the previous example, emCompress indicates that the original content is 1,089 bytes so, when decompressed, that is what we need:

```
// File: Ex2.c
// - Decompress encoded file in all-at-once mode.
//

#include "COMPRESS.h"
#include "Compressed_Jabberwocky.c"
#include <stdio.h>

void main(void) {
  static union {
    U8  Bytes[4472];        // Workspace reported by emCompress
    U32 Long;               // Force long alignment of workspace.
  } Workspace;
  static U8 aOutput[1089];  // Output reported by emCompress
  int       Status;
  //
  Status = COMPRESS_DecompressToMem(&Compressed_Jabberwocky,
                                    &Workspace, sizeof(Workspace),
                                    aOutput,
                                    0, ~0UL,
                                    0);
  if (Status >= 0) {
    printf("Decompressed %d bytes.\n\n", Status);
    printf("%.*s", Status, aOutput);
  } else {
    printf("Decompression error.");
  }
}
```


2.4 Defensive decompression

Applications may wish to verify the integrity of the input and output bitstreams. This may be particularly appropriate when programming FPGA devices with a configuration bitstream.

2.4.1 Using built-in integrity checks

To verify the CRC is straighforward: replace the call to the plain decompression function with a call to the function with builtin verification.

```
// File: Ex3.c
// - Decompress encoded file in stream mode with CRC check.
//

#include "COMPRESS.h"
#include "SEGGER_CRC.h"
#include "Compressed_Jabberwocky.c"
#include <stdio.h>

static int _PrintData(void *pUserContext, void *pData, unsigned NumBytesData) {
  return printf("%.*s", NumBytesData, pData);
}

void main(void) {
  static union {
    U8  Bytes[4472];  // Workspace reported by emCompress
    U32 Long;         // Force long alignment of workspace.
  } Workspace;
  int Status;
  //
  Status = COMPRESS_DecompressThruFunc(&Compressed_Jabberwocky,
                                       &Workspace, sizeof(Workspace),
                                       _PrintData, 0,
                                       0, ~0UL,
                                       SEGGER_CRC_Calc_04C11DB7);
  if (Status >= 0) {
    printf("\nDecompressed %d bytes.\n", Status);
  } else {
    printf("\nDecompression error.\n");
  }
}
```

The function SEGGER_CRC_Calc_04C11DB7 implements the CRC-32 check. The directory SEGGER contains the code for this function.

The verification functions perform two CRC checks, on the compressed and uncompressed bitstreams, to ensure data integrity:

- Before decompression, the compressed bitstream’s stored CRC is checked against a newly computed CRC calculated over the compressed bitstream. If the CRCs do not match, indicating a failure in the integrity of the comprssed bitstream, the bitstream is not decompressed and emCompress signals a decompression failure.
- If the compressed bitstream is intact, the bitstream is decompressed, passing the decompressed output to the application, and a running CRC is maintained. At the end of compression, if the uncompressed bitstream’s stored CRC does not match the calculated CRC, emCompress signals a decompression failure.

The two CRC checks are made as it is very difficult to detect errors in compressed bitstreams during decompression: checking the compressed bitstream’s integrity before decompressing ensures that the decompressors are presented with a clean bitstream. Checking that the decompressed output matches what is expected gives an extra level of assurance that the decompression algorithm executed correctly and has not suffered data corruption during decompression.

2.4.2 Extra-defensive decompression

If you are decompressing in stream mode and working with a decompressed bitstream that requires absolute integrity, you may wish to ensure that both compressed and decompressed bitstreams are intact before processing the bitstream. It could be, for instance, that you need to ensure that a bitstream sent to an FPGA for configuration is correct and the CRC check at the end of decompression is simply too late.

In this case, you can perform a "dry run" decompression before configuring the FPGA. To do this, run the decompression that you intend to run, but specify a null function pointer such that no data is handed to your processing function:

```
// File: Ex4.c
// - Decompress encoded file in stream mode with preflight CRC check.
//

#include "COMPRESS.h"
#include "SEGGER_CRC.h"
#include "Compressed_Jabberwocky.c"
#include <stdio.h>

static int _PrintData(void *pUserContext, void *pData, unsigned NumBytesData) {
  return printf("%.*s", NumBytesData, pData);
}

void main(void) {
  static union {
    U8  Bytes[4472];  // Workspace reported by emCompress
    U32 Long;         // Force long alignment of workspace.
  } Workspace;
  int Status;
  //
  // Run preflight checks checks.
  //
  Status = COMPRESS_DecompressThruFunc(&Compressed_Jabberwocky,
                                       &Workspace, sizeof(Workspace),
                                       0, 0,
                                       0, ~0UL,
                                       SEGGER_CRC_Calc_04C11DB7);
  if (Status >= 0) {
    //
    // Preflight OK...
    //
    printf("Preflight checks passed: decompress for real.\n\n");
    Status = COMPRESS_DecompressThruFunc(&Compressed_Jabberwocky,
                                         &Workspace, sizeof(Workspace),
                                         _PrintData, 0,
                                         0, ~0UL,
                                         0);
  }
  //
  if (Status >= 0) {
    printf("\nDecompressed %d bytes.\n", Status);
  } else {
    printf("\nDecompression error.\n");
  }
}
```


2.5 Limiting decompressor memory

As emCompress may well be used in highly memory-constrained devices, it is important to be able to limit the amount of memory that the decoder uses when decompressing a file. emCompress is able to do this using the `-m` command line switch when compressing a file.

Limiting the workspace that the decoder uses for decompressing will affect both the codecs that are eligible and the compression ratio that they can achieve.

Considering the FPGA example before, we can ask to compress the bitstream and use no more than 4 KB of workspace when decompressing:

```
C:> emCompress.exe -m4k FPGA.rbf

(c) 2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress V2.10 compiled Jun 23 2015 17:16:26

Input File:    FPGA.rbf
Optimization:  Level 5 (Balanced)
Restriction:   4096 bytes maximum decompressor RAM
Codec:         LZJU90(3583,3,256) chosen from 64 candidates
Decoding:      3880 bytes required for decompression
Compression:   24.1% (75.9% of original removed)
- Sizes        114557 -> 27599 bytes
Output File:   Compressed_FPGA.c
Elapsed time:  1.463 s

C:> _
```

In this particular instance, emCompress has selected the LZJU90 codec from only 64 candidates and the compressed size is slightly larger so the compressed image is 23.8% of the original rather than 20.6%. However, the decompression can run with a workspace of 3880 bytes.

Squeezing the workspace requirement even more and asking for 512 bytes:

```
C:> emCompress.exe -m512 FPGA.rbf

(c) 2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress V2.10 compiled Jun 23 2015 17:16:26

Input File:    FPGA.rbf
Optimization:  Level 5 (Balanced)
Restriction:   512 bytes maximum decompressor RAM
Codec:         LZSS(256,3,34) chosen from 10 candidates
Decoding:      332 bytes required for decompression
Compression:   32.7% (67.3% of original removed)
- Sizes        114557 -> 37465 bytes
Output File:   Compressed_FPGA.c
Elapsed time:  0.410 s

C:> _
```

emCompress has now chosen a new compressor again, LZSS. And as before the compressed image gets slightly larger but the workspace required is now only 332 bytes.

If RAM is especially tight in very small devices, emCompress can still perform very well. Constraining the workspace to only 100 bytes:

```
C:> emCompress.exe -m100 FPGA.rbf

(c) 2015 SEGGER Microcontroller GmbH & Co. KG
         www.segger.com
emCompress V2.10 compiled Jun  3 2015 18:22:25

Input File:    FPGA.rbf
Optimization:  Level 5 (Balanced)
```

```
Restriction:   100 bytes maximum decompressor RAM
Codec:         RLE-PAR(63,63) chosen from 2 candidates
Decoding:      72 bytes required for decompression
Compression:   42.8% (57.2% of original removed)
- Sizes        114557 -> 49032 bytes
Output File:   Compressed_FPGA.c
Elapsed time:  0.063 s

C:> _
```

If there is no suitable compressor or the data is incompressible, emCompress will select the STORE compressor and store the original bitstream uncompressed.

2.6 Adjusting compression performance

It's important not to waste time during development. If your compressed content is changing rapidly, you might want to spend less time compressing and sacrifice the very best compression ratios that emCompress can achieve.

2.6.1 Speeding up compression

To speed up compression, you can tell emCompress not to look so deeply for compression opportunities using the `-O` command line option. This option sets the optimization level that emCompress uses for its codecs, from 1 to 9: the higher the number, the more care emCompress takes when compressing.

Considering the FPGA example before, we can ask to compress the bitstream using its fastest compression:

```
C:> emCompress.exe -O1 FPGA.rbf

(c) 2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress V2.10 compiled Jun 23 2015 17:16:26

Input File:    FPGA.rbf
Optimization:  Level 1 (Fastest Compression)
Restriction:   None (assume unlimited decompressor RAM)
Codec:         DEFLATE(32k,3,258) chosen from 125 candidates
Decoding:      35248 bytes required for decompression
Compression:   20.9% (79.1% of original removed)
- Sizes        114557 -> 23903 bytes
Output File:   Compressed_FPGA.c
Elapsed time:  2.674 s

C:> _
```

Asking for the fastest compression reduces the time it takes to compress by a second, in this example, and compression is only slightly worse, the compressed image taking 23903 bytes rather than 23643 bytes at `-O5`.

2.6.2 Increasing the compression ratio

It may be possible to compress to smaller sizes by taking more time when compressing. emCompress offers compression levels 1 through 9, with 1 being the fastest and 9 being the slowest but with the best compression ratio. The effectiveness of the compression level directly depends upon the structure of the data to compress and the selected decoder memory constraints.

Considering the FPGA example again, we can ask to compress the bitstream using its best compression:

```
C:> emCompress.exe -O9 FPGA.rbf

(c) 2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress V2.10 compiled Jun 23 2015 17:16:26

Input File:    FPGA.rbf
Optimization:  Level 9 (Best Compression)
Restriction:   None (assume unlimited decompressor RAM)
Codec:         DEFLATE(32k,3,258) chosen from 125 candidates
Decoding:      35248 bytes required for decompression
Compression:   20.6% (79.4% of original removed)
- Sizes        114557 -> 23548 bytes
Output File:   Compressed_FPGA.c
Elapsed time:  11.922 s

C:> _
```


Compression at this level takes significantly longer but delivers a slightly better compression ratio, saving an additional 350 bytes.

2.7 Batch compression

It's quite common that read-only static content spans multiple files, for instance the content of a web server in an embedded device. As a convenience, emCompress accepts a wildcard file specification and will compress multiple files.

2.7.1 Compressing web server content

The following example compresses the entire content served by an embOS/IP embedded web server; the command line switch `-s` presents the results concisely:

```
C:> emCompress.exe -s *.htm *.js

(c) 2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress V2.10 compiled Jun 23 2015 17:16:26

------------------------------------------------------------------------
Filename                       Size  -> Size    Saved  Codec
------------------------------------------------------------------------
about.htm                     1,952       988      964  DEFLATE(2k,3,34)
authen.htm                      913       476      437  DEFLATE(1k,3,66)
embOSInfo.htm                   959       525      434  DEFLATE(1k,3,34)
embOSIPInfo.htm                 975       535      440  DEFLATE(1k,3,18)
formGET.htm                   3,509       987    2,522  DEFLATE(4k,3,66)
formPOST.htm                  3,515       989    2,526  DEFLATE(4k,3,66)
index.htm                       628       353      275  DEFLATE(1k,3,34)
Presentation.htm              3,790     1,301    2,489  DEFLATE(4k,3,66)
sendmail.htm                  3,568     1,243    2,325  DEFLATE(4k,3,130)
Shares.htm                    4,784     1,758    3,026  DEFLATE(8k,3,66)
SSEembOS.htm                  1,322       686      636  DEFLATE(2k,3,66)
SSEembOSIP.htm                1,326       693      633  DEFLATE(2k,3,66)
SSETime.htm                   1,212       599      613  DEFLATE(2k,3,66)
upload.htm                      802       442      360  DEFLATE(1k,3,18)
Upload_AJAX.htm               2,913       992    1,921  DEFLATE(4k,3,66)
virtfile.htm                    962       470      492  DEFLATE(1k,3,66)
eventsource.min.js            4,328     2,063    2,265  DEFLATE(4k,3,34)
jquery.min.js                93,106    33,554   59,552  DEFLATE(32k,3,66)
...ph.common.core.min.js     56,552    13,526   43,026  DEFLATE(32k,3,130)
...common.effects.min.js     27,594     3,147   24,447  DEFLATE(32k,3,258)
RGraph.line.min.js           62,868    13,016   49,852  DEFLATE(32k,3,258)
------------------------------------------------------------------------
Total                       277,578    78,343  199,235  28.2% of original
------------------------------------------------------------------------

Maximum decompressor memory required: 35232 bytes
Elapsed time:  10.955 s

C:> _
```

Because the files compressed are text, they compress very well using various DEFLATE compressors. The totals line shows that the original 277 KB compresses to 78 KB saving 199 KB and the compressed image is 28.2% of the original.

2.7.2 Applying limits to all files

You can limit the decompressor memory for all files. Running emCompress again but limiting decompressor memory workspace to 1 KB results in a completely different set of codecs selected:

```
C:> emCompress.exe -s -m1k *.htm *.js

(c) 2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress V2.10 compiled Jun 23 2015 17:16:26

------------------------------------------------------------------------------
Filename                       Size  -> Size    Saved  Codec
------------------------------------------------------------------------------
about.htm                     1,952    1,339      613  LZJU90(511,3,16)
authen.htm                      913      606      307  LZJU90(511,3,64)
embOSInfo.htm                   959      682      277  LZSS(512,3,18)
embOSIPInfo.htm                 975      688      287  LZSS(512,3,18)
formGET.htm                   3,509    1,370    2,139  LZSS(512,3,34)
formPOST.htm                  3,515    1,371    2,144  LZSS(512,3,34)
index.htm                       628      425      203  LZSS(512,3,18)
Presentation.htm              3,790    1,739    2,051  LZSS(512,3,18)
sendmail.htm                  3,568    1,620    1,948  LZSS(512,3,34)
Shares.htm                    4,784    2,492    2,292  LZSS(512,3,18)
SSEembOS.htm                  1,322      923      399  LZSS(512,3,18)
SSEembOSIP.htm                1,326      928      398  LZSS(512,3,18)
SSETime.htm                   1,212      795      417  LZSS(512,3,18)
upload.htm                      802      559      243  LZSS(512,3,18)
Upload_AJAX.htm               2,913    1,357    1,556  LZSS(512,3,34)
virtfile.htm                    962      603      359  LZJU90(511,3,64)
eventsource.min.js            4,328    2,968    1,360  LZSS(512,3,18)
jquery.min.js                93,106   57,096   36,010  LZSS(512,3,18)
...ph.common.core.min.js     56,552   23,051   33,501  LZSS(512,3,34)
...common.effects.min.js     27,594   12,306   15,288  LZSS(512,3,34)
RGraph.line.min.js           62,868   25,587   37,281  LZSS(512,3,34)
------------------------------------------------------------------------------
Total                       277,578  138,505  139,073  49.9% of original
------------------------------------------------------------------------------

Maximum decompressor memory required: 616 bytes
Elapsed time:  2.946 s

C:> _
```


2.7.3 Tuning compression

In the previous example, three of the codecs selected for compression were LZJU90 but the majority were LZSS. Because so few files use LZJU90 compression, and there is a code space overhead to include the LZJU90 decompressor in addition to the LZSS decompressor, it might be advantageous to use LZSS for `about.htm`, `authen.htm`, and `virtfile.htm`.

To compare the savings made between the two compressors, emCompress offers the ability to customize the set of codecs that are considered when compressing.

The `-X` option excludes all codecs (other than STORE) from consideration, and `-A` will add back a codec. To compare the difference between LZJU90 and LZSS on the three files, first run the compression using only LZJU90:

```
C:> emCompress.exe -s -m1k -X -ALZJU90 about.htm authen.htm virtfile.htm

(c) 2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress V2.10 compiled Jun 23 2015 17:16:26

-------------------------------------------------------------------------
Filename                       Size  -> Size    Saved  Codec
-------------------------------------------------------------------------
about.htm                     1,952    1,339      613  LZJU90(511,3,16)
authen.htm                      913      606      307  LZJU90(511,3,64)
virtfile.htm                    962      603      359  LZJU90(511,3,64)
-------------------------------------------------------------------------
Total                         3,827    2,548    1,279  66.6% of original
-------------------------------------------------------------------------

Maximum decompressor memory required: 616 bytes
Elapsed time:  0.079 s

C:> _
```

And then LZSS:

```
C:> emCompress.exe -s -m1k -X -ALZSS about.htm authen.htm virtfile.htm

(c) 2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress V2.10 compiled Jun 23 2015 17:16:26

-------------------------------------------------------------------------
Filename                       Size  -> Size    Saved  Codec
-------------------------------------------------------------------------
about.htm                     1,952    1,339      613  LZSS(512,3,18)
authen.htm                      913      608      305  LZSS(512,3,18)
virtfile.htm                    962      606      356  LZSS(512,3,34)
-------------------------------------------------------------------------
Total                         3,827    2,553    1,274  66.7% of original
-------------------------------------------------------------------------

Maximum decompressor memory required: 588 bytes
Elapsed time:  0.146 s

C:> _
```

From this you can see that using LZJU90 saves 1,279 bytes overall and using LZSS saves 1,274 bytes, a difference of just five bytes. Including the LZJU90 decompressor for these three files makes no sense as the code size of the LZJU90 decompressor is much more than five bytes.

Therefore, when compressing the entire content, it would make sense to simply exclude the LZJU90 codec from consideration using the emCompress `-X` option with a codec name:

```
C:> emCompress.exe -s -m1k -XLZJU90 *.htm *.js
```



```
(c) 2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress V2.10 compiled Jun 23 2015 17:16:26

--------------------------------------------------------------------------
Filename                      Size  -> Size    Saved  Codec
--------------------------------------------------------------------------
about.htm                    1,952    1,339      613  LZSS(512,3,18)
authen.htm                     913      608      305  LZSS(512,3,18)
embOSInfo.htm                  959      682      277  LZSS(512,3,18)
embOSIPInfo.htm                975      688      287  LZSS(512,3,18)
formGET.htm                  3,509    1,370    2,139  LZSS(512,3,34)
formPOST.htm                 3,515    1,371    2,144  LZSS(512,3,34)
index.htm                      628      425      203  LZSS(512,3,18)
Presentation.htm             3,790    1,739    2,051  LZSS(512,3,18)
sendmail.htm                 3,568    1,620    1,948  LZSS(512,3,34)
Shares.htm                   4,784    2,492    2,292  LZSS(512,3,18)
SSEembOS.htm                 1,322      923      399  LZSS(512,3,18)
SSEembOSIP.htm               1,326      928      398  LZSS(512,3,18)
SSETime.htm                  1,212      795      417  LZSS(512,3,18)
upload.htm                     802      559      243  LZSS(512,3,18)
Upload_AJAX.htm              2,913    1,357    1,556  LZSS(512,3,34)
virtfile.htm                   962      606      356  LZSS(512,3,34)
eventsource.min.js           4,328    2,968    1,360  LZSS(512,3,18)
jquery.min.js               93,106   57,096   36,010  LZSS(512,3,18)
...ph.common.core.min.js    56,552   23,051   33,501  LZSS(512,3,34)
...common.effects.min.js    27,594   12,306   15,288  LZSS(512,3,34)
RGraph.line.min.js          62,868   25,587   37,281  LZSS(512,3,34)
--------------------------------------------------------------------------
Total                      277,578  138,510  139,068  49.9% of original
--------------------------------------------------------------------------

Maximum decompressor memory required: 588 bytes
Elapsed time:  2.583 s

C:> _
```


2.8 Group compression

If your application has many small files, in the order of a few kilobytes each, it may well be worth compressing those files in *group mode*.

2.8.1 What is group compression?

Group compression combines all input files by concatenating them into a single image which is then compressed. The advantage of this type of compression scheme is that there is more opportunity for the encoders to find redundancy in the combined image than when considering each file individually (called *unit mode* in this guide).

Taking the previous web server example, all the HTML files are small and compress fairly well on their own. Running emCompress on the HTML files individually:

```
C:> emCompress.exe *.htm -s

(c) 2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress V2.10 compiled Jun 23 2015 17:16:26

------------------------------------------------------------------------
Filename                       Size  -> Size    Saved  Codec
------------------------------------------------------------------------
about.htm                     1,952      988      964  DEFLATE(2k,3,34)
authen.htm                      913      476      437  DEFLATE(1k,3,66)
embOSInfo.htm                   959      525      434  DEFLATE(1k,3,34)
embOSIPInfo.htm                 975      535      440  DEFLATE(1k,3,18)
formGET.htm                   3,509      987    2,522  DEFLATE(4k,3,66)
formPOST.htm                  3,515      989    2,526  DEFLATE(4k,3,66)
index.htm                       628      353      275  DEFLATE(1k,3,34)
Presentation.htm              3,790    1,300    2,490  DEFLATE(8k,3,66)
sendmail.htm                  3,568    1,243    2,325  DEFLATE(4k,3,130)
Shares.htm                    4,784    1,758    3,026  DEFLATE(8k,3,66)
SSEembOS.htm                  1,322      686      636  DEFLATE(2k,3,66)
SSEembOSIP.htm                1,326      693      633  DEFLATE(2k,3,66)
SSETime.htm                   1,212      599      613  DEFLATE(2k,3,66)
upload.htm                      802      442      360  DEFLATE(1k,3,18)
Upload_AJAX.htm               2,913      992    1,921  DEFLATE(4k,3,66)
virtfile.htm                    962      470      492  DEFLATE(1k,3,66)
------------------------------------------------------------------------
Total                        33,130   13,036   20,094  39.3% of original
------------------------------------------------------------------------

Maximum decompressor memory required: 10404 bytes
Elapsed time:  21.606 s

C:> _
```

In this case we have reduced 33 KB to 13 KB, which is good. However, in group mode we do better still:

```
C:> emCompress.exe -cCompressed_Files.c *.htm

(c) 2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress V2.10 compiled Jun 23 2015 17:16:26

Codec:         DEFLATE(32k,3,258) chosen from 126 candidates
Encoding:      33659168 bytes used during compression
Decoding:      35232 bytes required for decompression (258 for window)
Compression:   20.0% (80.0% of original removed)
- Sizes        33130 -> 6636 bytes
Elapsed time:  1.801 s

C:> _
```

Now the 33 KB input is reduced to 6.5 KB.

2.8.2 Compromises with group compression

Client software can deal with files compressed in either group or unit mode transparently: emCompress takes care of managing the details of decompressing both individual files and the "slices" of a group-compressed bitstream that correspond to the original input. As such, there is no need to alter the code that calls the emCompress API to decompresses a file if you flip the compression of a file between unit mode and group mode: all features, such as data integrity checks, work just the same in both modes.

There are, however, compromises that you should be aware of when using group mode.

Access time

In order to decompress the contents of a file compressed in group mode, emCompress must decode the bitstream from the start, passing over compressed content of other files, before starting to decompress the content of the requested file. Whilst there is no memory overhead associated with this process, there is a time overhead: it takes longer to access files at the end of the compressed bitstream than at the start.

You can compress your files in group mode such that the most frequently accessed files are given on the emCompress command line first, placing them at the front of the compressed bitstream, with less frequently used files towards the end of the command line.

Alternatively, you may well decide to compress different file sets in group mode and others in unit mode. This will require multiple invocations of emCompress, but the general strategy is workable.

For instance, you might want to compress all HTML files in group mode, all JavaScript files in group mode, and all images in unit mode, and you could use a set of commands such as this:

```
emCompress -gCompressed_HTML.c *.htm
emCompress -gCompressed_JS.c   *.js
emCompress *.png *.bmp
```

In this case, there are two group bitstreams (both completely independent of each other) along with as many individual bitstreams corresponding to each BMP and PNG file.

Dead data

All files that you compress in group mode are packaged into a single compressed bitstream. If you only use one file out of that bitstream, the entire bitstream is linked into your application, including the compressed content of files that you never reference. In this case, the linker has no opportunity to remove the redundant data that you never put to use. The advice would be, then, to group compress only the files that you know you use in your application.

In contrast, all files that are compressed individually in unit mode have separate bitstreams and decompressing a single file will not include the bitstreams of other compressed files (assuming your linker is capable of removing dead data).

emCompress provides the capabilitities for you to make appropriate decisions on how to compress your files and structure the compression that best suits your application's use.

2.8.3 Group compression walkthrough

In this example we will compress a set of files using group mode. The use of emCompress in group mode extends naturally from unit mode compression that you've seen previously.

Compress the files

We will compress three poems held in text files, each with the extension "`.txt`". A difference between unit mode and group mode is that you must tell emCompress where to write the compressed output source file—in unit mode, the generated C source file name is based on the uncompressed input file name. The `-g` command line option serves to invoke group mode compression and set the group-compressed output file name:

```
C:> emCompress.exe -gCompressed_Poems.c *.txt
```



```
(c) 2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emCompress V2.10 compiled Jun 23 2015 17:16:26

Codec:         DEFLATE(1k,3,66) chosen from 126 candidates
Encoding:      33563936 bytes used during compression
Decoding:      3220 bytes required for decompression (66 for window)
Compression:   49.9% (50.1% of original removed)
- Sizes        3114 -> 1554 bytes
Elapsed time:  1.401 s

C:> _
```

Now that all files are compressed in group mode, extracting the three files is no different to extracting individual bitstreams in unit mode. Here's a modification of the first example that prints the decompressed text of the three files to the console:

```
// File: Ex5.c
// - Decompress group-encoded file in stream mode.
//

#include "COMPRESS.h"
#include "Compressed_Poems.c"
#include <stdio.h>

static int _PrintData(void *pUserContext, void *pData, unsigned NumBytesData) {
  return printf("%.*s", NumBytesData, pData);
}

static void _WritePoem(const COMPRESS_ENCODED_FILE *pFile) {
  static union {
    U8  Bytes[3236];  // Workspace reported by emCompress
    U32 Long;         // Force long alignment of workspace.
  } Workspace;
  int Status;
  //
  Status = COMPRESS_DecompressThruFunc(pFile, &Workspace, sizeof(Workspace),
                                       _PrintData, 0,
                                       0, ~0UL,
                                       0);
  if (Status >= 0) {
    printf("\nDecompressed %d bytes.\n\n", Status);
  } else {
    printf("\nDecompression error.\n\n");
  }
}

void main(void) {
  _WritePoem(&Compressed_Jabberwocky);
  _WritePoem(&Compressed_IWanderedLonleyAsACloud);
  _WritePoem(&Compressed_AWinterNight);
}
```


2.9 Dynamic workspace allocation

You can configure emCompress to limit the memory required to decompress a file. In previous examples that memory is allocated statically, using a static array. You could place that static array on the stack, but it can also make sense to place it on the heap.

In this example the decompression workspace is allocated dynamically using `malloc` and the required size is retrieved using `COMPRESS_QueryWorkspaceSize()`.

```
// File: Ex6.c
// - Decode and verify group-encoded files in stream mode using
//   dynamic workspace allocation.
//

#include "COMPRESS.h"
#include "SEGGER_CRC.h"
#include "Compressed_Poems.c"
#include <stdio.h>
#include <stdlib.h>

static void _ValidatePoem(const COMPRESS_ENCODED_FILE *pFile) {
  void * pWorkspace;
  int    Status;
  //
  pWorkspace = malloc(COMPRESS_QueryWorkspaceSize(pFile));
  if (pWorkspace == 0) {
    printf("Can't allocate memory for workspace.\n");
  } else {
    Status = COMPRESS_DecompressThruFunc(pFile,

  pWorkspace, COMPRESS_QueryWorkspaceSize(pFile),
                                       0, 0,
                                       0, ~0UL,
                                       SEGGER_CRC_Calc_04C11DB7);
    if (Status >= 0) {
      printf("Verified %d bytes.\n", Status);
    } else {
      printf("Decompression error!\n");
    }
  }
  free(pWorkspace);
}

void main(void) {
  _ValidatePoem(&Compressed_IWanderedLonleyAsACloud);
  _ValidatePoem(&Compressed_Jabberwocky);
  _ValidatePoem(&Compressed_AWinterNight);
}
```


2.10 Command line options

emCompress accepts the following command line options.

2.10.1 Add codec (-A)

Syntax

`-A`*name*

Description

Add the given codec family to the list of codecs that are considered for compression. The letter case of the codec name does not matter.

By default all codecs are considered included.

2.10.2 Group compression (-g)

Syntax

`-g`*filename*

Description

Group files before compressing. The file *filename* is written to contain a single compressed bitstream and emCompress compressed file descriptors corresponding to each input file.

2.10.3 List performance (-l)

Syntax

`-l`

Description

List each compressor’s performance for every input file.

2.10.4 Decompressor memory limit (-m)

Syntax

`-m`*size*

Description

Set the maximum memory to be used by a decompressor. *size* is an integer but can also be suffixed by “`k`” to indicate that the units are in kilobytes (e.g. `-m4k` will set the maximum memory to 4096 bytes).

By default the compressor assumes that the decompressor has unlimited memory.

2.10.5 Dry run (-n)

Syntax

`-n`

Description

Run emCompress as normal but do not write any compressed files.

2.10.6 Optimization level (-O)

Syntax

`-O`*level*

Description

Set the optimization level from 1 (fastest compression) to 9 (best compression).

By default the optimization level is set to 5, which is balance between performance and compression ratio.

2.10.7 Summarize (-s)

Syntax

`-s`

Description

Print only a summary of each file that is compressed together with an summary of the overall savings made over all files.

2.10.8 Verbose (-v)

Syntax

`-v`

Description

Display additional statisical information and the computed CRCs for each compressed file.

2.10.9 Exclude codec (-X)

Syntax

`-X`

Description

Exclude all codecs, other than STORE, from consideration when compressing. You can add individual codecs using `-A`. By default no codecs are considered excluded.

Syntax

`-X`*name*

Description

Exclude the named codec family from list of codecs considered for compression. The letter case of the codec family does not matter.

2.10.10 Exit (--exit)

Syntax

`--exit`

Description

Force automatic termination of emCompress, even if an error occurred. By default emCompress terminates on success or waits for a key press on error.

2.10.11 Output directory (--outdir)

Syntax

`--outdir`*outputpath*

Description

Explicitly set the directory into which the compressed files are written. If not set, the compressed file is written into the source directory.

Chapter 3

API reference

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

3.1 Core functions

Function	Description
Decompression functions	
COMPRESS_DecompressThruFunc()	Decompress part of a file in streamed mode.
COMPRESS_DecompressToMem()	Decompress part of a file into memory.
Query functions	
COMPRESS_QueryEncodedData()	Query direct access to encoded bitstream.
COMPRESS_QueryEncodedDataCRC()	Query the CRC of the encoded (compressed) data image.
COMPRESS_QueryEncodedDataSize()	Query the size of the compressed (encoded) data image.
COMPRESS_QueryDecodedDataCRC()	Query the CRC of the decompressed (decoded) data image.
COMPRESS_QueryDecodedDataSize()	Query the size of the decompressed (decoded) data image.
COMPRESS_QueryWorkspaceSize()	Query the size of the decompressor workspace needed to decode a compressed file.

3.1.1 COMPRESS_DecompressThruFunc()

Description

Decompress part of a file in streamed mode.

Prototype

```
int COMPRESS_DecompressThruFunc(const COMPRESS_ENCODED_FILE *pSelf,
                                void *pWorkspace,
                                unsigned NumBytesWorkspace,
                                COMPRESS_OUTPUT_FUNC pfOutput,
                                void *pContext,
                                U32 Start,
                                U32 Len,
                                COMPRESS_CRC_FUNC pfCalcCRC);
```

Parameters

Parameter	Description
pSelf	Pointer to compressed file descriptor.
pWorkspace	Pointer to decompressor workspace. The workspace must be correctly aligned for the target architecture.
NumBytesWorkspace	Number of bytes in decompressor workspace.
pfOutput	Pointer to function that is fed decompressed output.
pContext	User context passed to output function.
Start	Byte offset, relative to start of decompressed file, from which to start delivering data.
Len	Total number of bytes to deliver starting from offset Start.
pfCalcCRC	Pointer to CRC calculation function for verification. If pfCalcCRC is null, no verification is performed.

Return value

≥ 0	Number of bytes successfully delivered. This will be no more than Len bytes.
< 0	Error in encoded bitstream or decoding aborted by user.

Additional information

If the end of file is encountered whilst decoding the bitstream, all bytes up to the end of file will be delivered to the application. The actual number of bytes delivered to the application is returned and, if less than Len, signals reaching the end of file.

The workspace pointed to by pWorkspace must be correctly aligned. For ARM and other 32-bit processors this generally means that the workspace must be aligned on a 32-bit boundary. You can ensure this by allocating a workspace using an array of unsigned or by using compiler-dependent pragmas or extensions.

Example

Please see *Single-file walkthrough* on page 18.

3.1.2 COMPRESS_DecompressToMem()

Description

Decompress part of a file into memory.

Prototype

```
int COMPRESS_DecompressToMem(const COMPRESS_ENCODED_FILE *pSelf,
                             void *pWorkspace,
                             unsigned NumBytesWorkspace,
                             void *pDest,
                             U32 Start,
                             U32 Len,
                             COMPRESS_CRC_FUNC pfCalcCRC);
```

Parameters

Parameter	Description
pSelf	Pointer to compressed file descriptor.
pWorkspace	Pointer to decompressor workspace. The workspace must be correctly aligned for the target architecture.
NumBytesWorkspace	Number of bytes in decompressor workspace.
pDest	Pointer to destination object that will receive decompressed output. The destination object must be at least Len bytes in size.
Start	Byte offset, relative to start of decompressed file, from which to start delivering data.
Len	Maximum number of bytes to deliver. The size of the object pointed to by pDest must be at least Len bytes in size.
pfCalcCRC	Pointer to CRC function implementation for verification. If pfCalcCRC is null, no verification is performed.

Return value

≥ 0	Number of bytes successfully stored into the object pointed to by pDest. This will be no more than Len bytes.
< 0	Error in encoded bitstream.

Additional information

If the end of file is encountered whilst decoding the bitstream, fewer than Len bytes will be delivered to the object pointed to by pDest. The actual number of bytes stored is returned and, if less than Len, signals reaching the end of file.

The workspace pointed to by pWorkspace must be correctly aligned. For ARM and other 32-bit processors this generally means that the workspace must be aligned on a 32-bit boundary. You can ensure this by allocating a workspace using an array of unsigned or by using compiler-dependent pragmas or extensions.

Example

Please see *Decompression into memory* on page 21.

3.1.3 COMPRESS_QueryEncodedData()

Description

Query direct access to encoded bitstream.

Prototype

```
void COMPRESS_QueryEncodedData(const COMPRESS_ENCODED_FILE *pSelf,
                               COMPRESS_ENCODED_DATA *pData);
```

Parameters

Parameter	Description
pSelf	Pointer to compressed file descriptor.
pData	Pointer to structure that receives the encoded data descriptor.

3.1.4 COMPRESS_QueryEncodedDataCRC()

Description

Query the CRC of the encoded (compressed) data image.

Prototype

```
U32 COMPRESS_QueryEncodedDataCRC(const COMPRESS_ENCODED_FILE *pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to compressed file descriptor.

Return value

The CRC-32 of the compressed bitstream.

3.1.5 COMPRESS_QueryEncodedDataSize()

Description

Query the size of the compressed (encoded) data image.

Prototype

```
U32 COMPRESS_QueryEncodedDataSize(const COMPRESS_ENCODED_FILE *pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to compressed file descriptor.

Return value

The size of the compressed data, measured in bytes.

3.1.6 COMPRESS_QueryDecodedDataCRC()

Description

Query the CRC of the decompressed (decoded) data image.

Prototype

```
U32 COMPRESS_QueryDecodedDataCRC(const COMPRESS_ENCODED_FILE *pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to compressed file descriptor.

Return value

The CRC-32 of the decompressed data.

3.1.7 COMPRESS_QueryDecodedDataSize()

Description

Query the size of the decompressed (decoded) data image.

Prototype

```
U32 COMPRESS_QueryDecodedDataSize(const COMPRESS_ENCODED_FILE *pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to compressed file descriptor.

Return value

The size of the decompressed data, measured in bytes.

3.1.8 COMPRESS_QueryWorkspaceSize()

Description

Query the size of the decompressor workspace needed to decode a compressed file.

Prototype

```
U32 COMPRESS_QueryWorkspaceSize(const COMPRESS_ENCODED_FILE *pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to compressed file descriptor.

Return value

The size of the workspace required to decompress a compressed file, in bytes.

Chapter 4

Resource use

This section describes the memory requirement in terms of RAM and ROM that emCompress requires for decompression which can be used to obtain sufficient estimates for most target systems.

4.1 General comments

emCompress is fully reentrant when decompressing a bitstream: there is no requirement to lock any shared data nor is there any static data requirement associated with its use.

There is no configuration required in order to use emCompress in your target system beyond setting compiler options for code generation strategy and setting up paths to include files.

4.2 Memory footprint

4.2.1 Target system configuration

The following table shows the hardware and the toolchain details of a typical emCompress target system:

Detail	Description
CPU	Cortex-M3
Tool chain	IAR Embedded Workbench for ARM V6.40
Model	Thumb-2 instructions
Compiler options	Highest size optimization

4.2.2 RAM use

The amount of RAM that emCompress uses is under complete control as it is specified at compression time. In addition to the workspace requirement specified for decompression, there is a small stack requirement to decompress any bitstream. The following section lists per-codec RAM requirements.

4.2.3 ROM use

The amount of ROM that emCompress uses for decompression varies with the codec selected. Some of the codecs share common code:

- All decoders require the BITIO functions.
- LZJU90 requires the start-step-stop decoder.
- DEFLATE requires the canonical Huffman decoder.

The following table measures the total ROM required for a single decoder instance, isolated from all other decoders, including all supporting functions and excluding integrity checks. Hence, if your compressed bitstreams only use DEFLATE, the amount of ROM required is easily read off from the table.

In addition to the ROM requirements, the following table summarizes the RAM needed for static data and stack.

Codec	ROM	RAM (static)	RAM (stack)
STORE	0.5 KB	0 bytes	256 bytes
RLE-PAR	0.8 KB	0 bytes	256 bytes
HUFF	1.3 KB	0 bytes	256 bytes
LZW	1.0 KB	0 bytes	256 bytes
LZSS	1.2 KB	0 bytes	256 bytes
LZJU90	1.3 KB	0 bytes	276 bytes
DEFLATE	2.1 KB	0 bytes	276 bytes

4.2.4 Integrity check overhead

Adding a table-driven CRC32 integrity check adds 1 KB of code to the total emCompress ROM requirement.

Chapter 5

Frequently asked questions

Q: *What's the purpose of the STORE compressor? It's useless isn't it?*

A: No, it is not useless. The STORE compressor ensures that incompressible data does not expand, as it generally does using other lossless compressors. It allows incompressible data to be handled in the same manner as compressed data, freeing the client from treating uncompressed data differently from compressed data.

Q: *Can I compress something with gzip and use that in emCompress?*

A: No. The primary function of emCompress is to correctly determine the amount of memory required by the decoder, not to establish the best compression ratio. emCompress cannot ensure that the decompressor will be able to decompress an arbitrary external bitstream within the decoder's stated memory requirements.

Q: *Are bitstreams portable across different versions of emCompress?*

A: No. We require that you recompress your files when upgrading emCompress so that you get the best compression and the decoder is correctly matched to the encoder.

Q: *emCompress is great! Where can I find the compressor sources?*

A: emCompress is designed for fast and effective *decompression* at runtime and the companion emCompress application is designed to run on a workstation or PC where memory is plentiful. Because the emCompress application is all that is required for preparing data to be decompressed, we do not ship the source code of the compressors built into the emCompress application.

Q: *I still need compression in my application, though…*

A: Should you wish to take advantage of on-device compression in your application, please contact us to discuss your requirements and how we might be able to help.

Q: *Why do you store two CRCs? Surely one is enough?*

A: The two CRCs serve different purposes, although both are integrity checks. The CRC of the compressed bitstream ensures that the compressed bitstream is intact before being decompressed. If the compressed bitstream is held in RAM, it may become corrupt by a store through a wild pointer; and if it is held in embedded flash, read disturbances may well corrupt the flash, but this is rare. Checking that the decompressed output CRC matches the stored CRC offers an extra level of assurance that the decompression process executed correctly and has not suffered data corruption during decompression.

Q: *Why are there so many compressors?*

A: emCompress ships with compression algorithms that can be parameterized to tune decompressor memory. As each decompressor has different memory requirements and compression capabilities, emCompress covers a wide range of use from excellent compression using moderate amounts of RAM to good compression using tiny amounts of RAM. No one compression algorithm will be applicable across the wide range of inputs and workspace requirements of target applications.

Chapter 6

Reference

6.1 Unabridged sample output

The following is the unabridged C file that is generated by emCompress on the sample `Jabberwocky.txt` file with default options.

```
//
//  Generated by emCompress V1.00 compiled Jun 23 2015 17:16:26
//
//  Input File:    Jabberwocky.txt
//  Optimization:  Level 5 (Balanced)
//  Restriction:   None (assume unlimited decompressor RAM)
//  Output File:   Compressed_Jabberwocky.c
//  Codec:         DEFLATE(2k,3,258) chosen from 125 candidates
//  Encoding:      33567008 bytes used during compression
//  - Speed        7644 ns/byte (8325 us total time) yields 0.12 MB/s
//  - Image CRC    5044A8E3 (polynomial is 04C11DB7, start with FFFFFFFF)
//  Decoding:      4472 bytes required for decompression (258 for window)
//  - Speed        92 ns/byte (101 us total time) yields 10.28 MB/s
//  - Image CRC    A33EE9AD (polynomial is 04C11DB7, start with FFFFFFFF)
//  Compression:   48.0% at 3.842 bits/byte (52.0% of original removed)
//  - Sizes        1089 -> 523 bytes (saving 566 bytes)
//

#include "COMPRESS_Int.h"

#if !defined(COMPRESS_VERSION) || COMPRESS_VERSION != 20100
  #error Incompatible version -- regenerate with emCompress
#endif

static const U8 _Jabberwocky__aBitstream[523] = {
  0xCC, 0x53, 0x4B, 0x8E, 0xDB, 0x30, 0x0C, 0xDD, 0x07, 0xC8, 0x1D, 0x98,
  0xD9, 0x74, 0x93, 0xF6, 0x00, 0xED, 0xA2, 0x48, 0xD2, 0x02, 0x6D, 0x11,
  0x20, 0xC0, 0x4C, 0x80, 0xA2, 0x4B, 0xCA, 0xA6, 0x2D, 0x25, 0x92, 0x39,
  0xD0, 0x67, 0x5C, 0x9F, 0xA4, 0xD7, 0x2D, 0xA9, 0xC4, 0x49, 0x8E, 0x30,
  0xC9, 0xC2, 0x86, 0x48, 0x3D, 0xBE, 0xF7, 0xF8, 0xFC, 0x0B, 0x8D, 0xA1,
  0x38, 0x72, 0x73, 0x9E, 0x96, 0x0B, 0x80, 0xED, 0x1F, 0xD8, 0x7F, 0xFF,
  0xFD, 0xF3, 0x05, 0x76, 0x9B, 0xE7, 0xE7, 0xC3, 0x7E, 0xBF, 0x5C, 0xE8,
  0xFF, 0xC3, 0x71, 0xC4, 0x04, 0x26, 0x3A, 0xEF, 0x5D, 0xBF, 0x06, 0x1C,
  0x5A, 0xC8, 0x96, 0x20, 0x79, 0x97, 0xED, 0x04, 0x99, 0xDF, 0x28, 0xE9,
  0x65, 0xFD, 0x7D, 0x73, 0x2D, 0xF4, 0x53, 0xA4, 0xDA, 0xD4, 0xBB, 0x60,
  0x3C, 0x81, 0x1B, 0x6A, 0xFB, 0x88, 0x86, 0x3E, 0x2F, 0x17, 0x1B, 0xEF,
  0x21, 0xB8, 0x90, 0x26, 0x18, 0x49, 0xFA, 0xB4, 0x62, 0x38, 0x72, 0xAF,
  0x28, 0xEB, 0x19, 0x66, 0x73, 0x1D, 0x11, 0x38, 0x10, 0x44, 0xCC, 0x36,
  0x01, 0x97, 0xDC, 0x47, 0x81, 0xF8, 0xA4, 0x8C, 0x9E, 0xB6, 0x34, 0xE2,
  0xF5, 0xF6, 0x5D, 0xC2, 0x1A, 0xC2, 0x04, 0x89, 0x87, 0xD5, 0x0C, 0x73,
  0x94, 0xF2, 0x09, 0xC7, 0x24, 0x7D, 0x98, 0xC1, 0xB8, 0x4C, 0xEB, 0x7A,
  0xA5, 0xF1, 0xB7, 0xC3, 0x06, 0x73, 0x63, 0xE5, 0xC2, 0x23, 0x60, 0x31,
  0xA7, 0x62, 0xA4, 0x3D, 0xB6, 0x17, 0xB1, 0xC9, 0x96, 0xE1, 0x11, 0xB2,
  0x8B, 0x25, 0x38, 0x2E, 0x09, 0xB6, 0x52, 0xA5, 0x98, 0x86, 0x8A, 0xF1,
  0xA4, 0xC4, 0x7E, 0x08, 0x04, 0xF3, 0x19, 0xAC, 0x4B, 0xF0, 0xC6, 0xF1,
  0x15, 0x3D, 0xA4, 0x91, 0x63, 0xAB, 0x26, 0x58, 0xE9, 0xFE, 0x32, 0xE3,
  0xEC, 0x79, 0xE8, 0x21, 0x3B, 0x91, 0x57, 0x75, 0xE2, 0xF0, 0x57, 0xA5,
  0x76, 0x4C, 0xA0, 0xCE, 0x72, 0xE9, 0x6D, 0xFE, 0xB7, 0x5C, 0xBC, 0x30,
  0x44, 0x4A, 0x99, 0x5A, 0x3D, 0x35, 0xE2, 0xB5, 0x3C, 0x8E, 0x25, 0xE4,
  0x12, 0x20, 0x47, 0xA2, 0x47, 0xBB, 0x92, 0xCC, 0x6D, 0x01, 0x47, 0xEB,
  0x66, 0xCB, 0x2B, 0x48, 0xB5, 0x4B, 0xEA, 0xA2, 0x24, 0xE9, 0x71, 0xE9,
  0x3A, 0x97, 0xEC, 0x5C, 0xAD, 0xC3, 0xF4, 0xE2, 0xCD, 0x79, 0xD5, 0xF7,
  0xE8, 0xE8, 0x28, 0x3B, 0x06, 0x9A, 0x48, 0x16, 0xD0, 0x41, 0xE7, 0x31,
  0x90, 0xB4, 0xEE, 0xE4, 0x01, 0x32, 0xA9, 0xEB, 0xBC, 0x53, 0x19, 0x36,
  0x2A, 0x5A, 0x65, 0x97, 0x8B, 0xEF, 0x49, 0x76, 0xFB, 0x88, 0xA9, 0xF4,
  0x4C, 0x89, 0x92, 0x05, 0x21, 0x28, 0x2C, 0xD4, 0xF5, 0x40, 0x62, 0xFA,
  0x72, 0x71, 0x18, 0x74, 0x25, 0x23, 0xAF, 0xE0, 0xFE, 0x76, 0x59, 0xFE,
  0x05, 0xF2, 0x92, 0xB5, 0xFA, 0xFE, 0xC8, 0xF0, 0x6A, 0xAD, 0xF1, 0xD8,
  0x0A, 0x11, 0x1A, 0x32, 0xA4, 0xC1, 0x35, 0x67, 0x8A, 0x1F, 0x65, 0x19,
  0xCD, 0x59, 0xA0, 0x65, 0x11, 0x9E, 0xBA, 0xAC, 0xC3, 0x5A, 0xC2, 0xEB,
  0x22, 0xAB, 0x18, 0x97, 0x93, 0xC8, 0xC6, 0x76, 0xC6, 0x93, 0xCE, 0x8A,
  0xD0, 0xA3, 0x2F, 0xE1, 0xD5, 0xAA, 0x20, 0x23, 0x18, 0x97, 0x9C, 0x29,
  0x17, 0x8B, 0x29, 0x57, 0xC3, 0x24, 0xF1, 0x78, 0x0D, 0xF3, 0xDD, 0xA2,
```



```
  0xAF, 0x33, 0xCE, 0x4E, 0x17, 0x98, 0x59, 0x33, 0x88, 0x31, 0xA4, 0x1A,
  0x46, 0x43, 0x18, 0xD4, 0x6E, 0xC3, 0x93, 0x70, 0x3A, 0x48, 0x72, 0xD0,
  0x9C, 0x34, 0x39, 0x2D, 0x4E, 0x2B, 0xD8, 0xA1, 0xF7, 0xCC, 0x76, 0x05,
  0x3B, 0xF4, 0x1E, 0x27, 0x4D, 0xD0, 0x8D, 0x52, 0x63, 0x39, 0x66, 0x35,
  0x4C, 0x83, 0x23, 0x61, 0x3A, 0xF1, 0x54, 0x19, 0xBD, 0xB7, 0x6F, 0xF1,
  0x3F, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
};

const COMPRESS_ENCODED_FILE Compressed_Jabberwocky = {
  _Jabberwocky__aBitstream,
  523,
  0x5044A8E3,
  &COMPRESS_DEFLATE_Decode,
  0,
  1089,
  0xA33EE9AD,
  4472,
  { 2048, 3, 258 }
};
```


6.2 Compression algorithms

This section provides a brief overview of the compression algorithms that emCompress uses to compress data.

emCompress implements the following algorithms, listed in increasing order of compression effectiveness:

Algorithm	Description
STORE	Data that is incompressible is stored uncompressed.
RLE-PAR	Run-length encoding that compresses FPGA and CPLD bitstreams well and has an ultra-fast, compact decoder.
HUFF	Huffman-coded bitstream using 257 symbols.
LZW	Lempel–Ziv–Welch compression with dictionary references stored using adaptive width coding.
LZSS	Lempel–Ziv–Storer–Szymanski compression with match distances and lengths stored as fixed-length bit sequences.
LZJU90	Lempel–Ziv–Storer–Szymanski compression with match distances and lengths stored using a start-step-stop unary coding.
DEFLATE	Standard DEFLATE compressor which is the basis of Zip compression using LZSS with Huffman coding of the match distance and symbol/length dictionaries. This is the most complex compressor and decompressor but usually provides the best compression ratio when decoder memory is more than 2 K.

6.2.1 STORE codec

The STORE codec stores the content of a file without compressing it and is the codec of last resort when confronted with a file that simply does not compress with any other codec.

This can make sense when you are compressing a directory containing many files and one of them cannot be effectively compressed without expansion (e.g. it is already compressed by some tool). Rather than needing to find those files and write special code that would handle uncompressed data differently from compressed data, the "store" compressor treats uncompressed data just the same as compressed data.

6.2.2 HUFF codec

The Huffman codec examines the content to be compressed as a whole and tries to reduce it. To do this it computes the frequency of each symbol (byte) in the input and constructs a canonical Huffman coding that covers each byte plus the "end of stream" symbol. Including the end of stream marker, the Huffman coding is computed using an alphabet of at most 257 symbols rather than just 256.

6.2.3 LZW codec

The LZW codec creates a dictionary of strings seen in the input such that when the string is seen again, the compressor encodes a reference to it rather than the string itself.

LZW has a fairly high workspace requirement for constructing and maintaining the dictionary. As the dictionary grows, encoding a reference becomes less efficient as more bits are required to encode it. The emCompress LZW codec adaptively encodes references with variable bit widths and, when the dictionary becomes full, the dictionary is emptied and filled over again with new content.

Emptying the dictionary like this may seem rather wasteful, but it helps maintain a fresh set of recently-seen data to work from, which is very helpful when the content to be encoded suddenly changes—old content expires rather than being held in the dictionary forever.

Both the encoder and the decoder need to keep identical models of the dictionary content, so both encoder and decoder need to maintain the state of the dictionary. This has an unfortunate impact on the decoder: it requires the same memory footprint as the encoder to maintain the dictionary and, therefore, is not the best compressor for static content.

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.

CRC
Cyclic Redundancy Check. An error detection code that can detect corruption of a bitstream.

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.

DEFLATE
A popular compression format defined by RFC 1951. The compression format is widely adopted in, for instance, Zip and gzip.

Group mode
An emCompress mode that compresses multiple files by combining them and compressing the combined image into a single bitstream in order to improve overall compression ratio. Compare *Unit mode*.

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.

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

LZW
Lempel–Ziv–Welch. A compression scheme that is based in LZ78.

RLE
Run-Length Encoding. A compression scheme where repeated bytes are replaced with a special code, or marker, that indicates the repeated byte and the number of repeats.

Unit mode
An emCompress mode that compresses each file into its own individual bitstream independent of any other file. Compare *Group mode*.

