emLoad

Bootstrap loader for
embedded applications

User Guide & Reference Manual

Document: UM04002
Software Version: 4.20
Revision: 0
Date: July 21, 2023

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

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

Copyright notice

Trademarks

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

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

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

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

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

Manual versions

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

Print date: July 21, 2023

Software	Revision	Date	By	Description
4.20	0	230721	OO	Change of variable type. • All U32 address variables in emLoad have been changed to use the type PTR_ADDR which for 32-bit systems still defaults to the U32 type. Chapter "emLoad BTL structures" updated. • Updated the structure BTL_FLASH_DRIVER .
4.18a	0	230206	OO	Chapter "emLoad Status Codes" updated. • Status code BTL_STATUS_ABORT_UPDATE_NO_FW added.
4.18	0	221020	OO	Chapter "Core functions" updated. • BTL_ConfigRetryUpdateOnTimeout() added. • BTL_SetCmdRecvTimeout() added. • BTL_SetCmdParaRecvTimeout() added. • BTL_COM_AddOnCmdHook() added. • BTL_COM_SetCustomData() added. • BTL_NET_DisableInterrupt() added. • BTL_NET_EnableInterrupt() added. Chapter "Configuration" updated. • Configuration define BTL_CMD_RECV_TIMEOUT added. • Configuration define BTL_CMD_PARA_RECV_TIMEOUT added. Chapter "emLoad Status Codes" updated. • Status code BTL_STATUS_BACK_TO_IF_READY added. • Status code BTL_STATUS_IF_READ_TIMEOUT added.
4.16f	0	220607	OO	Added new USB Bulk interface and description.
4.16e	0	211105	OO	Added new NFC NTAG interface and description.
4.16d	0	211018	OO	Chapter "Core functions" updated. • BTL_POLICY_UPDATE_DIFFERENT option added to BTL_ConfigUpdatePolicy() .
4.16c	0	201130	OO	Chapter "Configuration" updated. • Configuration define BTL_NO_INFINITE_FEED_WATCHDOG added. • Sub-chapter *Using a watchdog* on page 38 added.
4.16	0	200626	OO	Chapter "Configuration" updated. • Configuration define BTL_SKIP_CRC_BOOT_CHECK added. • Configuration define BTL_DISABLE_FW_CRC_CHECK added. Chapter "emLoad Status Codes" updated. • Status code BTL_STATUS_VERIFY_SIGNATURE added. Chapter "emLoad BTL structures" updated. • pfExec added to structure BTL_NET_API. Added new UART API BTL_IF_UART_ExecIF(). Added new USB HID API BTL_IF_USB_HID_ExecIF(). Added new USB HID API BTL_IF_USB_HID_Read().
4.14a	0	190220	OO	Chapter "Core functions" updated. • BTL_SetFlashProgProgressHook() added.
4.14	0	181219	OO	Renamed tools with "Updater" in their names to "Loader" to prevent Windows UAC dialog interaction. Chapter "emLoad BTL structures" updated. • pfResetIF of BTL_NET_API description updated.
4.12	0	181127	OO	Chapter "emLoad Status Codes" updated. • Status code BTL_STATUS_REBOOTED_START_FW added. • Status code BTL_STATUS_REBOOTING added. • Status code BTL_STATUS_ABORT_UPDATE_CHECK_FW added. Chapter "CPU specific API" updated. • BTL_CPU_DisableInt() added. • BTL_CPU_EnableInt() added. Chapter "emLoad BTL structures" updated. • pfSetMarkerReboot added to structure BTL_NET_API. • pfCheckMarker added to structure BTL_NET_API.
4.10e	0	180726	OO	Added new UART API BTL_IF_UART_Reset().
4.10b	0	180403	OO	Company name changed.

Software	Revision	Date	By	Description
				Chapter “emLoad BTL structures” updated. • `pfResetIF` of `BTL_NET_API` can now be used with all interfaces.
4.10a	0	171110	OO	Added new UART API `BTL_IF_UART_ConfigDelayBeforeExit()`.
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 *Digitally sign an update* on page 31 using the Signature add-on.
4.06c	0	170421	TG	Added new FS-NAND interface description.
4.06b	0	170221	OO	Added `pfResetIF` to `BTL_NET_API`.
4.06	3	170210	TG	Minor syntax correction.
4.06	2	170103	OO	Minor changes. Chapter “emLoad tools”. • Added `BTL_ImageCreator` 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 Ritchie (ISBN 0--13--1103628), which describes the standard in C programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

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

Typographic conventions for syntax

This manual uses the following typographic conventions:

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

Table of contents

Chapter 1

Introduction to 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 a PC driven interface is used - a loader application such as the `USB_HID_Loader`) 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 Bulk	USB connection from target to a PC (see *Updating via USB Bulk* on page 119).
USB Device HID	USB connection from target to a PC (see *Updating via USB HID* on page 129).
USB Host MSD	Update via USB stick (see *Updating via USBH MSD* on page 139).
FS MMC/SD card	Update via MMC/SD card (see *Updating via MMC/SD card* on page 97).
FS NAND	Update via file on NAND (see *Updating via NAND/NOR* on page 147).
FS NOR	Update via file on NOR (see *Updating via NAND/NOR* on page 147).
NFC NTAG	Update via wireless NFC connection between target and PC (see *Updating via NFC NTAG* on page 161).

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
`0x08000000`	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.
`0x08005FFF`	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.
`0x08006000`	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 (`0x40000`) - Start of firmware area (`0x6000`) - size of firmware info area (`0x10`).
`0x0803FFEF`	Start of the firmware info area. The firmware info area stores information about the current firmware such as a CRC checksum that is

Address	Used for
	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 `BTL_FIRMWARE_INFO_SIZE` (see *Compile-time configuration switches* on page 34).

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

Note on the emLoad size

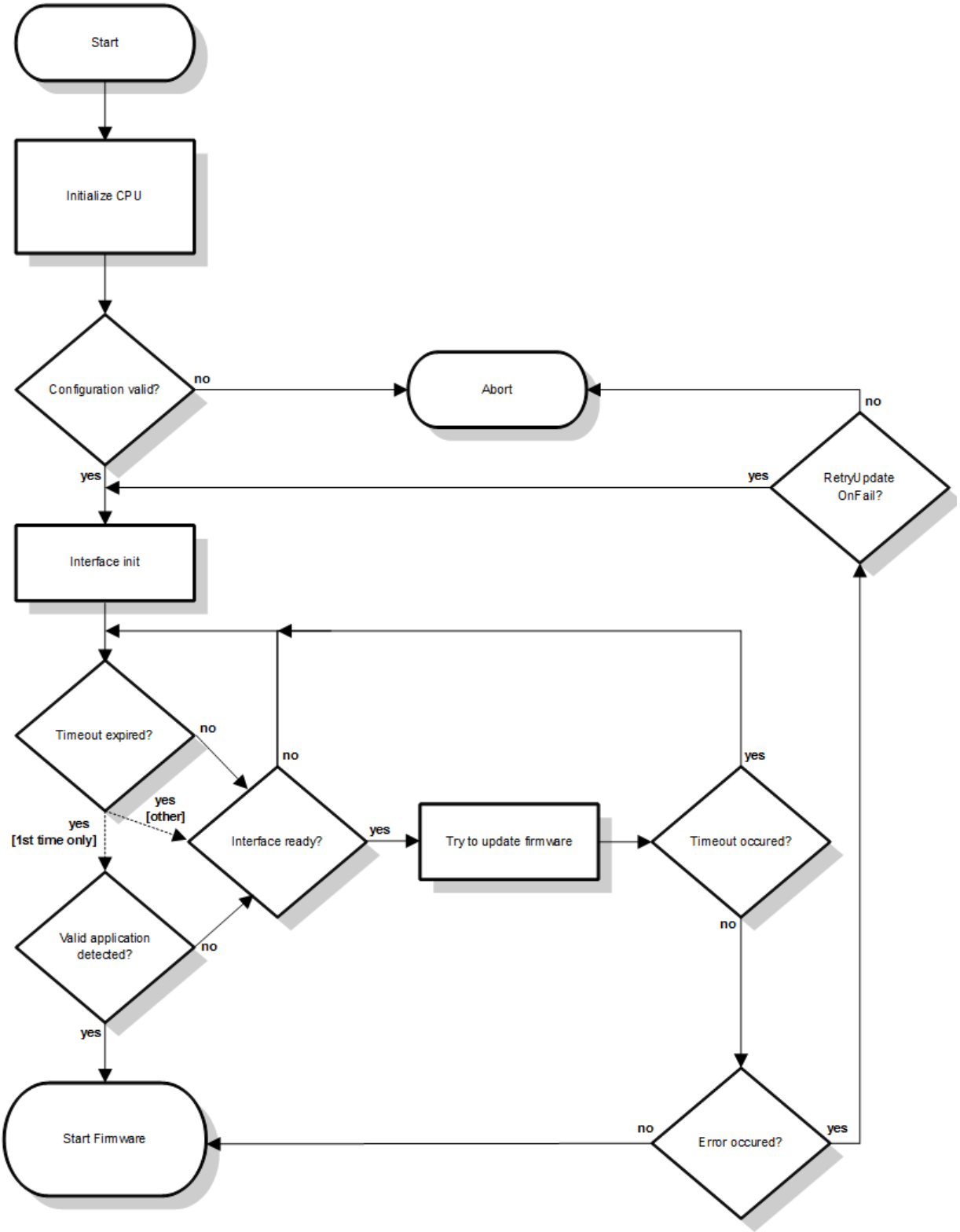
The size (i.e. the number of flash sectors) actually used by emLoad is of course different in DEBUG and in RELEASE configurations. By default, an emLoad project is delivered with a layout that supports both configurations.

Sometimes the difference could be quite important between DEBUG and RELEASE sizes of emLoad. Thus in case of constrains 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 53). 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 Loader, used for UART interface.
- USB Bulk Loader, used for USB Bulk interface.
- USB HID Loader, used for USB HID interface.
- NTAG Loader, used for NFC NTAG interface.

- BTL Image Creator, used to create a programmable image containing a firmware update with/without BTL.

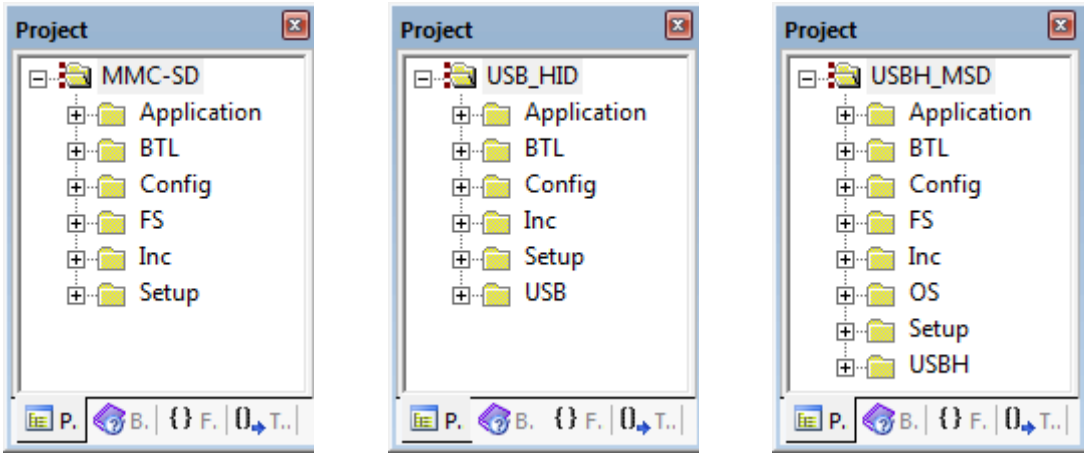
For more information, see chapter *emLoad tools* on page 23.

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 appropiate 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 or Bulk 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 34 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 24 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 34 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 31.

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 firmare update file use the PrepareFW[PRO] tool that can be found in "\emLoadV4\Windows\PrepareFW" or "\emLoadV4\Windows\PrepareFWPRO". See *PrepareFW / PrepareFWPRO* on page 24 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_Loader.exe` that can be found in "\emLoadV4\Windows\`UART_Loader`" with the `-update` and `-wait` options. See *UART Loader* on page 26 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 Bulk

- Run the `USB_BULK_Loader.exe` that can be found in "\emLoadV4\Windows \`USB_BULK_Loader`" with the `-update` and `-wait` options. See *USB Bulk Loader* on page 28 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 USB HID

- Run the `USB_HID_Loader.exe` that can be found in "\emLoadV4\Windows \`USB_HID_Loader`" with the `-update` and `-wait` options. See *USB HID Loader* on page 27 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.

1.3.7 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.

1.3.8 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.

1.3.9 Updating the firmware using emLoad for NTAG

- Run the `NTAG_Loader.exe` that can be found in "\emLoadV4\Windows\`NTAG_Loader`" with the `-update` and `-wait` options. See *NTAG Loader* on page 29 for more details.
- Connect the NFC reader to your PC.
- Place the NFC reader onto the NTAG token with the target in powered down state. The reader should recognize the token.
- Power on the target.
- emLoad updates and starts the firmware afterwards.

1.3.10 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".

1.3.11 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

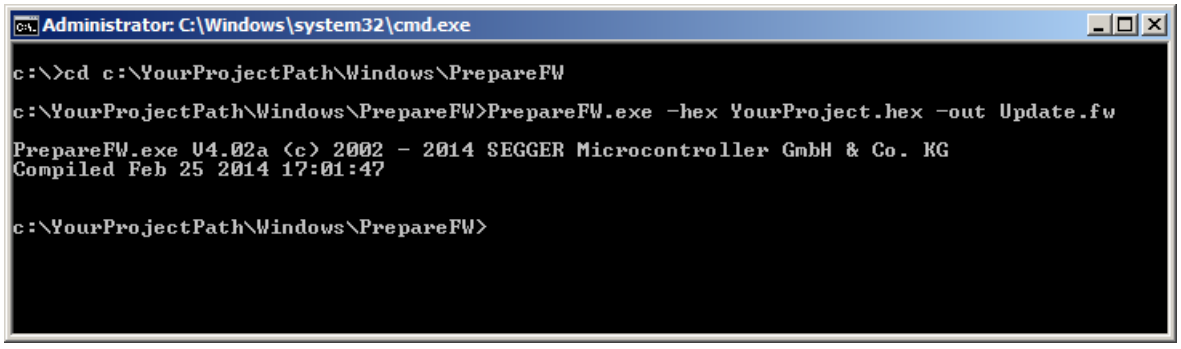
emLoad firmware updates need to adhere a specific format to be considered valid. To ensure this format, SEGGER provides the "PrepareFW" or "PrepareFWPRO" tool, which is a 32-bit Windows application specifically designed to transform binary files, Intel hex files or Motorola files into emLoad-compatible 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:



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 Loader

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

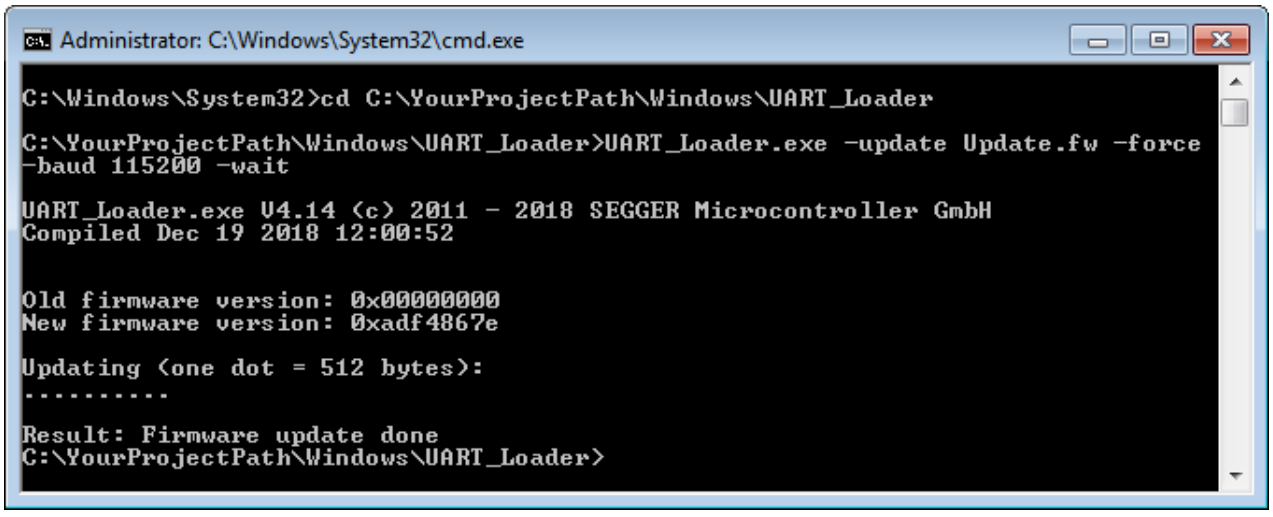
2.2.1 Running UART_Loader.exe

`UART_Loader` 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_Loader` can be used to identify the firmware in a target. By default the `UART_Loader` 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_Loader`:



2.3 USB HID Loader

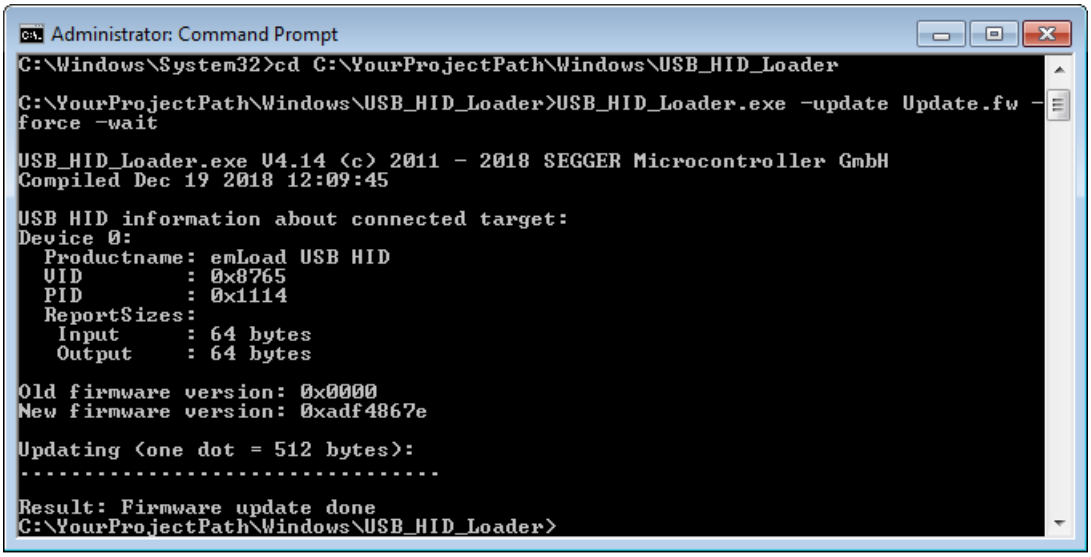
By using an emLoad-compatible firmware file, this program updates an emLoad target via USB HID interface.

2.3.1 Running USB_HID_Loader.exe

`USB_HID_Loader` 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_Loader` 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_Loader`:



2.4 USB Bulk Loader

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

2.4.1 Running USB_BULK_Loader.exe

USB_BULK_Loader 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_BULK_Loader 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

2.5 NTAG Loader

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

2.5.1 Compatible card readers

For the NFC NTAG interface a card reader that support the FastWrte and FastRead command is required. The following readers have been tested:

- uTrust 3700 F

2.5.2 Running NTAG_Loader.exe

`NTAG_Loader` 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, `NTAG_Loader` 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

2.6 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.

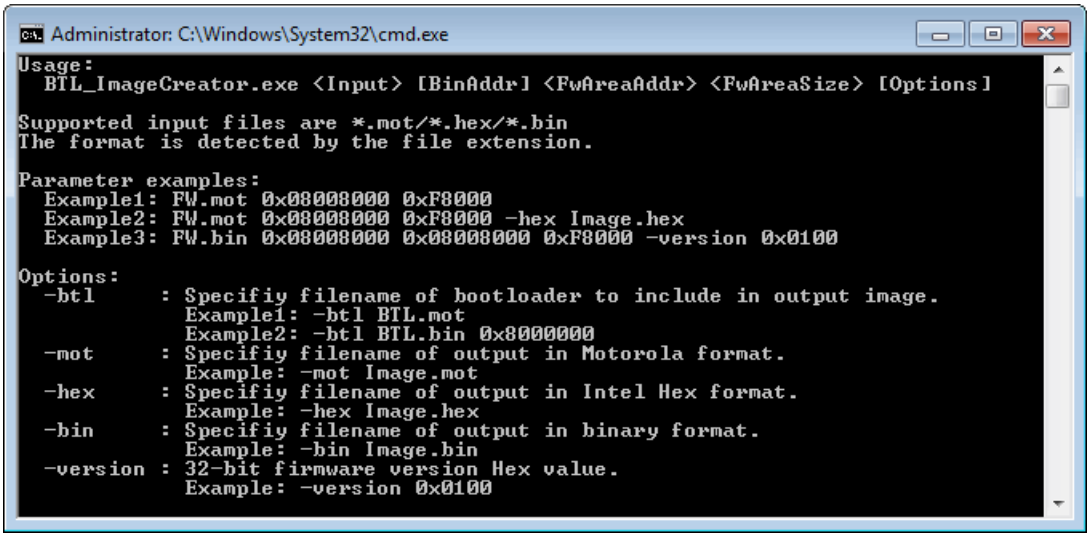
The BTL Image Creator tool, which is a 32-bit Windows application specifically designed to transform binary files, Intel hex files or Motorola files into an emLoad compatible format. `BTL_ImageCreator` is shipped as both, an executable that works out-of-the-box and also as a Microsoft Visual Studio project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

2.6.1 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`:



2.7 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.

2.7.1 Using the batch script to sign a firmware update

A batch script `CreateSignedUpdate.bat` can be found in the folder `\Windows\emSecure_ECDSA\/\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
emSecure.prv	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.
emSecure.pub	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.
Update.fw	The firmware update digitally signed with the private key in `emSecure.prv` . This file can now be used to update the firmware on the target running emLoad with Signature add-on.
`emLoad_PublicKey.c`	A code representation of the public key data, ready to be included in the `Main.c` sample file on the target.

2.7.2 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.

2.7.2.1 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).

2.7.2.2 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).

2.7.2.3 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.

2.7.2.4 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.

2.7.2.5 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.

2.7.2.6 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	BTL_ALLOW_CHANGE_UPDATE_POLICY	0	Defines if changes to the update policy are allowed.
T	BTL_COMPANY_NAME	“My Company”	Used to store the company name for a firmware.
T	BTL_DEVICE_NAME	“My Device”	Used to store the device name for a firmware.
N	BTL_INIT_TIMEOUT	500	Used to store the time [in ms] the BTL waits for a firmware update before starting a valid firmware.
N	BTL_CMD_RECV_TIMEOUT	0	Timeout [ms] after which a command based interface aborts if no new command is received. 0 disables the timeout.
N	BTL_CMD_PARA_RECV_TIMEOUT	5000	Timeout [ms] after which a command based interface aborts if no new data is received for an ongoing commmand (parameters etc.). 0 disables the timeout.

Type	Symbolic name	Default	Description
N	BTL_FIRMWARE_INFO_SIZE	16	Used to define the size of the firmware info area.
T	BTL_FW_PASS	""	Used to store a password for a firmware update.
T	BTL_FW_STRING	"SEG-GER-FWUpdate"	Used to store the firmware string of a firmware update.
T	BTL_UPDATE_FILE_NAME	"Update.fw" or "Update.fwc"	Used to store the file name of a firmware update. fwc extension when Crypto is present.
N	BTL_UPDATE_POLICY	0	Used to store the current update policy.
B	BTL_SKIP_CRC_BOOT_CHECK	0	Enables/disables checking the CRC of the entire firmware area against the CRC stored in the FirmwareInfoArea at boot. You might consider skipping the CRC verification at boot if you do not expect data corruption to appear after the update and CRC calculation takes a long time (due to architecture or large flash).
B	BTL_DISABLE_FW_CRC_CHECK	0	Enables/disables using a CRC over the whole flash area as second validation (next to a simple valid marker). You might consider disabling the use of CRC verification if this takes a long time (due to architecture or large flash).
B	BTL_NO_INFINITE_FEED_WATCHDOG	0	If enabled, the watchdog is not fed from true infinite loops like the "wait while FLASH BUSY flag is cleared" as used in many Flash Drivers (FD). This typically applies to FDs for internal memory as FDs for external memory might be far more complex and might come with their own limitations. For more information about this please refer to *Safer watchdog usage in Flash Drivers* on page 38 .

Type	Symbolic name	Default	Description
B	CHECK_SIGNATURE	0	Enables/disables the Signature check when available.
B	DECRYPT_FW	0	If the Crypto option is present, defines if crypto is used or not.
T	DECRYPT_PW	"SEGGER"	If the Crypto option is present, password to use for key and IV generation if not set by defines.
T	BTL_CRYPTO_KEY and BTL_CRYPTO_IV	not defined	If the Crypto option is present, key and IV to use.
Debug macros			
N	BTL_DEBUG	0	Macro to define the debug level of the emLoad build. Refer to *Debug level* on page 37 for a description of debug levels.
Replacement macros			
F	BTL_ADDR2PTR	(void BTL_HUGE *)(Addr)	Macro to convert an address to a pointer to that address.
F	BTL_MEMCMP	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	BTL_MEMCPY	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	BTL_MEMSET	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	BTL_PTR2ADDR	(long)Ptr	Macro to convert a pointer to an address to the numerical address it points at.
F	BTL_RAM_FUNC		Macro to define a RAM_FUNC 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.

3.3 Using a watchdog

emLoad comes prepared to be used with a watchdog while the default configuration delivered is typically not using a watchdog. This is due to various configurations that can be used with a watchdog and a specific behavior might be what the user is looking, making it very hard to provide a suitable implementation for everyone.

Therefore emLoad comes prepared for your own watchdog implementation by providing you with a place where you can install your watchdog feeding, while trying to trigger this routine as often as possible while being idle or where it would be necessary during bootloader work to do.

The routine `BTL_X_FeedWatchdog()` can typically be found in the `BTL_Config_<Interface>_<Hardware>.c` file. It is typically empty as no watchdog is used or is even disabled during start for simplicity ot the project. The routine is desined to be placed in RAM (if this is possible easily possible with your IDE/toolchain), so it can be called during flash operations that might need the flash untouched while changing it.

3.3.1 Safer watchdog usage in Flash Drivers

Typically in a `Flash Driver` (`FD`) the feed watchdog routine is called whenever necessary. One such a place is waiting for a flash operation to finish by waiting for a BUSY bit being set/cleared in a register of the flash controller. Many devices require you to not access the flash either in part or in whole while a flash operation is active. Therefore typically this part of an FD is often executed from a function loaded into RAM (`RAMfunc`).

A typical code for this looks as follows:

```
BTL_RAM_FUNC static void _WaitWhileBusy() {
  volatile U32 Status;

  do {
    BTL_X_FeedWatchdog();
    Status = FLASH_REG;
  } while ((FLASG_REG & FLASH_BUSY_BIT) != 0u);
}
```

Typically this is fine and is also described exactly like this in many CPU specific manuals when showing how to program the internal flash of the device. The flash controller and `FLASH_REG` never exiting out of the `BUSY` state is not considered something that typically happens but in theory this is a possible (while highly unlikely) scenario.

While being highly unlikely to happen, this still is a possible scenario. However, you would need to consider that if the flash and things around it already have started to corrupt, how likely would it be that you manage to successfully bring the device back into a functional state from a watchdog at this point. Also to mention that if something managed to cause this much trouble to the system, it is likely that the general execution of the CPU and even the RAMfunc itself can not be expected to work reliable anymore.

There are two more or less simple approaches on how to handle this when you want the extra security in this scenario.

Using the regular feed watchdog behavior

When using the regular `BTL_X_FeedWatchdog()` behavior, you could check from where the routine has been called, typically involving some assembler magic to access something like a `Link Register` (`LR`) or whetever is used by your architecture. If you always get called from the very same place and the bootloader system tick time does not increment for a longer period of time, you could assume that you are stuck in a Flash Driver infinite waiting loop.

Using the configuration define BTL_NO_INFINITE_FEED_WATCHDOG

When enabling the define `BTL_NO_INFINITE_FEED_WATCHDOG` the feed watchdog behavior in Flash Drivers is changed to the following:

```
BTL_RAM_FUNC static void _WaitWhileBusy() {
  volatile U32 Status;

  BTL_X_FeedWatchdog();
  do {
    Status = FLASH_REG;
  } while ((FLASG_REG & FLASH_BUSY_BIT) != 0u);
  BTL_X_FeedWatchdog();
}
```

Places that are not subject of infinite loops but are using some sort of timeout remain unchanged. With this change the watchdog is fed around the infinite loop but not within, causing a timeout if this never returns. This behavior is not the default for two reasons:

The first is that this is more like a theoretical problem as described before, as this is typically not even considered to happen by most CPU vendors. Some of them even instruct you to deactivate the watchdog for changing flash contents.

The second reason is this might cause trouble depending on how strict the watchdog is configured by the user (or even how it can be configured at all). For this solution to work, it is necessary that the watchdog timeout is configured long enough that every flash operation can safely finish its work. To avoid trouble with customer watchdog configurations, this method is not used as the default and might need some fine tuning from the user upon enabling it.

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

Function	Description
BTL_SetCmdRecvTimeout()	Sets/extends the timeout [ms] to wait for the next command to be received.
BTL_SetCmdParaRecvTimeout()	Sets the timeout [ms] to wait for new data to be received for an ongoing command.
BTL_SetFlashCrcProgressHook()	This function registers a callback that will be executed after each CRC chunk of the flash CRC calculation.
BTL_SetFlashProgProgressHook()	This function registers a callback that will be executed after each chunk programmed.
BTL_ConfigInterfaceReadyDelay()	Configures the length of the delay that is waited after interface reports that it is ready.
BTL_DisableUpdateInitTimeout()	This function disables the initial timeout configured for the interface to wait for a first data exchange with an update client.
BTL_GetUpdateInitTimeoutTimestamp()	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.
BTL_Delay()	Wait for the given time (parameter in ms).
Firmware configuration functions	
BTL_GetCompanyName()	Retrieves the pointer for the company name string.
BTL_GetDeviceName()	Retrieves the pointer for the device name string.
BTL_GetFwString()	Retrieves the pointer for the set firmware string.
BTL_GetFwVersion()	Retrieves the firmware version of the current firmware in flash.
BTL_GetUpdateFileName()	Retrieves the current pointer to the update file name string.
BTL_SetCompanyName()	Sets the pointer for the company name string to a given resource.
BTL_SetDeviceName()	Sets the pointer for the device name string to a given resource.
BTL_SetFwPassword()	Sets the pointer for the firmware password string to a given resource.
BTL_SetFwStartAddr()	Sets firmware start addr.
BTL_SetFwString()	Sets the pointer for the firmware string to a given resource.
BTL_SetUpdateFileName()	Sets the pointer for the update file name string to a given resource.
Interface specific functions	
BTL_COM_AddOnCmdHook()	Sets a callback that gets notified about commands received.
BTL_COM_SetCustomData()	Sets custom data that can be requested using the BTL_COM_READ_CUSTOM_DATA command.
Other functions	
BTL_NET_DisableInterrupt()	Globally disables interrupts.
BTL_NET_EnableInterrupt()	Globally enables interrupts.

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
pHook	Pointer to a structure of type BTL_HOOK_ON_STATE_CHANGE
pf	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
pMem	Pointer to a memory pool
NumBytes	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 92):

- BTL_STATUS_INIT
- BTL_STATUS_STARTED
- BTL_STATUS_CONFIG_ERROR

4.1.1.5 BTL_Exec()

Description

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.

Prototype

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

Additional information

If `BTL_NET_API.pfExec()` is set, it is the responsibility of the application `pfExec` callback to call `BTL_Exec()` for internal management after making sure that `BTL_NET_API.pfHasTick()` returns that one ms has passed and `BTL_Exec()` therefore will do something. The `BTL_NET_API.pfHasTick()` = true condition should be cleared by the `BTL_NET_API.pfHasTick()` callback in this case.

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
pKey	Pointer to 32-byte long key to use.
pIV	Pointer to 16-byte long Initialization Vector that is used as first IV for each new filestream.
KeyLen	Length of key to use. Used for parameter checking.
IVLen	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
pPublicKey	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
pPublicKey	Pointer to RSA public key.
pSalt	Pointer to salt. Can be NULL.
NumBytesSalt	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(const BTL_FLASH_INFO * p,
                         unsigned               NumEntries);
```

Parameters

Parameter	Description
p	Pointer to a structure of type BTL_FLASH_INFO.
NumEntries	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
NumBytes	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_ConfigRetryUpdateOnTimeout()

Description

This function configures the behavior in case of a communication interface timeout.

Prototype

```
void BTL_ConfigRetryUpdateOnTimeout(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

This routine allows to configure the behavior in case a read read timeout is reported that also comes with a state transition to BTL_STATUS_IF_READ_TIMEOUT .

Default behavior is to not retry.

4.1.1.13 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 • BTL_POLICY_UPDATE_DIFFERENT

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 34). By default, BTL_ALLOW_CHANGE_UPDATE_POLICY is defined as 0, so the policy can not be changed.

4.1.1.14 BTL_GetCpuClock()

Description

Retrieves the stored CPU clock value.

Prototype

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

Return value

Previously stored CPU clock in Hz

4.1.1.15 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

4.1.1.16 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

4.1.1.17 BTL_SetInitTimeout()

Description

Sets the 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
ms	Timeout in ms

Additional information

Default value is 500 ms.

4.1.1.18 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
pf	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);
```


4.1.1.19 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
Skip	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. Default is set via the BTL_SKIP_CRC_BOOT_CHECK define.

4.1.1.20 BTL_SetCmdRecvTimeout()

Description

Sets/extends the timeout [ms] to wait for the next command to be received.

Prototype

```
void BTL_SetCmdRecvTimeout(unsigned ms);
```

Parameters

Parameter	Description
ms	Timeout [ms] to wait for the next command to be received.

Additional information

The timeout is only useful for command driven interfaces.

The timeout is set/extended to NOW plus the timeout. It is automatically extended once after each successful command.

To avoid a race condition between interrupt and non-interrupt usage of this routine, interrupts are disabled and re-enabled for this operation. If used from an interrupt context, re-enabling interrupts might allow nestable interrupt behavior with undesired and erroneous outcome by race conditions. When using this routine from interrupt context the interrupt handler needs to disable/enable interrupts around calling at least this routine using `BTL_NET_DisableInterrupt()` before and `BTL_NET_EnableInterrupt()` after.

4.1.1.21 BTL_SetCmdParaRecvTimeout()

Description

Sets the timeout [ms] to wait for new data to be received for an ongoing command.

Prototype

```
void BTL_SetCmdParaRecvTimeout(unsigned ms);
```

Parameters

Parameter	Description
ms	Timeout [ms] to wait for new data like parameters for an ongoing command to be received.

Additional information

The timeout is only useful for command driven interfaces and only applies while commands are executed. It does not serve as a timeout for receiving commands.

4.1.1.22 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
pf	Pointer to a callback that will receive the current progress of the CRC calculation.

4.1.1.23 BTL_SetFlashProgProgressHook()

Description

This function registers a callback that will be executed after each chunk programmed. It can be used to show the update progress for example by using a GUI progress bar.

Prototype

```
void BTL_SetFlashProgProgressHook(BTL_FLASH_PROG_PROGRESS_HOOK_CB * pf);
```

Parameters

Parameter	Description
pf	Pointer to a callback that will receive the current progress of flash programming.

4.1.1.24 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
ms	Time to wait after interface reports ready before communicating.

4.1.1.25 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);
```


4.1.1.26 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

4.1.1.27 BTL_Delay()

Description

Wait for the given time (parameter in `ms`).

Prototype

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

Parameters

Parameter	Description
`ms`	Delay in `ms`

4.1.1.28 BTL_GetCompanyName()

Description

Retrieves the pointer for the company name string.

Prototype

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

Return value

Pointer to the company name string

4.1.1.29 BTL_GetDeviceName()

Description

Retrieves the pointer for the device name string.

Prototype

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

Return value

Pointer to the device name string

4.1.1.30 BTL_GetFwString()

Description

Retrieves the pointer for the set firmware string.

Prototype

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

Return value

Pointer to the firmware identifier string

4.1.1.31 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.

4.1.1.32 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

4.1.1.33 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.

4.1.1.34 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.

4.1.1.35 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.

4.1.1.36 BTL_SetFwStartAddr()

Description

Sets firmware start addr.

Prototype

```
void BTL_SetFwStartAddr(PTR_ADDR Addr);
```

Parameters

Parameter	Description
Addr	Addr of the entry point into the firmware

4.1.1.37 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.

4.1.1.38 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.

4.1.1.39 BTL_COM_AddOnCmdHook()

Description

Sets a callback that gets notified about commands received.

Prototype

```
void BTL_COM_AddOnCmdHook(BTL_COM_HOOK_ON_CMD * pHook,
                          BTL_COM_ON_CMD_FUNC * pf);
```

Parameters

Parameter	Description
pHook	Pointer to a structure of type BTL_COM_HOOK_ON_CMD .
pf	Pointer to a callback to be notified about the command status.

Example

```
//
// Static declaration.
//
static BTL_COM_HOOK_ON_CMD _OnBtlComCmdHook;
static void _OnBtlComCmd(BTL_COM_ON_CMD_INFO* pInfo) {
  if (pInfo->Cmd != 0u) {  // New command ?
    //
    // Notify about the command.
    //
  }
}
//
// Code running in main task.
//
BTL_COM_AddOnCmdHook(&_OnBtlComCmdHook, _OnBtlComCmd);
```


4.1.1.40 BTL_COM_SetCustomData()

Description

Sets custom data that can be requested using the BTL_COM_READ_CUSTOM_DATA command.

Prototype

```
void BTL_COM_SetCustomData(U8  * pData,
                           U32   NumBytes);
```

Parameters

Parameter	Description
pData	Pointer to the data that shall be available upon request of the BTL_COM_READ_CUSTOM_DATA command. Can be NULL for no custom data at all.
NumBytes	Length of the data at pData . Set to 0 if pData is NULL .

Additional information

By default “loader” applications expect to receive a single printable string without its ’’ termination. Therefore the NumBytes parameter can be used with strlen(String) .

Instead of a single string other customized data can be used as well but might require modifications to the “loader” application.

4.1.1.41 BTL_NET_DisableInterrupt()

Description

Globally disables interrupts. Callback needs to take care of nested calls.

Prototype

```
void BTL_NET_DisableInterrupt(void);
```


4.1.1.42 BTL_NET_EnableInterrupt()

Description

Globally enables interrupts. Callback needs to take care of nested calls.

Prototype

```
void BTL_NET_EnableInterrupt(void);
```


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)  (const BTL_FLASH_INFO* pFlashInfo);
  int (*pfRead)
  (const BTL_FLASH_INFO* pFlashInfo, U8 * pDest, PTR_ADDR Src , U32 NumBytes);
  int (*pfWrite) (const BTL_FLASH_INFO* pFlashInfo, PTR_ADDR Dest, U8 * pSrc , U32 NumBy
  int (*pfExit)  (const BTL_FLASH_INFO* pFlashInfo);
} BTL_FLASH_DRIVER;
```

Member	Description
pfInit	Pointer to a function that initializes the flash driver and allocates memory as needed.
pfRead	Pointer to a function that reads data from any address of the flash.
pfWrite	Pointer to a function that writes data into any section of the flash.
pfExit	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 {
  PTR_ADDR                 Addr;
  U32                      NumBytes;
  const BTL_FLASH_DRIVER * pDriver;
} BTL_FLASH_INFO;
```

Member	Description
Addr	Address of the beginning of the firmware area.
NumBytes	Firmware data size in Bytes.
pDriver	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
pf	Pointer to a function to be called from a hook.
pNext	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)(PTR_ADDR StartAddr);
  void (*pfStartBTL)        (void);
  int  (*pfRead)            (void * pData, unsigned NumBytes);
  void (*pfDisableInterrupt)(void);
  void (*pfEnableInterrupt) (void);
  int  (*pfExecIF)          (void);
  int  (*pfResetIF)         (void);
  void (*pfSetMarkerReboot) (void);
  int  (*pfCheckMarker)     (unsigned ClrMarker);
  void (*pfExec)            (void);
} BTL_NET_API;
```

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

Member	Description
pfResetIf	Pointer to a function that offers a reset of the update interface to be able to re-read the update file. If set, the update will be read twice. Once for a simple verification that it is not corrupted and a second time for the actual update. During the first read the flash content that would be programmed is read back and compared against the update. If the content is the same, the second run will not program the flash again and assume that the FirmwareInfoArea will not change as well. Set to NULL to disable pre-flash verification.
pfSetMarkerReboot	Pointer to a function that sets a marker in RAM (has to be non-volatile on reboot) and soft-resets the CPU. Set to NULL to disable CPU reset for peripheral reset strategy (CPU disable needs to be implemented).
pfCheckMarker	Pointer to a function that checks the marker in RAM and clears it if requested. Can be NULL if pfSetMarkerReboot is NULL.
pfExec	Pointer to a function executed whenever internal management should take place. If set the callback is expected to also call BTL_Exec() to execute internal management.

Additional information

Function pointers in BTL_NET_API are not checked for validity and are therefore not allowed to be NULL pointers if not explicitly stated. 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
BTL_STATUS_INIT	0	After booting, state after reset, not handled.
BTL_STATUS_STARTED	30	After booting, BTL running.
BTL_STATUS_REBOOTED_START_FW	40	BTL rebooted for peripheral reset, try to start new firmware.
BTL_STATUS_CONFIG_ERROR	60	Error during config init.
BTL_STATUS_TRY_UPDATE	90	Trying to update firmware.
BTL_STATUS_IF_PRE_FILE_OPEN	120	For FS based interfaces this means FS_Init() 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.
BTL_STATUS_IF_READY	150	Update interface initialized and ready for updating.
BTL_STATUS_BACK_TO_IF_READY	151	Intermediate state used to transition from deeper levels back to main read-update-loop when using command driven interfaces like USB_HID or UART . This state is not meant to be interacted with as it is for internal purposes only.
BTL_STATUS_IF_READ_TIMEOUT	152	Intermediate state set by command based interfaces when reading has failed due to a timeout. The initial timeout waiting for the first command to be received from a "loader" application is not reported as a timeout using this state as it is internally handled and signalled using the BTL_STATUS_ABORT_UPDATE_CHECK_FW status once the initial wait for interface ready timeout expires.
BTL_STATUS_UPDATING_TEST	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.
BTL_STATUS_UPDATING	160	Update in progress.
BTL_STATUS_VERIFY_SIGNATURE	170	Signature verification in progress.
BTL_STATUS_UPDATE_SUCCESS	180	Update succeeded, new firmware will be started.
BTL_STATUS_REBOOTING	195	Reboot for peripheral reset.
BTL_STATUS_UPDATE_FAILED	210	Update failed.
BTL_STATUS_ABORT_UPDATE_CHECK_FW	239	No update found, abort updating and check if the old firmware is valid.

Symbolic name	Value	Description
BTL_STATUS_ABORT_UPDATE_START_FW	240	No update found, abort updating and try to start old firmware.
BTL_STATUS_ABORT_UPDATE_NO_FW	242	No update found, aborted updating but no (valid) firmware to start.

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
BTL_CPU_StartApplication()	Starts the application by configuring the vector table offset register to point to the firmware and loading stackpointer and PC with appropiate firmware values.
BTL_CPU_DisableInt()	Disables all interrupts on global level.
BTL_CPU_EnableInt()	Enables all interrupts on global level.

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 appropiate firmware values.

Prototype

```
void BTL_CPU_StartApplication ( PTR_ADDR BaseAddr );
```

Parameters

Parameter	Description
BaseAddr	in Base address of the firmware.

Additional information

This routine is often written in assembler.

5.1.1.2 BTL_CPU_DisableInt()

Description

Disables all interrupts on global level.

Prototype

```
void BTL_CPU_DisableInt ( void );
```

5.1.1.3 BTL_CPU_EnableInt()

Description

Enables all interrupts on global level.

Prototype

```
void BTL_CPU_EnableInt ( void );
```


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 24 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
BTL_IF_FS_MMC_SD_Exit()	This routine is a generic routine for de-initialization of the interface.
BTL_IF_FS_MMC_SD_Init()	This routine is a generic routine for initialization of the interface.
BTL_IF_FS_MMC_SD_IsInterfaceReady()	This routine is a generic routine for checking the current status of the interface.
BTL_IF_FS_MMC_SD_Read()	This routine is a generic routine for reading data from the interface.
BTL_IF_FS_MMC_SD_Reset()	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
pData	Pointer to buffer where data should be read into.
NumBytes	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 24 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_Loader`.exe with the switch "-update". See *UART Loader* on page 26 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
BTL_IF_UART_ConfigDelayBeforeExit()	This routine configures a delay to wait before the UART hardware gets de-initialized.
BTL_IF_UART_ExecIF()	Triggers the interface command execution loop to wait for further commands.
BTL_IF_UART_Exit()	This routine is a generic routine for de-initialization of the interface.
BTL_IF_UART_Init()	This routine is a generic routine for initialization of the interface.
BTL_IF_UART_IsInterfaceReady()	This routine is a generic routine for checking the current status of the interface.
BTL_IF_UART_OnRx()	This routine is a generic routine for receiving and storing bytes.
BTL_IF_UART_OnTx()	This routine is a generic routine for sending the next byte from the output buffer.
BTL_IF_UART_Reset()	This routine is a generic routine for resetting the interface for re-reading the update.
BTL_IF_UART_SetAPI()	Sets the API to use for UART communication.

7.2.1 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.

7.2.2 BTL_IF_UART_ExecIF()

Description

Triggers the interface command execution loop to wait for further commands.

Prototype

```
int BTL_IF_UART_ExecIF(void);
```

Return value

	1 - Timeout
0	OK Other - Error

7.2.3 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);
```


7.2.4 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);
```


7.2.5 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.

7.2.6 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.

7.2.7 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.

7.2.8 BTL_IF_UART_Reset()

Description

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

Prototype

```
int BTL_IF_UART_Reset(void);
```

Return value

= 0	OK.
≠ 0	Error.

7.2.9 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 .

7.3 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
pfWrite1	Callback to send out the next byte.
pfEnDisable-Transmitter	Callbacks that enables/disables a transmitter chip (typically used for RS485).

Chapter 8

Updating via USB Bulk

This chapter provides descriptions for using emLoad for USB Bulk.

8.1 Update Procedure with USB Bulk

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 Bulk interface, the "PrepareFW[PRO]" tool has to be used for preparation of the firmware. See *PrepareFW / PrepareFWPRO* on page 24 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_BULK_Loader`.exe with the switch "-update". See *USB Bulk Loader* on page 28 for more information on how to use this tool. emLoad is now updating and starting the firmware afterwards.

8.2 USB Bulk interface API

The table below lists the available API functions:

Function	Description
BTL_IF_USB_BULK_ConfigDelayAfterDetach()	This routine configures a delay to wait after detach from host.
BTL_IF_USB_BULK_ExecIF()	Triggers the interface command execution loop to wait for further commands.
BTL_IF_USB_BULK_Exit()	This routine is a generic routine for de-initialization of the interface.
BTL_IF_USB_BULK_Init()	This routine is a generic routine for initialization of the interface.
BTL_IF_USB_BULK_IsInterfaceReady()	This routine is a generic routine for checking the current status of the interface.
BTL_IF_USB_BULK_Read()	This routine is a generic routine for reading data from the interface.

8.2.1 BTL_IF_USB_BULK_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_BULK_ConfigDelayAfterDetach(U32 ms);
```

Parameters

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

8.2.2 BTL_IF_USB_BULK_ExecIF()

Description

Triggers the interface command execution loop to wait for further commands.

Prototype

```
int BTL_IF_USB_BULK_ExecIF(void);
```

Return value

	1 - Timeout
0	OK Other - Error

8.2.3 BTL_IF_USB_BULK_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_BULK_Exit(void);
```


8.2.4 BTL_IF_USB_BULK_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_BULK_Init(void);
```


8.2.5 BTL_IF_USB_BULK_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_BULK_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.

8.2.6 BTL_IF_USB_BULK_Read()

Description

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

Prototype

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

Parameters

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

Return value

= 0 OK.
≠ 0 Error.

8.3 Interface specific data structures

8.3.1 Structure BTL_USB_BULK_PACKET

Description

This structure holds USB Bulk transmissions.

Prototype

```
struct {
  U32 NumBytes;
  U8  acData[508];
} BTL_USB_BULK_PACKET;
```

Member	Description
NumBytes	Number of bytes of payload in the package.
acData	Buffer for the packet.

Chapter 9

Updating via USB HID

This chapter provides descriptions for using emLoad for USB HID.

9.1 Update Procedure with USB HID

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 USB HID interface, the "PrepareFW[PRO]" tool has to be used for preparation of the firmware. See *PrepareFW / PrepareFWPRO* on page 24 for more information on how to use this tool.

9.1.3 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_Loader`.exe with the switch "-update". See *USB HID Loader* on page 27 for more information on how to use this tool. emLoad is now updating and starting the firmware afterwards.

9.2 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_ExecIF()	Triggers the interface command execution loop to wait for further commands.
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.
BTL_IF_USB_HID_Read()	This routine is a generic routine for reading data from the interface.

9.2.1 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.

9.2.2 BTL_IF_USB_HID_ExecIF()

Description

Triggers the interface command execution loop to wait for further commands.

Prototype

```
int BTL_IF_USB_HID_ExecIF(void);
```

Return value

	1 - Timeout
0	OK Other - Error

9.2.3 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);
```


9.2.4 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);
```


9.2.5 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.

9.2.6 BTL_IF_USB_HID_Read()

Description

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

Prototype

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

Parameters

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

Return value

= 0 OK.
≠ 0 Error.

9.3 Interface specific data structures

9.3.1 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
NumBytes	Number of bytes of payload in the package.
acData	Buffer for the packet.

Chapter 10

Updating via USBH MSD

This chapter provides descriptions for using emLoad for USBH MSD.

10.1 Update Procedure with USBH MSD

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 USBH MSD interface, the "PrepareFW[PRO]" tool has to be used for preparation of the firmware. See *PrepareFW / PrepareFWPRO* on page 24 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 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.

10.2 USBH MSD interface API

The table below lists the available API functions:

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

10.2.1 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);
```


10.2.2 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);
```


10.2.3 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.

10.2.4 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
pData	Pointer to buffer where data should be read into.
NumBytes	Number of bytes to read from interface.

Return value

= 0 OK
≠ 0 Error

10.2.5 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 11

Updating via NAND/NOR

This chapter provides descriptions for using emLoad for NAND.

11.1 Update Procedure with NAND/NOR

11.1.1 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.

11.1.2 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 24 for more information on how to use this tool.

11.1.3 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.

11.2 NAND interface API

The table below lists the available API functions:

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

11.2.1 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);
```


11.2.2 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);
```


11.2.3 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.

11.2.4 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
pData	Pointer to buffer where data should be read into.
NumBytes	Number of bytes to read from interface.

Return value

= 0 OK.
≠ 0 Error.

11.2.5 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.

11.3 NOR interface API

The table below lists the available API functions:

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

11.3.1 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);
```


11.3.2 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);
```


11.3.3 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.

11.3.4 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
pData	Pointer to buffer where data should be read into.
NumBytes	Number of bytes to read from interface.

Return value

= 0 OK.
≠ 0 Error.

11.3.5 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 12

Updating via NFC NTAG

This chapter provides descriptions for using emLoad for NTAG.

12.1 Update Procedure with NTAG

12.1.1 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.

12.1.2 Preparing an emLoad firmware update

For the NTAG interface, the "PrepareFW[PRO]" tool has to be used for preparation of the firmware. See *PrepareFW / PrepareFWPRO* on page 24 for more information on how to use this tool.

12.1.3 Update the firmware using emLoad

Connect the NFC card reader to your PC. Run the `NTAG_Loader`.exe with the switches "-wait" and "-update". See *NTAG Loader* on page 29 for more information on how to use this tool. Place the reader near the NTAG device of the powered down target. The *NTAG Loader* on page 29 should recognize the token. Power on the target. emLoad is now updating and starting the firmware afterwards.

12.2 NTAG interface API

The table below lists the available API functions:

Function	Description
BTL_IF_NTAG_ConfigDelayBeforeExit()	This routine configures a delay to wait before the NTAG hardware gets de-initialized.
BTL_IF_NTAG_ExecIF()	Triggers the interface command execution loop to wait for further commands.
BTL_IF_NTAG_Exit()	This routine is a generic routine for de-initialization of the interface.
BTL_IF_NTAG_Init()	This routine is a generic routine for initialization of the interface.
BTL_IF_NTAG_IsInterfaceReady()	This routine is a generic routine for checking the current status of the interface.
BTL_IF_NTAG_Read()	This routine is a generic routine for reading data from the interface.
BTL_IF_NTAG_Reset()	This routine is a generic routine for resetting the interface for re-reading the update.

12.2.1 BTL_IF_NTAG_ConfigDelayBeforeExit()

Description

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

Prototype

```
void BTL_IF_NTAG_ConfigDelayBeforeExit(U32 ms);
```

Parameters

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

12.2.2 BTL_IF_NTAG_ExecIF()

Description

Triggers the interface command execution loop to wait for further commands.

Prototype

```
int BTL_IF_NTAG_ExecIF(void);
```

Return value

-1	Timeout
0	O.K Other: Error

12.2.3 BTL_IF_NTAG_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_NTAG_Exit(void);
```


12.2.4 BTL_IF_NTAG_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_NTAG_Init(void);
```


12.2.5 BTL_IF_NTAG_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_NTAG_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.

12.2.6 BTL_IF_NTAG_Read()

Description

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

Prototype

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

Parameters

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

Return value

= 0 O.K.
≠ 0 Error.

12.2.7 BTL_IF_NTAG_Reset()

Description

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

Prototype

```
int BTL_IF_NTAG_Reset(void);
```

Return value

= 0	O.K.
≠ 0	Error.

12.3 Interface specific data structures

12.3.1 Structure BTL_NTAG_PACKET

Description

This structure holds NFC NTAG transmissions.

Prototype

```
#define BTL_NTAG_MAX_PAYLOAD_SIZE  (60u)

struct {
  U8  PacketId;
  U8  LRC;
  U16 FlagsPayloadLen;
  U8  acData[BTL_NTAG_MAX_PAYLOAD_SIZE];
} BTL_NTAG_PACKET;
```

Member	Description
PacketId	Packet id range is 1..255 and then rolls over back to 1 again. The packet id is continuous (even between sending pre-verify and real update) . Packets with BTL_NTAG_FLAGS_CONTENT_TYPE_CMD_MASK set the current packet id for the BTL to be able to resend commands. ACKs (with or without payload) are sent back with the packet id + 1 (also rolling over to 1 if the received packet id was 255).
LRC	ISO 1155 Longitudinal Redundancy Check (LRC) over acData[PayloadLen] . Instead of 0 the start value is <PacketId> . Packets from the updater are ACKed with <PacketId + 1> . ACKs can contain a response or have no payload if the command does not expect a reply.
FlagsPayloadLen	bit[10..0] PayloadLen: Fits up to 2047 bytes length which is suitable for future Ethernet usage as well. bit[15..11] Flags : Start from high bits to allow extending PayloadLen if necessary.
acData	Buffer for the packet.

Chapter 13

Debugging

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

13.1 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.

13.1.1 Debug API functions

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

13.1.1.1 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);
```


13.1.1.2 BTL_Panic()

Description

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

Prototype

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


13.1.1.3 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);
```