

# emLoad

Bootstrap loader for  
embedded applications

User Guide & Reference Manual

Document: UM04002  
Software Version: 4.10a  
Revision: 0  
Date: November 10, 2017



A product of SEGGER Microcontroller GmbH & Co. KG

[www.segger.com](http://www.segger.com)

## Disclaimer

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

## Copyright notice

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

© 2010-2017 SEGGER Microcontroller GmbH & Co. KG, Hilden / Germany

## Trademarks

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

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

## Contact address

SEGGER Microcontroller GmbH & 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: November 10, 2017

Software	Revision	Date	By	Description
4.10a	0	171110	OO	Added new UART API <code>BTL_IF_UART_ConfigDelayBeforeExit()</code> .
4.10	0	170717	OO	Added new UART interface description.
4.08a	0	170712	TG	Added new FS-NOR interface description.
4.08	0	170612	OO	Added information about how to <i>Digitally sign an update</i> on page 27 using the Signature add-on.
4.06c	0	170421	TG	Added new FS-NAND interface description.
4.06b	0	170221	OO	Added <code>pfResetIF</code> to <code>BTL_NET_API</code> .
4.06	3	170210	TG	Minor syntax correction.
4.06	2	170103	OO	Minor changes. Chapter "emLoad tools". • Added <code>BTL_ImageCreator</code> information.
4.06	1	161025	TG	Add quick start guide.
4.06	0	161021	TG	Initial 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.
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.
<b>GUIElement</b>	Buttons, dialog boxes, menu names, menu commands.
<b>Emphasis</b>	Very important sections.



# Table of contents

---

1	Introduction to emLoad .....	11
1.1	What is emLoad? .....	12
1.1.1	Functionality of the emLoad software .....	12
1.1.2	emLoad memory layout .....	12
1.1.3	Flow chart .....	14
1.2	Contents of emLoad .....	15
1.2.1	Folder structure of an emLoad shipment .....	15
1.2.2	Shipped tools .....	15
1.2.3	Project structure .....	16
1.2.4	Availability and FLASH devices .....	16
1.2.5	Development environment (compiler) .....	16
1.2.6	Crypto Add-On .....	17
1.2.7	Signature Add-On .....	17
1.3	Quick start guide .....	18
1.3.1	Loading emLoad into a target .....	18
1.3.2	Preparing an emLoad firmware update .....	18
1.3.3	Updating the firmware using emLoad for MMC/SD .....	18
1.3.4	Updating the firmware using emLoad for UART .....	18
1.3.5	Updating the firmware using emLoad for USB HID .....	18
1.3.6	Updating the firmware using emLoad for USBH MSD .....	18
1.3.7	Updating the firmware using emLoad for NAND/NOR .....	19
1.3.8	Configuration of firmware passwords and other firmware identifiers .....	19
1.3.9	Configuration of the firmware area .....	19
2	emLoad tools .....	21
2.1	PrepareFW / PrepareFWPRO .....	22
2.1.1	Running PrepareFW.exe / PrepareFWPRO.exe .....	22
2.1.2	Additional options of PrepareFWPRO .....	23
2.2	UART Updater .....	24
2.2.1	Running UART_Updater.exe .....	24
2.3	USB HID Updater .....	25
2.3.1	Running USB_HID_Updater.exe .....	25
2.4	BTL Image Creator .....	26
2.4.1	Running BTL_ImageCreator.exe .....	26
2.5	Digitally sign an update .....	27
2.5.1	Using the batch script to sign a firmware update .....	27
2.5.2	Signing a firmware update manually .....	27
2.5.2.1	ECDSA: Creating a private/public key pair using P-256 .....	27
2.5.2.2	RSA: Creating a 2048 bit private/public key pair .....	28
2.5.2.3	Preparing the data to be signed .....	28

2.5.2.4	Signing the data .....	28
2.5.2.5	Generating the firmware update file .....	28
2.5.2.6	Extracting the private key content for use with emLoad .....	28
3	Configuration .....	29
3.1	Compile-time configuration .....	30
3.1.1	Compile-time configuration switches .....	30
3.2	Debug level .....	32
4	Core functions .....	33
4.1	emLoad BTL API .....	34
4.1.1	Detailed core functions .....	35
4.1.1.1	BTL_AddStateChangeHook() .....	36
4.1.1.2	BTL_AssignMemory() .....	37
4.1.1.3	BTL_Start() .....	38
4.1.1.4	BTL_StartFW() .....	39
4.1.1.5	BTL_Exec() .....	40
4.1.1.6	BTL_AddCrypto() .....	41
4.1.1.7	BTL_AddSignature_ECDSA() .....	42
4.1.1.8	BTL_AddSignature_RSA() .....	43
4.1.1.9	BTL_ConfigFlashInfo() .....	44
4.1.1.10	BTL_ConfigReadMemSize() .....	45
4.1.1.11	BTL_ConfigRetryUpdateOnFail() .....	46
4.1.1.12	BTL_ConfigUpdatePolicy() .....	47
4.1.1.13	BTL_GetCpuClock() .....	48
4.1.1.14	BTL_GetStatus() .....	49
4.1.1.15	BTL_SetCpuClock() .....	50
4.1.1.16	BTL_SetInitTimeout() .....	51
4.1.1.17	BTL_SetTickHook() .....	52
4.1.1.18	BTL_ConfigSkipBootCrcCheck() .....	53
4.1.1.19	BTL_SetFlashCrcProgressHook() .....	54
4.1.1.20	BTL_ConfigInterfaceReadyDelay() .....	55
4.1.1.21	BTL_DisableUpdateInitTimeout() .....	56
4.1.1.22	BTL_GetUpdateInitTimeoutTimestamp() .....	57
4.1.1.23	BTL_Delay() .....	58
4.1.1.24	BTL_GetCompanyName() .....	59
4.1.1.25	BTL_GetDeviceName() .....	60
4.1.1.26	BTL_GetFwString() .....	61
4.1.1.27	BTL_GetFwVersion() .....	62
4.1.1.28	BTL_GetUpdateFileName() .....	63
4.1.1.29	BTL_SetCompanyName() .....	64
4.1.1.30	BTL_SetDeviceName() .....	65
4.1.1.31	BTL_SetFwPassword() .....	66
4.1.1.32	BTL_SetFwStartAddr() .....	67
4.1.1.33	BTL_SetFwString() .....	68
4.1.1.34	BTL_SetUpdateFileName() .....	69
4.2	emLoad BTL structures .....	70
4.2.1	Structure BTL_FLASH_DRIVER .....	70
4.2.2	Structure BTL_FLASH_INFO .....	71
4.2.3	Structure BTL_HOOK_ON_STATE_CHANGE .....	72
4.2.4	Structure BTL_NET_API .....	73
4.3	emLoad status codes .....	75
5	CPU specific functions .....	77
5.1	CPU specific API .....	78
5.1.1	CPU specific functions detailed .....	78
5.1.1.1	BTL_CPU_StartApplication() .....	78



6	Updating via MMC/SD card .....	79
6.1	Update Procedure with MMC/SD .....	80
6.1.1	Loading emLoad into a target .....	80
6.1.2	Preparing an emLoad firmware update .....	80
6.1.3	Update the firmware using emLoad .....	80
6.2	MMC/SD interface API .....	81
6.2.1	BTL_IF_FS_MMC_SD_Exit() .....	82
6.2.2	BTL_IF_FS_MMC_SD_Init() .....	83
6.2.3	BTL_IF_FS_MMC_SD_IsInterfaceReady() .....	84
6.2.4	BTL_IF_FS_MMC_SD_Read() .....	85
6.2.5	BTL_IF_FS_MMC_SD_Reset() .....	86
7	Updating via UART .....	87
7.1	Update Procedure with UART .....	88
7.1.1	Loading emLoad into a target .....	88
7.1.2	Preparing an emLoad firmware update .....	88
7.1.3	Update the firmware using emLoad .....	88
7.2	UART interface API .....	89
7.2.1	BTL_IF_UART_Exit() .....	90
7.2.2	BTL_IF_UART_Init() .....	91
7.2.3	BTL_IF_UART_IsInterfaceReady() .....	92
7.2.4	BTL_IF_UART_OnRx() .....	93
7.2.5	BTL_IF_UART_OnTx() .....	94
7.2.6	BTL_IF_UART_SetAPI() .....	95
7.2.7	BTL_IF_UART_ConfigDelayBeforeExit() .....	96
7.3	Interface specific data structures .....	97
7.3.1	Structure BTL_UART_API .....	97
8	Updating via USB HID .....	99
8.1	Update Procedure with USB HID .....	100
8.1.1	Loading emLoad into a target .....	100
8.1.2	Preparing an emLoad firmware update .....	100
8.1.3	Update the firmware using emLoad .....	100
8.2	USB HID interface API .....	101
8.2.1	BTL_IF_USB_HID_ConfigDelayAfterDetach() .....	102
8.2.2	BTL_IF_USB_HID_Exit() .....	103
8.2.3	BTL_IF_USB_HID_Init() .....	104
8.2.4	BTL_IF_USB_HID_IsInterfaceReady() .....	105
8.3	Interface specific data structures .....	106
8.3.1	Structure BTL_USB_HID_PACKET .....	106
9	Updating via USBH MSD .....	107
9.1	Update Procedure with USBH MSD .....	108
9.1.1	Loading emLoad into a target .....	108
9.1.2	Preparing an emLoad firmware update .....	108
9.1.3	Update the firmware using emLoad .....	108
9.2	USBH MSD interface API .....	109
9.2.1	BTL_IF_USBH_MSD_Exit() .....	110
9.2.2	BTL_IF_USBH_MSD_Init() .....	111
9.2.3	BTL_IF_USBH_MSD_IsInterfaceReady() .....	112
9.2.4	BTL_IF_USBH_MSD_Read() .....	113
9.2.5	BTL_IF_USBH_MSD_Reset() .....	114
10	Updating via NAND/NOR .....	115
10.1	Update Procedure with NAND/NOR .....	116
10.1.1	Loading emLoad into a target .....	116
10.1.2	Preparing an emLoad firmware update .....	116

10.1.3	Update the firmware using emLoad .....	116
10.2	NAND interface API .....	117
10.2.1	BTL_IF_FS_NAND_Exit() .....	118
10.2.2	BTL_IF_FS_NAND_Init() .....	119
10.2.3	BTL_IF_FS_NAND_IsInterfaceReady() .....	120
10.2.4	BTL_IF_FS_NAND_Read() .....	121
10.2.5	BTL_IF_FS_NAND_Reset() .....	122
10.3	NOR interface API .....	123
10.3.1	BTL_IF_FS_NOR_Exit() .....	124
10.3.2	BTL_IF_FS_NOR_Init() .....	125
10.3.3	BTL_IF_FS_NOR_IsInterfaceReady() .....	126
10.3.4	BTL_IF_FS_NOR_Read() .....	127
10.3.5	BTL_IF_FS_NOR_Reset() .....	128
11	Debugging .....	129
11.1	Message output .....	130
11.1.1	Debug API functions .....	130
11.1.1.1	BTL_Log() .....	131
11.1.1.2	BTL_Panic() .....	132
11.1.1.3	BTL_Warn() .....	133

# Chapter 1

## Introduction to emLoad

---

This chapter provides an introduction to using emLoad. It explains the basic concepts behind emLoad.

## 1.1 What is emLoad?

emLoad is a software that allows program updates in embedded applications. The software consists of one or two Windows executables on the one hand (PrepareFW[PRO] and - if USB HID is the chosen interface - USB\_HID\_Updater) and another program for the target application (BTL) on the other hand. It can be used with one of the following already available interfaces:

Interface	Explanation
USB Device HID	USB connection from target to a PC (see <i>Updating via USB HID</i> on page 99).
USB Host MSD	Update via USB stick (see <i>Updating via USBH MSD</i> on page 107).
FS MMC/SD card	Update via MMC/SD card (see <i>Updating via MMC/SD card</i> on page 79).
FS NAND	Update via file on NAND (see <i>Updating via NAND/NOR</i> on page 115).
FS NOR	Update via file on NOR (see <i>Updating via NAND/NOR</i> on page 115).

### 1.1.1 Functionality of the emLoad software

After `RESET`, instead of starting the application program immediately, the emLoad BTL gets started first. The BTL then waits for an update interface to signal an available update for a configurable amount of time (default: 0,5 seconds). If no update interface provides an update, the BTL checks the flash memory for a valid application program and starts it in case there is one. For the application program, the only difference while running with the BTL is the program's location at a different area of the flash memory and that it is not started immediately after `RESET`, but after a certain delay. Except for this, the application program is not affected by the BTL in any way and has all resources available; it can use interrupts and the entire RAM of the target system without limitation.

### 1.1.2 emLoad memory layout

The basic memory layout for emLoad consists of 3 areas: one for emLoad itself and another area for the firmware, while the third area constitutes the firmware info area. For an easier understanding, the following memory map shows a typical layout for a CPU with flash beginning at `0x08000000` with 256kBytes of flash:

Address	Used for
<code>0x08000000</code>	Beginning of the emLoad area including vectors and vector forwarding code, if necessary for the CPU. emLoad has to reside in the same area as the reset vector to be able to start directly after reset.
<code>0x08005FFF</code>	End of the emLoad area. In this sample emLoad needs 24kBytes of flash. The actual size necessary for emLoad is the size of emLoad rounded up to complete sectors. So if emLoad needs 23kBytes of flash but the target has 2kBytes sectors the emLoad area has to be widened to 24kBytes.
<code>0x08006000</code>	Start of the firmware area. This is the location where the user firmware will be stored. The size of the firmware area is typically calculated as follows: Size of CPU flash Memory ( <code>0x40000</code> ) - Start of firmware area ( <code>0x6000</code> ) - size of firmware info area ( <code>0x10</code> ).
<code>0x0803FFEF</code>	Start of the firmware info area. The firmware info area stores information about the current firmware such as a CRC checksum that is used to verify the firmware before a start of the firmware. The location of the firmware info area is always the last bytes of the last configured flash range. The size of the firmware info area can be configured with the define <code>BTL_FIRMWARE_INFO_SIZE</code> (see <i>Compile-time configuration switches</i> on page 30).

*This is just an example and needs to be adapted to every CPU.*

### **Note on the emLoad size**

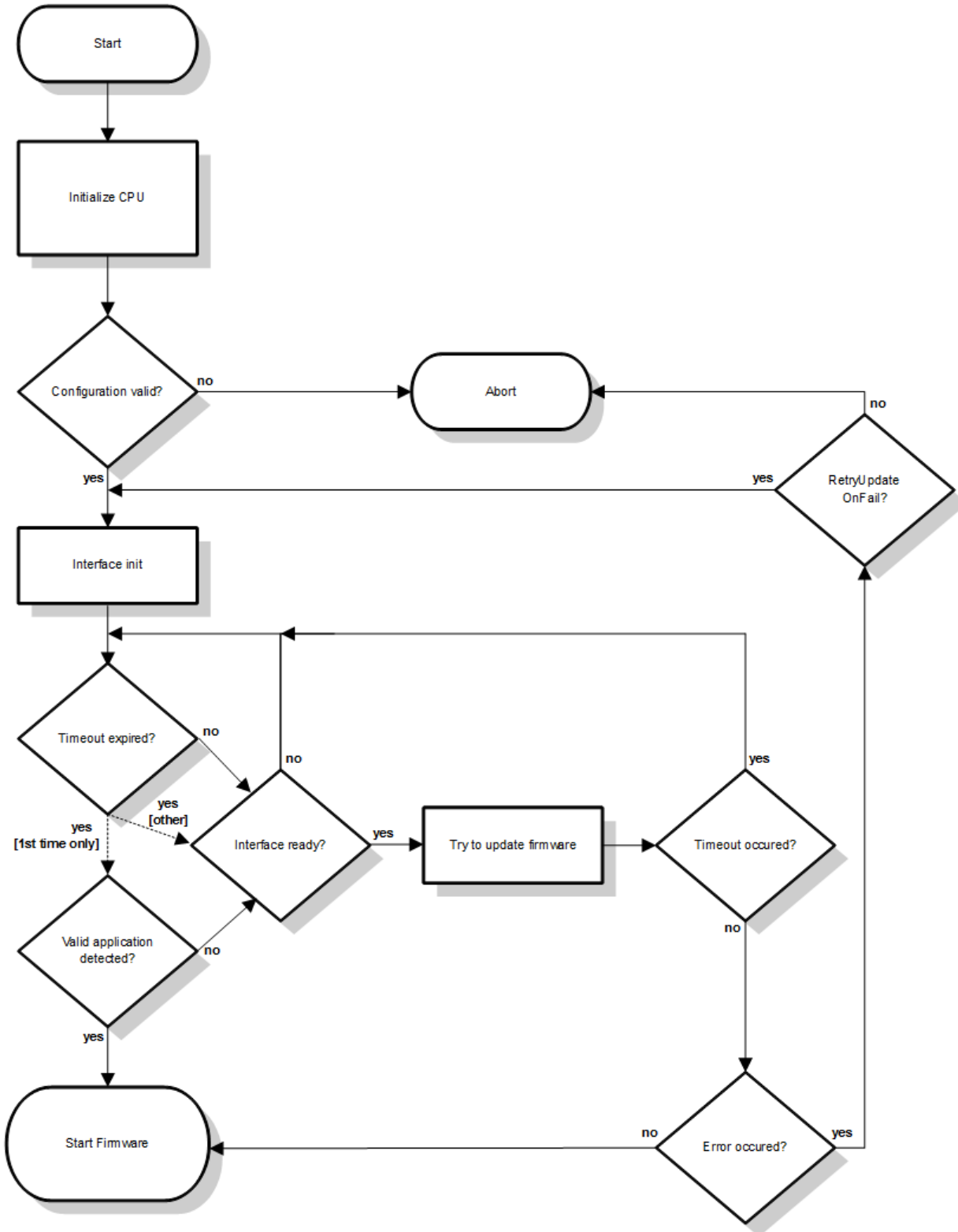
The size (i.e. the number of flash sectors) actually used by emLoad is of course different in DEBUG and in RELEASE configurations. By default, an emLoad project is delivered with a layout that supports both configurations.

Sometimes the difference could be quite important between DEBUG and RELEASE sizes of emLoad. Thus in case of constraints on the flash size needed by the firmware, it could be interesting to use a smaller footprint for emLoad by compiling it in RELEASE and changing the layout accordingly. Note that in this case, emLoad will probably not work anymore when compiled in DEBUG.

In order to change the layout, update the define `BTL_SIZE` or call `BTL_ConfigFlashInfo` (see *BTL\_ConfigFlashInfo* on page 44). Note that the size given should be rounded up to a sector size. As the firmware will be shifted, it will be also needed to update the firmware to start from the new address.

### 1.1.3 Flow chart

The diagram below shows the flowchart of the BTL software:



## 1.2 Contents of emLoad

### 1.2.1 Folder structure of an emLoad shipment

The following table shows the contents of the emLoad root directory:

Directory	Contents
Doc\	Contains manuals and documentations.
emLoadV4\	Contains the emLoad start project.
emLoadV4\Application	Contains Main.c which contains a basic configuration that can be altered by the user.
emLoadV4\BTL	Contains the emLoad bootstrap loader.
emLoadV4\Config	Contains several generic configuration routines e.g. for debug output messages as well as the memory layout of the firmware update.
emLoadV4\CRYPTO	Contains functions for the firmware signature check. May not be included when the Signature add-on is not present.
emLoadV4\FS	Contains a file system stack that is needed with some update interfaces. May not be included if not necessary for the selected update interface.
emLoadV4\Inc	Generic include files between several middleware components.
emLoadV4\LIB	Contains functions for the firmware encryption. May not be included when the Crypto add-on is not present.
emLoadV4\OS	Contains an OS that is needed with some update interfaces. May not be included if not necessary for the selected update interface.
emLoadV4\SECURE	Contains functions for the firmware signature check. May not be included when the Signature add-on is not present.
emLoadV4\SEGGER	Contains generic optimized functionalities.
emLoadV4\Setup	Contains device specific hardware and configuration modules.
emLoadV4\USB	Contains a USB-Device stack that is needed with some update interfaces. May not be included if not necessary for the selected update interface.
emLoadV4\USBH	Contains a USB-Host stack that is needed with some update interfaces. May not be included if not necessary for the selected update interface.
FirmwareSample\	Contains sample projects for firmware updates, prebuilt sample firmwares and linker files that can be used to build firmware updates for emLoad.
Windows\	Contains PC tools that are necessary to prepare a firmware update and transfer it to the target.

### 1.2.2 Shipped tools

emLoad is shipped with the following executable tools for Windows PCs:

- PrepareFW or PrepareFWPRO (if the Crypto add-on is present).
- UART Updater, used for UART interface.
- USB HID Updater, used for USB HID interface.
- BTL Image Creator, used to create a programmable image containing a firmware update with/without BTL.

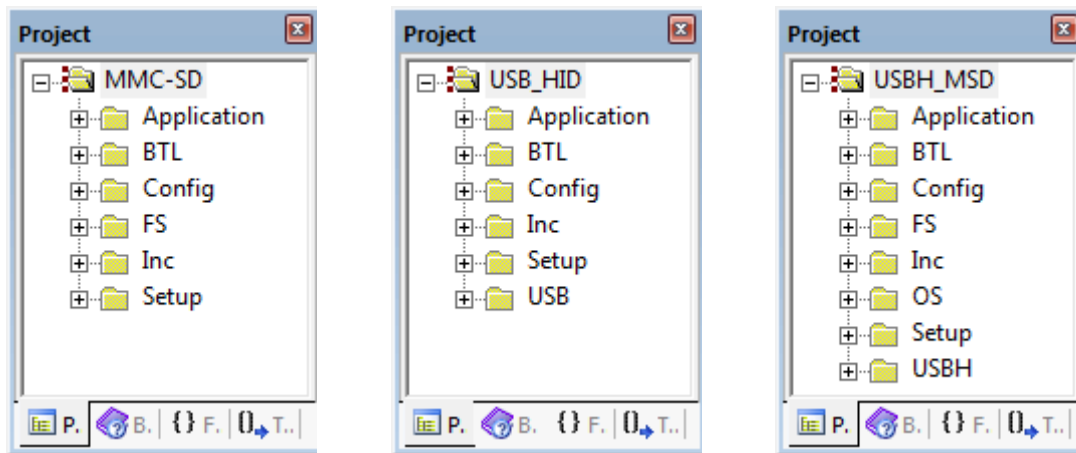
For more information, see chapter *emLoad tools* on page 21.

### 1.2.3 Project structure

emLoad shipments include one or several start project(s) for the chosen interface(s) and tool chain(s). The actual structure of these start projects depends on the particular interface, but closely resemble the folder structure of the shipment. Regardless of the interface, each start project includes the \*.c, \*.h and \*.s files contained in the folders "Application", "BTL", "Config", "Inc" and "Setup". In addition, each project includes the files for its appropriate interface, i.e.:

- the files contained in the folder "FS" for MMC/SD card, NAND or NOR interfaces.
- the files contained in the folder "USB" for USB HID interface.
- the files contained in the folders "FS", "OS" and "USBH" for USBH MSD interface.

The following screenshots illustrate the project structure of each interface (from left to right: MMC/SD card (or NAND or NOR), USB HID, USBH MSD):



Depending on the chosen toolchain, an emLoad shipment may also contain additional files that are required for terminal output. If so, for each interface these files are located in the folder "OS".

If the Crypto add-on is present, the folder "LIB" contains sources for firmware encryption. If the Signature add-on is present, the folder "CRYPTO" and "SECURE" contain sources for firmware signature verification.

### 1.2.4 Availability and FLASH devices

The software is completely written in ANSI-C and can therefore be used on virtually any CPU. The only requirements to port the BTL for a particular application are a "C"-module for accessing the peripherals of the microcontroller and a "C"-module containing the programming algorithm for the FLASH-memory chip(s). For latest information about supported devices, please visit our website. Ports for other microcontrollers can be made within short time.

### 1.2.5 Development environment (compiler)

An ANSI-compliant compiler complying with at least one of the following international standard is required:

- ISO/IEC/ANSI 9899:1990 (C90) with support for C++ style comments (//)
- ISO/IEC 9899:1999 (C99)
- ISO/IEC 14882:1998 (C++)

If your compiler has some limitations, let us know and we will inform you if these will be a problem when compiling the software. Any compiler for 16/32/64-bit CPUs or DSPs that we know of can be used; most 8-bit compilers can be used as well.



A C++ compiler is not required, but can be used. The application program can therefore also be programmed in C++ if desired.

## 1.2.6 Crypto Add-On

An add-on to encrypt the firmware and check the encryption during the update process is available to purchase. This add-on is activated by calling one API and can be easily activated with the shipped sample `Main.c` by activating the compilation switch `DECRYPT_FW` (see *Compile-time configuration switches* on page 30 for details).

The only difference in usage concerns the tool `PrepareFWPRO` which is used instead of `PrepareFW`. `PrepareFWPRO` has additional options to encrypt the firmware. See *PrepareFW / PrepareFWPRO* on page 22 for more details on these options.

## 1.2.7 Signature Add-On

An add-on to digitally sign the firmware and verify the signature during the update process is available to purchase. This add-on is activated by calling one API and can be easily activated with the shipped sample `Main.c` by activating the compilation switch `CHECK_SIGNATURE` (see *Compile-time configuration switches* on page 30 for details).

When the Signature add-on is used, some additional steps are required when creating the update image. For a description of this process, please refer to *Digitally sign an update* on page 27.

## 1.3 Quick start guide

The following steps describe how to use emLoad for the first time.

### 1.3.1 Loading emLoad into a target

- Open the start project that can be found in “\emLoadV4” .
- If not already done select the DEBUG configuration (the DEBUG configuration enables several checks that make sure that the configuration as setup by the user is valid and is recommended to be run at least once every time parameters are changed).
- Build the configuration and download it to the target.
- Run the program.

### 1.3.2 Preparing an emLoad firmware update

emLoad firmware updates have to be in a special format to be recognized as valid firmware update. For this to achieve the “PrepareFW” or “PrepareFWPRO” tool can be used. It can be found in the “\emLoadV4\Windows” folder. The PrepareFW[PRO] tool is able to generate firmware update files from the following source file types:

- \*.mot
- \*.hex
- \*.bin

To generate a firmware update file you will need to have a firmware in one of the supported formats. To generate a firmware update file use the PrepareFW[PRO] tool that can be found in “\emLoadV4\Windows\PrepareFW” or “\emLoadV4\Windows\PrepareFWPRO”. See *PrepareFW / PrepareFWPRO* on page 22 for syntax and parameters details.

In case your firmware memory layout is split into multiple ranges the parameter `-multi-range` needs to be used with the PrepareFW[PRO] tool as well!

### 1.3.3 Updating the firmware using emLoad for MMC/SD

- Store the generated firmware update file (typically Update.fw or Update.fwc when Crypto is present, if not configured otherwise) onto your MMC/SD card.
- Insert the MMC/SD card into the target.
- Power or power cycle into emLoad that updates and starts the firmware afterwards.

### 1.3.4 Updating the firmware using emLoad for UART

- Run the `UART_Updater.exe` that can be found in “\emLoadV4\Windows\UART\_Updater” with the `-update` and `-wait` options. See *UART Updater* on page 24 for more details.
- Connect the target powered down to your PC using a serial cable.
- Power on the target. The target will wait a limited time for a first command from the PC.
- emLoad updates and starts the firmware afterwards.

### 1.3.5 Updating the firmware using emLoad for USB HID

- Run the `USB_HID_Updater.exe` that can be found in “\emLoadV4\Windows\USB\_HID\_Updater” with the `-update` and `-wait` options. See *USB HID Updater* on page 25 for more details.
- Connect the target powered down to your PC using a USB cable.
- Power on the target. The target will enumerate with the PC.
- emLoad updates and starts the firmware afterwards.

### 1.3.6 Updating the firmware using emLoad for USBH MSD

- Store the generated firmware update file (typically Update.fw or Update.fwc when Crypto is present, if not configured otherwise) onto your USB stick.
- Insert the USB stick into the target.
- Power or power cycle into emLoad that updates and starts the firmware afterwards.

### 1.3.7 Updating the firmware using emLoad for NAND/NOR

- Store the generated firmware update file (typically Update.fw or Update.fwc when Crypto is present, if not configured otherwise) onto your NAND/NOR.
- Power cycle into emLoad that updates and starts the firmware afterwards.
- After the update, the firmware Update.fw is by default removed from the NAND to avoid having an update at each start.

### 1.3.8 Configuration of firmware passwords and other firmware identifiers

A firmware password and other firmware identifiers can be set directly in the `main()` routine that can be found in the file `"\emFileV4\Application\Main.c"`.

### 1.3.9 Configuration of the firmware area

The firmware area used by emLoad to store the firmware can be configured providing a `BTL_FLASH_INFO` structure using the routine `BTL_ConfigFlashInfo()`. The configuration of the flash ranges can typically be found in the `BTL_Config_<UpdateInterface>_<CPU>_<BoardManufacturer>_<Board>.c` file that is located in `"\emFileV4\Setup"`.



# Chapter 2

## emLoad tools

---

This chapter gives an introduction to the PC-programs shipped with emLoad.

All tools are shipped as pre-compiled binaries and in source with a Visual Studio project. The tools typically do not require any installation process, DLLs or runtime libraries.

## 2.1 PrepareFW / PrepareFWPRO

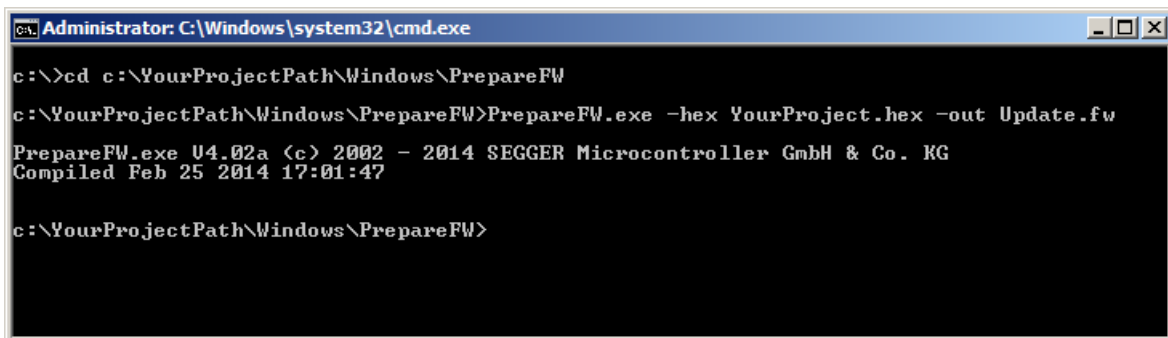
emLoad firmware updates need to adhere a specific format to be considered valid. To ensure this format, SEGGER provides the "PrepareFW" or "PrepareFWPRO" tool, which is a 32-bit Windows application specifically designed to transform binary files, Intel hex files or Motorola files into emLoad-compatible firmware files. PrepareFW[PRO] is shipped as both, an executable that works out-of-the-box and also as a Microsoft Visual Studio project to recompile the executable when needed.

### 2.1.1 Running PrepareFW.exe / PrepareFWPRO.exe

PrepareFW[PRO] runs as a Windows command line application. As such, it requires the user to identify the input file's format and name via command line parameters. In addition, some optional configuration parameters can be specified via command line. These parameters include:

- Input file (mandatory) and start address (mandatory for binary input files).  
Example:  
-bin Input.bin 0x11000  
-hex Input.hex  
-mot Input.mot
- Desired name of the output file (optional).  
Example:  
-out Filename.fw
- Firmware identifier string (optional).  
Example:  
-firmwareid "FWUpdate"
- 32-bit firmware version (optional).  
Example:  
-version 0x0100
- Company identifier string (optional).  
Example:  
-company "My Company"
- Device identifier string (optional).  
Example:  
-device "My Device"

The following screenshot shows the successful conversion of a hex file into a valid firmware file:



```

Administrator: C:\Windows\system32\cmd.exe
c:\>cd c:\YourProjectPath\Windows\PrepareFW
c:\YourProjectPath\Windows\PrepareFW>PrepareFW.exe -hex YourProject.hex -out Update.fw
PrepareFW.exe U4.02a (c) 2002 - 2014 SEGGER Microcontroller GmbH & Co. KG
Compiled Feb 25 2014 17:01:47
c:\YourProjectPath\Windows\PrepareFW>

```

## 2.1.2 Additional options of PrepareFWPRO

PrepareFWPRO supports the same options as PrepareFW but additionally allows to encrypt the firmware. The extension of the firmware could then be `.fwc` instead of `.fw`. To do so, there are two possible options.

- Encryption based on a password.  
Example:  
-cryptopass "PASSWORD"
- Encryption based on a key (32 bytes) and IV (16 bytes).  
Example:  
-cryptokeyiv <32 hex. bytes> <16 hex. bytes>

## 2.2 UART Updater

By using an emLoad-compatible firmware file, this program updates an emLoad target via UART interface.

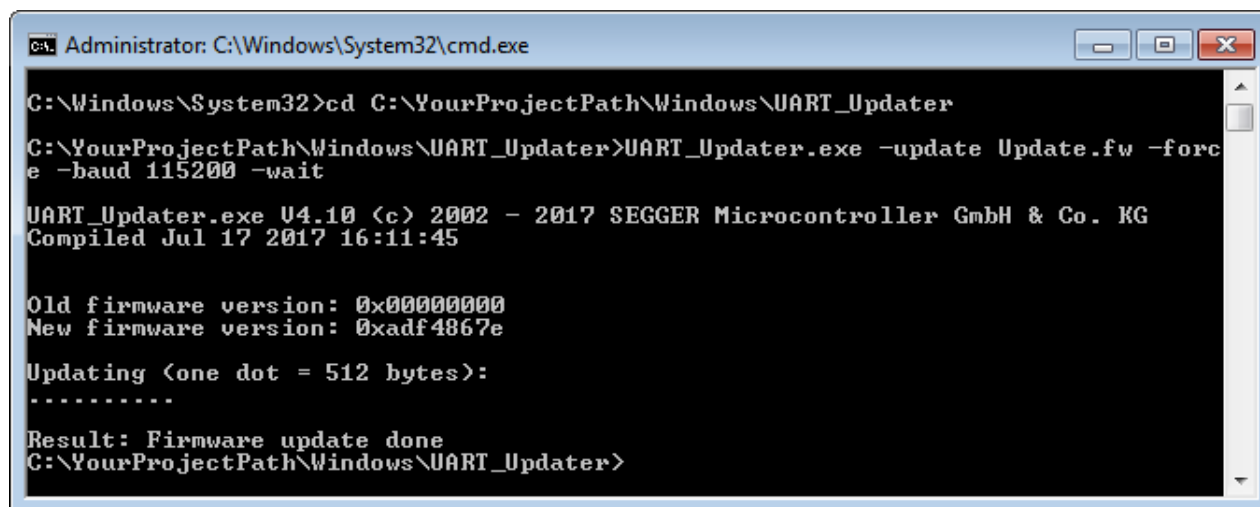
### 2.2.1 Running UART\_Updater.exe

UART\_Updater runs as a Windows command line application. Note that it is sometimes necessary to run it with administrator privileges. It requires the user to identify the firmware update file via command line parameter and whether an update should be forced (regardless of the firmware version in the target as, per default, update will not take place if the given firmware version is smaller than the one already flashed). Alternatively, UART\_Updater can be used to identify the firmware in a target. By default the UART\_Updater uses COM1 with a 9600 baud and an 8-N-1 configuration. The serial parameters can be configured using the optional commands like `-baud 115200`. For more options please have a look at the online help of the program.

Possible parameters are:

- Update file and (optional) update policy.  
Example:
  - update Update.fw
  - update Update.fw -force
- Identify firmware.  
Example:
  - identify

The following screenshot shows the successful update using the UART\_Updater:



```
C:\Windows\System32>cd C:\YourProjectPath\Windows\UART_Updater
C:\YourProjectPath\Windows\UART_Updater>UART_Updater.exe -update Update.fw -force -baud 115200 -wait
UART_Updater.exe U4.10 (c) 2002 - 2017 SEGGER Microcontroller GmbH & Co. KG
Compiled Jul 17 2017 16:11:45

Old firmware version: 0x00000000
New firmware version: 0xadf4867e

Updating (one dot = 512 bytes):
.....

Result: Firmware update done
C:\YourProjectPath\Windows\UART_Updater>
```



## 2.3 USB HID Updater

By using an emLoad-compatible firmware file, this program updates an emLoad target via USB HID interface.

### 2.3.1 Running USB\_HID\_Updater.exe

USB\_HID\_Updater runs as a Windows command line application. Note that it is sometimes necessary to run it with administrator privileges. It requires the user to identify the firmware update file via command line parameter and whether an update should be forced (regardless of the firmware version in the target as, per default, update will not take place if the given firmware version is smaller than the one already flashed). Alternatively, USB\_HID\_Updater can be used to identify the firmware in a target. Possible parameters are:

- Update file and (optional) update policy.

Example:

```
-update Update.fw
-update Update.fw -force
```

- Identify firmware.

Example:

```
-identify
```

The following screenshot shows the successful update using the USB\_HID\_Updater:

```
Administrator: Command Prompt
C:\Windows\system32>cd c:\YourProjectPath\Windows\USB_HID_Updater
c:\YourProjectPath\Windows\USB_HID_Updater>USB_HID_Updater.exe -update Update.fw
c -force

USB_HID_Updater.exe U4.04 (c) 2002 - 2013 SEGGER Microcontroller GmbH & Co. KG
Compiled Apr 13 2016 12:01:38

USB HID information about connected target:
Device 0:
  Productname: emLoad USB HID
  UID        : 0x0765
  PID        : 0x1114
  ReportSize:
    Input    : 64 bytes
    Output   : 64 bytes

Old firmware version: 0x0000
New firmware version: 0xadf4867e

Updating (one dot = 512 bytes):
.....

Result: Firmware update done
c:\YourProjectPath\Windows\USB_HID_Updater>
```

## 2.4 BTL Image Creator

emLoad stores additional information about the current firmware image in the target in a firmware information area. Simply programming the content of a firmware onto the target will therefore not result in the BTL to detect a valid firmware that can be booted. The BTL Image Creator tool is able to generate a programmable image that can be used to program a first default firmware during production. It is even possible to include the BTL itself to program BTL and a first firmware in a single step during production.

The BTL Image Creator tool, which is a 32-bit Windows application specifically designed to transform binary files, Intel hex files or Motorola files into an emLoad compatible format. `BTL_ImageCreator` is shipped as both, an executable that works out-of-the-box and also as a Microsoft Visual Studio project to recompile the executable when needed.

### 2.4.1 Running BTL\_ImageCreator.exe

`BTL_ImageCreator` runs as a Windows command line application. As such, it requires the user to identify the input file's format and name via command line parameters. In addition, some optional configuration parameters can be specified via command line. These parameters include:

- Firmware file (mandatory) and start address (mandatory for binary input files). The type (binary, Intel Hex, Motorola is detected by the file extension).

Example:

```
FW.bin 0x08008000
```

```
FW.hex
```

```
FW.mot
```

- Firmware area information (mandatory) including start address and size of the area. The example shows a configuration for a target with 1MByte flash and an emLoad that uses 32 kBytes of flash.

Example:

```
0x08008000 0xF8000
```

- Desired output name and format (optional, default Motorola).

Example:

```
-bin Image.bin
```

```
-hex Image.hex
```

```
-mot Image.mot
```

- 32-bit firmware version (optional).

Example:

```
-version 0x0100
```

The following screenshot shows the online help of the `BTL_ImageCreator`:

```
Administrator: C:\Windows\System32\cmd.exe

Usage:
  BTL_ImageCreator.exe <Input> [BinAddr] <FwAreaAddr> <FwAreaSize> [Options]

Supported input files are *.mot/*.hex/*.bin
The format is detected by the file extension.

Parameter examples:
  Example1: FW.mot 0x08008000 0xF8000
  Example2: FW.mot 0x08008000 0xF8000 -hex Image.hex
  Example3: FW.bin 0x08008000 0x08008000 0xF8000 -version 0x0100

Options:
  -btl      : Specify filename of bootloader to include in output image.
              Example1: -btl BTL.mot
              Example2: -btl BTL.bin 0x80000000
  -mot      : Specify filename of output in Motorola format.
              Example: -mot Image.mot
  -hex      : Specify filename of output in Intel Hex format.
              Example: -hex Image.hex
  -bin      : Specify filename of output in binary format.
              Example: -bin Image.bin
  -version  : 32-bit firmware version Hex value.
              Example: -version 0x0100
```

## 2.5 Digitally sign an update

emLoad together with the Signature add-on is able to digitally sign your firmware update. This allows you to prevent unauthorized and potentially malicious firmware to be loaded onto your device.

The Signature add-on is available for RSA 2048 bit or ECDSA P-256 up to P-521 encryption. RSA and ECDSA Signature add-ons are available as separate packages.

The update process and data validity is ensured by using a RSA/ECDSA private/public key pair. The public key gets stored on the target and is able to verify a firmware update that was previously signed with the private key of the same key pair.

For this the emLoad Signature add-on includes the tools that come with SEGGER's emSecure package that allow to easily sign the firmware update on the PC side.

### 2.5.1 Using the batch script to sign a firmware update

A batch script `CreateSignedUpdate.bat` can be found in the folder `\Windows\emSecure_ECDSA\` or `\Windows\emSecure_RSA\`. The tools that are required to digitally sign a file can be found here as well.

Using the prepared batch script is easy. Just place a file `Firmware.hex` into this folder and execute the batch script `CreateSignedUpdate.bat`. If your file is a `.mot` file, please change the variable `_FIRMWARE_EXT_` in the script from "hex" to "mot".

The batch should output the following files:

File	Content
<code>emSecure.prv</code>	Private key of a digital key pair. To be used on the PC to sign an update. Created based on random data if not already present.
<code>emSecure.pub</code>	Public key of a digital key pair. To be used on the target to verify an update. Created based on random data if not already present.
<code>Update.fw</code>	The firmware update digitally signed with the private key in <code>emSecure.prv</code> . This file can now be used to update the firmware on the target running emLoad with Signature add-on.
<code>emLoad_PublicKey.C</code>	A code representation of the public key data, ready to be included in the <code>Main.c</code> sample file on the target.

### 2.5.2 Signing a firmware update manually

With different requirements when preparing the update using `PrepareFW`, the batch script `CreateSignedUpdate.bat` might not fully fit your needs. The following steps explain what the batch script does. Based on this information you should be able to vary on different use cases like `Signature+Encryption` or `Signature+Multirange` input on `PrepareFW`.

#### 2.5.2.1 ECDSA: Creating a private/public key pair using P-256

```
KeyGenECDSA.exe -p256
```

`KeyGenECDSA.exe` generates a private/public key pair based on random data of your system. By default this generates the files `emSecure.prv` (your private key) and `emSecure.pub` (your public key).

### 2.5.2.2 RSA: Creating a 2048 bit private/public key pair

```
KeyGenRSA.exe -l 2048
```

`KeyGenRSA.exe` generates a private/public key pair based on random data of your system. By default this generates the files `emSecure.prv` (your private key) and `emSecure.pub` (your public key).

### 2.5.2.3 Preparing the data to be signed

```
PrepareFW.exe -hex Firmware.hex -out ToSign.fw -noheader
```

The `-noheader` option of `PrepareFW` instructs the tool to generate a firmware update file without the main header that comes before the actual data. The reason for this is that the signature that we will generate is stored within the header itself.

The data part that the signature will include is written to the file `ToSign.fw`.

### 2.5.2.4 Signing the data

```
SignECDSA.exe/SignRSA.exe ToSign.fw
```

`SignECDSA.exe/SignRSA.exe` by default uses the key from the file `emSecure.prv` to calculate the signature of the given file `ToSign.fw`. The tool outputs a signature file in `emSecure` format of the same name with `.sig` extension. In this example a file `ToSign.fw.sig` is generated.

### 2.5.2.5 Generating the firmware update file

```
PrepareFW.exe -hex Firmware.hex -out Firmware.fw -sigfile ToSign.fw.sig
```

This step is similar to generating a firmware update file without Signature add-on. The only difference is that by the option `-sigfile` the signature of the signed data will be included into the header of the firmware update.

### 2.5.2.6 Extracting the private key content for use with emLoad

```
PrintKeyECDSA.exe/PrintKeyRSA.exe emSecure.pub > emLoad_PublicKey.c
```

`PrintKeyECDSA.exe/PrintKeyRSA.exe` can be used to extract the public key data in a form suitable for inclusion with `emLoad`. The sample code shown above outputs the generated content into the file `emLoad_PublicKey.c`. The content from this file can then be included into the `Main.c` that is shipped as an example together with `emLoad`.

This step is only required once or if you change the private/public key pair.

# Chapter 3

## Configuration

---

emLoad can be used without changing any of the compile-time flags. All compile-time configuration flags are preconfigured with valid values, which match the requirements of most applications. The default configuration of emLoad can be changed via compile-time flags which can be added to `BTL_Conf.h`. `BTL_Conf.h` is the main configuration file for emLoad.

## 3.1 Compile-time configuration

The following types of configuration macros exist:

### Binary switches "B"

Switches can have a value of either 0 or 1, for deactivated and activated respectively. Actually, anything other than 0 works, but 1 makes it easier to read a configuration file. These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macros.

### Numerical values "N"

Numerical values are used somewhere in the code in place of a numerical constant. A typical example is the configuration of the sector size of a storage medium.

### Text replacements "T"

Text replacements are used somewhere in the code in place of a constant string. A typical example is the configuration of filename.

### Function replacements "F"

Macros can basically be treated like regular functions although certain limitations apply, as a macro is still put into the code as simple text replacement. Function replacements are mainly used to add specific functionality to a module which is highly hardware-dependent. This type of macro is always declared using brackets (and optional parameters).

### 3.1.1 Compile-time configuration switches

Type	Symbolic name	Default	Description
Common bootloader definitions			
B	<code>BTL_ALLOW_CHANGE_UPDATE_POLICY</code>	0	Defines if changes to the update policy are allowed.
T	<code>BTL_COMPANY_NAME</code>	"My Company"	Used to store the company name for a firmware.
T	<code>BTL_DEVICE_NAME</code>	"My Device"	Used to store the device name for a firmware.
N	<code>BTL_INIT_TIMEOUT</code>	500	Used to store the time [in ms] the BTL waits for a firmware update before starting a valid firmware.
N	<code>BTL_FIRMWARE_INFO_SIZE</code>	16	Used to define the size of the firmware info area.
T	<code>BTL_FW_PASS</code>	""	Used to store a password for a firmware update.
T	<code>BTL_FW_STRING</code>	"SEG-GER-FWUpdate"	Used to store the firmware string of a firmware update.
T	<code>BTL_UPDATE_FILE_NAME</code>	"Update.fw" or "Update.fwc"	Used to store the file name of a firmware update. <code>fwc</code> extension when Crypto is present.

Type	Symbolic name	Default	Description
N	<a href="#">BTL_UPDATE_POLICY</a>	0	Used to store the current update policy.
B	<a href="#">CHECK_SIGNATURE</a>	0	Enables/disables the Signature check when available.
B	<a href="#">DECRYPT_FW</a>	0	If the Crypto option is present, defines if crypto is used or not.
T	<a href="#">DECRYPT_PW</a>	"SEGGGER"	If the Crypto option is present, password to use for key and IV generation if not set by defines.
T	<a href="#">BTL_CRYPTO_KEY</a> and <a href="#">BTL_CRYPTO_IV</a>	not defined	If the Crypto option is present, key and IV to use.
Debug macros			
N	<a href="#">BTL_DEBUG</a>	0	Macro to define the debug level of the emLoad build. Refer to <i>Debug level</i> on page 32 for a description of debug levels.
Replacement macros			
F	<a href="#">BTL_ADDR2PTR</a>	(void <a href="#">BTL_HUGE</a> *) (Addr)	Macro to convert an address to a pointer to that address.
F	<a href="#">BTL_MEMCMP</a>	memcmp (C-routine in standard C-library)	Macro to define an optimized memcmp routine. An optimized memcmp routine is typically implemented in assembly language.
F	<a href="#">BTL_MEMCPY</a>	memcpy (C-routine in standard C-library)	Macro to define an optimized memcpy routine. An optimized memcpy routine is typically implemented in assembly language.
F	<a href="#">BTL_MEMSET</a>	memset (C-routine in standard C-library)	Macro to define an optimized memset routine. An optimized memset routine is typically implemented in assembly language.
F	<a href="#">BTL_PTR2ADDR</a>	(long)Ptr	Macro to convert a pointer to an address to the numerical address it points at.
F	<a href="#">BTL_RAM_FUNC</a>		Macro to define a <a href="#">RAM_FUNC</a> implementation for various compilers.

## 3.2 Debug level

emLoad can be configured to display debug information at higher debug levels to locate a problem (Error) or potential problem. To display information, emLoad uses several logging routines (see *Message output* on page ). These routines can be blank, as they are not required for the functionality of emLoad. Typically, they are not present in release (production) builds at all, as classical production builds utilize lower debug levels.

If (BTL\_DEBUG = 0) : No checks are enabled.

If (BTL\_DEBUG = 1) : Warn and "Panic" checks are enabled.

If (BTL\_DEBUG >= 2) : Warn, log and "Panic" checks are enabled.



# Chapter 4

## Core functions

---

In this chapter, you will find a description of each emLoad core function.

## 4.1 emLoad BTL API

The table below lists the available API functions within their respective categories.

Function	Description
Core functions	
<code>BTL_AddStateChangeHook()</code>	This function adds a hook function to the <code>BTL_HOOK_ON_STATE_CHANGE</code> list.
<code>BTL_AssignMemory()</code>	Assigns a memory pool to the BTL.
<code>BTL_Start()</code>	Initializes the CPU, LEDs and initializes the configuration.
<code>BTL_StartFW()</code>	Allows starting the firmware from several states causing the BTL to close down its operations in case a valid firmware is found in flash.
<code>BTL_Exec()</code>	Check if one ms has been passed and maintains periodic tasks such as setting LEDs and calling a registered tick hook if any.
Security add-on API	
<code>BTL_AddCrypto()</code>	Hooks in the decryption module that decrypts a firmware and converts a password string to an AES256 usable key and sets it for later usage by the crypto module.
<code>BTL_AddSignature_ECDSA()</code>	Hooks in the signature module that is able to verify a firmware update to be of trusted source.
<code>BTL_AddSignature_RSA()</code>	Hooks in the signature module that is able to verify a firmware update to be of trusted source.
BTL configuration functions	
<code>BTL_ConfigFlashInfo()</code>	Sets the internal pointer to a structure of type <code>BTL_FLASH_INFO</code> .
<code>BTL_ConfigReadMemSize()</code>	Overwrites the default configuration for the size of the buffer used for reading from the update interface.
<code>BTL_ConfigRetryUpdateOnFail()</code>	This function configures the behavior in case of a failed update.
<code>BTL_ConfigUpdatePolicy()</code>	Configures the update policy to use.
<code>BTL_GetCpuClock()</code>	Retrieves the stored CPU clock value.
<code>BTL_GetStatus()</code>	Retrieves the information if the firmware is valid or not.
<code>BTL_SetCpuClock()</code>	Stores a CPU clock value for all modules to retrieve.
<code>BTL_SetInitTimeout()</code>	Sets initial timeout [ms] to wait before update interface is ready.
<code>BTL_SetTickHook()</code>	This function registers a callback that will be tried to be executed every tick (typically 1ms).
<code>BTL_ConfigSkipBootCrcCheck()</code>	This function configures if the CRC check on boot is executed or skipped.

Function	Description
<code>BTL_SetFlashCrcProgressHook()</code>	This function registers a callback that will be executed after each CRC chunk of the flash CRC calculation.
<code>BTL_ConfigInterfaceReadyDelay()</code>	Configures the length of the delay that is waited after interface reports that it is ready.
<code>BTL_DisableUpdateInitTimeout()</code>	This function disables the initial timeout configured for the interface to wait for a first data exchange with an update client.
<code>BTL_GetUpdateInitTimeoutTimestamp()</code>	This function returns the timestamp of the timeout when reading from the update interface for the first time which typically is the firmware update header.
<code>BTL_Delay()</code>	Wait for the given time (parameter in ms).
<b>Firmware configuration functions</b>	
<code>BTL_GetCompanyName()</code>	Retrieves the pointer for the company name string.
<code>BTL_GetDeviceName()</code>	Retrieves the pointer for the device name string.
<code>BTL_GetFwString()</code>	Retrieves the pointer for the set firmware string.
<code>BTL_GetFwVersion()</code>	Retrieves the firmware version of the current firmware in flash.
<code>BTL_GetUpdateFileName()</code>	Retrieves the current pointer to the update file name string.
<code>BTL_SetCompanyName()</code>	Sets the pointer for the company name string to a given resource.
<code>BTL_SetDeviceName()</code>	Sets the pointer for the device name string to a given resource.
<code>BTL_SetFwPassword()</code>	Sets the pointer for the firmware password string to a given resource.
<code>BTL_SetFwStartAddr()</code>	Sets firmware start addr.
<code>BTL_SetFwString()</code>	Sets the pointer for the firmware string to a given resource.
<code>BTL_SetUpdateFileName()</code>	Sets the pointer for the update file name string to a given resource.

### 4.1.1 Detailed core functions

### 4.1.1.1 BTL\_AddStateChangeHook()

#### Description

This function adds a hook function to the BTL\_HOOK\_ON\_STATE\_CHANGE list.

#### Prototype

```
void BTL_AddStateChangeHook(BTL_HOOK_ON_STATE_CHANGE * pHook,  
                             void (*pf)(U32 State ));
```

#### Parameters

Parameter	Description
<code>pHook</code>	Pointer to a structure of type BTL_HOOK_ON_STATE_CHANGE
<code>pf</code>	Pointer to a callback to be notified with the new state

#### Example

```
//  
// Static declaration.  
//  
static BTL_HOOK_ON_STATE_CHANGE _OnStateChangeHook;  
static void _OnStateChange(U32 State) {  
    //  
    // Notify about state change.  
    //  
}  
//  
// Code running in main task.  
//  
BTL_AddStateChangeHook(&_OnStateChangeHook, _OnStateChange);
```

### 4.1.1.2 BTL\_AssignMemory()

#### Description

Assigns a memory pool to the BTL. From this memory pool buffers will be allocated for reading from the update interface and if necessary for caching of writeing/reading to/from flash areas.

#### Prototype

```
void BTL_AssignMemory(void * pMem,
                     U32   NumBytes);
```

#### Parameters

Parameter	Description
<a href="#">pMem</a>	Pointer to a memory pool
<a href="#">NumBytes</a>	Size of the memory pool

#### Example

```
//
// Static declaration.
//
static U32 _aPoolBTL[256];
//
// Code running during initialization.
//
BTL_AssignMemory(&_aPoolBTL, sizeof(_aPoolBTL));
```

### 4.1.1.3 BTL\_Start()

#### Description

Initializes the CPU, LEDs and initializes the configuration. Afterwards the update process is tried to be started.

#### Prototype

```
void BTL_Start(void);
```

#### Example

```
//  
// Static declarations.  
//  
static OS_STACKPTR int _BTL_MainStack[512];  
static OS_TASK      _BTL_MainTCB;  
//  
// Code running in main().  
//  
OS_CREATETASK(&_BTL_MainTCB, "BTL Task", BTL_Start, 100, _BTL_MainStack);
```

#### 4.1.1.4 BTL\_StartFW()

##### Description

Allows starting the firmware from several states causing the BTL to close down its operations in case a valid firmware is found in flash. In case this function returns this means there is no valid firmware in flash.

##### Prototype

```
void BTL_StartFW(void);
```

##### Additional information

This function is prohibited to be called from the following states (see *emLoad status codes* on page 75):

- BTL\_STATUS\_INIT
- BTL\_STATUS\_STARTED
- BTL\_STATUS\_CONFIG\_ERROR

### 4.1.1.5 BTL\_Exec()

#### Description

Check if one ms has been passed and maintains periodic tasks such as setting LEDs and calling a registered tick hook if any. It is intended to be called at least every ms and might be called from various places.

#### Prototype

```
int BTL_Exec(void);
```

#### Return value

- 0 No ms has been passed
- 1 At least one ms has been passed



### 4.1.1.6 BTL\_AddCrypto()

#### Description

Hooks in the decryption module that decrypts a firmware and converts a password string to an AES256 usable key and sets it for later usage by the crypto module.

#### Prototype

```
void BTL_AddCrypto(const U8      * pKey,
                  const U8      * pIV,
                  unsigned   KeyLen,
                  unsigned   IVLen);
```

#### Parameters

Parameter	Description
<a href="#">pKey</a>	Pointer to 32-byte long key to use.
<a href="#">pIV</a>	Pointer to 16-byte long Initialization Vector that is used as first IV for each new filestream.
<a href="#">KeyLen</a>	Length of key to use. Used for parameter checking.
<a href="#">IVLen</a>	Length of IV to use. Used for parameter checking.

### 4.1.1.7 BTL\_AddSignature\_ECDSA()

#### Description

Hooks in the signature module that is able to verify a firmware update to be of trusted source.

#### Prototype

```
void BTL_AddSignature_ECDSA(void * pPublicKey);
```

#### Parameters

Parameter	Description
<code>pPublicKey</code>	Pointer to ECDSA public key.

### 4.1.1.8 BTL\_AddSignature\_RSA()

#### Description

Hooks in the signature module that is able to verify a firmware update to be of trusted source. RSA with 2048 bit key is expected.

#### Prototype

```
void BTL_AddSignature_RSA(void * pPublicKey,
                          U8   * pSalt,
                          int   NumBytesSalt);
```

#### Parameters

Parameter	Description
<code>pPublicKey</code>	Pointer to RSA public key.
<code>pSalt</code>	Pointer to salt. Can be NULL.
<code>NumBytesSalt</code>	Size of salt. Can be 0.

### 4.1.1.9 BTL\_ConfigFlashInfo()

#### Description

Sets the internal pointer to a structure of type `BTL_FLASH_INFO`.

#### Prototype

```
void BTL_ConfigFlashInfo(BTL_FLASH_INFO * p,
                        unsigned NumEntries);
```

#### Parameters

Parameter	Description
<code>p</code>	Pointer to a structure of type <code>BTL_FLASH_INFO</code> .
<code>NumEntries</code>	Number of Flash info entries.

#### Example

```
//
// Static declaration.
//
static BTL_FLASH_INFO _FlashInfo[] = {
    (FLASH_START_ADDR + BTL_SIZE),
    ((FLASH_END_ADDR - FLASH_START_ADDR + 1) - BTL_SIZE),
    &BTL_FLASH_Driver_ST_STM32F2xx,
};
//
// Code running during initialization.
//
BTL_ConfigFlashInfo(_FlashInfo, SEGGER_COUNTOF(_FlashInfo));
```

#### Additional information

The firmware area used by emLoad to store the firmware can be configured providing a `BTL_FLASH_INFO` structure using the routine `BTL_ConfigFlashInfo()`. The configuration of the flash ranges can typically be found in the folder "`\emFileV4\Setup`" in the file `BTL_Config_<CPU manufacturer>_<CPU>.c`.

### 4.1.1.10 BTL\_ConfigReadMemSize()

#### Description

Overwrites the default configuration for the size of the buffer used for reading from the update interface.

#### Prototype

```
void BTL_ConfigReadMemSize(U32 NumBytes);
```

#### Parameters

Parameter	Description
<code>NumBytes</code>	Number of bytes to allocate for reading from the update interface.

#### Additional information

Default buffer size is 0x200.

### 4.1.1.11 BTL\_ConfigRetryUpdateOnFail()

#### Description

This function configures the behavior in case of a failed update.

#### Prototype

```
void BTL_ConfigRetryUpdateOnFail(U8 Retry);
```

#### Parameters

Parameter	Description
Retry	OnOff switch to retry or not retry updating in case an update attempt failed. Default is do not retry.

#### Additional information

Default behavior is to not retry.

### 4.1.1.12 BTL\_ConfigUpdatePolicy()

#### Description

Configures the update policy to use. By default the firmware is updated always.

#### Prototype

```
void BTL_ConfigUpdatePolicy(U8 Policy);
```

#### Parameters

Parameter	Description
Policy	Policy to use when updating: BTL_POLICY_UPDATE_ALWAYS BTL_POLICY_UPDATE_NEWER BTL_POLICY_UPDATE_SAME_OR_NEWER

#### Additional information

Default policy is BTL\_POLICY\_UPDATE\_ALWAYS. Changes to the update policy may be prohibited by defining BTL\_ALLOW\_CHANGE\_UPDATE\_POLICY accordingly (see *Compile-time configuration switches* on page 30). By default, BTL\_ALLOW\_CHANGE\_UPDATE\_POLICY is defined as 0, so the policy can not be changed.

### 4.1.1.13 BTL\_GetCpuClock()

#### Description

Retrieves the stored CPU clock value.

#### Prototype

```
U32 BTL_GetCpuClock(void);
```

#### Return value

Previously stored CPU clock in Hz



#### 4.1.1.14 BTL\_GetStatus()

##### Description

Retrieves the information if the firmware is valid or not.

##### Prototype

```
U32 BTL_GetStatus(void);
```

##### Return value

Current status of the BTL

### 4.1.1.15 BTL\_SetCpuClock()

#### Description

Stores a CPU clock value for all modules to retrieve.

#### Prototype

```
void BTL_SetCpuClock(U32 Hz);
```

#### Parameters

Parameter	Description
Hz	CPU clock in Hz

### 4.1.1.16 BTL\_SetInitTimeout()

#### Description

Sets initial timeout [[ms](#)] to wait before update interface is ready. Afterwards the firmware present (if any) will be started.

#### Prototype

```
void BTL_SetInitTimeout(unsigned ms);
```

#### Parameters

Parameter	Description
<a href="#">ms</a>	Timeout in <a href="#">ms</a>

#### Additional information

Default value is 500 [ms](#).

### 4.1.1.17 BTL\_SetTickHook()

#### Description

This function registers a callback that will be tried to be executed every tick (typically 1ms). Due to excessive time consumed by interface or flash routines the tick is not accurate.

#### Prototype

```
void BTL_SetTickHook(BTL_TICK_HOOK_CB * pf);
```

#### Parameters

Parameter	Description
<code>pf</code>	Pointer to a callback that will be tried to be called every tick (typically 1ms)

#### Example

```
//  
// Static declarations.  
//  
static U32 _Time;  
static void _OnTick(void) {  
    _Time++;  
}  
//  
// Code running in main task.  
//  
BTL_SetTickHook(_OnTick);
```

### 4.1.1.18 BTL\_ConfigSkipBootCrcCheck()

#### Description

This function configures if the CRC check on boot is executed or skipped.

#### Prototype

```
void BTL_ConfigSkipBootCrcCheck(U8 Skip);
```

#### Parameters

Parameter	Description
<code>Skip</code>	OnOff switch to skip or execute the CRC check of the whole firmware area on boot. Default is to NOT skip CRC check on boot.

### 4.1.1.19 BTL\_SetFlashCrcProgressHook()

#### Description

This function registers a callback that will be executed after each CRC chunk of the flash CRC calculation.

#### Prototype

```
void BTL_SetFlashCrcProgressHook(BTL_FLASH_CRC_PROGRESS_HOOK_CB * pf);
```

#### Parameters

Parameter	Description
<code>pf</code>	Pointer to a callback that will receive the current progress of the CRC calculation.

### 4.1.1.20 BTL\_ConfigInterfaceReadyDelay()

#### Description

Configures the length of the delay that is waited after interface reports that it is ready.

#### Prototype

```
void BTL_ConfigInterfaceReadyDelay(U32 ms);
```

#### Parameters

Parameter	Description
<a href="#">ms</a>	Time to wait after interface reports ready before communicating.

### 4.1.1.21 BTL\_DisableUpdateInitTimeout()

#### Description

This function disables the initial timeout configured for the interface to wait for a first data exchange with an update client. For more information please refer to `BTL_GetUpdateInitTimeoutTimestamp()`.

#### Prototype

```
void BTL_DisableUpdateInitTimeout(void);
```



### 4.1.1.22 BTL\_GetUpdateInitTimeoutTimestamp()

#### Description

This function returns the timestamp of the timeout when reading from the update interface for the first time which typically is the firmware update header. The interface read routine shall be non-blocking and periodically calling `BTL_Exec()` to maintain internal timers and indicators. This is necessary for interfaces such as USB as the interface reports ready as soon as the target has enumerated with a PC. If no update client is running on the PC the plan is to come back from the read function periodically to maintain indicators and to finally start the firmware if no update shall be done.

#### Prototype

```
I32 BTL_GetUpdateInitTimeoutTimestamp(void);
```

#### Return value

I32 timestamp

### 4.1.1.23 BTL\_Delay()

#### Description

Wait for the given time (parameter in [ms](#)).

#### Prototype

```
void BTL_Delay(unsigned ms);
```

#### Parameters

Parameter	Description
<a href="#">ms</a>	Delay in <a href="#">ms</a>

#### 4.1.1.24 BTL\_GetCompanyName()

##### Description

Retrieves the pointer for the company name string.

##### Prototype

```
char *BTL_GetCompanyName(void);
```

##### Return value

Pointer to the company name string

#### 4.1.1.25 BTL\_GetDeviceName()

##### Description

Retrieves the pointer for the device name string.

##### Prototype

```
char *BTL_GetDeviceName(void);
```

##### Return value

Pointer to the device name string

#### 4.1.1.26 BTL\_GetFwString()

##### Description

Retrieves the pointer for the set firmware string.

##### Prototype

```
char *BTL_GetFwString(void);
```

##### Return value

Pointer to the firmware identifier string

### 4.1.1.27 BTL\_GetFwVersion()

#### Description

Retrieves the firmware version of the current firmware in flash.

#### Prototype

```
U32 BTL_GetFwVersion(void);
```

#### Return value

U32 firmware version as provided by last update. The value will be simply stored for reference during update and has to be interpreted by the user.

#### 4.1.1.28 BTL\_GetUpdateFileName()

##### Description

Retrieves the current pointer to the update file name string.

##### Prototype

```
char *BTL_GetUpdateFileName(void);
```

##### Return value

Pointer to the string containing the name of the firmware update filename

### 4.1.1.29 BTL\_SetCompanyName()

#### Description

Sets the pointer for the company name string to a given resource.

#### Prototype

```
void BTL_SetCompanyName(const char * s);
```

#### Parameters

Parameter	Description
s	Pointer to a string with the desired company name

#### Additional information

Default company name is "My Company". Company name strings may not be longer than 0x30 characters.



### 4.1.1.30 BTL\_SetDeviceName()

#### Description

Sets the pointer for the device name string to a given resource.

#### Prototype

```
void BTL_SetDeviceName(const char * s);
```

#### Parameters

Parameter	Description
s	Pointer to a string with the desired device name

#### Additional information

Default device name is "My Device". Device name strings may not be longer than 0x40 characters.

### 4.1.1.31 BTL\_SetFwPassword()

#### Description

Sets the pointer for the firmware password string to a given resource.

#### Prototype

```
void BTL_SetFwPassword(const char * s);
```

#### Parameters

Parameter	Description
s	Pointer to a string with the password

#### Additional information

Default firmware password is an empty string. Firmware passwords may not be longer than 0x10 characters.

### 4.1.1.32 BTL\_SetFwStartAddr()

#### Description

Sets firmware start addr.

#### Prototype

```
void BTL_SetFwStartAddr(U32 Addr);
```

#### Parameters

Parameter	Description
<a href="#">Addr</a>	<a href="#">Addr</a> of the entry point into the firmware

### 4.1.1.33 BTL\_SetFwString()

#### Description

Sets the pointer for the firmware string to a given resource.

#### Prototype

```
void BTL_SetFwString(const char * s);
```

#### Parameters

Parameter	Description
s	Pointer to a string with the desired firmware identifier string

#### Additional information

Default firmware string is "SEGGER-FWUpdate". Firmware strings may not be longer than 0x10 characters.

### 4.1.1.34 BTL\_SetUpdateFileName()

#### Description

Sets the pointer for the update file name string to a given resource. If not set the default one is used.

#### Prototype

```
void BTL_SetUpdateFileName(const char * s);
```

#### Parameters

Parameter	Description
s	Pointer to a string with the desired update filename

#### Additional information

Default firmware string is "Update.fw" (or "Update.fwc" when the Crypto add-on is present). Firmware strings may not be longer than 0x0C characters.

## 4.2 emLoad BTL structures

### 4.2.1 Structure BTL\_FLASH\_DRIVER

#### Description

This structure holds function pointers for the BTL flash driver.

#### Prototype

```
struct {
    int (*pfInit) (void);
    int (*pfRead) (U8 * pDest, U8 * pSrc , U32 NumBytes);
    int (*pfWrite) (U8 * pDest, U8 * pSrc , U32 NumBytes, char FlushCache);
    int (*pfExit) (void);
} BTL_FLASH_DRIVER;
```

Member	Description
<code>pfInit</code>	Pointer to a function that initializes the flash driver and allocates memory as needed.
<code>pfRead</code>	Pointer to a function that reads data from any address of the flash.
<code>pfWrite</code>	Pointer to a function that writes data into any section of the flash.
<code>pfExit</code>	Pointer to a function that de-initializes the flash driver.

## 4.2.2 Structure BTL\_FLASH\_INFO

### Description

This structure holds the firmware area configuration used by emLoad.

### Prototype

```
struct {
    U32                Addr;
    U32                NumBytes;
    const BTL_FLASH_DRIVER * pDriver;
} BTL_FLASH_INFO;
```

Member	Description
<a href="#">Addr</a>	Address of the beginning of the firmware area.
<a href="#">NumBytes</a>	Firmware data size in Bytes.
<a href="#">pDriver</a>	Pointer to a flash driver of type BTL_FLASH_DRIVER.

## 4.2.3 Structure BTL\_HOOK\_ON\_STATE\_CHANGE

### Description

This structure holds pointers to functions that are called from hook.

### Prototype

```
struct {  
    void (*pf) (U32 State);  
    struct BTL_HOOK_ON_STATE_CHANGE * pNext;  
} BTL_HOOK_ON_STATE_CHANGE;
```

Member	Description
<a href="#">pf</a>	Pointer to a function to be called from a hook.
<a href="#">pNext</a>	Pointer to the next BTL_HOOK_ON_STATE_CHANGE structure.



## 4.2.4 Structure BTL\_NET\_API

### Description

Abstraction layer for several routines that allows to switch between configurations and interfaces for several purposes.

### Prototype

```

struct {
    void (*pfInitCPU)          (void);
    void (*pfExitCPU)         (void);
    void (*pfInitIf)          (void);
    void (*pfExitIf)         (void);
    void (*pfInitIfHw)        (void);
    void (*pfExitIfHw)        (void);
    int  (*pfHasTick)         (void);
    int  (*pfIsInterfaceReady) (void);
    void (*pfStartApplication) (U32 StartAddr);
    void (*pfStartBTL)        (void);
    int  (*pfRead)            (void * pData, unsigned NumBytes);
    void (*pfDisableInterrupt) (void);
    void (*pfEnableInterrupt)  (void);
    int  (*pfExecIF)          (void);
    int  (*pfResetIF)         (void);
} BTL_NET_API;

```

Member	Description
<code>pfInitCPU</code>	Pointer to a function that initializes the CPU module (if necessary).
<code>pfExitCPU</code>	Pointer to a function that revokes the initializations that were done during CPU initialization.
<code>pfInitIf</code>	Pointer to a function that initializes the update interface.
<code>pfExitIf</code>	Pointer to a function that closes the update interface.
<code>pfInitIfHW</code>	Pointer to a function that initializes the hardware for the update interface.
<code>pfExitIfHW</code>	Pointer to a function that revokes the initializations that were done during hardware initialization.
<code>pfHasTick</code>	Pointer to a function that checks if one microsecond has been passed.
<code>pfIsInterfaceReady</code>	Pointer to a function that checks if an interface is ready to communicate and retrieve data.
<code>pfStartApplication</code>	Pointer to a function that is called to start the user application program.
<code>pfStartBTL</code>	Pointer to a function that initializes the basic hardware for status outputs and starts the BTL and OS (if necessary).
<code>pfRead</code>	Pointer to a function that reads de data.
<code>pfDisableInterrupt</code>	Pointer to a function that disables interrupts if an RTOS is present, otherwise disabling of interrupts not needed.
<code>pfEnableInterrupt</code>	Pointer to a function that enables interrupts.
<code>pfExecIf</code>	Pointer to a function that offers the update interface a chance to execute commands.
<code>pfResetIf</code>	Pointer to a function that offers a reset of the update interface to be able to re-read the update file. Only valid for file system based interfaces. If set, the update file will be read

Member	Description
	twice. Once for a simple verification that it is not corrupted and a second time for the actual update.

### Additional information

Function pointers in `BTL_NET_API` are not checked for validity and are therefore not allowed to be `NULL` pointers. Instead of using a `NULL` pointer, the function pointer has to point at a dummy routine.

## 4.3 emLoad status codes

The following table contains a list of emLoad status codes.

Symbolic name	Value	Description
<code>BTL_STATUS_INIT</code>	0	After booting, state after reset, not handled.
<code>BTL_STATUS_STARTED</code>	30	After booting, BTL running.
<code>BTL_STATUS_CONFIG_ERROR</code>	60	Error during config init.
<code>BTL_STATUS_TRY_UPDATE</code>	90	Trying to update firmware.
<code>BTL_STATUS_IF_PRE_FILE_OPEN</code>	120	For FS based interfaces this means <code>FS_Init()</code> has been called but the update file has not yet been opened. Indicates the last chance to change the update file name. Only used by FS based interfaces.
<code>BTL_STATUS_IF_READY</code>	150	Update interface initialized and ready for updating.
<code>BTL_STATUS_UPDATING_TEST</code>	157	For FS based interfaces a check over the update file before the actual update and therefore erasing the old firmware is simple as we can re-read the file. If the test update succeeds, the real update with writing flash follows immediately.
<code>BTL_STATUS_UPDATING</code>	160	Update in progress.
<code>BTL_STATUS_UPDATE_SUCCESS</code>	180	Update succeeded, new firmware will be started.
<code>BTL_STATUS_UPDATE_FAILED</code>	210	Update failed.
<code>BTL_STATUS_ABORT_UPDATE_START_FW</code>	240	No update found, abort updating and try to start old firmware.



# Chapter 5

## CPU specific functions

---

For some CPUs generic CPU modules are delivered that can be used in the function tables. In case the user does not wish to use the generic routines or it is not available for the CPU, the user will have to implement his own routines instead.

## 5.1 CPU specific API

The table below lists the available CPU specific API functions:

Function	Description
<code>BTL_CPU_StartApplication()</code>	Starts the application by configuring the vector table offset register to point to the firmware and loading stackpointer and PC with appropriate firmware values.

### 5.1.1 CPU specific functions detailed

#### 5.1.1.1 BTL\_CPU\_StartApplication()

##### Description

Starts the application by configuring the vector table offset register to point to the firmware and loading stackpointer and PC with appropriate firmware values.

##### Prototype

```
void BTL_CPU_StartApplication ( U32 BaseAddr );
```

##### Parameters

Parameter	Description
<code>BaseAddr</code>	<b>In</b> Base address of the firmware.

##### Additional information

This routine is often written in assembler.

# Chapter 6

## Updating via MMC/SD card

---

This chapter provides descriptions for using emLoad for MMC/SD card.

## 6.1 Update Procedure with MMC/SD

### 6.1.1 Loading emLoad into a target

Open the start project that can be found in the shipment folder “\emLoadV4” and build the DEBUG configuration, download it to the target and run the program. We recommend to run the debug configuration at least once everytime a parameter was changed, as it includes several checks to make sure the user configuration is actually valid.

### 6.1.2 Preparing an emLoad firmware update

For the MMC/SD card interface, the “PrepareFW[PRO]” tool has to be used for preparation of the firmware. See *PrepareFW / PrepareFWPRO* on page 22 for more information on how to use this tool.

### 6.1.3 Update the firmware using emLoad

Store the generated firmware update file (typically Update.fw unless configured otherwise) onto your MMC/SD card. Insert the MMC/SD card into the target. emLoad is now updating and starting the firmware afterwards.



## 6.2 MMC/SD interface API

The table below lists the available API functions:

Function	Description
<code>BTL_IF_FS_MMC_SD_Exit()</code>	This routine is a generic routine for de-initialization of the interface.
<code>BTL_IF_FS_MMC_SD_Init()</code>	This routine is a generic routine for initialization of the interface.
<code>BTL_IF_FS_MMC_SD_IsInterfaceReady()</code>	This routine is a generic routine for checking the current status of the interface.
<code>BTL_IF_FS_MMC_SD_Read()</code>	This routine is a generic routine for reading data from the interface.
<code>BTL_IF_FS_MMC_SD_Reset()</code>	This routine is a generic routine for resetting the interface for re-reading the update.

## 6.2.1 BTL\_IF\_FS\_MMC\_SD\_Exit()

### Description

This routine is a generic routine for de-initialization of the interface. It calls further routines as needed for hardware de-initialization for this interface.

### Prototype

```
void BTL_IF_FS_MMC_SD_Exit(void);
```

## 6.2.2 BTL\_IF\_FS\_MMC\_SD\_Init()

### Description

This routine is a generic routine for initialization of the interface. It calls further routines as needed for hardware initialization for this interface.

### Prototype

```
void BTL_IF_FS_MMC_SD_Init(void);
```

### 6.2.3 BTL\_IF\_FS\_MMC\_SD\_IsInterfaceReady()

#### Description

This routine is a generic routine for checking the current status of the interface. Once the interface is ready a communication channel might be opened by this routine.

#### Prototype

```
int BTL_IF_FS_MMC_SD_IsInterfaceReady(void);
```

#### Return value

- 0 Interface is NOT ready (maybe not finished with initialization yet).
- 1 Interface is ready to communicate.
- 2 Interface is ready but no update file has been found.

## 6.2.4 BTL\_IF\_FS\_MMC\_SD\_Read()

### Description

This routine is a generic routine for reading data from the interface.

### Prototype

```
int BTL_IF_FS_MMC_SD_Read(void * pData,  
                           unsigned NumBytes);
```

### Parameters

Parameter	Description
<code>pData</code>	Pointer to buffer where data should be read into.
<code>NumBytes</code>	Number of bytes to read from interface.

### Return value

= 0      OK.  
≠ 0      Error.

## 6.2.5 BTL\_IF\_FS\_MMC\_SD\_Reset()

### Description

This routine is a generic routine for resetting the interface for re-reading the update.

### Prototype

```
int BTL_IF_FS_MMC_SD_Reset(void);
```

### Return value

= 0      OK.  
≠ 0      Error.

# Chapter 7

## Updating via UART

---

This chapter provides descriptions for using emLoad for UART.

## 7.1 Update Procedure with UART

### 7.1.1 Loading emLoad into a target

Open the start project that can be found in the shipment folder “\emLoadV4” and build the DEBUG configuration, download it to the target and run the program. We recommend to run the debug configuration at least once everytime a parameter was changed, as it includes several checks to make sure the user configuration is actually valid.

### 7.1.2 Preparing an emLoad firmware update

For the UART interface, the “PrepareFW[PRO]” tool has to be used for preparation of the firmware. See *PrepareFW / PrepareFWPRO* on page 22 for more information on how to use this tool.

### 7.1.3 Update the firmware using emLoad

Connect the powered down target to your PC using a serial cable connection, then power on the target. The target will wait for a limited time for the PC to send a first command. Run the `UART_Updater.exe` with the switch “-update”. See *UART Updater* on page 24 for more information on how to use this tool. emLoad is now updating and starting the firmware afterwards.



## 7.2 UART interface API

The table below lists the available API functions:

Function	Description
<code>BTL_IF_UART_Exit()</code>	This routine is a generic routine for de-initialization of the interface.
<code>BTL_IF_UART_Init()</code>	This routine is a generic routine for initialization of the interface.
<code>BTL_IF_UART_IsInterfaceReady()</code>	This routine is a generic routine for checking the current status of the interface.
<code>BTL_IF_UART_OnRx()</code>	This routine is a generic routine for receiving and storing bytes.
<code>BTL_IF_UART_OnTx()</code>	This routine is a generic routine for sending the next byte from the output buffer.
<code>BTL_IF_UART_SetAPI()</code>	Sets the API to use for UART communication.
<code>BTL_IF_UART_ConfigDelayBeforeExit()</code>	This routine configures a delay to wait before the UART hardware gets de-initialized.

## 7.2.1 BTL\_IF\_UART\_Exit()

### Description

This routine is a generic routine for de-initialization of the interface. It calls further routines as needed for hardware de-initialization for this interface.

### Prototype

```
void BTL_IF_UART_Exit(void);
```

## 7.2.2 BTL\_IF\_UART\_Init()

### Description

This routine is a generic routine for initialization of the interface. It calls further routines as needed for hardware initialization for this interface.

### Prototype

```
void BTL_IF_UART_Init(void);
```

### 7.2.3 BTL\_IF\_UART\_IsInterfaceReady()

#### Description

This routine is a generic routine for checking the current status of the interface. Once the interface is ready a communication channel might be opened by this routine.

#### Prototype

```
int BTL_IF_UART_IsInterfaceReady(void);
```

#### Return value

- 0        Interface is NOT ready (maybe not finished with initialization yet).
- 1        Interface is ready to communicate
- 2        Interface is ready but no update file has been found.

## 7.2.4 BTL\_IF\_UART\_OnRx()

### Description

This routine is a generic routine for receiving and storing bytes. To be called from interrupt context or with interrupts disabled.

### Prototype

```
void BTL_IF_UART_OnRx(unsigned Unit,  
                     U8      c);
```

### Parameters

Parameter	Description
Unit	Unit number (typically zero-based).
c	Received character.

## 7.2.5 BTL\_IF\_UART\_OnTx()

### Description

This routine is a generic routine for sending the next byte from the output buffer. To be called from interrupt context or with interrupts disabled.

### Prototype

```
int BTL_IF_UART_OnTx(unsigned Unit);
```

### Parameters

Parameter	Description
Unit	Unit number (typically zero-based).

### Return value

- 0 More data sent.
- 1 No more data to send.

## 7.2.6 BTL\_IF\_UART\_SetAPI()

### Description

Sets the API to use for UART communication.

### Prototype

```
void BTL_IF_UART_SetAPI(    unsigned    Unit,
                           const BTL_UART_API * pAPI);
```

### Parameters

Parameter	Description
Unit	Unit number (typically zero-based).
pAPI	API of type BTL_UART_API .

## 7.2.7 BTL\_IF\_UART\_ConfigDelayBeforeExit()

### Description

This routine configures a delay to wait before the UART hardware gets de-initialized. This might be necessary to make sure the last ACK of the UART protocol gets sent out before the hardware is disabled.

### Prototype

```
void BTL_IF_UART_ConfigDelayBeforeExit(U32 ms);
```

### Parameters

Parameter	Description
ms	Time [ms] to wait before UART is de-initialized.



## 7.3 Interface specific data structures

### 7.3.1 Structure BTL\_UART\_API

#### Description

This structure is used for UART interface abstraction.

#### Prototype

```
struct {
    void (*pfWrite1)          (unsigned Unit, U8 Data);
    void (*pfEnDisableTransmitter)(unsigned Unit, char OnOff);
} BTL_UART_API;
```

Member	Description
<a href="#">pfWrite1</a>	Callback to send out the next byte.
<a href="#">pfEnDisable-Transmitter</a>	Callbacks that enables/disables a transmitter chip (typically used for RS485).



# Chapter 8

## Updating via USB HID

---

This chapter provides descriptions for using emLoad for USB HID.

## 8.1 Update Procedure with USB HID

### 8.1.1 Loading emLoad into a target

Open the start project that can be found in the shipment folder “\emLoadV4” and build the DEBUG configuration, download it to the target and run the program. We recommend to run the debug configuration at least once everytime a parameter was changed, as it includes several checks to make sure the user configuration is actually valid.

### 8.1.2 Preparing an emLoad firmware update

For the USB HID interface, the “PrepareFW[PRO]” tool has to be used for preparation of the firmware. See *PrepareFW / PrepareFWPRO* on page 22 for more information on how to use this tool.

### 8.1.3 Update the firmware using emLoad

Connect the powered down target to your PC using an ordinary USB cable, then power on the target. The target will enumerate with the PC. Run the `USB_HID_Updater.exe` with the switch “-update”. See *USB HID Updater* on page 25 for more information on how to use this tool. emLoad is now updating and starting the firmware afterwards.

## 8.2 USB HID interface API

The table below lists the available API functions:

Function	Description
BTL_IF_USB_HID_ConfigDelayAfterDetach()	This routine configures a delay to wait after detach from host.
BTL_IF_USB_HID_Exit()	This routine is a generic routine for de-initialization of the interface.
BTL_IF_USB_HID_Init()	This routine is a generic routine for initialization of the interface.
BTL_IF_USB_HID_IsInterfaceReady()	This routine is a generic routine for checking the current status of the interface.

## 8.2.1 BTL\_IF\_USB\_HID\_ConfigDelayAfterDetach()

### Description

This routine configures a delay to wait after detach from host. This might be necessary as the USB host might not recognize a detach in BTL and attach in firmware if these two events occur too fast.

### Prototype

```
void BTL_IF_USB_HID_ConfigDelayAfterDetach(U32 ms);
```

### Parameters

Parameter	Description
ms	Time [ms] to wait after HW detach before proceeding.

## 8.2.2 BTL\_IF\_USB\_HID\_Exit()

### Description

This routine is a generic routine for de-initialization of the interface. It calls further routines as needed for hardware de-initialization for this interface.

### Prototype

```
void BTL_IF_USB_HID_Exit(void);
```

### 8.2.3 BTL\_IF\_USB\_HID\_Init()

#### Description

This routine is a generic routine for initialization of the interface. It calls further routines as needed for hardware initialization for this interface.

#### Prototype

```
void BTL_IF_USB_HID_Init(void);
```



## 8.2.4 BTL\_IF\_USB\_HID\_IsInterfaceReady()

### Description

This routine is a generic routine for checking the current status of the interface. Once the interface is ready a communication channel might be opened by this routine.

### Prototype

```
int BTL_IF_USB_HID_IsInterfaceReady(void);
```

### Return value

- 0 Interface is NOT ready (maybe not finished with initialization yet).
- 1 Interface is ready to communicate
- 2 Interface is ready but no update file has been found.

## 8.3 Interface specific data structures

### 8.3.1 Structure BTL\_USB\_HID\_PACKET

#### Description

This structure holds USB HID transmissions.

#### Prototype

```
struct {  
    U32 NumBytes;  
    U8  acData[60];  
} BTL_USB_HID_PACKET;
```

Member	Description
<a href="#">NumBytes</a>	Number of bytes of payload in the package.
<a href="#">acData</a>	Buffer for the packet.

# Chapter 9

## Updating via USBH MSD

---

This chapter provides descriptions for using emLoad for USBH MSD.

## 9.1 Update Procedure with USBH MSD

### 9.1.1 Loading emLoad into a target

Open the start project that can be found in the shipment folder “\emLoadV4” and build the DEBUG configuration, download it to the target and run the program. We recommend to run the debug configuration at least once everytime a parameter was changed, as it includes several checks to make sure the user configuration is actually valid.

### 9.1.2 Preparing an emLoad firmware update

For the USBH MSD interface, the “PrepareFW[PRO]” tool has to be used for preparation of the firmware. See *PrepareFW / PrepareFWPRO* on page 22 for more information on how to use this tool.

### 9.1.3 Update the firmware using emLoad

Store the generated firmware update file (typically Update.fw or Update.fwc if Crypto is used unless configured otherwise) onto your USB stick. Insert the USB stick into the target. emLoad is now updating and starting the firmware afterwards.

## 9.2 USBH MSD interface API

The table below lists the available API functions:

Function	Description
<code>BTL_IF_USBH_MSD_Exit()</code>	This routine is a generic routine for de-initialization of the interface.
<code>BTL_IF_USBH_MSD_Init()</code>	This routine is a generic routine for initialization of the interface.
<code>BTL_IF_USBH_MSD_IsInterfaceReady()</code>	This routine is a generic routine for checking the current status of the interface.
<code>BTL_IF_USBH_MSD_Read()</code>	This routine is a generic routine for reading data from the interface.
<code>BTL_IF_USBH_MSD_Reset()</code>	This routine is a generic routine for resetting the interface for re-reading the update.

## 9.2.1 BTL\_IF\_USBH\_MSD\_Exit()

### Description

This routine is a generic routine for de-initialization of the interface. It calls further routines as needed for hardware de-initialization for this interface.

### Prototype

```
void BTL_IF_USBH_MSD_Exit(void);
```

## 9.2.2 BTL\_IF\_USBH\_MSD\_Init()

### Description

This routine is a generic routine for initialization of the interface. It calls further routines as needed for hardware initialization for this interface.

### Prototype

```
void BTL_IF_USBH_MSD_Init(void);
```

### 9.2.3 BTL\_IF\_USBH\_MSD\_IsInterfaceReady()

#### Description

This routine is a generic routine for checking the current status of the interface. Once the interface is ready a communication channel might be opened by this routine.

#### Prototype

```
int BTL_IF_USBH_MSD_IsInterfaceReady(void);
```

#### Return value

- 0 Interface is NOT ready (maybe not finished with initialization yet).
- 1 Interface is ready to communicate
- 2 Interface is ready but no update file has been found.



## 9.2.4 BTL\_IF\_USBH\_MSD\_Read()

### Description

This routine is a generic routine for reading data from the interface.

### Prototype

```
int BTL_IF_USBH_MSD_Read(void * pData,
                        unsigned NumBytes);
```

### Parameters

Parameter	Description
<code>pData</code>	Pointer to buffer where data should be read into.
<code>NumBytes</code>	Number of bytes to read from interface.

### Return value

= 0      OK  
 ≠ 0      Error

## 9.2.5 BTL\_IF\_USBH\_MSD\_Reset()

### Description

This routine is a generic routine for resetting the interface for re-reading the update.

### Prototype

```
int BTL_IF_USBH_MSD_Reset(void);
```

### Return value

= 0      OK.  
≠ 0      Error.

# Chapter 10

## Updating via NAND/NOR

---

This chapter provides descriptions for using emLoad for NAND.

## 10.1 Update Procedure with NAND/NOR

### 10.1.1 Loading emLoad into a target

Open the start project that can be found in the shipment folder “\emLoadV4” and build the DEBUG configuration, download it to the target and run the program. We recommend to run the debug configuration at least once everytime a parameter was changed, as it includes several checks to make sure the user configuration is actually valid.

### 10.1.2 Preparing an emLoad firmware update

For the NAND or NOR interface, the “PrepareFW[PRO]” tool has to be used for preparation of the firmware. See *PrepareFW / PrepareFWPRO* on page 22 for more information on how to use this tool.

### 10.1.3 Update the firmware using emLoad

Store the generated firmware update file (typically Update.fw unless configured otherwise) onto your NAND/NOR. Restart and emLoad is now updating and starting the firmware afterwards.

## 10.2 NAND interface API

The table below lists the available API functions:

Function	Description
<a href="#">BTL_IF_FS_NAND_Exit()</a>	This routine is a generic routine for de-initialization of the interface.
<a href="#">BTL_IF_FS_NAND_Init()</a>	This routine is a generic routine for initialization of the interface.
<a href="#">BTL_IF_FS_NAND_IsInterfaceReady()</a>	This routine is a generic routine for checking the current status of the interface.
<a href="#">BTL_IF_FS_NAND_Read()</a>	This routine is a generic routine for reading data from the interface.
<a href="#">BTL_IF_FS_NAND_Reset()</a>	This routine is a generic routine for resetting the interface for re-reading the update.

## 10.2.1 BTL\_IF\_FS\_NAND\_Exit()

### Description

This routine is a generic routine for de-initialization of the interface. It calls further routines as needed for hardware de-initialization for this interface.

### Prototype

```
void BTL_IF_FS_NAND_Exit(void);
```

## 10.2.2 BTL\_IF\_FS\_NAND\_Init()

### Description

This routine is a generic routine for initialization of the interface. It calls further routines as needed for hardware initialization for this interface.

### Prototype

```
void BTL_IF_FS_NAND_Init(void);
```

### 10.2.3 BTL\_IF\_FS\_NAND\_IsInterfaceReady()

#### Description

This routine is a generic routine for checking the current status of the interface. Once the interface is ready a communication channel might be opened by this routine.

#### Prototype

```
int BTL_IF_FS_NAND_IsInterfaceReady(void);
```

#### Return value

- 0 Interface is NOT ready (maybe not finished with initialization yet).
- 1 Interface is ready to communicate.
- 2 Interface is ready but no update file has been found.



## 10.2.4 BTL\_IF\_FS\_NAND\_Read()

### Description

This routine is a generic routine for reading data from the interface.

### Prototype

```
int BTL_IF_FS_NAND_Read(void * pData,  
                        unsigned NumBytes);
```

### Parameters

Parameter	Description
<a href="#">pData</a>	Pointer to buffer where data should be read into.
<a href="#">NumBytes</a>	Number of bytes to read from interface.

### Return value

= 0      OK.  
≠ 0      Error.

## 10.2.5 BTL\_IF\_FS\_NAND\_Reset()

### Description

This routine is a generic routine for resetting the interface for re-reading the update.

### Prototype

```
int BTL_IF_FS_NAND_Reset(void);
```

### Return value

= 0      OK.  
≠ 0      Error.

## 10.3 NOR interface API

The table below lists the available API functions:

Function	Description
<code>BTL_IF_FS_NOR_Exit()</code>	This routine is a generic routine for de-initialization of the interface.
<code>BTL_IF_FS_NOR_Init()</code>	This routine is a generic routine for initialization of the interface.
<code>BTL_IF_FS_NOR_IsInterfaceReady()</code>	This routine is a generic routine for checking the current status of the interface.
<code>BTL_IF_FS_NOR_Read()</code>	This routine is a generic routine for reading data from the interface.
<code>BTL_IF_FS_NOR_Reset()</code>	This routine is a generic routine for resetting the interface for re-reading the update.

## 10.3.1 BTL\_IF\_FS\_NOR\_Exit()

### Description

This routine is a generic routine for de-initialization of the interface. It calls further routines as needed for hardware de-initialization for this interface.

### Prototype

```
void BTL_IF_FS_NOR_Exit(void);
```

## 10.3.2 BTL\_IF\_FS\_NOR\_Init()

### Description

This routine is a generic routine for initialization of the interface. It calls further routines as needed for hardware initialization for this interface.

### Prototype

```
void BTL_IF_FS_NOR_Init(void);
```

### 10.3.3 BTL\_IF\_FS\_NOR\_IsInterfaceReady()

#### Description

This routine is a generic routine for checking the current status of the interface. Once the interface is ready a communication channel might be opened by this routine.

#### Prototype

```
int BTL_IF_FS_NOR_IsInterfaceReady(void);
```

#### Return value

- 0 Interface is NOT ready (maybe not finished with initialization yet).
- 1 Interface is ready to communicate.
- 2 Interface is ready but no update file has been found.

## 10.3.4 BTL\_IF\_FS\_NOR\_Read()

### Description

This routine is a generic routine for reading data from the interface.

### Prototype

```
int BTL_IF_FS_NOR_Read(void * pData,  
                      unsigned NumBytes);
```

### Parameters

Parameter	Description
<a href="#">pData</a>	Pointer to buffer where data should be read into.
<a href="#">NumBytes</a>	Number of bytes to read from interface.

### Return value

= 0      OK.  
≠ 0      Error.

## 10.3.5 BTL\_IF\_FS\_NOR\_Reset()

### Description

This routine is a generic routine for resetting the interface for re-reading the update.

### Prototype

```
int BTL_IF_FS_NOR_Reset(void);
```

### Return value

= 0      OK.  
≠ 0      Error.



# Chapter 11

## Debugging

---

emLoad comes with debugging options including optional warning and log outputs. These are explained in the following chapter.

## 11.1 Message output

The debug builds of emLoad include a debug system which helps to analyze the correct implementation of your application. All modules can output logging and warning messages via terminal I/O.

### 11.1.1 Debug API functions

Function	Description
<code>BTL_Log()</code>	This function is called by the stack in debug builds with log output.
<code>BTL_Panic()</code>	This function is called if the BTL encounters a critical situation.
<code>BTL_Warn()</code>	This function is called by the stack in debug builds with log output.

### 11.1.1.1 BTL\_Log()

#### Description

This function is called by the stack in debug builds with log output. In a release build, this function may not be linked in.

#### Prototype

```
void BTL_Log(const char * s);
```

### 11.1.1.2 BTL\_Panic()

#### Description

This function is called if the BTL encounters a critical situation.

#### Prototype

```
void BTL_Panic(const char * s);
```

### 11.1.1.3 BTL\_Warn()

#### Description

This function is called by the stack in debug builds with log output. In a release build, this function may not be linked in.

#### Prototype

```
void BTL_Warn(const char * s);
```

