emCompress-ToGo

On-target compression system

User Guide & Reference Manual

Document: UM17003
Software Version: 3.30
Revision: 0
Date: March 28, 2019

A product of SEGGER Microcontroller GmbH

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 (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

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

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

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

Manual versions

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

Print date: March 28, 2019

Software	Revision	Date	By	Description
3.30	0	190328	RH	Added fast compression functions. • Added `CTG_FastCompressM2M()`. • Added `CTG_FastCompress()`. • Added `CTG_FastCompressInit()`.
3.20	0	190312	RH	Chapter "emCompress-ToGo API", Added streaming compression and decompression functions. • Added `CTG_Compress()`. • Added `CTG_Decompress()`. • Added `CTG_CompressRead()`. • Added `CTG_DecompressRead()`. • Added `CTG_CompressWrite()`. • Added `CTG_DecompressWrite()`. Updated chapter *Resource use and performance*
3.10	0	190130	PC	Chapter "emCompress-ToGo API" • Added `CTG_CompressM2M_256()` … `CTG_CompressM2M_8k()`. • Added `CTG_DecompressM2M_256()` … `CTG_DecompressM2M_8k()`. • Added `CTG_GetStatusText()`. • Added `CTG_GetCopyrightText()`. • Added `CTG_GetVersionText()`.
2.10	0	180221	PC	First release.

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.
`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.

Table of contents

Chapter 1

Introducing emCompress-ToGo

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

1.1 What is emCompress-ToGo?

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

1.2 Features

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

- 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.

1.3 Package content

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

Files	Description
`Config`	Configuration header files.
`Doc`	emCompress-ToGo documentation.
`COMPRESS`	emCompress-ToGo source code.
`SEGGER`	SEGGER software component source code used in emCompress-ToGo.
`Windows`	Supporting applications in binary form.

1.4 Sample applications

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

The sample applications are:

Application	Description
CTG_M2M_Encode.c	Memory to memory compression.
CTG_M2M_Decode.c	Memory to memory decompression.
CTG_M2F_Encode.c	Memory to function compression.
CTG_F2M_Decode.c	Function to memory decompression.
CTG_F2M_Encode.c	Function to memory compression.
CTG_M2F_Decode.c	Memory to function decompression.
CTG_BenchEncode.c	Benchmark for all compression modes.
CTG_BenchDecode.c	Benchmark for all decompression modes.
CTG_Util.c	Compress and decompress files.
CTG_Test.c	Compressor and decompressor test application.
CTG_Optimize.c	Search for optimal compressor parameters.

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.

1.4.2 Where to find the sample code

Target-independent code

All target-independent samples are included in the `Application` directory of the emCompress-ToGo distribution.

The entry point for all target-dependent code is the function `MainTask()`, which is usually run by the host operating system in a task context. To run this code on a PC or workstation, simply call `MainTask()` from `main()`:

```
void MainTask(void);

int main(int argc, char **argv) {
  MainTask();
  return 0;
}
```

Host-only code

All applications that require a file system and are intended to run on a PC or workstation are included in the `Windows/COMPRESS` directory of the emCompress-ToGo distribution.

1.5 Recommended project structure

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

> **Note**
>
> When updating to a newer emCompress-ToGo 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`

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

Chapter 2

Using emCompress-ToGo

---

emCompress-ToGo divides into two parts:

- A compressor that is responsible for compressing data using SEGGER's SMASH-2 algorithm, and
- A decompressor that is responsible for decompression data compressed with the SMASH-2 algorithm.

2.1 Memory-to-memory compression

The most straightforward way of compressing data is to use a simple memory-to-memory compression function. For this purpose, emCompress-ToGo offers `CTG_CompressM2M()` where the "M2M" suffix indicates that the source is memory and the destination is memory.

This example compresses some text and displays the compressed size:

```
// File: CTG_M2M_Encode.c
// - Compress memory to memory
//

#include "CTG.h"  ❶
#include <stdio.h>

static const U8 _aInput[] = {  ❷
  "Jabberwocky\n  BY LEWIS CARROLL\n\n\n"
  "'Twas brillig, and the slithy toves\n   Did gyre and gimble in the wabe:\n"
  "All mimsy were the borogoves,\n   And the mome raths outgrabe.\n\n"
  "\"Beware the Jabberwock, my son!\n   The jaws that bite, the claws that catch!\n"
  "Beware the Jubjub bird, and shun\n   The frumious Bandersnatch!\"\n\n"
  "He took his vorpal sword in hand;\n   Long time the manxome foe he sought-\n"
  "So rested he by the Tumtum tree\n   And stood awhile in thought.\n\n"
  "And, as in uffish thought he stood,\n   The Jabberwock, with eyes of flame,\n"
  "Came whiffling through the tulgey wood,\n   And burbled as it came!\n\n"
  "One, two! One, two! And through and through\n   The vorpal blade went snicker-snack!\n\n"
  "He left it dead, and with its head\n   He went galumphing back.\n\n"
  "\"And hast thou slain the Jabberwock?\n   Come to my arms, my beamish boy!\n"
  "O frabjous day! Callooh! Callay!\"\n\n   He chortled in his joy.\n"
  "'Twas brillig, and the slithy toves\n   Did gyre and gimble in the wabe:\n"
  "All mimsy were the borogoves,\n   And the mome raths outgrabe.\n"
};

static U8 _aOutput[sizeof(_aInput)*2];  ❸

void MainTask(void);
void MainTask(void) {
  int Status;
  //
  Status = CTG_CompressM2M(&_aInput[0],  sizeof(_aInput),  ❹
                           &_aOutput[0], sizeof(_aOutput),
                           0);
  if (Status >= 0) {  ❺
    printf("Compressed %u to %u bytes.\n", sizeof(_aInput), Status);
  } else {
    printf("Compression error.\n");
  }
}
```

❶ Include emCompress-ToGo header

All of emCompress-ToGo's API is made available by including the single header file `CTG.h`.

❷ Uncompressed input data

The array `_aInput` contains some text which is the data to be compressed.

❸ Compressed output

The array `_aOutput` will contain the compressed bitstream. It is possible for some uncompressible data to expand, so the output array is made double size in case this happens. (This happens to be far larger than required: the maximum size that uncompressible data expands to is documented in *SMASH-2 format and bitstream expansion* on page 87.)

❹ Run compressor to compress data

The function `CTG_CompressM2M()` compresses the input, writes the compressed bitstream, and delivers a status to the application which indicates whether the compression succeeded or not. The compression function is provided the input and output arrays, their sizes, and

an additional “flags” parameter that customizes the compression algorithm: the default of zero is used here, and this parameter is described in *Adjusting compression parameters* on page 25.

❺ Interpret the result

Any negative status code is an error, indicating that compression failed. The only failure that can happen, in this case, is that the output array is not big enough to contain the compressed data. We don’t expect compression to ever fail, as the output array is of sufficient capacity here, but the status is decoded anyway.

Compile and test

To build the program, compile `CTG_M2M_Encode.c` with the emCompress-ToGo source code in `COMPRESS` and SEGGER software components in `SEGGER`. After it’s compiled and linked without errors, running the example prints the uncompressed and compressed sizes:

```
C:> CTG_M2M_Encode.exe

Compressed 1011 to 700 bytes.

C:> _
```


2.2 Memory-to-memory decompression

Now that we have some compressed data, it's time to decompress it. The function `CTG_DecompressM2M()` uses the same memory-to-memory pattern.

This example compresses some text, decompresses it, and checks that the decompressed image is identical to the image given to the compressor:

```
// File: CTG_M2M_Decode.c
// - Compress and decompress memory to memory
//

#include "CTG.h"
#include <stdio.h>

static const U8 _aInput[] = {
  "Jabberwocky\n  BY LEWIS CARROLL\n\n\n"
  "'Twas brillig, and the slithy toves\n   Did gyre and gimble in the wabe:\n"
  "All mimsy were the borogoves,\n   And the mome raths outgrabe.\n\n"
  "\"Beware the Jabberwock, my son!\n   The jaws that bite, the claws that catch!\n"
  "Beware the Jubjub bird, and shun\n   The frumious Bandersnatch!\"\n\n"
  "He took his vorpal sword in hand;\n   Long time the manxome foe he sought-\n"
  "So rested he by the Tumtum tree\n   And stood awhile in thought.\n\n"
  "And, as in uffish thought he stood,\n   The Jabberwock, with eyes of flame,\n"
  "Came whiffling through the tulgey wood,\n   And burbled as it came!\n\n"
  "One, two! One, two! And through and through\n   The vorpal blade went snicker-snack!\n\n"
  "He left it dead, and with its head\n   He went galumphing back.\n\n"
  "\"And hast thou slain the Jabberwock?\n   Come to my arms, my beamish boy!\n"
  "O frabjous day! Callooh! Callay!\"\n\n   He chortled in his joy.\n"
  "'Twas brillig, and the slithy toves\n   Did gyre and gimble in the wabe:\n"
  "All mimsy were the borogoves,\n   And the mome raths outgrabe.\n"
};

static U8 _aOutput[sizeof(_aInput)*2];
static U8 _aMirror[sizeof(_aInput)];  ❶

void MainTask(void);
void MainTask(void) {
  int EncodeLen;
  int DecodeLen;
  //
  EncodeLen = CTG_CompressM2M(&_aInput[0],  sizeof(_aInput),  ❷
                              &_aOutput[0], sizeof(_aOutput),
                              0);
  if (EncodeLen < 0) {
    printf("Compression error.\n");
  } else {
    printf("Compressed %u to %u bytes.\n", sizeof(_aInput), EncodeLen);
    DecodeLen = CTG_DecompressM2M(&_aOutput[0], EncodeLen,  ❸
                                  &_aMirror[0], sizeof(_aMirror));
    if (DecodeLen < 0) {  ❹
      printf("Decompression error.\n");
    } else if (DecodeLen != sizeof(_aInput)) {  ❺
      printf("Decompression expected-length error.\n");
    } else {
      printf("Decompressed %u to %u bytes.\n", EncodeLen, DecodeLen);  ❻
      if (memcmp(_aInput, _aMirror, DecodeLen) == 0) {  ❼
        printf("Images match!\n");
      } else {
        printf("Images do not match!\n");
      }
    }
  }
}
```

❶ Introduce a mirror

The array `_aMirror` is introduced to be a mirror-image of the input data after the compression-decompression process. If the input and mirrored input are not identical, there is an error in the compression-decompression process.

❷ Compress the input

This proceeds as in the previous example, and the length of the encoded data is stored because we need it later.

❸ Decompress the compressed data

Calling `CTG_DecompressM2M()` decompresses the compressed data. Notice that we do not need to pass an additional "flags" parameter when decompressing as the compression flags are stored, along with the compressed image, in the bitstream to be recoverd by the decompressor.

❹ Interpret the result

`CTG_DecompressM2M()` returns a status code the same as `CTG_CompressM2M()` does. However, it is possible for the compressed bitstream to be corrupt, so additional errors can be reported by the decompressor.

❺ Check lengths

If the length of the decompressed data does not match the length of the original data, that's a problem. We don't expect this to be the case.

❻ Report lengths

As a check, the decompression phase statistics are shown.

❼ Check images mirror each other

Once the compressed bitstream has decompressed correctly, an additional check ensures that the decompressed data is identical to the original data. If this isn't the case, there is an error in the compression-decompression process.

Compile and run

To build the program, compile `CTG_M2M_Decode.c` with the emCompress-ToGo source code in `COMPRESS` and SEGGER software components in `SEGGER`. After it's compiled and linked without errors, running the example prints the uncompressed and compressed sizes:

```
C:> CTG_M2M_Decode.exe

Compressed 1011 to 700 bytes.
Decompressed 700 to 1011 bytes.
Images match!

C:> _
```


2.3 Memory-to-function compression

Rather than the all-at-once compression offered by `CTG_CompressM2M()`, emCompress-ToGo can compress the input and write the output incrementally through a user-provided function. In this way it's possible for the client, for instance, to write the output to a file or to memory, or even to a network connection. A further example is compressing an FPGA configuration bitstream where the uncompressed FPGA bitstream is too large to fit into memory—in this case, the compressed bitstream is decompressed and written to the FPGA by the user-provided function as it is decompressed.

This example compresses some text and sends the compressed output through a function which stores it into memory.

```
// File: CTG_M2F_Encode.c
// - Compress memory to function.
//

#include "CTG.h"
#include <stdio.h>

static const U8 _aInput[] = {
  "Jabberwocky\n  BY LEWIS CARROLL\n\n\n"
  "'Twas brillig, and the slithy toves\n   Did gyre and gimble in the wabe:\n"
  "All mimsy were the borogoves,\n   And the mome raths outgrabe.\n\n"
  "\"Beware the Jabberwock, my son!\n   The jaws that bite, the claws that catch!\n"
  "Beware the Jubjub bird, and shun\n   The frumious Bandersnatch!\"\n\n"
  "He took his vorpal sword in hand;\n   Long time the manxome foe he sought-\n"
  "So rested he by the Tumtum tree\n   And stood awhile in thought.\n\n"
  "And, as in uffish thought he stood,\n   The Jabberwock, with eyes of flame,\n"
  "Came whiffling through the tulgey wood,\n   And burbled as it came!\n\n"
  "One, two! One, two! And through and through\n   The vorpal blade went snicker-snack!\n\n"
  "He left it dead, and with its head\n   He went galumphing back.\n\n"
  "\"And hast thou slain the Jabberwock?\n   Come to my arms, my beamish boy!\n"
  "O frabjous day! Callooh! Callay!\"\n\n   He chortled in his joy.\n"
  "'Twas brillig, and the slithy toves\n   Did gyre and gimble in the wabe:\n"
  "All mimsy were the borogoves,\n   And the mome raths outgrabe.\n"
};

static U8 _aOutput[sizeof(_aInput)*2];

typedef struct {  ❶
  U8       * pOutput;
  unsigned   OutputLen;
  unsigned   Cursor;
} CTG_WR_CONTEXT;

static int _Wr(const U8 *pData, unsigned DataLen, void *pWrCtx) {  ❷
  CTG_WR_CONTEXT * pCtx;
  unsigned         N;
  //
  pCtx = (CTG_WR_CONTEXT *)pWrCtx;  ❸
  for (N = 0; N < DataLen; ++N) {
    if (pCtx->Cursor >= pCtx->OutputLen) {  ❹
      return CTG_STATUS_OUTPUT_OVERFLOW;
    }
    pCtx->pOutput[pCtx->Cursor++] = *pData++;
  }
  return N;
}

void MainTask(void);
void MainTask(void) {
  CTG_WR_CONTEXT WrCtx;
  int            EncodeLen;
  //
  WrCtx.pOutput   = &_aOutput[0];  ❺
  WrCtx.OutputLen = sizeof(_aOutput);
  WrCtx.Cursor    = 0;
  EncodeLen = CTG_CompressM2F(&_aInput[0],  sizeof(_aInput),  ❻
                              _Wr, &WrCtx,
                              0);
  if (EncodeLen < 0) {
```



```
    printf("Compression error.\n");
  } else {
    printf("Compressed %u to %u bytes.\n", sizeof(_aInput), EncodeLen);
  }
}
```

❶ The “write context”

The user-defined type `CTG_WR_CONTEXT` contains state information that is relevant by the output function. In this case there is a pointer to the start of the output buffer, `pOutput`, the size of the output buffer, `OutputLen`, and a pointer to the next output position to be written, `Cursor`.

❷ Accept write parameters

The function `_Wr` is responsible for taking the data given to it and somehow acting upon it. In this case we store the data to the output buffer, but other implementations could write the output to a file or to a network connection.

❸ Recover context

The incoming parameter `pWrCtx` is a blind pointer, but we know that a pointer to the client’s “write context” is passed in, so the pointer is unblinded by a cast.

❹ Write data

A `for` loop iterates over the incoming data bytes and transfers them to the output buffer. If the output buffer becomes full, an error indication is returned which immediately terminates compression.

❺ Set up write context

The write context is set up to deliver the compressed data into the array `_aOutput`.

❻ Compress the data

The call to `CTG_CompressM2F()` passes in the array to compress, the function to call when compressed data are ready, and a pointer to the user’s write context. On return, any error returned by the write function is propagated to the client.

Compile and run

To build the program, compile `CTG_M2F_Encode.c` with the emCompress-ToGo source code in `COMPRESS` and SEGGER software components in `SEGGER`. After it’s compiled and linked without errors, running the example prints the uncompressed and compressed sizes:

```
C:> CTG_M2F_Encode.exe

Compressed 1011 to 700 bytes.

C:> _
```


2.4 Function-to-memory decompression

Just as CTG_CompressM2F() can deliver its compressed data using a function, decompression can receive its compressed data using a function. In this way it's possible for the client, for instance, to read the input from a file or network connection and write the output to memory.

This example decompresses a valid bitstream delivered through repeated function calls and stores it into memory.

```
// File: CTG_F2M_Decode.c
// - Decompress function to memory.
//

#include "CTG.h"
#include <stdio.h>

static const U8 _aInput[] = {
  0xA2, 0x24, 0x8C, 0x18, 0x31, 0x65, 0xE4, 0xDC, 0x79, 0x33, 0x66, 0x4D,
  0x9E, 0x06, 0x0A, 0x40, 0x80, 0x10, 0x92, 0x05, 0x04, 0x93, 0x22, 0x57,
  0x92, 0x4C, 0x01, 0x31, 0x24, 0x88, 0x14, 0x29, 0x4F, 0x98, 0x30, 0x45,
  0x73, 0x09, 0x27, 0xA8, 0xDC, 0x09, 0x33, 0x07, 0x84, 0x18, 0x39, 0x69,
  0xD8, 0xB0, 0x49, 0x73, 0x86, 0x05, 0x88, 0x30, 0x6E, 0xC8, 0x80, 0xA0,
  0x83, 0xA6, 0x0C, 0x88, 0xB9, 0xA4, 0x13, 0x93, 0x97, 0xA8, 0x37, 0x76,
  0xCA, 0xCC, 0xDD, 0xF8, 0x05, 0x10, 0x49, 0x8B, 0xD6, 0x99, 0x3C, 0x72,
  0x11, 0xDF, 0x88, 0x33, 0x69, 0xDA, 0x88, 0x61, 0x4B, 0x54, 0x1A, 0xB7,
  0x9F, 0x0B, 0x87, 0x30, 0xA5, 0x93, 0x1D, 0x41, 0x21, 0x01, 0x44, 0x5B,
  0x84, 0xE6, 0x32, 0xEC, 0x2E, 0xC4, 0xA3, 0x3A, 0x25, 0x31, 0x6F, 0xE4,
  0xBC, 0x39, 0x3B, 0x16, 0x58, 0x86, 0x34, 0x04, 0x6D, 0x8A, 0xDA, 0xBC,
  0x69, 0x4B, 0x88, 0x9C, 0xB0, 0x44, 0x43, 0xDF, 0x9B, 0x3A, 0x74, 0xCE,
  0x92, 0x0C, 0x05, 0x2E, 0xD7, 0x08, 0x11, 0x42, 0xCA, 0x42, 0x67, 0xAF,
  0x8F, 0x2B, 0x4C, 0x35, 0xDA, 0xC2, 0xCA, 0x9C, 0x37, 0x6E, 0x42, 0x86,
  0x34, 0x50, 0x55, 0x4E, 0x4D, 0x98, 0xBB, 0x8C, 0x67, 0x27, 0x1C, 0x42,
  0x33, 0x75, 0x29, 0xCB, 0xC4, 0x4D, 0x1A, 0xC3, 0xB6, 0x14, 0x19, 0x4B,
  0xC6, 0x18, 0xB4, 0x0E, 0x0E, 0x24, 0xA4, 0x8E, 0x18, 0xB5, 0x44, 0x3A,
  0x47, 0x0E, 0xD9, 0x2C, 0xDB, 0x1C, 0x34, 0x75, 0xDC, 0x06, 0x52, 0x62,
  0x46, 0x4E, 0x5D, 0x1A, 0x4C, 0x1F, 0x61, 0x40, 0xA8, 0xEA, 0x50, 0x37,
  0xC7, 0xED, 0x39, 0x10, 0xB9, 0x62, 0x20, 0x95, 0x82, 0x37, 0x6F, 0xD6,
  0x80, 0x40, 0x93, 0x16, 0x35, 0xBB, 0xD4, 0x0A, 0x4E, 0x58, 0xCA, 0x36,
  0x97, 0x6A, 0x42, 0xE4, 0x4C, 0x10, 0xBA, 0x59, 0xB3, 0x33, 0xE4, 0x81,
  0x29, 0x35, 0x39, 0xCB, 0x54, 0x7C, 0x30, 0x29, 0x6B, 0x8B, 0x14, 0xDE,
  0x55, 0xCD, 0xCC, 0x5B, 0x32, 0xE7, 0xAC, 0x90, 0x73, 0x06, 0x0D, 0x9D,
  0xCB, 0x6C, 0x99, 0xF2, 0x96, 0xF6, 0x63, 0x8B, 0xD4, 0x96, 0xFD, 0xB3,
  0x49, 0xBD, 0xD6, 0x18, 0xAA, 0x90, 0x47, 0x97, 0x48, 0xA2, 0x11, 0x2A,
  0x1B, 0x68, 0x97, 0xAC, 0x84, 0x77, 0x52, 0xE9, 0x35, 0xCC, 0x87, 0x96,
  0x7B, 0x12, 0x6F, 0x97, 0xEA, 0xDB, 0x64, 0x1A, 0xA9, 0x4C, 0x1D, 0x33,
  0x66, 0xA9, 0x06, 0xDA, 0x50, 0xFD, 0x64, 0x6F, 0x78, 0xE9, 0xBC, 0xEB,
  0x77, 0xA0, 0xA3, 0xDC, 0xD5, 0xE3, 0x40, 0x94, 0xC9, 0x4B, 0x09, 0x31,
  0x20, 0xB3, 0xD4, 0x13, 0xDF, 0xA4, 0xA4, 0x4E, 0x19, 0x6A, 0x2C, 0xCA,
  0x30, 0x0D, 0x17, 0x6C, 0xA1, 0xD4, 0x1A, 0xA1, 0xC5, 0xBF, 0x65, 0xF3,
  0xED, 0xA4, 0x1A, 0x6C, 0xCE, 0x32, 0x17, 0xF1, 0x40, 0x4E, 0x57, 0x0F,
  0x31, 0x75, 0xE4, 0xFA, 0x05, 0x2D, 0xA8, 0x2D, 0x9C, 0xEB, 0x42, 0xA0,
  0xA6, 0x4B, 0xC2, 0x13, 0xB7, 0xB3, 0x64, 0xD6, 0x84, 0x00, 0x19, 0x27,
  0xA7, 0xF9, 0x27, 0xA5, 0xF9, 0x35, 0x9F, 0x1D, 0xD8, 0xD0, 0xB6, 0x56,
  0x18, 0x09, 0x8B, 0xFF, 0xFA, 0x23, 0xDC, 0xC2, 0x28, 0x26, 0x94, 0x96,
  0xBA, 0x52, 0x9A, 0x96, 0x9A, 0x31, 0xB9, 0x52, 0xAB, 0x11, 0x43, 0x97,
  0x59, 0x74, 0x25, 0x9B, 0x89, 0xAC, 0x4C, 0x23, 0x7F, 0x99, 0xC9, 0xA4,
  0x82, 0x50, 0x88, 0x64, 0xC8, 0xA7, 0x4B, 0xCF, 0xC0, 0x59, 0x08, 0xC5,
  0x56, 0xE0, 0x52, 0x6A, 0x0B, 0x25, 0x56, 0x0E, 0xF3, 0x67, 0xAE, 0x8B,
  0xF8, 0x36, 0x16, 0x79, 0xC3, 0xF4, 0xF1, 0x84, 0xED, 0x33, 0x1B, 0x5B,
  0xC4, 0xCF, 0x90, 0x87, 0xA1, 0x33, 0x70, 0xBC, 0x74, 0x1F, 0x24, 0x8C,
  0x9C, 0x36, 0x73, 0x7F, 0x46, 0xD9, 0x48, 0xD1, 0x67, 0x37, 0x62, 0xDE,
  0xE4, 0xB5, 0x13, 0x4F, 0x7D, 0x8E, 0xEC, 0x52, 0xBB, 0xE7, 0x40, 0x26,
  0x2C, 0x82, 0x2C, 0x86, 0x31, 0xB6, 0xF8, 0x04, 0xDA, 0x0C, 0x56, 0xD0,
  0x07, 0x59, 0x56, 0x62, 0x2C, 0x7C, 0xC9, 0xA1, 0x3B, 0xC3, 0xDC, 0x71,
  0xF4, 0x4C, 0xB2, 0x2A, 0xEF, 0x1B, 0x19, 0x14, 0xF8, 0xE2, 0x55, 0x55
};

static U8 _aOutput[sizeof(_aInput)*2];

typedef struct {  ❶
  const U8 * pInput;
  unsigned   InputLen;
  unsigned   Cursor;
```



```
} CTG_RD_CONTEXT;

static int _Rd(U8 *pData, unsigned DataLen, void *pRdCtx) {  ❷
  CTG_RD_CONTEXT * pCtx;
  unsigned         N;
  //
  pCtx = (CTG_RD_CONTEXT *)pRdCtx;  ❸
  for (N = 0; N < DataLen; ++N) {   ❹
    if (pCtx->Cursor >= pCtx->InputLen) {
      break;
    }
    *pData++ = pCtx->pInput[pCtx->Cursor++];
  }
  return N;
}

void MainTask(void);
void MainTask(void) {
  CTG_RD_CONTEXT RdCtx;
  int            DecodeLen;
  //
  RdCtx.pInput   = &_aInput[0];  ❺
  RdCtx.InputLen = sizeof(_aInput);
  RdCtx.Cursor   = 0;
  DecodeLen = CTG_DecompressF2M(_Rd, &RdCtx, ❻
                                &_aOutput[0], sizeof(_aOutput));
  if (DecodeLen < 0) {
    printf("Decompression error.\n");
  } else {
    printf("Decompressed %u to %u bytes.\n", sizeof(_aInput), DecodeLen);
  }
}
```

❶ The “read context”

The user-defined type `CTG_RD_CONTEXT` contains state information that is relevant by the input function. In this case there is a pointer to the start of the input buffer, `pInput`, the size of the input buffer, `InputLen`, and a pointer to the next input position to be read, `Cursor`.

❷ Accept read parameters

The function `_Rd` is responsible for reading source data and returning the number of bytes read to the caller. In this case we read the data from the input buffer, but other implementations could read from a file or from a network connection.

❸ Recover context

The incoming parameter `pRdCtx` is a blind pointer, but we know that a pointer to the client’s “read context” is passed in, so the pointer is unblinded by a cast.

❹ Read data

Return as much data as possible up to ’DataLen’. The function returns less bytes as ’DataLen’ or even 0 at the end of the input data.

❺ Set up read context

The read context is set up to read the compressed data from the array `_aInput`.

❻ Decompress the data

The call to `CTG_DecompressF2M()` passes in the function to call when more data are required, a pointer to the user’s read context, and the array that receives the decompressed data. On return, any error returned by the read function is propagated to the client.

Compile and run

To build the program, compile `CTG_F2M_Decode.c` with the emCompress-ToGo source code in `COMPRESS` and SEGGER software components in `SEGGER`. After it's compiled and linked without errors, running the example prints the uncompressed and compressed sizes:

```
C:> CTG_F2M_Decode.exe

Decompressed 576 to 1089 bytes.

C:> _
```


2.5 Adjusting compression parameters

In all examples the `Flags` parameter has been set to zero. This parameter adjusts how the compressor works and affects its runtime performance, i.e. how long it takes to find matches, and the compression ratio. As you might imagine, better compression takes longer, and in a real-time system you may wish to favor compression speed to reduce latency on a communications channel or you may wish to favor size when writing sensor data or a log file to a file system.

Currently the flags passed into the compressor control the window size.

2.5.1 Window size

The window size affects how much history is examined to find and compress a "string match": when a substring is received that matches part of the history, a reference to the match position is encoded in place of the matched string. Clearly, having more history provides more opportunities for a match but also reduces the runtime performance of the compressor because more history needs to be searched. There is also another consequence of larger windows: with larger windows, you have longer distances which, when encoded, require more bits in the output bitstream.

emCompress-ToGo has the capability of selecting history sizes from 256 bytes to 16 KB in powers of two. For low-latency real time systems, a small history of 256 bytes to 1 KB should suffice. For good compression, a medium history of 2KB or 4 KB is an excellent choice.

If this parameter is not specified, it defaults to a match distance of 256 bytes.

See *Compressor encoding flags* on page 35 for the compression flags you can select.

2.5.2 A small example

The effect of the window size upon compression for a sample input file (HTML code, size 324751 bytes) is shown in the following table.

Window size	Compressed file size	
256 bytes	121586 bytes	37.4 %
512 bytes	103809 bytes	31.9 %
1k	91455 bytes	28.1 %
2k	67199 bytes	20.6 %
4k	51234 bytes	15.7 %
8k	48802 bytes	15.0 %
16k	47588 bytes	14.6 %

2.6 Function-to-memory compression

The compression modes function-to-memory and function-to-function require an additional parameter of a "work buffer" to maintain history information to make forward match decisions. These modes are useful when the uncompressed data cannot fit into the target's memory or the source data is stored externally, such as in a file or in a serial flash, and can only be read incrementally.

The size of this work buffer can be determined using the macro `CTG_COMPRESS_WS_SIZE`. The compression functions ensure that the work buffer is large enough and is consistent with the selected compression flags: if there is a problem, the compression function returns an error.

The functions `CTG_CompressF2M()` and `CTG_CompressF2F()` use the same functional parameters seen in function-to-memory decompression, and as such this mechanism is not described further. The example `CTG_F2M_Encode.c` demonstrates the concepts that have been used before and shows how the buffer is provided:

```
// File: CTG_F2M_Encode.c
// - Compress function to memory
//

#include "CTG.h"
#include <stdio.h>

static const U8 _aInput[] = {
  "Jabberwocky\n  BY LEWIS CARROLL\n\n\n"
  "'Twas brillig, and the slithy toves\n   Did gyre and gimble in the wabe:\n"
  "All mimsy were the borogoves,\n   And the mome raths outgrabe.\n\n"
  "\"Beware the Jabberwock, my son!\n   The jaws that bite, the claws that catch!\n"
  "Beware the Jubjub bird, and shun\n   The frumious Bandersnatch!\"\n\n"
  "He took his vorpal sword in hand;\n   Long time the manxome foe he sought-\n"
  "So rested he by the Tumtum tree\n   And stood awhile in thought.\n\n"
  "And, as in uffish thought he stood,\n   The Jabberwock, with eyes of flame,\n"
  "Came whiffling through the tulgey wood,\n   And burbled as it came!\n\n"
  "One, two! One, two! And through and through\n   The vorpal blade went snicker-snack!\n\n"
  "He left it dead, and with its head\n   He went galumphing back.\n\n"
  "\"And hast thou slain the Jabberwock?\n   Come to my arms, my beamish boy!\n"
  "O frabjous day! Callooh! Callay!\"\n\n   He chortled in his joy.\n"
  "'Twas brillig, and the slithy toves\n   Did gyre and gimble in the wabe:\n"
  "All mimsy were the borogoves,\n   And the mome raths outgrabe.\n"
};

static U8 _aOutput[sizeof(_aInput)*2];

typedef struct {
  const U8 * pInput;
  unsigned   InputLen;
  unsigned   Cursor;
} CTG_RD_CONTEXT;

static int _Rd(U8 *pData, unsigned DataLen, void *pRdCtx) {
  CTG_RD_CONTEXT * pCtx;
  unsigned         N;
  //
  pCtx = (CTG_RD_CONTEXT *)pRdCtx;
  for (N = 0; N < DataLen; ++N) {
    if (pCtx->Cursor >= pCtx->InputLen) {
      break;
    }
    *pData++ = pCtx->pInput[pCtx->Cursor++];
  }
  return N;
}

void MainTask(void);
void MainTask(void) {
  CTG_RD_CONTEXT RdCtx;
  int            Status;
  U8             aWork[CTG_COMPRESS_WS_SIZE(CTG_FLAG_WINDOW_SIZE_1K)];  ❶
  //
  RdCtx.pInput   = &_aInput[0];
```



```
  RdCtx.InputLen = sizeof(_aInput);
  RdCtx.Cursor   = 0;
  //
  Status = CTG_CompressF2M(_Rd, &RdCtx,  ❷
                           &_aOutput[0], sizeof(_aOutput),
                           &aWork[0], sizeof(aWork),  ❸
                           CTG_FLAG_WINDOW_SIZE_1K);  ❹
  if (Status >= 0) {
    printf("Compressed %u to %u bytes.\n", sizeof(_aInput), Status);
  } else {
    printf("Compression error.\n");
  }
}
```

❶ Declare work buffer

The work buffer is sized to the compression parameters we will select.

❷ Provide read context

The "read function and context" mechanism is identical to the decompress function-to-memory example.

❸ Provide work buffer

The work buffer and its size are provided to the compression function.

❹ Compress with nondefault flags

The selected flag of a 1 KB maximum match distance is consistent with the work buffer provided to the function.

2.7 Memory-to-function decompression

The decompression modes memory-to-function and function-to-function require an additional parameter of a "work buffer" to maintain history information to copy backward references.

The size of this work buffer can be determined using the macro `CTG_DECOMPRESS_WS_SIZE`. The decompression functions ensure that the work buffer is large enough and is consistent with the selected compression flags: if there is a problem, the decompression function returns an error.

The functions `CTG_DecompressM2F()` and `CTG_DecompressF2F()` use the same functional parameters seen before and this mechanism is not described further. The example `CTG_M2F_Decode.c` demonstrates the concepts that have been used before and shows how the buffer is provided:

```
// File: CTG_M2F_Decode.c
// - Decompress memory to function.
//

#include "CTG.h"
#include <stdio.h>

static const U8 _aInput[] = {
  0xA2, 0x24, 0x8C, 0x18, 0x31, 0x65, 0xE4, 0xDC, 0x79, 0x33, 0x66, 0x4D,
  0x9E, 0x06, 0x0A, 0x40, 0x80, 0x10, 0x92, 0x05, 0x04, 0x93, 0x22, 0x57,
  0x92, 0x4C, 0x01, 0x31, 0x24, 0x88, 0x14, 0x29, 0x4F, 0x98, 0x30, 0x45,
  0x73, 0x09, 0x27, 0xA8, 0xDC, 0x09, 0x33, 0x07, 0x84, 0x18, 0x39, 0x69,
  0xD8, 0xB0, 0x49, 0x73, 0x86, 0x05, 0x88, 0x30, 0x6E, 0xC8, 0x80, 0xA0,
  0x83, 0xA6, 0x0C, 0x88, 0xB9, 0xA4, 0x13, 0x93, 0x97, 0xA8, 0x37, 0x76,
  0xCA, 0xCC, 0xDD, 0xF8, 0x05, 0x10, 0x49, 0x8B, 0xD6, 0x99, 0x3C, 0x72,
  0x11, 0xDF, 0x88, 0x33, 0x69, 0xDA, 0x88, 0x61, 0x4B, 0x54, 0x1A, 0xB7,
  0x9F, 0x0B, 0x87, 0x30, 0xA5, 0x93, 0x1D, 0x41, 0x21, 0x01, 0x44, 0x5B,
  0x84, 0xE6, 0x32, 0xEC, 0x2E, 0xC4, 0xA3, 0x3A, 0x25, 0x31, 0x6F, 0xE4,
  0xBC, 0x39, 0x3B, 0x16, 0x58, 0x86, 0x34, 0x04, 0x6D, 0x8A, 0xDA, 0xBC,
  0x69, 0x4B, 0x88, 0x9C, 0xB0, 0x44, 0x43, 0xDF, 0x9B, 0x3A, 0x74, 0xCE,
  0x92, 0x0C, 0x05, 0x2E, 0xD7, 0x08, 0x11, 0x42, 0xCA, 0x42, 0x67, 0xAF,
  0x8F, 0x2B, 0x4C, 0x35, 0xDA, 0xC2, 0xCA, 0x9C, 0x37, 0x6E, 0x42, 0x86,
  0x34, 0x50, 0x55, 0x4E, 0x4D, 0x98, 0xBB, 0x8C, 0x67, 0x27, 0x1C, 0x42,
  0x33, 0x75, 0x29, 0xCB, 0xC4, 0x4D, 0x1A, 0xC3, 0xB6, 0x14, 0x19, 0x4B,
  0xC6, 0x18, 0xB4, 0x0E, 0x0E, 0x24, 0xA4, 0x8E, 0x18, 0xB5, 0x44, 0x3A,
  0x47, 0x0E, 0xD9, 0x2C, 0xDB, 0x1C, 0x34, 0x75, 0xDC, 0x06, 0x52, 0x62,
  0x46, 0x4E, 0x5D, 0x1A, 0x4C, 0x1F, 0x61, 0x40, 0xA8, 0xEA, 0x50, 0x37,
  0xC7, 0xED, 0x39, 0x10, 0xB9, 0x62, 0x20, 0x95, 0x82, 0x37, 0x6F, 0xD6,
  0x80, 0x40, 0x93, 0x16, 0x35, 0xBB, 0xD4, 0x0A, 0x4E, 0x58, 0xCA, 0x36,
  0x97, 0x6A, 0x42, 0xE4, 0x4C, 0x10, 0xBA, 0x59, 0xB3, 0x33, 0xE4, 0x81,
  0x29, 0x35, 0x39, 0xCB, 0x54, 0x7C, 0x30, 0x29, 0x6B, 0x8B, 0x14, 0xDE,
  0x55, 0xCD, 0xCC, 0x5B, 0x32, 0xE7, 0xAC, 0x90, 0x73, 0x06, 0x0D, 0x9D,
  0xCB, 0x6C, 0x99, 0xF2, 0x96, 0xF6, 0x63, 0x8B, 0xD4, 0x96, 0xFD, 0xB3,
  0x49, 0xBD, 0xD6, 0x18, 0xAA, 0x90, 0x47, 0x97, 0x48, 0xA2, 0x11, 0x2A,
  0x1B, 0x68, 0x97, 0xAC, 0x84, 0x77, 0x52, 0xE9, 0x35, 0xCC, 0x87, 0x96,
  0x7B, 0x12, 0x6F, 0x97, 0xEA, 0xDB, 0x64, 0x1A, 0xA9, 0x4C, 0x1D, 0x33,
  0x66, 0xA9, 0x06, 0xDA, 0x50, 0xFD, 0x64, 0x6F, 0x78, 0xE9, 0xBC, 0xEB,
  0x77, 0xA0, 0xA3, 0xDC, 0xD5, 0xE3, 0x40, 0x94, 0xC9, 0x4B, 0x09, 0x31,
  0x20, 0xB3, 0xD4, 0x13, 0xDF, 0xA4, 0xA4, 0x4E, 0x19, 0x6A, 0x2C, 0xCA,
  0x30, 0x0D, 0x17, 0x6C, 0xA1, 0xD4, 0x1A, 0xA1, 0xC5, 0xBF, 0x65, 0xF3,
  0xED, 0xA4, 0x1A, 0x6C, 0xCE, 0x32, 0x17, 0xF1, 0x40, 0x4E, 0x57, 0x0F,
  0x31, 0x75, 0xE4, 0xFA, 0x05, 0x2D, 0xA8, 0x2D, 0x9C, 0xEB, 0x42, 0xA0,
  0xA6, 0x4B, 0xC2, 0x13, 0xB7, 0xB3, 0x64, 0xD6, 0x84, 0x00, 0x19, 0x27,
  0xA7, 0xF9, 0x27, 0xA5, 0xF9, 0x35, 0x9F, 0x1D, 0xD8, 0xD0, 0xB6, 0x56,
  0x18, 0x09, 0x8B, 0xFF, 0xFA, 0x23, 0xDC, 0xC2, 0x28, 0x26, 0x94, 0x96,
  0xBA, 0x52, 0x9A, 0x96, 0x9A, 0x31, 0xB9, 0x52, 0xAB, 0x11, 0x43, 0x97,
  0x59, 0x74, 0x25, 0x9B, 0x89, 0xAC, 0x4C, 0x23, 0x7F, 0x99, 0xC9, 0xA4,
  0x82, 0x50, 0x88, 0x64, 0xC8, 0xA7, 0x4B, 0xCF, 0xC0, 0x59, 0x08, 0xC5,
  0x56, 0xE0, 0x52, 0x6A, 0x0B, 0x25, 0x56, 0x0E, 0xF3, 0x67, 0xAE, 0x8B,
  0xF8, 0x36, 0x16, 0x79, 0xC3, 0xF4, 0xF1, 0x84, 0xED, 0x33, 0x1B, 0x5B,
  0xC4, 0xCF, 0x90, 0x87, 0xA1, 0x33, 0x70, 0xBC, 0x74, 0x1F, 0x24, 0x8C,
  0x9C, 0x36, 0x73, 0x7F, 0x46, 0xD9, 0x48, 0xD1, 0x67, 0x37, 0x62, 0xDE,
  0xE4, 0xB5, 0x13, 0x4F, 0x7D, 0x8E, 0xEC, 0x52, 0xBB, 0xE7, 0x40, 0x26,
  0x2C, 0x82, 0x2C, 0x86, 0x31, 0xB6, 0xF8, 0x04, 0xDA, 0x0C, 0x56, 0xD0,
  0x07, 0x59, 0x56, 0x62, 0x2C, 0x7C, 0xC9, 0xA1, 0x3B, 0xC3, 0xDC, 0x71,
  0xF4, 0x4C, 0xB2, 0x2A, 0xEF, 0x1B, 0x19, 0x14, 0xF8, 0xE2, 0x55, 0x55
};
```



```
static U8 _aOutput[sizeof(_aInput)*2];

typedef struct {
  U8       * pOutput;
  unsigned   OutputLen;
  unsigned   Cursor;
} CTG_WR_CONTEXT;

static int _Wr(const U8 *pData, unsigned DataLen, void *pWrCtx) {
  CTG_WR_CONTEXT * pCtx;
  unsigned         N;
  //
  pCtx = (CTG_WR_CONTEXT *)pWrCtx;
  for (N = 0; N < DataLen; ++N) {
    if (pCtx->Cursor >= pCtx->OutputLen) {
      return CTG_STATUS_OUTPUT_OVERFLOW;
    }
    pCtx->pOutput[pCtx->Cursor++] = *pData++;
  }
  return N;
}

void MainTask(void);
void MainTask(void) {
  CTG_WR_CONTEXT WrCtx;
  U8             aWork[CTG_DECOMPRESS_WS_SIZE(CTG_FLAG_WINDOW_SIZE_1K)];  ❶
  int            DecodeLen;
  //
  WrCtx.pOutput   = &_aOutput[0];
  WrCtx.OutputLen = sizeof(_aOutput);
  WrCtx.Cursor    = 0;
  DecodeLen = CTG_DecompressM2F(&_aInput[0], sizeof(_aInput),
                                _Wr, &WrCtx,
                                &aWork[0], sizeof(aWork));  ❷
  if (DecodeLen < 0) {
    printf("Decompression error.\n");
  } else {
    printf("Decompressed %u to %u bytes.\n", sizeof(_aInput), DecodeLen);
  }
}
```

❶ Declare work buffer

The work buffer is sized to the compression parameters used to compress the input stream. In this case the window size was selected to be 1 KB and therefore the buffer is sized appropriately.

❷ Provide work buffer

The work buffer and its size are provided to the decompression function to use and check correctness.

2.8 Fast compression

emCompress-ToGo also provides compression functions that are much faster than the ordinary ones, but require an additional or bigger “work buffer” in RAM to speed up the operation.

The size of this work buffer can be determined using the macros `CTG_FCOMPRESS_WS_SIZE()` and `CTG_FCOMPRESS_M_WS_SIZE()`.

See `CTG_FastCompressM2M()`.

For details on the performance see *Runtime performance* on page 89.

Chapter 3

emCompress-ToGo API

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

3.1 Preprocessor definitions

3.1.1 Version number

Description

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

Definition

```
#define CTG_VERSION    33000
```

Symbols

Definition	Description
CTG_VERSION	Format is “Mmmrr” so, for example, 31000 corresponds to version 3.10.

3.1.2 Function status values

Description

Compression and decompression errors.

Definition

```
#define CTG_STATUS_OUTPUT_OVERFLOW     (-100)
#define CTG_STATUS_INPUT_UNDERFLOW     (-101)
#define CTG_STATUS_BUFFER_TOO_SMALL    (-102)
#define CTG_STATUS_BAD_PARAMETER       (-103)
#define CTG_STATUS_DECODING_ERROR      (-104)
```

Symbols

Definition	Description
CTG_STATUS_OUTPUT_OVERFLOW	The output buffer is not big enough to contain the compressed or decompressed output.
CTG_STATUS_INPUT_UNDERFLOW	The input buffer contains a bitstream that is truncated.
CTG_STATUS_BUFFER_TOO_SMALL	The work buffer provided to the compressor or decompressor is too small.
CTG_STATUS_BAD_PARAMETER	A parameter given to the function is invalid.
CTG_STATUS_DECODING_ERROR	Error while decoding, bad encoded data stream.

Additional information

Values returned as errors by the compression and decompression functions. All errors are negative.

3.1.3 Workspace buffer sizes

Description

Gives the size of the workspace required for compression and decompression depending on the 'Flags' parameter (CTG_FLAG_WINDOW_SIZE_...) used during compression.

Definition

```
#define CTG_COMPRESS_WS_SIZE(Flags)
  ((0x100uL << (Flags)) + CTG_MAX_REFERENCE_LEN)
#define CTG_DECOMPRESS_WS_SIZE(Flags)     (0x100uL << (Flags))
#define CTG_FCOMPRESS_M_WS_SIZE(Flags)    ((0x200uL << (Flags)) + 512u)
#define CTG_FCOMPRESS_WS_SIZE(Flags)
  ((0x300uL << (Flags)) + CTG_MAX_REFERENCE_LEN + 512u)
```

Symbols

Definition	Description
CTG_COMPRESS_WS_SIZE(Flags)	Gives the workspace size required for compression functions depending on the window size used.
CTG_DECOMPRESS_WS_SIZE(Flags)	Gives the workspace size required for decompression functions depending on the window size used for compression.
CTG_FCOMPRESS_M_WS_SIZE(Flags)	Gives the workspace size required for the fast compression functions from memory depending on the window size used.
CTG_FCOMPRESS_WS_SIZE(Flags)	Gives the workspace size required for the fast compression functions depending on the window size used.

3.2 Compressor encoding flags

3.2.1 Window size parameter

Description

Sets the window size to use when compressing.

Definition

```
#define CTG_FLAG_WINDOW_SIZE_256    0u
#define CTG_FLAG_WINDOW_SIZE_512    1u
#define CTG_FLAG_WINDOW_SIZE_1K     2u
#define CTG_FLAG_WINDOW_SIZE_2K     3u
#define CTG_FLAG_WINDOW_SIZE_4K     4u
#define CTG_FLAG_WINDOW_SIZE_8K     5u
#define CTG_FLAG_WINDOW_SIZE_16K    6u
```

Symbols

Definition	Description
CTG_FLAG_WINDOW_SIZE_256	Window set to 256 bytes
CTG_FLAG_WINDOW_SIZE_512	Window set to 512 bytes
CTG_FLAG_WINDOW_SIZE_1K	Window set to 1K bytes
CTG_FLAG_WINDOW_SIZE_2K	Window set to 2K bytes
CTG_FLAG_WINDOW_SIZE_4K	Window set to 4K bytes
CTG_FLAG_WINDOW_SIZE_8K	Window set to 8K bytes
CTG_FLAG_WINDOW_SIZE_16K	Window set to 16K bytes

Additional information

Smaller windows will make compression run faster, larger windows tend to make the compressed output smaller.

3.3 Data types

3.3.1 CTG_STREAM

Description

Streaming interface for compression / decompression.

Type definition

```
typedef struct {
  unsigned   AvailIn;
  const U8 * pIn;
  int        Flush;
  unsigned   AvailOut;
  U8       * pOut;
} CTG_STREAM;
```

Structure members

Member	Description
AvailIn	Number of elements available as input octets.
pIn	Pointer to available input octets.
Flush	Set to 1, if no more input data are available.
AvailOut	Size of buffer for output data.
pOut	Pointer to the buffer for output data.

3.3.2 CTG_RD_FUNC

Description

Read data form source. The function must provide at least one byte of data to the caller, except if the end of the input data is reached. If the function returns a negative value, the compression/decompression operation will be aborted and the value if forward to the caller of the compression/decompression function.

Type definition

```
typedef int (CTG_RD_FUNC)(U8     * pData,
                          unsigned DataLen,
                          void   * pRdCtx);
```

Parameters

Parameter	Description
pData	Pointer to object that receives the data.
DataLen	Maximum number of bytes to read into object, will be nonzero.
pRdCtx	Client-supplied read context.

Return value

< 0 Error reading data.
= 0 Success, end of stream, no further data.
> 0 Success, number of bytes placed into object, more data to read.

3.3.3 CTG_WR_FUNC

Description

Write data to sink. The function must consume all 'DataLen' bytes of data. If the function returns a negative value, the compression/decompression operation will be aborted and the value if forward to the caller of the compression/decompression function.

Type definition

```
typedef int (CTG_WR_FUNC)(const U8     * pData,
                          unsigned DataLen,
                          void   * pWrCtx);
```

Parameters

Parameter	Description
pData	Pointer to object to write.
DataLen	Number of bytes to write.
pWrCtx	Client-supplied read context.

Return value

< 0 Error writing data.
= 0 Success.

3.4 Compression functions

emCompress-ToGo defines the following compression functions:

Function	Description
One-call compression functions	
`CTG_CompressM2M()`	Compress, memory to memory.
`CTG_CompressM2F()`	Compress, memory to function.
`CTG_CompressF2M()`	Compress, function to memory.
`CTG_CompressF2F()`	Compress, function to function.
`CTG_FastCompressM2M()`	Compress, memory to memory.
Multi-call compression functions	
`CTG_CompressInit()`	Initialize for compression with `CTG_Compress()`.
`CTG_Compress()`	Run compress on (part of) data.
`CTG_FastCompressInit()`	Initialize for compression with `CTG_FastCompress()`.
`CTG_FastCompress()`	Run compress on (part of) data.
`CTG_CompressReadInit()`	Initialize for compression with `CTG_CompressRead()`.
`CTG_CompressRead()`	Read compressed data from a compression-stream.
`CTG_CompressWriteInit()`	Initialize for compression with `CTG_CompressWrite()`.
`CTG_CompressWrite()`	Write data to be compressed to a compression-stream.

3.4.1 CTG_CompressM2M()

Description

Compress, memory to memory.

Prototype

```
int CTG_CompressM2M(const U8       * pInput,
                    unsigned         InputLen,
                    U8       * pOutput,
                    unsigned         OutputSize,
                    unsigned         Flags);
```

Parameters

Parameter	Description
pInput	Pointer to octet string to encode.
InputLen	Byte length of the input octet string.
pOutput	Pointer to the buffer that receives the compressed output.
OutputSize	Size of the output buffer in bytes.
Flags	Window size to be used for compression, see CTG_FLAG_WINDOW_SIZE_… macros.

Return value

≥ 0 Success, compressed output length.
< 0 Error indication.

Example

See *Memory-to-memory compression* on page 16.

See also

Function status values on page 33 and *Compressor encoding flags* on page 35.

3.4.2 CTG_CompressM2F()

Description

Compress, memory to function.

Prototype

```
int CTG_CompressM2F(const U8          * pInput,
                    unsigned            InputLen,
                    CTG_WR_FUNC       * pfWr,
                    void              * pWrCtx,
                    unsigned            Flags);
```

Parameters

Parameter	Description
pInput	Pointer to octet string to encode.
InputLen	Octet length of the input octet string.
pfWr	Pointer to function that writes output.
pWrCtx	Pointer to user-supplied write context.
Flags	Encoding flags.

Return value

≥ 0 Success, compressed output length.
< 0 Error indication.

Example

See *Memory-to-function compression* on page 20.

See also

Function status values on page 33 and *Compressor encoding flags* on page 35.

3.4.3 CTG_CompressF2M()

Description

Compress, function to memory.

Prototype

```
int CTG_CompressF2M(CTG_RD_FUNC * pfRd,
                    void        * pRdCtx,
                    U8          * pOutput,
                    unsigned      OutputSize,
                    U8          * pWork,
                    unsigned      WorkLen,
                    unsigned      Flags);
```

Parameters

Parameter	Description
pfRd	Pointer to function that reads input.
pRdCtx	Pointer to user-supplied context for read function.
pOutput	Pointer to buffer that receives the compressed output.
OutputSize	Size of the output buffer in bytes.
pWork	Pointer to work buffer.
WorkLen	Length of work buffer.
Flags	Encoding flags.

Return value

≥ 0 Success, compressed output length.
< 0 Error indication.

Additional information

The work buffer size (value of WorkLen) must be at least CTG_COMPRESS_WS_SIZE(Flags). This requirement is enforced by the compressor and, should the buffer be too small, CTG_STATUS_BUFFER_TOO_SMALL is returned by the compress functions.

Example

See *Function-to-memory compression* on page 26.

See also

Function status values on page 33 and *Compressor encoding flags* on page 35.

3.4.4 CTG_CompressF2F()

Description

Compress, function to function.

Prototype

```
int CTG_CompressF2F(CTG_RD_FUNC * pfRd,
                    void        * pRdCtx,
                    CTG_WR_FUNC * pfWr,
                    void        * pWrCtx,
                    U8          * pWork,
                    unsigned      WorkLen,
                    unsigned      Flags);
```

Parameters

Parameter	Description
pfRd	Pointer to function that reads input.
pRdCtx	Pointer to user-supplied context for read function.
pfWr	Pointer to function that writes output.
pWrCtx	Pointer to user-supplied write context.
pWork	Pointer to work buffer.
WorkLen	Length of work buffer.
Flags	Encoding flags.

Return value

≥ 0 Success, compressed output length.
< 0 Error indication.

Additional information

The work buffer size (value of WorkLen) must be at least CTG_COMPRESS_WS_SIZE(Flags). This requirement is enforced by the compressor and, should the buffer be too small, CTG_STATUS_BUFFER_TOO_SMALL is returned by the compress functions.

Example

See *Function-to-memory compression* on page 26.

See also

Function status values on page 33 and *Compressor encoding flags* on page 35.

3.4.5 CTG_FastCompressM2M()

Description

Compress, memory to memory. Much faster than `CTG_CompressM2M()`.

Prototype

```
int CTG_FastCompressM2M(const U8       * pInput,
                        unsigned   InputLen,
                        U8       * pOutput,
                        unsigned   OutputSize,
                        U16      * pWork,
                        unsigned   WorkLen,
                        unsigned   Flags);
```

Parameters

Parameter	Description
`pInput`	Pointer to octet string to encode.
`InputLen`	Byte length of the input octet string.
`pOutput`	Pointer to the buffer that receives the compressed output.
`OutputSize`	Size of the output buffer in bytes.
`pWork`	Pointer to work buffer.
`WorkLen`	Length of work buffer in bytes.
`Flags`	Window size to be used for compression, see `CTG_FLAG_WINDOW_SIZE_...` macros.

Return value

≥ 0 Success, compressed output length.
< 0 Error indication.

Additional information

The work buffer size (value of `WorkLen`) must be at least `CTG_FCOMPRESS_M_WS_SIZE(Flags)`. This requirement is enforced by the compressor and, should the buffer be too small, `CTG_STATUS_BUFFER_TOO_SMALL` is returned by the compress function.

See also

Function status values on page 33 and *Compressor encoding flags* on page 35.

3.4.6 CTG_CompressInit()

Description

Initialize for compression with `CTG_Compress()`.

Prototype

```
void CTG_CompressInit(CTG_COMPRESS_CTX * pCtx,
                      U8               * pWork,
                      unsigned           WorkLen,
                      unsigned           Flags);
```

Parameters

Parameter	Description
`pCtx`	Pointer to compress context to be initialized.
`pWork`	Pointer to work buffer.
`WorkLen`	Length of work buffer.
`Flags`	Window size to be used for compression, see `CTG_FLAG_WINDOW_SIZE_...` macros.

Additional information

The work buffer size (value of `WorkLen`) must be at least `CTG_COMPRESS_WS_SIZE(Flags)`. This requirement is enforced by the compressor and, should the buffer be too small, `CTG_STATUS_BUFFER_TOO_SMALL` is returned by the compress function.

Example

See *`CTG_Util.c` complete listing* on page 72.

See also

Compressor encoding flags on page 35.

3.4.7 CTG_Compress()

Description

Run compress on (part of) data. The function will process as many data from the input stream as possible and copy the compressed data to the output stream. While processing input data, `pStream->pIn` is increased and `pStream->AvailIn` is reduced. While generating output data, `pStream->pOut` is increased and `pStream->AvailOut` is reduced. The function stops if either `pStream->AvailIn` or `pStream->AvailOut` reaches 0. On calling this function, the data stream must meet the following conditions: `pStream->AvailOut > 0 && (pStream->AvailIn > 0 || pStream->Flush > 0)`.

Prototype

```
int CTG_Compress(CTG_COMPRESS_CTX * pCtx,
                 CTG_STREAM       * pStream);
```

Parameters

Parameter	Description
`pCtx`	Pointer to compress context, created by `CTG_CompressInit()`.
`pStream`	Pointer to data stream (input and output).

Return value

< 0	Error indication.
= 0	Processing suspended, needs more input data or more output buffer. Then the function must be called again to resume processing.
> 0	Compression finished successfully.

Example

See *`CTG_Util.c` complete listing* on page 72.

See also

Function status values on page 33

3.4.8 CTG_FastCompressInit()

Description

Initialize for compression with `CTG_FastCompress()`.

Prototype

```
void CTG_FastCompressInit(CTG_FCOMPRESS_CTX * pCtx,
                          U16               * pWork,
                          unsigned            WorkLen,
                          unsigned            Flags);
```

Parameters

Parameter	Description
`pCtx`	Pointer to compress context to be initialized.
`pWork`	Pointer to work buffer.
`WorkLen`	Length of work buffer.
`Flags`	Window size to be used for compression, see `CTG_FLAG_WINDOW_SIZE_...` macros.

Additional information

The work buffer size (value of `WorkLen`) must be at least `CTG_FCOMPRESS_WS_SIZE(Flags)`. This requirement is enforced by the compressor and, should the buffer be too small, `CTG_STATUS_BUFFER_TOO_SMALL` is returned by the compress function.

See also

Compressor encoding flags on page 35.

3.4.9 CTG_FastCompress()

Description

Run compress on (part of) data. Much faster than `CTG_Compress()`. The function will process as many data from the input stream as possible and copy the compressed data to the output stream. While processing input data, `pStream->pIn` is increased and `pStream->AvailIn` is reduced. While generating output data, `pStream->pOut` is increased and `pStream->AvailOut` is reduced. The function stops if either `pStream->AvailIn` or `pStream->AvailOut` reaches 0. On calling this function, the data stream must meet the following conditions: `pStream->AvailOut > 0 && (pStream->AvailIn > 0 || pStream->Flush > 0)`.

Prototype

```
int CTG_FastCompress(CTG_FCOMPRESS_CTX * pCtx,
                     CTG_STREAM        * pStream);
```

Parameters

Parameter	Description
`pCtx`	Pointer to compress context, created by `CTG_FastCompressInit()`.
`pStream`	Pointer to data stream (input and output).

Return value

< 0	Error indication.
= 0	Processing suspended, needs more input data or more output buffer. Then the function must be called again to resume processing.
> 0	Compression finished successfully.

See also

Function status values on page 33

3.4.10 CTG_CompressReadInit()

Description

Initialize for compression with `CTG_CompressRead()`.

Prototype

```
void CTG_CompressReadInit(CTG_COMPRESS_RD_CTX * pCtx,
                          U8                  * pWork,
                          unsigned              WorkLen,
                          U8                  * pBuff,
                          unsigned              BuffSize,
                          CTG_RD_FUNC         * pfRd,
                          void                * pRdCtx,
                          unsigned              Flags);
```

Parameters

Parameter	Description
`pCtx`	Pointer to compress context to be initialized.
`pWork`	Pointer to work buffer.
`WorkLen`	Length of work buffer.
`pBuff`	Pointer to a buffer used for input data. This buffer is used when calling the read function: `pfRd(pBuff, BuffSize, pRdCtx);`
`BuffSize`	Size of the buffer in bytes. Every buffer size ≥ 1 will work. However, small sizes will decrease performance.
`pfRd`	Pointer to function that reads input (uncompressed data).
`pRdCtx`	Pointer to user-supplied context for read function.
`Flags`	Window size to be used for compression, see `CTG_FLAG_WINDOW_SIZE_...` macros.

Additional information

The work buffer size (value of `WorkLen`) must be at least `CTG_COMPRESS_WS_SIZE(Flags)`. This requirement is enforced by the compressor and, should the buffer be too small, `CTG_STATUS_BUFFER_TOO_SMALL` is returned by the compress function.

See also

Compressor encoding flags on page 35.

3.4.11 CTG_CompressRead()

Description

Read compressed data from a compression-stream. Input data to be compressed is fetched using the callback read function given via `CTG_CompressReadInit()`. To read the whole compressed data this function must be called repeatedly until it returns a value < `NumBytes`.

Prototype

```
int CTG_CompressRead(CTG_COMPRESS_RD_CTX * pCtx,
                     U8                  * pBuff,
                     unsigned              NumBytes);
```

Parameters

Parameter	Description
`pCtx`	Pointer to compress context, created by `CTG_CompressReadInit()`.
`pBuff`	Pointer to the buffer to store the compressed data.
`NumBytes`	Number of bytes to be read.

Return value

≥ 0	Success, number of bytes read from stream. A value < `NumBytes` indicates the end of the compressed data.
< 0	Error indication.

See also

Function status values on page 33

3.4.12 CTG_CompressWriteInit()

Description

Initialize for compression with `CTG_CompressWrite()`.

Prototype

```
void CTG_CompressWriteInit(CTG_COMPRESS_WR_CTX * pCtx,
                           U8                  * pWork,
                           unsigned              WorkLen,
                           U8                  * pBuff,
                           unsigned              BuffSize,
                           CTG_WR_FUNC         * pfWr,
                           void                * pWrCtx,
                           unsigned              Flags);
```

Parameters

Parameter	Description
`pCtx`	Pointer to compress context to be initialized.
`pWork`	Pointer to work buffer.
`WorkLen`	Length of work buffer.
`pBuff`	Pointer to a buffer used to collect output data. This buffer is used when calling the write function: `pfWr(pBuff`, Len, `pWrCtx);`
`BuffSize`	Size of the buffer in bytes. Every buffer size ≥ 1 will work. However, small sizes will decrease performance.
`pfWr`	Pointer to function that writes output (compressed data).
`pWrCtx`	Pointer to user-supplied write context.
`Flags`	Window size to be used for compression, see `CTG_FLAG_WINDOW_SIZE_…` macros.

Additional information

The work buffer size (value of `WorkLen`) must be at least `CTG_COMPRESS_WS_SIZE(Flags)`. This requirement is enforced by the compressor and, should the buffer be too small, `CTG_STATUS_BUFFER_TOO_SMALL` is returned by the compress function.

See also

Compressor encoding flags on page 35.

3.4.13 CTG_CompressWrite()

Description

Write data to be compressed to a compression-stream. Compressed data is output using the callback write function given via `CTG_CompressWriteInit()`. This function must be called repeatedly until all data to be compressed are written. To indicate the end of the data to be compressed, the function must be called with 'NumBytes' = 0.

Prototype

```
int CTG_CompressWrite(      CTG_COMPRESS_WR_CTX * pCtx,
                      const U8                  * pData,
                            unsigned              NumBytes);
```

Parameters

Parameter	Description
pCtx	Pointer to compress context, created by `CTG_CompressWriteInit()`.
pData	Pointer to the uncompressed data to be compressed.
NumBytes	Number of bytes to write.

Return value

= 0 Success.
< 0 Error indication.

See also

Function status values on page 33

Example

```
static int _Fd;

//
// Write callback function that stores data into a file.
// The function is called for every 512 bytes of compressed data.
//
static int _WriteCb(const U8 *pData, unsigned DataLen, void *pWrCtx) {
  FileWrite(_Fd, pData, DataLen);
  return 0;
}

//
// Main function: Collect sensor data and write them compressed to a file.
//
void main() {
  U8  Len;
  U8  Buff[100];
  U8  FileBuff[512];
  U8  WorkBuff[CTG_COMPRESS_WS_SIZE(CTG_FLAG_WINDOW_SIZE_1K)];
  CTG_COMPRESS_WR_CTX Ctx;

  _Fd = FileOpen("SensorData.bin", "w");
  CTG_CompressWriteInit(&Ctx, WorkBuff, sizeof(WorkBuff),
                              FileBuff, sizeof(FileBuff),
                              _WriteCb, NULL, CTG_FLAG_WINDOW_SIZE_1K);
  //
  // Super loop
  //
  while (...) {
    if (..more sensor data available..) {
```



```
      //
      // Get sonsor data (variable length)
      //
      Len = GenSensorData(Buff);
      //
      // Store length in front of data
      //
      CTG_CompressWrite(&Ctx, &Len, sizeof(Len));
      CTG_CompressWrite(&Ctx, Buff, Len);
    }
    //
    // Handle other tasks
    //
    ....

  }
  //
  // Finish compression
  //
  CTG_CompressWrite(&Ctx, NULL, 0);
  FileClose(_Fd);
}
```


3.5 Decompression functions

emCompress-ToGo defines the following decompression functions:

Function	Description
One-call decompression functions	
`CTG_DecompressM2M()`	Decompress, memory to memory.
`CTG_DecompressM2F()`	Decompress, memory to function.
`CTG_DecompressF2M()`	Decompress, function to memory.
`CTG_DecompressF2F()`	Decompress, function to function.
Multi-call decompression functions (streaming interface)	
`CTG_DecompressInit()`	Initialize for decompression with `CTG_Decompress()`.
`CTG_Decompress()`	Run decompress on (part of) data.
`CTG_DecompressReadInit()`	Initialize for decompression with `CTG_DecompressRead()`.
`CTG_DecompressRead()`	Read decompressed data from a decompression-stream.
`CTG_DecompressWriteInit()`	Initialize for decompression with `CTG_DecompressWrite()`.
`CTG_DecompressWrite()`	Write compressed data to be decompressed to a stream.

3.5.1 CTG_DecompressM2M()

Description

Decompress, memory to memory.

Prototype

```
int CTG_DecompressM2M(const U8       * pInput,
                      unsigned         InputLen,
                      U8             * pOutput,
                      unsigned         OutputSize);
```

Parameters

Parameter	Description
pInput	Pointer to object that contains the compressed input.
InputLen	Octet length of the input object.
pOutput	Pointer to buffer that receives the decompressed output.
OutputSize	Size of the output buffer in bytes.

Return value

< 0 Error decoding bitstream.
≥ 0 Number of bytes output.

Example

See *Memory-to-memory decompression* on page 18.

See also

Function status values on page 33

3.5.2 CTG_DecompressM2F()

Description

Decompress, memory to function.

Prototype

```
int CTG_DecompressM2F(const U8          * pInput,
                      unsigned            InputLen,
                      CTG_WR_FUNC       * pfWr,
                      void              * pWrCtx,
                      U8                * pWork,
                      unsigned            WorkLen);
```

Parameters

Parameter	Description
pInput	Pointer to object that contains the compressed input.
InputLen	Octet length of the input object.
pfWr	Pointer to function that writes output.
pWrCtx	Pointer to user-supplied write context.
pWork	Pointer to work buffer.
WorkLen	Length of work buffer.

Return value

< 0	Error decoding bitstream.
≥ 0	Number of bytes output.

Additional information

The work buffer size (value of WorkLen) must be at least CTG_DECOMPRESS_WS_SIZE(Flag), where 'Flag' is the parameter that was used during compression of the data (Flag = CTG_FLAG_WINDOW_SIZE_...). This requirement is enforced by the decompressor and, should the buffer be too small, CTG_STATUS_BUFFER_TOO_SMALL is returned.

Example

See *Memory-to-function decompression* on page 28.

See also

Function status values on page 33

3.5.3 CTG_DecompressF2M()

Description

Decompress, function to memory.

Prototype

```
int CTG_DecompressF2M(CTG_RD_FUNC   pfRd,
                      void        * pRdCtx,
                      U8          * pOutput,
                      unsigned      OutputSize);
```

Parameters

Parameter	Description
pfRd	Pointer to function that reads input.
pRdCtx	Pointer to user-supplied context for read function.
pOutput	Pointer to buffer that receives the decompressed output.
OutputSize	Size of the output buffer in bytes.

Return value

< 0	Error decoding bitstream.
≥ 0	Number of bytes output.

Example

See *Function-to-memory decompression* on page 22.

See also

Function status values on page 33

3.5.4 CTG_DecompressF2F()

Description

Decompress, function to function.

Prototype

```
int CTG_DecompressF2F(CTG_RD_FUNC   pfRd,
                      void        * pRdCtx,
                      CTG_WR_FUNC * pfWr,
                      void        * pWrCtx,
                      U8          * pWork,
                      unsigned      WorkLen);
```

Parameters

Parameter	Description
pfRd	Pointer to function that reads input.
pRdCtx	Pointer to user-supplied read context.
pfWr	Pointer to function that writes output.
pWrCtx	Pointer to user-supplied write context.
pWork	Pointer to work buffer.
WorkLen	Length of work buffer.

Return value

< 0	Error decoding bitstream.
≥ 0	Number of bytes output.

Additional information

The work buffer size (value of WorkLen) must be at least CTG_DECOMPRESS_WS_SIZE(Flag), where 'Flag' is the parameter that was used during compression of the data (Flag = CTG_FLAG_WINDOW_SIZE_...). This requirement is enforced by the decompressor and, should the buffer be too small, CTG_STATUS_BUFFER_TOO_SMALL is returned.

Example

See *Memory-to-function decompression* on page 28.

See also

Function status values on page 33

3.5.5 CTG_DecompressInit()

Description

Initialize for decompression with `CTG_Decompress()`.

Prototype

```
void CTG_DecompressInit(CTG_DECOMPRESS_CTX * pCtx,
                        U8                 * pWork,
                        unsigned             WorkLen);
```

Parameters

Parameter	Description
`pCtx`	Pointer to decompress context to be initialized.
`pWork`	Pointer to work buffer.
`WorkLen`	Length of work buffer.

Additional information

The work buffer size (value of `WorkLen`) must be at least `CTG_DECOMPRESS_WS_SIZE(Flags)`, where 'Flags' is the parameter that was used during compression of the data (Flag = `CTG_FLAG_WINDOW_SIZE_...`). This requirement is enforced by the decompressor and, should the buffer be too small, `CTG_STATUS_BUFFER_TOO_SMALL` is returned by the decompress function.

Example

See *`CTG_Util.c` complete listing* on page 72.

See also

Compressor encoding flags on page 35.

3.5.6 CTG_Decompress()

Description

Run decompress on (part of) data. The function will process as many data from the input stream as possible and copy the decompressed data to the output stream. While processing input data, `pStream->pIn` is increased and `pStream->AvailIn` is reduced. While generating output data, `pStream->pOut` is increased and `pStream->AvailOut` is reduced. The function stops if either `pStream->AvailIn` or `pStream->AvailOut` reaches 0. On calling this function, the data stream must meet the following conditions: `pStream->AvailOut > 0 && (pStream->AvailIn > 0 || pStream->Flush > 0)`.

Prototype

```
int CTG_Decompress(CTG_DECOMPRESS_CTX * pCtx,
                   CTG_STREAM         * pStream);
```

Parameters

Parameter	Description
`pCtx`	Pointer to decompress context, created by `CTG_DecompressInit()`.
`pStream`	Pointer to data stream (input and output).

Return value

< 0	Error decoding bitstream.
= 0	Processing suspended, needs more input data or more output buffer. Then the function must be called again to resume processing.
> 0	Decompression finished successfully.

Example

See *`CTG_Util.c` complete listing* on page 72.

See also

Function status values on page 33

3.5.7 CTG_DecompressReadInit()

Description

Initialize for decompression with `CTG_DecompressRead()`.

Prototype

```
void CTG_DecompressReadInit(CTG_DECOMPRESS_RD_CTX * pCtx,
                            U8                    * pWork,
                            unsigned                WorkLen,
                            U8                    * pBuff,
                            unsigned                BuffSize,
                            CTG_RD_FUNC           * pfRd,
                            void                  * pRdCtx);
```

Parameters

Parameter	Description
`pCtx`	Pointer to decompress context to be initialized.
`pWork`	Pointer to work buffer.
`WorkLen`	Length of work buffer.
`pBuff`	Pointer to a buffer used for input data. This buffer is used when calling the read function: `pfRd(pBuff, BuffSize, pRdCtx);`
`BuffSize`	Size of the buffer in bytes. Every buffer size ≥ 1 will work. However, small sizes will decrease performance.
`pfRd`	Pointer to function that reads input (compressed data).
`pRdCtx`	Pointer to user-supplied context for read function.

Additional information

The work buffer size (value of `WorkLen`) must be at least `CTG_DECOMPRESS_WS_SIZE(Flag)`, where 'Flag' is the parameter that was used during compression of the data (Flag = `CTG_FLAG_WINDOW_SIZE_...`). This requirement is enforced by the decompressor and, should the buffer be too small, `CTG_STATUS_BUFFER_TOO_SMALL` is returned by the decompress function.

See also

Compressor encoding flags on page 35.

3.5.8 CTG_DecompressRead()

Description

Read decompressed data from a decompression-stream. Compressed input data to be decompressed is fetched using the callback read function given via `CTG_DecompressReadInit()`. To read the whole decompressed data this function must be called repeatedly until it returns a value < `NumBytes`.

Prototype

```
int CTG_DecompressRead(CTG_DECOMPRESS_RD_CTX * pCtx,
                       U8                    * pBuff,
                       unsigned                NumBytes);
```

Parameters

Parameter	Description
`pCtx`	Pointer to decompress context, created by `CTG_DecompressReadInit()`.
`pBuff`	Pointer to the buffer to store the decompressed data.
`NumBytes`	Number of bytes to be read.

Return value

≥ 0	Success, number of bytes read from stream. A value < `NumBytes` indicates the end of the decompressed data.
< 0	Error indication.

See also

Function status values on page 33

Example

```
static int _Fd;

//
// Read callback function that reads data from a file.
// On EOF the function returns 0.
//
static int _ReadCb(U8 *pData, unsigned DataLen, void *pWrCtx) {
  return FileRead(_Fd, pData, DataLen);
}

//
// Main function: Read sensor data from a compressed file and process them.
//
void main() {
  U8  Len;
  U8  Buff[100];
  U8  FileBuff[512];
  U8  WorkBuff[CTG_COMPRESS_WS_SIZE(CTG_FLAG_WINDOW_SIZE_1K)];
  CTG_DECOMPRESS_RD_CTX Ctx;

  _Fd = FileOpen("SensorData.bin", "r");
  CTG_DecompressReadInit(&Ctx, WorkBuff, sizeof(WorkBuff),
                               FileBuff, sizeof(FileBuff),
                               _ReadCb, NULL);
  //
  // Each sensor data record is preceeded by its length.
  // Read all records until end of the file.
  //
  while (CTG_DecompressRead(&Ctx, &Len, sizeof(Len)) == sizeof(Len)) {
```



```
    CTG_DecompressRead(&Ctx, Len, Buff);
    //
    // Process data in Buff.
    //
    ....

  }
  FileClose(_Fd);
}
```


3.5.9 CTG_DecompressWriteInit()

Description

Initialize for decompression with `CTG_DecompressWrite()`.

Prototype

```
void CTG_DecompressWriteInit(CTG_DECOMPRESS_WR_CTX * pCtx,
                             U8                    * pWork,
                             unsigned                WorkLen,
                             U8                    * pBuff,
                             unsigned                BuffSize,
                             CTG_WR_FUNC           * pfWr,
                             void                  * pWrCtx);
```

Parameters

Parameter	Description
`pCtx`	Pointer to decompress context to be initialized.
`pWork`	Pointer to work buffer.
`WorkLen`	Length of work buffer.
`pBuff`	Pointer to a buffer used to collect output data. This buffer is used when calling the write function: `pfWr(pBuff`, Len, `pWrCtx`);
`BuffSize`	Size of the buffer in bytes. Every buffer size ≥ 1 will work. However, small sizes will decrease performance.
`pfWr`	Pointer to function that writes output (decompressed data).
`pWrCtx`	Pointer to user-supplied write context.

Additional information

The work buffer size (value of `WorkLen`) must be at least `CTG_DECOMPRESS_WS_SIZE(Flag)`, where 'Flag' is the parameter that was used during compression of the data (Flag = `CTG_FLAG_WINDOW_SIZE_`...). This requirement is enforced by the decompressor and, should the buffer be too small, `CTG_STATUS_BUFFER_TOO_SMALL` is returned by the decompress function.

See also

Compressor encoding flags on page 35.

3.5.10 CTG_DecompressWrite()

Description

Write compressed data to be decompressed to a stream. Decompressed data is output using the callback write function given via `CTG_DecompressWriteInit()`. This function must be called repeatedly until all compressed data are written. To indicate the end of the compressed data, the function must be called with '`NumBytes`' = 0.

Prototype

```
int CTG_DecompressWrite(      CTG_DECOMPRESS_WR_CTX * pCtx,
                        const U8                    * pData,
                              unsigned                NumBytes);
```

Parameters

Parameter	Description
`pCtx`	Pointer to compress context, created by `CTG_CompressReadInit()`.
`pData`	Pointer to the compressed data to be decompressed.
`NumBytes`	Number of bytes to write.

Return value

= 0 Success.
< 0 Error indication.

See also

Function status values on page 33

3.6 Utility functions

emCompress-ToGo defines the following utility functions:

Function	Description
`CTG_GetStatusText()`	Decode emCompress-ToGo status code.
`CTG_GetCopyrightText()`	Get emCompress-ToGo copyright as printable string.
`CTG_GetVersionText()`	Get emCompress-ToGo version as printable string.

3.6.1 CTG_GetStatusText()

Description

Decode emCompress-ToGo status code.

Prototype

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

Parameters

Parameter	Description
Status	Code to decode.

Return value

Non-zero pointer to status description.

3.6.2 CTG_GetCopyrightText()

Description

Get emCompress-ToGo copyright as printable string.

Prototype

```
char *CTG_GetCopyrightText(void);
```

Return value

Zero-terminated copyright string.

3.6.3 CTG_GetVersionText()

Description

Get emCompress-ToGo version as printable string.

Prototype

```
char *CTG_GetVersionText(void);
```

Return value

Zero-terminated version string.

Chapter 4

Complete applications

This section presents two full applications developed with emCompress-ToGo:

- A compression-decompression utility to compress and decompress files.
- A self-test utility.

4.1 Compressor-decompressor utility

emCompress-ToGo ships with an application, in source form, which compresses files into SMASH-2 format. These files can be fed into the emCompress-ToGo decompressor, byte for byte, and be decompressed on target hardware.

For details, see *CTG_Util.c complete listing* on page 72.

4.1.1 Command line and options

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

The command line syntax is:

`CTG_Util` [*options*] *inputfile* [*outputfile*]

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

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

Example

```
C:> CTG_Util.exe -wb=14 FPGA.input

emCompress-ToGo Compression Utility V3.20 compiled Mar 11 2019 10:59:40
Copyright (c) 2015-2019 SEGGER Microcontroller GmbH    www.segger.com

Statistics:
  Original:     114618 bytes
  Compressed:   25402 bytes, or 22.16% of original
  Saving:       89216 bytes, or 77.84% of original
  Standardized: 1.77 bits/byte

C:> _
```

4.1.1.1 Set window bits (-wb)

Syntax

`-wb=`*number*

Description

Set the number of window bits, and therefore the window size, to use when compressing. Acceptable values are 8 through 14 inclusive with the default being 11.

This parameter sets the maximum match distance and, therefore, the maximum number of octets that must be stored to satisfy references made by the compressor (when not compressing from memory) and the decompressor (when not decompressing to memory).

Increasing the window size will usually increase compression ratios and reduce the size of the compressed bitstream.

4.1.1.2 Compress (-c)

Syntax

`-c`

Description

Compress input file to output file. This is the default action.

4.1.1.3 Decompress (-d)

Syntax

```
-d
```

Description

Decompress input file to output file.

4.1.1.4 Run silently (-q)

Syntax

```
-q
```

Description

Instructs the utility to run quietly and not display compression and decompression statistics.

4.1.1.5 Verbose (-v)

Syntax

```
-v
```

Description

Instructs the utility to display compression and decompression statistics.

4.1.2 CTG_Util.c complete listing

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

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

File        : CTG_Util.c
Purpose     : Example emCompress-ToGo application.

*/

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

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

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

static const unsigned _aDistBitsEncoding[] = {
  0, 0, 0, 0, 0, 0, 0, 0,
  CTG_FLAG_WINDOW_SIZE_256,
  CTG_FLAG_WINDOW_SIZE_512,
  CTG_FLAG_WINDOW_SIZE_1K,
```



```
  CTG_FLAG_WINDOW_SIZE_2K,
  CTG_FLAG_WINDOW_SIZE_4K,
  CTG_FLAG_WINDOW_SIZE_8K,
  CTG_FLAG_WINDOW_SIZE_16K,
};

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

static U8       _aWork     [CTG_COMPRESS_WS_SIZE(CTG_FLAG_WINDOW_SIZE_16K)];
static U8       _InBuff    [1024];
static U8       _OutBuff   [1024];


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

/*********************************************************************
*
*       _Encode()
*
*  Function description
*    Encode a file.
*
*  Parameters
*    pInFile  - Pointer to file to encode.
*    pOutFile - Pointer to file that receives the encoded bitstream.
*    Flags    - Encoding parameters.
*    Verbose  - Flag indicating verbose output.
*/
static void _Encode(const char *sFileName, FILE *pInFile, FILE *pOutFile, unsigned Flags, int Verbose) {
  CTG_COMPRESS_CTX  Ctx;
  CTG_STREAM        Stream;
  int               Status;
  unsigned          InputLen;
  unsigned          OutputLen;
  //
  memset(&Stream, 0, sizeof(Stream));
  InputLen  = 0;
  OutputLen = 0;
  CTG_CompressInit(&Ctx, &_aWork[0], sizeof(_aWork), Flags);
  Stream.pOut     = _OutBuff;
  Stream.AvailOut = sizeof(_OutBuff);
  //
  for (;;) {
    if (Stream.AvailIn == 0) {
      Stream.AvailIn = fread(_InBuff, 1, sizeof(_InBuff), pInFile);
      InputLen += Stream.AvailIn;
      if (Stream.AvailIn == 0) {
        Stream.Flush = 1;
      }
      Stream.pIn = _InBuff;
    }
    Status = CTG_Compress(&Ctx, &Stream);
    if (Status != 0) {
      break;
    }
    if (Stream.AvailOut == 0) {
      if (pOutFile != NULL) {
        fwrite(_OutBuff, 1, sizeof(_OutBuff), pOutFile);
      }
      Stream.pOut     = _OutBuff;
      Stream.AvailOut = sizeof(_OutBuff);
      OutputLen += sizeof(_OutBuff);
    }
  }
  if (Status < 0) {
    printf("FATAL: %s\n", CTG_GetStatusText(Status));
```



```
  } else {
    if (Stream.AvailOut < sizeof(_OutBuff)) {
      if (pOutFile != NULL) {
        fwrite(_OutBuff, 1, sizeof(_OutBuff) - Stream.AvailOut, pOutFile);
      }
      OutputLen += sizeof(_OutBuff) - Stream.AvailOut;
    }
    if (Verbose) {
      printf("Statistics:\n");
      printf("  Original:     %u bytes\n",
             InputLen);
      printf("  Compressed:   %u bytes, or %.2f%% of original\n",
             OutputLen,
             OutputLen * 100.0f / InputLen);
      printf("  Saving:       %d bytes, or %.2f%% of original\n",
             InputLen - OutputLen,
             100.0f - OutputLen * 100.0f / InputLen);
      printf("  Standardized: %.2f bits/byte\n",
             OutputLen * 8.0f / InputLen);
    } else {
      printf("%-20s: Compressed %7u -> %7u bytes, %.2f%% of original\n",
             sFileName,
             InputLen,
             OutputLen,
             OutputLen * 100.0f / InputLen);
    }
  }
}

/*********************************************************************
*
*       _Decode()
*
*  Function description
*    Decode a file.
*
*  Parameters
*    pInFile  - Pointer to file containing the encoded bitstream.
*    pOutFile - Pointer to file that receives the decoded bitstream.
*    Verbose  - Flag indicating verbose output.
*/
static void _Decode(FILE *pInFile, FILE *pOutFile, int Verbose) {
  CTG_DECOMPRESS_CTX  Ctx;
  CTG_STREAM          Stream;
  int                 Status;
  unsigned            InputLen;
  unsigned            OutputLen;
  //
  memset(&Stream, 0, sizeof(Stream));
  InputLen  = 0;
  OutputLen = 0;
  CTG_DecompressInit(&Ctx, &_aWork[0], sizeof(_aWork));
  Stream.pOut     = _OutBuff;
  Stream.AvailOut = sizeof(_OutBuff);
  //
  for (;;) {
    if (Stream.AvailIn == 0) {
      Stream.AvailIn = fread(_InBuff, 1, sizeof(_InBuff), pInFile);
      InputLen += Stream.AvailIn;
      if (Stream.AvailIn == 0) {
        Stream.Flush = 1;
      }
      Stream.pIn = _InBuff;
    }
    Status = CTG_Decompress(&Ctx, &Stream);
    if (Status != 0) {
      break;
    }
    if (Stream.AvailOut == 0) {
      if (pOutFile != NULL) {
        fwrite(_OutBuff, 1, sizeof(_OutBuff), pOutFile);
      }
      Stream.pOut     = _OutBuff;
      Stream.AvailOut = sizeof(_OutBuff);
      OutputLen += sizeof(_OutBuff);
    }
```



```
  }
  if (Status < 0) {
    printf("FATAL: %s\n", CTG_GetStatusText(Status));
  } else {
    if (Stream.AvailOut < sizeof(_OutBuff)) {
      if (pOutFile != NULL) {
        fwrite(_OutBuff, 1, sizeof(_OutBuff) - Stream.AvailOut, pOutFile);
      }
      OutputLen += sizeof(_OutBuff) - Stream.AvailOut;
    }
    if (Verbose) {
      printf("Statistics:\n");
      printf("  Encoded size:   %u bytes\n",             InputLen);
      printf("  Decoded size:   %u bytes \n",            OutputLen);
      printf("  Ratio:          %.2fx\n",                (float)OutputLen / InputLen);
      printf("  Space savings:  %.2f%% (of
 original)\n", 100.0f - InputLen * 100.0f / OutputLen);
      printf("  Standardized:   %.2f bits/byte\n",       InputLen * 8.0f / OutputLen);
    }
  }
}

/*********************************************************************
*
*       _ShowBanner()
*
*  Function description
*    Show sign-on banner.
*/
static void _ShowBanner(void) {
  printf("\n");
  printf("emCompress-ToGo Compression Utility V%s ", CTG_GetVersionText());
  printf("compiled " __DATE__ " " __TIME__ "\n");
  printf("%s    www.segger.com\n\n", CTG_GetCopyrightText());
}

/*********************************************************************
*
*       _ShowHelp()
*
*  Function description
*    Show utility help.
*/
static void _ShowHelp(void) {
  printf("Syntax: CTG_Util [options] <input-file> [<output-file>]\n");
  printf("Options:\n");
  printf("  -c       Compress input file                    [default]\n");
  printf("  -d       Decompress input file\n");
  printf("  -wb=n    Set number of window bits to 'n'       [8...14]\n");
  printf("             (also sets 'fb' and 'lb' parameters to their\n");
  printf("             associated defaults)\n");
  printf("  -v       Verbose mode\n");
  printf("  -q       Silent mode\n");
  exit(0);
}

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

/*********************************************************************
*
*       main()
*
*  Function description
*    Application entry point.
*
*  Parameters
*    argc - Argument count.
*    argv - Argument vector.
*/
int main(int argc, char **argv) {
  int          i;
```



```
int          Verbose;
int          Action;
const char * sInputPathname;
const char * sOutputPathname;
FILE       * pInFile;
FILE       * pOutFile;
unsigned     DistBits;
unsigned     Flags;
//
if (argc == 1) {
  _ShowBanner();
  _ShowHelp();
  exit(0);
}
//
DistBits        = 11;
sInputPathname  = NULL;
sOutputPathname = NULL;
Action          = -1;  // Compress
Verbose         = 1;
//
for (i = 1; i < argc; ++i) {
  const char *sArg = argv[i];
  if (strncmp(sArg, "-wb=", 4) == 0 || strncmp(sArg, "-db=", 4) == 0) {
    DistBits = atoi(&sArg[4]);
    if (DistBits < 8 || 14 < DistBits) {
      printf("%s: invalid value: -db=%d\n", argv[0], DistBits);
      exit(100);
    }
  } else if (strcmp(sArg, "-d") == 0) {
    Action = +1;
  } else if (strcmp(sArg, "-c") == 0) {
    Action = -1;
  } else if (strcmp(sArg, "-v") == 0) {
    Verbose = 1;
  } else if (strcmp(sArg, "-q") == 0) {
    Verbose = 0;
  } else if (sArg[0] == '-') {
    printf("%s: unknown option %s\n", argv[0], sArg);
    exit(100);
  } else if (sInputPathname == NULL) {
    sInputPathname = sArg;
  } else if (sOutputPathname == NULL) {
    sOutputPathname = sArg;
  } else {
    printf("%s: too many filenames\n", argv[0]);
    exit(100);
  }
}
//
if (sInputPathname == NULL) {
  printf("%s: require input filename\n", argv[0]);
  exit(100);
}
//
pInFile = fopen(sInputPathname, "rb");
if (pInFile == NULL) {
  printf("%s: can't open %s for reading\n", argv[0], sInputPathname);
  exit(100);
}
//
pOutFile = NULL;
if (sOutputPathname != NULL) {
  pOutFile = fopen(sOutputPathname, "wb");
  if (pOutFile == NULL) {
    printf("%s: can't open %s for writing\n", argv[0], sInputPathname);
    fclose(pInFile);
    exit(100);
  }
}
//
Flags = _aDistBitsEncoding[DistBits];
//
if (Verbose) {
  _ShowBanner();
}
```



```
  if (Action < 0) {
    _Encode(sInputPathname, pInFile, pOutFile, Flags, Verbose);
  } else {
    _Decode(pInFile, pOutFile, Verbose);
  }
  //
  if (pInFile) {
    fclose(pInFile);
  }
  if (pOutFile) {
    fclose(pOutFile);
  }
  return 0;
}

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


4.2 Compressor and decompressor self-test

This utility takes as input a single file and runs all compressors and decompressors over the file to ensure that a compression-decompression cycle results in the original content. This provides an assurance of compressor and decompressor correctness after compilation and linking.

4.2.1 CTG_Test.c complete listing

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

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

File        : CTG_Test.c
Purpose     : Compression dataset test tool.

Additional information:

  This application takes an input file and compresses and decompresses
  it to ensure that all compression modes, all compression parameters,
  and all specialized compressors and decompressors can complete their
  task without error.

  At the end of execution, if all tests pass, the following is expected:

    Passed: no errors

  If not, and errors are reported, it is either an error in compilation
  or an error in the emCompress-ToGo compressor or decompressor.  If you
  see the error and believe your compiler produces correct object code,
  please supply SEGGER with the input file that you used and we will
  investigate whether this is indeed an error in our software.

*/

/*********************************************************************
*
*       #include Section
*
**********************************************************************
*/

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

/*********************************************************************
*
*       Local types
*
**********************************************************************
*/

typedef int (*ENCODE_FUNC)(const U8 *pInput,  unsigned InputLen,
                           U8 *pOutput, unsigned OutputLen,
                           unsigned Flags);
typedef int (*DECODE_FUNC)(const U8 *pInput,  unsigned InputLen,
                           U8 *pOutput, unsigned OutputLen);

typedef struct {
  const char  * sMode;
  ENCODE_FUNC   pfEncode;
  DECODE_FUNC   pfDecode;
} COMPRESS_GENERIC_TEST_MODE;

typedef struct {
  unsigned    DistanceBits;
  DECODE_FUNC pfSpecificEncode;
  DECODE_FUNC pfSpecificDecode;
  ENCODE_FUNC pfGenericEncode;
  DECODE_FUNC pfGenericDecode;
} COMPRESS_SPECIFIC_TEST_MODE;

typedef struct {
  const U8 * pInput;
  unsigned   InputLen;
  unsigned   Cursor;
```



```
} CTG_RD_CONTEXT;

typedef struct {
  U8       * pOutput;
  unsigned   OutputLen;
  unsigned   Cursor;
} CTG_WR_CONTEXT;

/*********************************************************************
*
*       Prototypes
*
**********************************************************************
*/

static int _TestCompressM2F  (const U8 *pInput,  unsigned InputLen,
                                    U8 *pOutput, unsigned OutputLen,
                                    unsigned Flags);
static int _TestCompressF2M  (const U8 *pInput,  unsigned InputLen,
                                    U8 *pOutput, unsigned OutputLen,
                                    unsigned Flags);
static int _TestCompressF2F  (const U8 *pInput,  unsigned InputLen,
                                    U8 *pOutput, unsigned OutputLen,
                                    unsigned Flags);
static int _TestDecompressM2F(const U8 *pInput,  unsigned InputLen,
                                    U8 *pOutput, unsigned OutputLen);
static int _TestDecompressF2M(const U8 *pInput,  unsigned InputLen,
                                    U8 *pOutput, unsigned OutputLen);
static int _TestDecompressF2F(const U8 *pInput,  unsigned InputLen,
                                    U8 *pOutput, unsigned OutputLen);

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

static const COMPRESS_GENERIC_TEST_MODE _aGenericTestModes[] = {
  { "M2M", CTG_CompressM2M,  CTG_DecompressM2M  },
  { "M2F", _TestCompressM2F, _TestDecompressM2F },
  { "F2M", _TestCompressF2M, _TestDecompressF2M },
  { "F2F", _TestCompressF2F, _TestDecompressF2F },
  { NULL,  NULL,             NULL               }
};

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

static U8 _aWork       [CTG_COMPRESS_WS_SIZE(CTG_FLAG_WINDOW_SIZE_16K)];
static U8 _aInputImage [16*1024*1024];
static U8 _aSmashImage [16*1024*1024];
static U8 _aMirrorImage[16*1024*1024];

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

/*********************************************************************
*
*       _Wr()
*
*  Function description
*    Write compressed or decompressed data.
*
*  Parameters
*    pData   - Pointer to octet string to write.
*    DataLen - Octet length of the octet string to write.
*    pWrCtx  - Pointer to write context.
```



```
*
*  Return value
*    1 if written, -1 if not.
*/
static int _Wr(const U8 *pData, unsigned DataLen, void *pWrCtx) {
  CTG_WR_CONTEXT * pCtx;
  unsigned         N;
  //
  pCtx = (CTG_WR_CONTEXT *)pWrCtx;
  for (N = 0; N < DataLen; ++N) {
    if (pCtx->Cursor >= pCtx->OutputLen) {
      return -1;  // Buffer overflow
    }
    pCtx->pOutput[pCtx->Cursor++] = *pData++;
  }
  return 1;
}

/*********************************************************************
*
*       _Rd()
*
*  Function description
*    Read compressed or decompressed data.
*
*  Parameters
*    pData   - Pointer to the octet string that receives data.
*    DataLen - Octet length of the octet string that receives data.
*    pRdCtx  - Pointer to read context.
*
*  Return value
*    Number of bytes read.
*/
static int _Rd(U8 *pData, unsigned DataLen, void *pRdCtx) {
  CTG_RD_CONTEXT * pCtx;
  unsigned         N;
  //
  pCtx = (CTG_RD_CONTEXT *)pRdCtx;
  for (N = 0; N < DataLen; ++N) {
    if (pCtx->Cursor >= pCtx->InputLen) {
      break;
    }
    *pData++ = pCtx->pInput[pCtx->Cursor++];
  }
  return N;
}

/*********************************************************************
*
*       _TestCompressM2F()
*
*  Function description
*    Shim for M2F compression to provide M2M-like interface.
*
*  Parameters
*    pInput    - Pointer to octet string to encode.
*    InputLen  - Octet length of the input octet string.
*    pOutput   - Pointer to octet string that receives the compressed output.
*    OutputLen - Octet length of the output octet string.
*    Flags     - Encoding flags.
*
*  Return value
*    Status returned from target compressor.
*/
static int _TestCompressM2F(const U8       * pInput,
                                  unsigned   InputLen,
                                  U8       * pOutput,
                                  unsigned   OutputLen,
                                  unsigned   Flags) {
  CTG_WR_CONTEXT W;
  //
  W.pOutput   = pOutput;
  W.OutputLen = OutputLen;
  W.Cursor    = 0;
  //
  return CTG_CompressM2F(pInput, InputLen, _Wr, &W, Flags);
```



```
}

/*********************************************************************
*
*       _TestCompressF2M()
*
*  Function description
*    Shim for F2M compression to provide M2M-like interface.
*
*  Parameters
*    pInput    - Pointer to octet string to encode.
*    InputLen  - Octet length of the input octet string.
*    pOutput   - Pointer to octet string that receives the compressed output.
*    OutputLen - Octet length of the output octet string.
*    Flags     - Encoding flags.
*
*  Return value
*    Status returned from target compressor.
*/
static int _TestCompressF2M(const U8       * pInput,
                                  unsigned   InputLen,
                                  U8       * pOutput,
                                  unsigned   OutputLen,
                                  unsigned   Flags) {
  CTG_RD_CONTEXT R;
  //
  R.pInput   = pInput;
  R.InputLen = InputLen;
  R.Cursor   = 0;
  //
  return CTG_CompressF2M(_Rd, &R, pOutput, OutputLen, &_aWork[0], sizeof(_aWork), Flags);
}

/*********************************************************************
*
*       _TestCompressF2F()
*
*  Function description
*    Shim for F2F compression to provide M2M-like interface.
*
*  Parameters
*    pInput    - Pointer to octet string to encode.
*    InputLen  - Octet length of the input octet string.
*    pOutput   - Pointer to octet string that receives the compressed output.
*    OutputLen - Octet length of the output octet string.
*    Flags     - Encoding flags.
*
*  Return value
*    Status returned from target compressor.
*/
static int _TestCompressF2F(const U8       * pInput,
                                  unsigned   InputLen,
                                  U8       * pOutput,
                                  unsigned   OutputLen,
                                  unsigned   Flags) {
  CTG_RD_CONTEXT R;
  CTG_WR_CONTEXT W;
  //
  R.pInput   = pInput;
  R.InputLen = InputLen;
  R.Cursor   = 0;
  //
  W.pOutput   = pOutput;
  W.OutputLen = OutputLen;
  W.Cursor    = 0;
  //
  return CTG_CompressF2F(_Rd, &R, _Wr, &W, &_aWork[0], sizeof(_aWork), Flags);
}

/*********************************************************************
*
*       _TestDecompressM2F()
*
*  Function description
*    Shim for M2F decompression to provide M2M-like interface.
*
```



```
*  Parameters
*    pInput    - Pointer to octet string to encode.
*    InputLen  - Octet length of the input octet string.
*    pOutput   - Pointer to octet string that receives the compressed output.
*    OutputLen - Octet length of the output octet string.
*
*  Return value
*    Status returned from target compressor.
*/
static int _TestDecompressM2F(const U8       * pInput,
                                    unsigned   InputLen,
                                    U8       * pOutput,
                                    unsigned   OutputLen) {
  CTG_WR_CONTEXT W;
  //
  W.pOutput   = pOutput;
  W.OutputLen = OutputLen;
  W.Cursor    = 0;
  //
  return CTG_DecompressM2F(pInput, InputLen, _Wr, &W, &_aWork[0], sizeof(_aWork));
}

/*********************************************************************
*
*       _TestDecompressF2M()
*
*  Function description
*    Shim for F2M decompression to provide M2M-like interface.
*
*  Parameters
*    pInput    - Pointer to octet string to encode.
*    InputLen  - Octet length of the input octet string.
*    pOutput   - Pointer to octet string that receives the compressed output.
*    OutputLen - Octet length of the output octet string.
*
*  Return value
*    Status returned from target compressor.
*/
static int _TestDecompressF2M(const U8       * pInput,
                                    unsigned   InputLen,
                                    U8       * pOutput,
                                    unsigned   OutputLen) {
  CTG_RD_CONTEXT R;
  //
  R.pInput   = pInput;
  R.InputLen = InputLen;
  R.Cursor   = 0;
  //
  return CTG_DecompressF2M(_Rd, &R, pOutput, OutputLen);
}

/*********************************************************************
*
*       _TestDecompressF2F()
*
*  Function description
*    Shim for F2F decompression to provide M2M-like interface.
*
*  Parameters
*    pInput    - Pointer to octet string to encode.
*    InputLen  - Octet length of the input octet string.
*    pOutput   - Pointer to octet string that receives the compressed output.
*    OutputLen - Octet length of the output octet string.
*
*  Return value
*    Status returned from target compressor.
*/
static int _TestDecompressF2F(const U8       * pInput,
                                    unsigned   InputLen,
                                    U8       * pOutput,
                                    unsigned   OutputLen) {
  CTG_RD_CONTEXT R;
  CTG_WR_CONTEXT W;
  //
  R.pInput   = pInput;
  R.InputLen = InputLen;
```



```
  R.Cursor   = 0;
  //
  W.pOutput   = pOutput;
  W.OutputLen = OutputLen;
  W.Cursor    = 0;
  //
  return CTG_DecompressF2F(_Rd, &R, _Wr, &W, &_aWork[0], sizeof(_aWork));
}

/*********************************************************************
*
*       _ShowBanner()
*
*  Function description
*    Show sign-on banner.
*/
static void _ShowBanner(void) {
  printf("\n");
  printf("emCompress-ToGo Test Application V%d.%02d ",
         CTG_VERSION / 10000,
         CTG_VERSION / 100 % 100);
  printf("compiled " __DATE__ " " __TIME__ "\n");
  printf("(c) 2017-2019 SEGGER Microcontroller GmbH    www.segger.com\n");
  printf("\n");
}

/*********************************************************************
*
*       _ShowHelp()
*
*  Function description
*    Show utility help.
*/
static void _ShowHelp(void) {
  printf("Syntax:\n");
  printf("  CTG_Test <input-file>\n");
}

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

/*********************************************************************
*
*       main()
*
*  Function description
*    Main application.
*
*  Parameters
*    argc - Argument count.
*    argv - Argument vector.
*/
int main(int argc, char **argv) {
  FILE      * pFile;
  unsigned    InputLen;
  unsigned    MaxDistBits;
  unsigned    i;
  int         Status;
  int         ErrorCnt;
  //
  ErrorCnt = 0;
  //
  _ShowBanner();
  //
  if (argc != 2) {
    _ShowHelp();
    exit(100);
  }
  //
  pFile = fopen(argv[1], "rb");
  if (pFile == NULL) {
    printf("Can't open %s for reading\n", argv[1]);
```



```
    exit(100);
  }
  InputLen = fread(_aInputImage, 1, sizeof(_aInputImage), pFile);
  //
  // Test all-parameters-specified mode.
  //
  printf("*** All parameters specified mode ***\n");
  //
  for (i = 0; _aGenericTestModes[i].sMode; ++i) {
    printf("\n%s:\n", _aGenericTestModes[i].sMode);
    //
    for (MaxDistBits = 8; MaxDistBits <= 14; ++MaxDistBits) {
      printf("  -wb=%-6u", MaxDistBits);
      CTG_MEMSET(_aSmashImage, 0, sizeof(_aSmashImage));
      CTG_MEMSET(_aMirrorImage, 0, sizeof(_aMirrorImage));
      Status = _aGenericTestModes[i].pfEncode(_aInputImage, InputLen,
                                              _aSmashImage, sizeof(_aSmashImage),
                                              MaxDistBits - 8);
      if (Status == CTG_STATUS_BAD_PARAMETER) {
        printf("%6s", "-");
      } else if (Status < 0) {
        printf("%6d [compress error]", 0);
        ++ErrorCnt;
      } else {
        printf("%6d", Status);
        Status = _aGenericTestModes[i].pfDecode(_aSmashImage, Status,
                                                _aMirrorImage, sizeof(_aMirrorImage));
        if (Status < 0) {
          printf(" [bitstream error]");
          ++ErrorCnt;
        } else if ((unsigned)Status < InputLen) {
          printf(" [length error, too short]");
          ++ErrorCnt;
        } else if ((unsigned)Status > InputLen) {
          printf(" [length error, too long]");
          ++ErrorCnt;
        } else if (memcmp(_aInputImage, _aMirrorImage, InputLen) != 0) {
          printf(" [match error]");
          ++ErrorCnt;
        }
      }
      printf("\n");
    }
  }
  //
  printf("\n*** Test run complete ***\n\n");
  if (ErrorCnt == 0) {
    printf("Passed: no errors\n");
  } else {
    printf("Failed: %u errors\n", ErrorCnt);
  }
  return 0;
}

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


Chapter 5

Resource use and performance

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

5.1 General comments

5.1.1 Reentrancy

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

5.1.2 Configuration

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

5.1.3 SMASH-2 format and bitstream expansion

emCompress-ToGo uses SEGGER's *Small Microcontroller Advanced Super-High* (SMASH-2) format to compress data. The proprietary SMASH-2 format is an excellent all-round, tunable format.

When used in emCompress-ToGo, the maximum expansion of a bitstream is one bit per byte plus 20 bits of header information and end-of-stream marker. Hence, the number of bits to encode an 8*n*-bit bitstream is 9*n*+20; round this up to the next multiple of 8 bits to determine the maximum buffer size required in bytes.

5.2 Memory footprint

5.2.1 Target system configuration

The following table shows the hardware and the toolchain details of the real-world target system used for benchmarking:

Detail	Description
CPU	STM32F7, Cortex-M7, 200 MHz, run from flash
Toolchain	Embedded Studio for ARM Release 4.12 (gcc compiler)
Model	Thumb-2 instructions
Compiler options	Highest size optimization

5.2.2 ROM and RAM use

The amount of ROM and RAM that emCompress-ToGo uses for compression and decompression varies depending upon the function used.

The following table measures the total ROM required for a single function. The compressor and decompressor do not require any static data themselves, any working data is transitory and provided by the client to the compressor and decompressor (if needed) and is shown in the "RAM (client)" column. The size of the RAM required depends on the window size selected upon compression.

Compression

Function	ROM	RAM (client)	RAM (stack)
`CTG_CompressM2M()`	1095 bytes	0 bytes	80 bytes
`CTG_CompressM2F()`	1099 bytes	0 bytes	80 bytes
`CTG_CompressF2M()`	1381 bytes	514 to 16642 bytes	160 bytes
`CTG_CompressF2F()`	1403 bytes	514 to 16642 bytes	192 bytes
`CTG_Compress()`	1275 bytes	514 to 16642 bytes	64 bytes
`CTG_FastCompressM2M()`	1200 bytes	768 to 33280 bytes	100 bytes

Decompression

Function	ROM	RAM (client)	RAM (stack)
`CTG_DecompressM2M()`	370 bytes	0 bytes	80 bytes
`CTG_DecompressM2F()`	630 bytes	514 to 16642 bytes	160 bytes
`CTG_DecompressF2M()`	394 bytes	0 bytes	80 bytes
`CTG_DecompressF2F()`	668 bytes	514 to 16642 bytes	192 bytes
`CTG_Decompress()`	536 bytes	514 to 16642 bytes	64 bytes

5.3 Runtime performance

5.3.1 Target system configuration

The following table shows the hardware and the toolchain details of the real-world target system used for benchmarking:

Detail	Description
CPU	STM32F7, Cortex-M7, 200 MHz, run from flash
Toolchain	Embedded Studio for ARM Release 4.12 (gcc compiler)
Model	Thumb-2 instructions
Compiler options	-O2 optimization

5.3.2 Performance results

The following tables show the average throughput of the compression and decompression functions calculated over a set of different input files. The actual performance depends on the data to be compressed / decompressed.

Decompression

Function	Compressed input data	Decompressed output data
`CTG_DecompressM2M()`	6.3 MB/sec	13.0 MB/sec
`CTG_Decompress()`	4.3 MB/sec	9.0 MB/sec

Compression, memory to memory, CTG_CompressM2M().

Window size	`CTG_CompressM2M()`	`CTG_FastCompressM2M()`
256 bytes	410 KB/sec	1510 KB/sec
512 bytes	263 KB/sec	1130 KB/sec
1k	171 KB/sec	850 KB/sec
2k	107 KB/sec	580 KB/sec
4k	67 KB/sec	390 KB/sec
8k	44 KB/sec	250 KB/sec
16k	31 KB/sec	180 KB/sec

Compression, streaming, CTG_Compress().

Window size	`CTG_Compress()`	`CTG_FastCompress()`
256 bytes	220 KB/sec	1010 KB/sec
512 bytes	133 KB/sec	750 KB/sec
1k	79 KB/sec	540 KB/sec
2k	46 KB/sec	350 KB/sec
4k	26 KB/sec	240 KB/sec
8k	15 KB/sec	165 KB/sec
16k	9 KB/sec	120 KB/sec

Chapter 6

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.

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.

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

Chapter 7

Indexes

7.1 Index of functions

7.2 Index of types

