emLib CRC

Library collection: Cyclic redundancy check

User Guide & Reference Manual

Document: UM12003
Software Version: 1.00
Revision: 2
Date: Januar 26, 2016

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Contact address

SEGGER Microcontroller GmbH & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

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

Manual versions

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

Print date: Januar 26, 2016

Software	Revision	Date	By	Description
1.00	2	160126	MC	Minor corrections.
1.00	1	160107	MC	Minor corrections.
1.00	0	150810	MC	Initial version.

About this document

Assumptions

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

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

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

How to use this manual

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

Typographic conventions for syntax

This manual uses the following typographic conventions:

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

Table of contents

Chapter 1

Introduction to emLib CRC

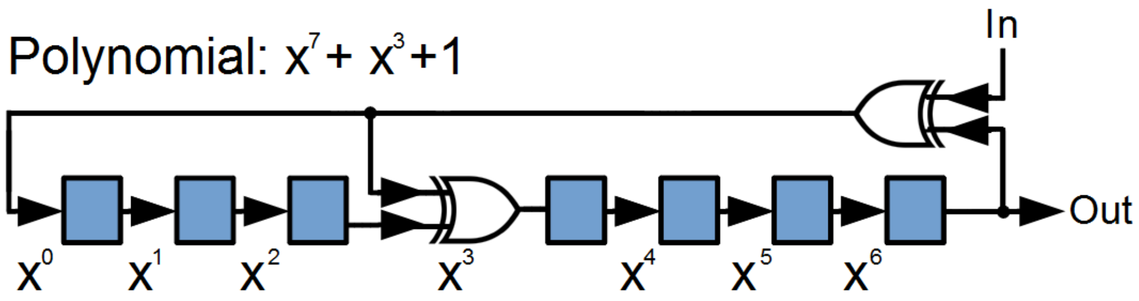
This chapter provides an introduction to emLib CRC. It explains the basic concepts behind emLib CRC and its modules.

1.1 What is CRC?

The *Cyclic Redundancy Check*, short *CRC*, is a common technique for the calculation of checksums in order to provide an error-detection on data transfers in digital networks and storage devices. The basic principle of CRC calculations was invented by W. Wesley Peterson in 1961, while many of today's standard implementations were published later on by several other researchers and mathematicians.

CRCs are named *cyclic* and *redundant* as they utilize cyclic block codes to calculate checksums, which themselves contain no information that was not already included in the raw data for which the CRC was calculated. The usefulness of CRC checksums originates from their inherent capacity to detect erroneous data in communication channels and storage devices, including any uneven number of bit errors and most two-bit errors, as well as any burst error of certain length. In order to do so, CRC algorithms require a generator polynomial that is used as a divisor in a polynomial division of the raw data for which to calculate the CRC. The remainder of this division is then considered the result.

In hardware, CRC checksum calculations may be easily implemented using linear feedback shift-registers. The following example diagram illustrates the basic principle of these implementations



Generally, a CRC is called *n*-bit CRC if its checksum is *n* bits. Hence, for a given *n*, several different generator polynomials may be used to calculate several different *n*-bit CRCs. These generator polynomials have in common that they consist of *n+1* terms (thus having a maximum degree of *n*) and that they are typically capable of detecting burst errors that affect less than *n* bits. Some CRCs are calculated using the most-significant-bit-first notation of polynomials, while others utilize the least-significant-bit-first notation of polynomials.

Usage of different generator polynomials results in different error-detection properties, thus the selection of the proper generator polynomial is a question of vital importance. Some of the factors in this selection process are the length of the data for which to caluclate the CRC, the desired error-detection properties, and the desired performance of the calculation. Also, certain CRC calculation algorithms have been incorporated into technical standards that do not only define the generator polynomial to be used in the polynomial division, but also require compliance to further specifications such as initialization vectors. For example, at least 17 different variants of 16-bit CRCs have been published (see http://reveng.sourceforge.net/crc-catalogue/all.htm). Therefore, selecting the proper generator polynomials was also subject to many scientific papers, one of which is *Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks* by Philip Koopman and Tridib Chakravarty, which is suggested for further information (it is publicly available at http://users.ece.cmu.edu/~koopman/roses/dsn04/koopman04_crc_poly_embedded.pdf).

1.2 What is emLib CRC?

emLib CRC is a collection of CRC implementations for different purposes. It currently includes two modules for CRC calculations using arbitrary polynomials and eight modules for CRC calculations using specified polynomials.

The software is designed for portabilty to any device. Each module can be used individually in PC applications as well as on embedded target devices. An additional module is included in emLib CRC to allow the validation of the implementation on each of these devices.

emLib CRC is optimized for speed performance and a small memory footprint. The sources are completely written in ANSI-C. Validation code for the APIs using standard test patterns is included.

1.2.1 Features

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

- Clean ISO/ANSI C source code.
- Easy-to-understand and simple-to-use API.
- Same modules and same API can be used in PC programs and on embedded targets.
- Includes validation and further sample applications.
- Royalty free.

1.2.2 Package content

emLib CRC is provided in source code and contains everything needed. The following table shows the contents of the emLib CRC Package:

Files	Description
`Config`	Configuration header files.
`Doc`	emLib CRC documentation.
`SEGGER`	emLib CRC source code.
`Windows`	Executable application samples and their source code.

1.2.2.1 Included modules

emLib CRC does not offer API functions that are fully compliant to one specific parametrised standard, but delivers generic CRC implementations instead. This allows their usage for more than one parametrised standard at a time, but requires the user to initialize and finalize the calculation according to the standard.

The following modules are included in emLib CRC.

1.2.2.2 Generic implementations

The following modules contain generic implementations that may be used with arbitrary polynomials. The table-driven implementations dynamically create a 16-entry-table for the chosen polynomial.

SEGGER_CRC.c

Includes table-driven and bitwise implementations using the least-significant-bit-first notation of arbitrary polynomials.

SEGGER_CRC_MSB.c

Includes table-driven and bitwise implementations using the most-significant-bit-first notation of arbitrary polynomials.

1.2.2.3 Specific implementations

The following modules contain implementations for specified polynomials and use static 256-entry-tables for the respective polynomial.

SEGGER_CRC_09.c

Includes a table-driven implementation for the 7-bit polynomial `0x09` (most-significant-bit-first notation), which typically is used with MultiMediaCards. To use this implementation in compliance to one specific parametrised standard, pass the appropiate initial value to the function and finalize the result through exclusive-or with the appropiate Xor value. Initial value and Xor value are given in the table below:

Standard	Initial value	Xor value
CRC-7	0x0000	0x0000

SEGGER_CRC_48.c

Includes a table-driven implementation for the 7-bit polynomial `0x48` (least-significant-bit-first notation).

SEGGER_CRC_1021.c

Includes a table-driven implementation for the 16-bit polynomial `0x1021` (most-significant-bit-first notation), which typically is used with MultiMediaCards, RFID tags, and other file transfer protocols. To use this implementation in compliance to one specific parametrised standard, pass the appropiate initial value to the function and finalize the result through exclusive-or with the appropiate Xor value. Initial value and Xor value are given in the table below:

Standard	Initial value	Xor value
CRC-16/DARC	0xFFFF	0xFFFF
CRC-16/SPI	0x1D0F	0x0000
CRC-16/XMODEM	0x0000	0x0000

SEGGER_CRC_8408.c

Includes a table-driven implementation for the 16-bit polynomial `0x8408` (least-significant-bit-first notation), which typically is used with Bluetooth, integrated circuit cards, and uninterruptible power supplies. To use this implementation in compliance to one specific parametrised standard, pass the appropiate initial value to the function and finalize the result through exclusive-or with the appropiate Xor value.Initial value and Xor value are given in the table below:

Standard	Initial value	Xor value
CRC-16/A	0xC6C6	0x0000
CRC-16/KERMIT	0x0000	0x0000
CRC-16/MCRF4xx	0xFFFF	0x0000
CRC-16/RIELLO	0xB2AA	0x0000
CRC-16/TMS37157	0x89EC	0x0000
CRC-16/X-25	0xFFFF	0xFFFF

SEGGER_CRC_04C11DB7.c

Includes a table-driven implementation for the 32-bit polynomial `0x04C11DB7` (most-significant-bit-first notation), which typically is used with Ethernet communications. To use this implementation in compliance to one specific parametrised standard, pass the appropiate initial value to the function and finalize the result through exclusive-or with the appropiate Xor value. Initial value and Xor value are given in the table below:

Standard	Initial value	Xor value
CRC-32/DECT	`0xFFFFFFFF`	`0xFFFFFFFF`
CRC-32/MPEG2	`0xFFFFFFFF`	`0x00000000`
CRC-32/POSIX	`0x00000000`	`0xFFFFFFFF`

SEGGER_CRC_1EDC6F41.c

Includes a table-driven implementation for the 32-bit polynomial `0x1EDC6F41` (most-significant-bit-first notation).

SEGGER_CRC_82F63B78.c

Includes a table-driven implementation for the 32-bit polynomial `0x82F63B78` (least-significant-bit-first notation), which typically is used in with SCSI, and in the Stream Control Transmission Protocol. To use this implementation in compliance to one specific parametrised standard, pass the appropiate initial value to the function and finalize the result through exclusive-or with the appropiate Xor value. Initial value and Xor value are given in the table below:

Standard	Initial value	Xor value
CRC-32C	`0xFFFFFFFF`	`0xFFFFFFFF`

SEGGER_CRC_EDB88320.c

Includes a table-driven implementation for the 32-bit polynomial `0xEDB88320` (least-significant-bit-first notation), which typically is used with Ethernet and Serial ATA. To use this implementation in compliance to one specific parametrised standard, pass the appropiate initial value to the function and finalize the result through exclusive-or with the appropiate Xor value. Initial value and Xor value are given in the table below:

Standard	Initial value	Xor value
CRC-32	`0xFFFFFFFF`	`0xFFFFFFFF`
CRC-32/JAM	`0xFFFFFFFF`	`0x00000000`

1.3 Usage

emLib CRC has a simple yet powerful API. It can be easily integrated into an existing application. The code is completely written in ANSI-C.

All functionality can be verified with standard test patterns using the validation API. The functions for generating the lookup tables used for specific polynomials are also included for full transparency.

To simply calculate a CRC for contiguous raw data, the application would only need to call one function. If two or more distinct data areas need to be processed into the same CRC, subsequent calls to the function may be used for incremental calculations over each of these areas.

1.3.1 Recommended project structure

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

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

1.3.2 Include directories

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

- `Config`
- `SEGGER`

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

It is frequently a major problem when updating to a new version of emLib CRC if you have old files included and therefore mix different versions. If you keep emLib CRC 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 `SEGGER` directories before updating.

Chapter 2

API reference

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

2.1 Core functions

Function	Description
Generic API functions	
SEGGER_CRC_Calc()	Provides a table-driven implementation of CRC calculations over a given range of memory using arbitrary polynomials in LSB-first notation.
SEGGER_CRC_CalcBitByBit()	Provides a bitwise implementation of CRC calculations over a given range of memory using arbitrary polynomials in LSB-first notation.
SEGGER_CRC_Calc_MSB()	Provides a table-driven implementation of CRC calculations over a given range of memory using arbitrary polynomials in MSB-first notation.
SEGGER_CRC_CalcBitByBit_MSB()	Provides a bitwise implementation of CRC calculations over a given range of memory using arbitrary polynomials in MSB-first notation.
Specific API functions	
SEGGER_CRC_Calc_09()	Computes the CRC checksum over the given range of memory using a static table for the polynomial 0x09.
SEGGER_CRC_Calc_48()	Computes the CRC checksum over the given range of memory using a static table for the polynomial 0x48.
SEGGER_CRC_Calc_1021()	Computes the CRC checksum over the given range of memory using a static table for the polynomial 0x1021.
SEGGER_CRC_Calc_8408()	Computes the CRC checksum over the given range of memory using a static table for the polynomial 0x8408.
SEGGER_CRC_Calc_04C11DB7()	Computes the CRC checksum over the given range of memory using a static table for the polynomial 0x04C11DB7.
SEGGER_CRC_Calc_1EDC6F41()	Computes the CRC checksum over the given range of memory using a static table for the polynomial 0x1EDC6F41.
SEGGER_CRC_Calc_82F63B78()	Computes the CRC checksum over the given range of memory using a static table for the polynomial 0x82F63B78.
SEGGER_CRC_Calc_EDB88320()	Computes the CRC checksum over the given range of memory using a static table for the polynomial 0xEDB88320.
Validation API functions	
SEGGER_CRC_Validate()	Validates the implementation of the emLib CRC API.

2.1.1 SEGGER_CRC_Calc()

Description

Provides a table-driven implementation of CRC calculations over a given range of memory using arbitrary polynomials in LSB-first notation. Dynamically creates a 16-entry-table for the given polynomial, which is then used to calculate the CRC for each nibble in each byte of data.

Prototype

```
U32 SEGGER_CRC_Calc(const U8   * pData,
                          U32    NumBytes,
                          U32    Crc,
                          U32    Poly);
```

Parameters

Parameter	Description
pData	Pointer to the source buffer which holds the data for which to calculate the CRC checksum.
NumBytes	Number of bytes to include in the calculation. May be zero.
Crc	Initial value for incremental calculation of the CRC. May be zero.
Poly	LSB-first notation of the polynomial to use in this calculation. Must not exceed a polynomial degree of 32, but may be zero. For example, LSB-first notations of common polynomials are: 32-bit: `0xEDB88320`, 16-bit: `0x8408`, 8-bit: `0xE0`.

Return value

Resulting CRC checksum.

Example

Please see *Using API functions for arbitrary polynomials (LSB)* on page 32.

2.1.2 SEGGER_CRC_CalcBitByBit()

Description

Provides a bitwise implementation of CRC calculations over a given range of memory using arbitrary polynomials in LSB-first notation. Calculates the CRC for each bit in each byte of data.

Prototype

```
U32 SEGGER_CRC_CalcBitByBit(const U8  * pData,
                                  U32   NumBytes,
                                  U32   Crc,
                                  U32   Poly);
```

Parameters

Parameter	Description
`pData`	Pointer to the source buffer which holds the data for which to calculate the CRC checksum.
`NumBytes`	Number of bytes to include in the calculation. May be zero.
`Crc`	Initial value for incremental calculation of the CRC. May be zero.
`Poly`	LSB-first notation of the polynomial to use in this calculation. Must not exceed a polynomial degree of 32, but may be zero. For example, LSB-first notations of common polynomials are: 32-bit: `0xEDB88320`, 16-bit: `0x8408`, 8-bit: `0xE0`.

Return value

Resulting CRC checksum.

Example

Please see *Using API functions for arbitrary polynomials (LSB)* on page 32.

2.1.3 SEGGER_CRC_Calc_MSB()

Description

Provides a table-driven implementation of CRC calculations over a given range of memory using arbitrary polynomials in MSB-first notation. Dynamically creates a 16-entry-table for the given polynomial, which is then used to calculate the CRC for each nibble in each byte of data.

Prototype

```
U32 SEGGER_CRC_Calc_MSB(const U8   * pData,
                              U32    NumBytes,
                              U32    Crc,
                              U32    Poly,
                              U8     SizeOfPoly);
```

Parameters

Parameter	Description
`pData`	Pointer to the source buffer which holds the data for which to calculate the CRC checksum.
`NumBytes`	Number of bytes to include in the calculation. May be zero.
`Crc`	Initial value for incremental calculation of the CRC. May be zero.
`Poly`	MSB-first notation of the polynomial to use in this calculation. Must not exceed a polynomial degree of 32, but must be 4 bits in size minimum. For example, MSB-first notations of common polynomials are: 32-bit: `0xEDB88320`, 16-bit: `0x8408`, 8-bit: `0xE0`.
`SizeOfPoly`	Size of the polynomial to use for calculation of the CRC. Must contain the correct size of `Poly` (at least 4).

Return value

Resulting CRC checksum.

Example

Please see *Using API functions for arbitrary polynomials (MSB)* on page 33.

2.1.4 SEGGER_CRC_CalcBitByBit_MSB()

Description

Provides a bitwise implementation of CRC calculations over a given range of memory using arbitrary polynomials in MSB-first notation. Calculates the CRC for each bit in each byte of data.

Prototype

```
U32 SEGGER_CRC_CalcBitByBit_MSB(const U8  * pData,
                                      U32   NumBytes,
                                      U32   Crc,
                                      U32   Poly,
                                      U8    SizeOfPoly);
```

Parameters

Parameter	Description
`pData`	Pointer to the source buffer which holds the data for which to calculate the CRC checksum.
`NumBytes`	Number of bytes to include in the calculation. May be zero.
`Crc`	Initial value for incremental calculation of the CRC. May be zero.
`Poly`	MSB-first notation of the polynomial to use in this calculation. Must not exceed a polynomial degree of 32. For example, MSB-first notations of common polynomials are: 32-bit: `0xEDB88320`, 16-bit: `0x8408`, 8-bit: `0xE0`.
`SizeOfPoly`	Size of the polynomial to use for calculation of the CRC. Must contain the correct size of `Poly`.

Return value

Resulting CRC checksum.

Example

Please see *Using API functions for arbitrary polynomials (MSB)* on page 33.

2.1.5 SEGGER_CRC_Calc_09()

Description

Computes the CRC checksum over the given range of memory using a static table for the polynomial `0x09`.

Prototype

```
U8 SEGGER_CRC_Calc_09(const U8  * pData,
                            U32   NumBytes,
                            U8    Crc);
```

Parameters

Parameter	Description
`pData`	Pointer to the source buffer which holds the data for which to calculate the CRC checksum.
`NumBytes`	Number of bytes to include in the calculation. May be zero.
`Crc`	Initial value for incremental calculation of the CRC. May be zero.

Return value

Resulting CRC checksum.

Example

Please see *Using API functions for specified polynomials* on page 34.

2.1.6 SEGGER_CRC_Calc_48()

Description

Computes the CRC checksum over the given range of memory using a static table for the polynomial `0x48`.

Prototype

```
U8 SEGGER_CRC_Calc_48(const U8  * pData,
                            U32   NumBytes,
                            U8    Crc);
```

Parameters

Parameter	Description
`pData`	Pointer to the source buffer which holds the data for which to calculate the CRC checksum.
`NumBytes`	Number of bytes to include in the calculation. May be zero.
`Crc`	Initial value for incremental calculation of the CRC. May be zero.

Return value

Resulting CRC checksum.

Example

Please see *Using API functions for specified polynomials* on page 34.

2.1.7 SEGGER_CRC_Calc_1021()

Description

Computes the CRC checksum over the given range of memory using a static table for the polynomial `0x1021`.

Prototype

```
U16 SEGGER_CRC_Calc_1021(const U8  * pData,
                               U32   NumBytes,
                               U16   Crc);
```

Parameters

Parameter	Description
`pData`	Pointer to the source buffer which holds the data for which to calculate the CRC checksum.
`NumBytes`	Number of bytes to include in the calculation. May be zero.
`Crc`	Initial value for incremental calculation of the CRC. May be zero.

Return value

Resulting CRC checksum.

Example

Please see *Using API functions for specified polynomials* on page 34.

2.1.8 SEGGER_CRC_Calc_8408()

Description

Computes the CRC checksum over the given range of memory using a static table for the polynomial `0x8408`.

Prototype

```
U16 SEGGER_CRC_Calc_8408(const U8   * pData,
                               U32    NumBytes,
                               U16    Crc);
```

Parameters

Parameter	Description
`pData`	Pointer to the source buffer which holds the data for which to calculate the CRC checksum.
`NumBytes`	Number of bytes to include in the calculation. May be zero.
`Crc`	Initial value for incremental calculation of the CRC. May be zero.

Return value

Resulting CRC checksum.

Example

Please see *Using API functions for specified polynomials* on page 34.

2.1.9 SEGGER_CRC_Calc_04C11DB7()

Description

Computes the CRC checksum over the given range of memory using a static table for the polynomial `0x04C11DB7`.

Prototype

```
U32 SEGGER_CRC_Calc_04C11DB7(const U8   * pData,
                                   U32    NumBytes,
                                   U32    Crc);
```

Parameters

Parameter	Description
`pData`	Pointer to the source buffer which holds the data for which to calculate the CRC checksum.
`NumBytes`	Number of bytes to include in the calculation. May be zero.
`Crc`	Initial value for incremental calculation of the CRC. May be zero.

Return value

Resulting CRC checksum.

Example

Please see *Using API functions for specified polynomials* on page 34.

2.1.10 SEGGER_CRC_Calc_1EDC6F41()

Description

Computes the CRC checksum over the given range of memory using a static table for the polynomial `0x1EDC6F41`.

Prototype

```
U32 SEGGER_CRC_Calc_1EDC6F41(const U8  * pData,
                                   U32   NumBytes,
                                   U32   Crc);
```

Parameters

Parameter	Description
`pData`	Pointer to the source buffer which holds the data for which to calculate the CRC checksum.
`NumBytes`	Number of bytes to include in the calculation. May be zero.
`Crc`	Initial value for incremental calculation of the CRC. May be zero.

Return value

Resulting CRC checksum.

Example

Please see *Using API functions for specified polynomials* on page 34.

2.1.11 SEGGER_CRC_Calc_82F63B78()

Description

Computes the CRC checksum over the given range of memory using a static table for the polynomial `0x82F63B78`.

Prototype

```
U32 SEGGER_CRC_Calc_82F63B78(const U8  * pData,
                                   U32   NumBytes,
                                   U32   Crc);
```

Parameters

Parameter	Description
`pData`	Pointer to the source buffer which holds the data for which to calculate the CRC checksum.
`NumBytes`	Number of bytes to include in the calculation. May be zero.
`Crc`	Initial value for incremental calculation of the CRC. May be zero.

Return value

Resulting CRC checksum.

Example

Please see *Using API functions for specified polynomials* on page 34.

2.1.12 SEGGER_CRC_Calc_EDB88320()

Description

Computes the CRC checksum over the given range of memory using a static table for the polynomial `0xEDB88320`.

Prototype

```
U32 SEGGER_CRC_Calc_EDB88320(const U8  * pData,
                                   U32   NumBytes,
                                   U32   Crc);
```

Parameters

Parameter	Description
`pData`	Pointer to the source buffer which holds the data for which to calculate the CRC checksum.
`NumBytes`	Number of bytes to include in the calculation. May be zero.
`Crc`	Initial value for incremental calculation of the CRC. May be zero.

Return value

Resulting CRC checksum.

Example

Please see *Using API functions for specified polynomials* on page 34.

2.1.13 SEGGER_CRC_Validate()

Description

Validates the implementation of the emLib CRC API. Returns upon the first error encountered, omitting any further validation steps. Performs the validation using the following polynomials: `0xEDB88320`, `0x04C11DB7`, `0x82F63B78`, `0x1EDC6F41`, `0x8408`, `0x1021`, `0x48`, and `0x09`.

Prototype

```
I8 SEGGER_CRC_Validate();
```

Return value

Return value	Description
SEGGER_CRC_VALIDATE_SUCCESS	Validation completed successfully.
SEGGER_CRC_VALIDATE_ERROR_ARBITRARY	Failed to validate the table-driven implementation using the LSB-first notation of arbitrary polynomials.
SEGGER_CRC_VALIDATE_ERROR_SPECIFIC	Failed to validate the table-driven implementation using the LSB-first notation of a specified polynomial.
SEGGER_CRC_VALIDATE_ERROR_BITWISE	Failed to validate the bit-by-bit implementation using the LSB-first notation of arbitrary polynomials.
SEGGER_CRC_VALIDATE_ERROR_ARBITRARY_MSB	Failed to validate the table-driven implementation using the MSB-first notation of arbitrary polynomials.
SEGGER_CRC_VALIDATE_ERROR_SPECIFIC_MSB	Failed to validate the table-driven implementation using the MSB-first notation of a specified polynomial.
SEGGER_CRC_VALIDATE_ERROR_BITWISE_MSB	Failed to validate the bit-by-bit implementation using the MSB-first notation of arbitrary polynomials.

Chapter 3

Examples

This chapter explains the usage of the emLib CRC API.

3.1 Example code

3.1.1 Using API functions for arbitrary polynomials (LSB)

This sample shows the incremental calculation of CRC checksums for two distinct blocks of data using the emLib CRC API functions for arbitrary polynomials in least-significant-bit-first notation.

```
#include "SEGGER_CRC.h"

//
// Sample data.
//
const U8 _aData1[4]  = {
  0x00, 0x11, 0x22, 0x33
};
const U8 _aData2[48] = {
  0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF,
  0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x44,
  0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x44, 0x55,
  0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x44, 0x55, 0x66
};

int main() {
  //
  // Declare variables.
  //
  U8  NumBytes1;
  U8  NumBytes2;
  U32 CRC;
  U32 Poly;
  U32 InitialCRC;

  //
  // Init variables.
  //
  NumBytes1  = 4;           // Number of bytes at _aData1.
  NumBytes2  = 48;          // Number of bytes at _aData2.
  Poly       = 0xEDB88320;  // Arbitrary polynomial in LSB-first notation.
  InitialCRC = 0x1A2B3C4D;  // Initial value for incremental calculation.
  //
  // Update CRC for the first block of data (bitwise).
  //
  CRC = SEGGER_CRC_CalcBitByBit(_aData1, NumBytes1, InitialCRC, Poly);
  //
  // Update CRC for the second block of data (table-driven).
  //
  CRC = SEGGER_CRC_Calc(_aData2, NumBytes2, CRC, Poly);

  return CRC;
}
```

3.1.2 Using API functions for arbitrary polynomials (MSB)

This sample shows the incremental calculation of CRC checksums for two distinct blocks of data using the emLib CRC API functions for arbitrary polynomials in most-significant-bit-first notation.

```
#include "SEGGER_CRC.h"

//
// Sample data.
//
const U8 _aData1[4]  = {
  0x00, 0x11, 0x22, 0x33
};
const U8 _aData2[48] = {
  0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF,
  0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x44,
  0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x44, 0x55,
  0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x44, 0x55, 0x66
};

int main() {
  //
  // Declare variables.
  //
  U8  NumBytes1;
  U8  NumBytes2;
  U8  SizeOfPoly;
  U32 CRC;
  U32 Poly;
  U32 InitialCRC;

  //
  // Init variables.
  //
  NumBytes1  = 4;           // Number of bytes at _aData1.
  NumBytes2  = 48;          // Number of bytes at _aData2.
  SizeOfPoly = 32;          // Size of the polynomial to use.
  Poly       = 0x04C11DB7;  // Arbitrary polynomial in MSB-first notation.
  InitialCRC = 0x1A2B3C4D;  // Initial value for incremental calculation.
  //
  // Update CRC for the first block of data (bitwise).
  //
  CRC = SEGGER_CRC_CalcBitByBit_MSB(_aData1, NumBytes1, InitialCRC, Poly, SizeOfPoly);
  //
  // Update CRC for the second block of data (table-driven).
  //
  CRC = SEGGER_CRC_Calc_MSB(_aData2, NumBytes2, CRC, Poly, SizeOfPoly);

  return CRC;
}
```

3.1.3 Using API functions for specified polynomials

This sample shows the incremental calculation of CRC checksums for two distinct blocks of data using an emLib CRC API function for a specified polynomial. This sample uses `SEGGER_CRC_Calc_EDB88320()`, resulting in a 32-bit CRC for the polynomial `0xEDB88320` (LSB-first notation). Using emLib CRC API functions for other specified polynomials may require modifications to `CRC` and `InitialCRC`.

```
#include "SEGGER_CRC.h"

//
// Sample data.
//
const U8 _aData1[4]  = {
  0x00, 0x11, 0x22, 0x33
};
const U8 _aData2[48] = {
  0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF,
  0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x44,
  0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x44, 0x55,
  0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x44, 0x55, 0x66
};

int main() {
  //
  // Declare variables.
  //
  U8  NumBytes1;
  U8  NumBytes2;
  U32 CRC;
  U32 InitialCRC;

  //
  // Init variables.
  //
  NumBytes1  = 4;           // Number of bytes at _aData1.
  NumBytes2  = 48;          // Number of bytes at _aData2.
  InitialCRC = 0x1A2B3C4D;  // Initial value for incremental calculation.
  //
  // Update CRC for the first block of data.
  //
  CRC = SEGGER_CRC_Calc_EDB88320(_aData1, NumBytes1, InitialCRC);
  //
  // Update CRC for the second block of data.
  //
  CRC = SEGGER_CRC_Calc_EDB88320(_aData2, NumBytes2, CRC);

  return CRC;
}
```

3.1.4 Using API functions to check a message's integrity

Using the polynomial `0x04C11DB7`, this sample calculates a 32-bit CRC for a given message and appends the resulting checksum to that message. Typically, this is done by the sender before sending the message towards its recepient. The sample then proceeds to verify the message’s integrity through calculation of the 32-bit CRC for the augmented message, which typically is done by the recipient.

```
#include "SEGGER_CRC.h"

int main(void) {
  //
  // Declare variables.
  //
  U8  NumBytes;
  U8  SizeOfPoly;
  U32 CRC;
  U32 Poly;
  U32 InitialCRC;

  //
  // Sample message (8 bytes of data + 4 bytes for augmentation of 32-bit CRC).
  //
  U8  aMessage[12] = {
    0x8C, 0x4C, 0xCC, 0x2C, 0xAC, 0x6C, 0xEC, 0x1C,
    0x0,  0x0,  0x0,  0x0
  };
  //
  // Init variables.
  //
  NumBytes   = 8;            // Number of bytes at aMessage.
  SizeOfPoly = 32;           // Size of the polynomial to use.
  Poly       = 0x04C11DB7;   // Arbitrary polynomial in MSB-first notation.
  InitialCRC = 0x1A2B3C4D;   // Initial value for incremental calculation.
  //
  // Calc CRC for the given message.
  //
  CRC = SEGGER_CRC_CalcBitByBit_MSB(aMessage, NumBytes, InitialCRC, Poly, SizeOfPoly);
  //
  // Append the resulting CRC (4 bytes) to the message.
  //
  aMessage[ 8] = (CRC >> 24) & 0xFF;
  aMessage[ 9] = (CRC >> 16) & 0xFF;
  aMessage[10] = (CRC >> 8)  & 0xFF;
  aMessage[11] = (CRC >> 0)  & 0xFF;
  //
  // Calc CRC for the augmented message (NumBytes + 4 Bytes CRC).
  // Non-zero result indicates erroneuos data.
  // Otherwise, no error has been detected.
  //
  CRC = SEGGER_CRC_CalcBitByBit_MSB(aMessage, NumBytes+4, InitialCRC, Poly, SizeOfPoly);
  if (CRC != 0) {
    return -1;
  }
  return 0;
}
```

3.2 Sample applications

emLib CRC includes sample applications to demonstrate its functionality and to provide an easy to use starting point for your application. The applications' source code is included in the shipment. The following applications are delivered with emLib CRC:

Application name	Target platform	Description
CRCAugmentFile.exe	Windows	Command line tool to augment a file with its calculated CRC. Uses the 32-bit polynomial `0x04C11DB7`.
CRCCalc.exe	Windows	Command line tool to calculate the CRC for a given file using the bitwise API functions for arbitrary polynomials. Calculation parameters are configurable by the user.
CRCValidate.exe	Windows	Console application to validate the emLib CRC API using standard test patterns.
CRCVerifyAugmentedFile.exe	Windows	Command line tool to verify the integrity of an augmented file. Uses the 32-bit polynomial `0x04C11DB7`.

3.2.1 CRCAugmentFile

CRCAugmentFile is a Windows command line tool to augment a file with its calculated CRC. Typically, this is done by a sender before sending the file towards its recepient. CRCAugmentFile uses the 32-bit polynomial `0x04C11DB7` (MSB-first notation) and requires the file to not be write-protected. The application *CRCVerifyAugmentedFile* on page 38 may be used to verify the augmented file's integrity.

WARNING: It is recommended to back up the target file prior to using this application sample.

```
C:> CRCAugmentFile sample.bin

(c) 2015 SEGGER Microcontroller GmbH & Co. KG
         www.segger.com

CRCAugmentFile V1.00
Sample appplication demonstrating the augmentation of messages using their CRC.
This sample uses the 32-bit polynomial 0x04C11DB7 (MSB-first notation).
Compiled at Jul 22 2015, 09:55:46.

Calculating CRC for the given file.
The calculation included 1981810 bytes in total.
The resulting CRC is 0x5226DB8C.
Appending resulting CRC to the given file... completed.

C:> _
```

Usage

`CRCAugmentFile` <sourcefile>

Parameter	Description
<sourcefile>	Path to the file for which to calculate the CRC. Must not be write-protected.

3.2.2 CRCCalc

CRCCalc is a Windows command line tool to calculate the CRC for a given file using the bitwise API functions for arbitrary polynomials. Further calculation parameters are configurable by the user, including the polynomial to use and an initial value for the calculation. CRCCalc may be used to easily calculate CRC checksums for arbitrary files.

```
C:> CRCCalc sample.bin -MSB 32 -poly 0x04C11DB7 -init 0x1A2B3C4D

(c) 2015 SEGGER Microcontroller GmbH & Co. KG
         www.segger.com

CRCCalc V1.00
Sample appplication demonstrating the calculation of CRC checksums for a given
 file.
All parameters are configurable, including polynomial and initial value to use.
Compiled at Jul 22 2015, 09:55:47.

Calculating the CRC for file sample.bin using the following parameters:
  Polynomial   : 0x4C11DB7.
  Initial value: 0x1A2B3C4D.
  Notation     : MSB-first.
The calculation included 1981810 bytes in total.
The resulting CRC is 0x5226DB8C.

C:> _
```

Usage

`CRCCalc` <sourcefile> [-MSB [<SizeOfPoly>]] [-Poly <Polynomial>] [-init <InitialCRC>]

Parameter	Description
<sourcefile>	Path to the file for which to calculate the CRC.
-MSB [<SizeOfPoly>]	(Optional) Used to indicate MSB-first notation. If not set, calculation is performed LSB-first. If set and Polynomial is given, SizeOfPoly must indicate that polynomial's size. If set and Polynomial is not given, a default polynomial of known size is used, hence SizeOfPoly is ignored and may be omitted.
-poly <Polynomial>	(Optional) Defines the polynomial to be used, which must not exceed a polynomial degree of 32. Expects hexadecimal notation. If not set, the 16-bit polynomial `0x8408` is used for LSB-first calculations and the 16-bit polynomial `0x1021` is used for MSB-first calculations.
-init <InitialCRC>	(Optional) Initial value for incremental calculations. Expects hexadecimal notation. If not set, `0x0` is used as initial value.

3.2.3 CRCValidate

CRCValidate is a Windows console application to validate the emLib CRC API using standard test patterns. It uses the emLib CRC validation API to validate the implementation of the emLib CRC API functions. CRCValidate will display an error code if any validation fails, omitting further validation steps. For a list of error codes, refer to *SEGGER_CRC_Validate* on page 29.

```
C:> CRCValidate

(c) 2015 SEGGER Microcontroller GmbH & Co. KG
         www.segger.com

CRCValidate V1.00
Sample program to validate the implementation.
Compiled at Jul 22 2015, 09:55:48.

Validation of CRC API functions... completed successfully.

C:> _
```

3.2.4 CRCVerifyAugmentedFile

CRCVerifyAugmentedFile is a Windows command line tool to verify the integrity of an augmented file. Typically, this is done by the file's recipient. CRCVerifyAugmentedFile uses the 32-bit polynomial `0x04C11DB7` (MSB-first notation). The application *CRCAugmentFile* on page 36 may be used to augment a file with its calculated CRC.

```
C:> CRCVerifyAugmentedFile sample.bin

(c) 2015 SEGGER Microcontroller GmbH & Co. KG
         www.segger.com

CRCVerifyAugmentedFile V1.00
Sample appplication demonstrating the verification of augmented messages.
This sample uses the 32-bit polynomial 0x04C11DB7 (MSB-first notation).
Compiled at Jul 22 2015, 09:55:49.

Calculating CRC for the augmented file.
The calculation included 1981814 bytes in total.
Checking integrity... completed successfully.

C:> _
```

Usage

`CRCVerifyAugmentedFile` <sourcefile>

Parameter	Description
<sourcefile>	Path to the file for which to perform the verification.

Chapter 4

Performance and resource usage

This chapter covers the perfomance and resource usage of emLib CRC. Contained information may be used to obtain sufficient estimates for most target systems.

emLib CRC is designed to cater for many different embedded design requirements, from constrained microcontrollers to high performance microprocessors. Each single module might be excluded from build in order to construct a highly compact, minimal system.

4.1 Target system configuration

The following table shows the hardware and toolchain details used to measure the given values:

Detail	Description
CPU	Cortex-M7, 200 MHz
Tool chain	IAR Embedded Workbench for ARM V7.40
Model	Thumb-2 instructions
Compiler options	Highest speed optimization

4.1.1 Performance

The following table shows performance values for emLib CRC:

LSB-first (using `0xEDB88320`)	**Setup time (cycles)**	**Cycles per byte**	**MByte per second**	**Computation time for 1 MByte data**
Bitwise, arbitrary polynomial	-	124	1.53	650.12 ms
Table-driven, arbitrary polynomial	784	17	11.21	89.13 ms
Table-driven, specified polynomial	-	13	14.67	68.16 ms
MSB-first (using `0x04C11DB7`)	**Setup time (cycles)**	**Cycles per byte**	**MByte per second**	**Computation time for 1 MByte data**
Bitwise, arbitrary polynomial	148	134	1.42	702.54 ms
Table-driven, arbitrary polynomial	878	20	9.53	104.86 ms
Table-driven, specified polynomial	-	11	17.33	57.67 ms

4.1.2 Resource usage

The following table shows ROM and RAM requirements for emLib CRC:

LSB-first	**ROM**	**RAM (static)**	**RAM (stack)**
Bitwise, arbitrary polynomial	56 bytes	0 bytes	15 bytes
Table-driven, arbitrary polynomial	92 bytes	0 bytes	83 bytes
Table-driven, specified polynomial (7-bit)	284 bytes	0 bytes	6 bytes
Table-driven, specified polynomial (16-bit)	548 bytes	0 bytes	7 bytes
Table-driven, specified polynomial (32-bit)	1060 bytes	0 bytes	9 bytes
MSB-first	**ROM**	**RAM (static)**	**RAM (stack)**
Bitwise, arbitrary polynomial	168 bytes	0 bytes	28 bytes
Table-driven, arbitrary polynomial	234 bytes	0 bytes	92 bytes
Table-driven, specified polynomial (7-bit)	284 bytes	0 bytes	6 bytes
Table-driven, specified polynomial (16-bit)	548 bytes	0 bytes	9 bytes
Table-driven, specified polynomial (32-bit)	1056 bytes	0 bytes	13 bytes

Chapter 5

Support

This chapter should help if any problem occurs, e.g. with the use of the emLib CRC functions, and describes how to contact the SEGGER support.

5.1 Contacting SEGGER support

If you are a registered emLib CRC user and need to contact the SEGGER support, please send the following information via email to support@segger.com:

- The emLib CRC version.
- Your emLib CRC registration number.
- If you are unsure about the above information, you may also use the name of the shipped emLib CRC ZIP-file (which contains the above information).
- A detailed description of the problem.
- (Optional) A project with which we can reproduce the problem.

Chapter 6

Glossary

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

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

Final XOR
Scalar that is applied to the calculated CRC checksum just prior to appending the input data.

Generator polynomial
The divisor in the polynomial division. Specified by a scalar or vector.

Initial value
Scalar that specifies the initial state of the CRC calculation.

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.

Lookup table
An array that replaces runtime computation with a simpler array indexing operation. May be precalculated and stored in static program storage.

LSB-first notation
Least-significant-bit-first notation. Binary notation of a number in which the first bit holds the least significant value and determines whether the number is even or odd.

MSB-first notation
Most-significant-bit-first notation. Binary notation of a number in which the first bit holds the most significant value.