VECTOR

Flash Bootloader Updater

Technical Reference

Updating the Flash Bootloader to a new version

Version 1.0

Authors	Torben Stößel, Jörn Herwig, Marco Riedl
Status	Released

Document Information

History

Author	Date	Version	Remarks
Torben Stößel Jörn Herwig Marco Riedl	2016-10-20	1.0	Initial version

Contents

Illustrations

Tables

1 Introduction

1.1 Overview

On a regular reprogrammable ECU, an application can be reprogrammed by the Flash Bootloader (FBL). The FBL itself cannot be replaced without an additional software component. The FBL Updater is intended to be downloaded to the ECU as a regular application to replace an existing Flash Bootloader with a newer version.

This document gives an overview of the basic properties and tasks of the Updater.

2 Update process

The general update process is shown in Figure 2-1 .

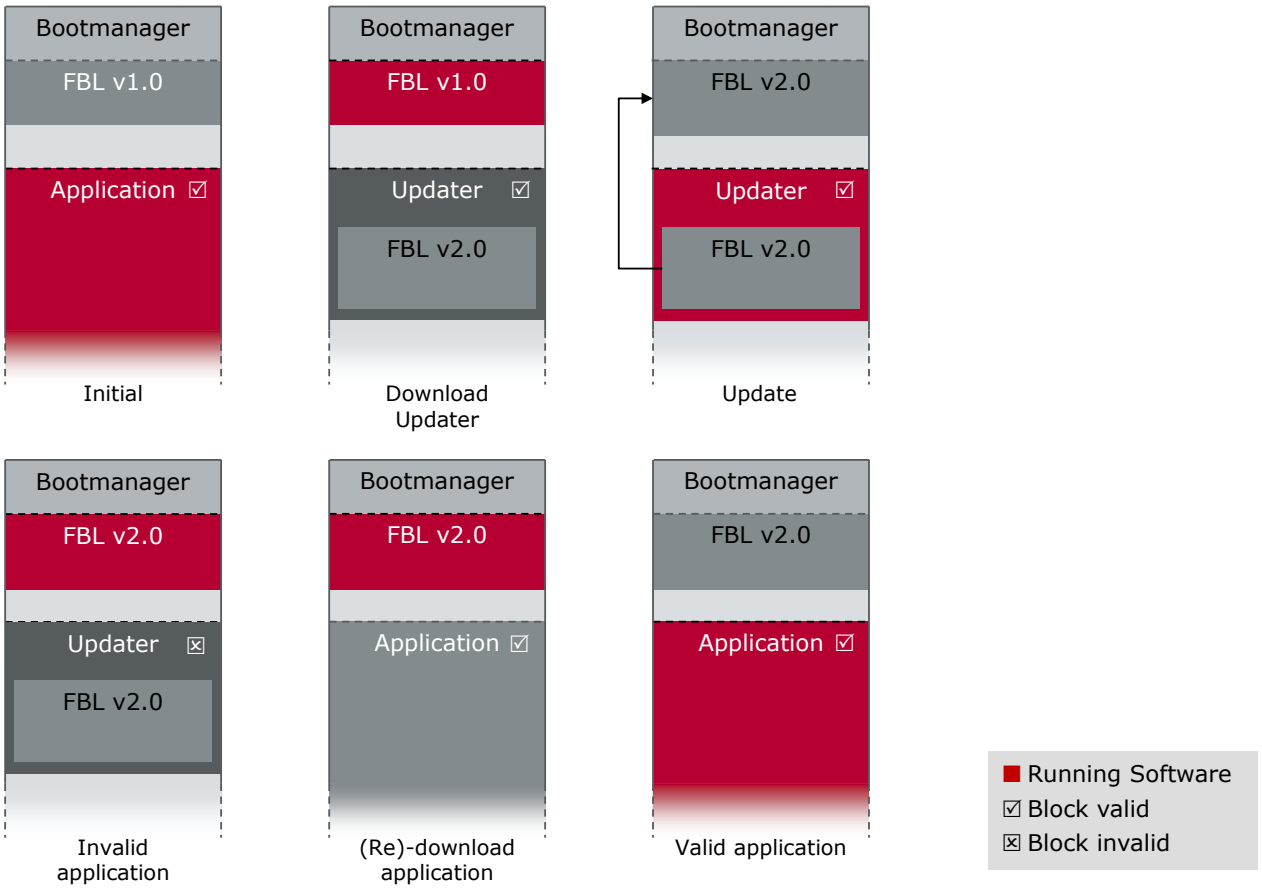

Figure 2-1 Updater concept

Note
The Bootmanager is optional and is mainly relevant, if the Controller cannot be configured to be reset-safe (see chapter 2.2.3).

2.1 Download of Updater

The Updater is a component that is downloaded to the ECU like a regular application. The FBL does not have any knowledge about the Updater and therefor this process is completely transparent. The Updater has to pass the OEM specific validation process (e.g. checksum and/or signature verification) like any other application.

The Updater is started if all OEM conditions are met (e.g. all mandatory blocks present).

In order to avoid potential disturbances, the Updater does not perform a download of the flash driver and the new FBL version. Instead, the flash driver and the new FBL version are part of the Updater.

2.2 Update process

The Updater performs the following steps:

1. Initialization of the communication stack (optional)
2. Initialization of the flash driver
3. Preparing reset safe FBL Updater
4. Erasure of the FBL memory area
5. Programming of new FBL
6. Verification of new FBL
7. Validation of new FBL
8. Invalidation of the FBL Updater (OEM specific)
9. Erasure of the FBL Updater (optional)
10. Reset

If the update process fails at some point, the Updater retries the process until a configurable maximum number of retries exceeded.

2.2.1 Initialization of communication stack

This step is optional. If the Updater has to send Response Pending messages for a certain request to keep the tester alive, the communication stack and the RP message has to be initialized.

Note
The Updater only sends Response Pending messages during the updater process. It does not process any requests to this ECU.

2.2.2 Initialization of the Flash Driver

With this step the flash driver is initialized in a hardware-specific manner. If applicable, the flash programming voltage has to be activated before.

2.2.3 Preparing reset safe FBL Updater

There is more than one approach to make the FBL Updater reset safe.

2.2.3.1 Hardware support

On some hardware platform a reset safe FBL Updater is achievable by using a hardware dependent startup mechanism. Some startup mechanisms are described in the following table:

Hardware	Mechanism
MPC	The PowerPC controller has a so called reset configuration half-word (RCHW) located on specific addresses. The controller evaluates all RCHW addresses of this controller in a specific order. If an RCHW is evaluated as valid, the controller jumps to the entry address defined in the RCHW. The FBL Updater uses this feature to set its own RCHW to a higher priority address or erase the higher priority RCHW of the Bootloader.
RL78	The RL78 uses the boot swap feature. The first 4KB flash block can be changed with the second 4KB. The FBL does occupy the second 4KB flash block, as there the FBL Updater will write own code and configures this 4KB block as the boot block.
RH850	The FBL Updater configures the variable reset vector to the startup code to itself on the RH850 controller. At the end of the update process the FBL Updater changes the reset vector back to the startup of the new FBL.

Table 2-1 Hardware support mechanisms

Caution
For some hardware platforms, no reset-safe configuration can be achieved. If the update process is interrupted e.g. by a power failure or a reset, the ECU is probably no longer reprogrammable. In this case a boot manager is required (see chapter 2.2.3.2).

2.2.3.2 Bootmanager

It is also possible to use a Bootmanager to make the controller reset safe. In this case the Bootmanager is always started first and determines if the FBL should be started or the FBL Updater. This is determined by a Bootmanager header structure which is inside the FBL and the FBL Updater. The FBL Updater is always started, if the structure is available. This means also that this structure needs to be removed at the end of the update process (see chapter 2.2.9).

2.2.4 Erasure of FBL Memory Area

Before the new FBL version can be programmed, the FBL memory area has to be erased. The Updater will only erase the flash blocks which are occupied by the new FBL.

2.2.5 Programming of new FBL version

After erasing the relevant flash blocks, the new FBL version is flashed.

2.2.6 Verification of new FBL

After the flash process has finished, a verification step has to make sure that all bytes of the new FBL version have been programmed correctly. As the updater has been downloaded via the OEM specific download process, the new FBL version has already been checked with the OEM specific verification procedure. Therefore the Updater makes a simple 1:1 comparison of the programmed data against the intended data.

2.2.7 Validation of new FBL

After the FBL Updater confirmed that the new FBL has been written successfully, the FBL Updater writes the validity range (see chapter 4.3.1) into the new FBL. This validity range can be hardware specific (e.g. MPC controller the RCHW, TriCore controller the BMI header) or the Bootmanager header structure.

2.2.8 Invalidation of Updater

The validity status of the Updater is set to the invalid state after the FBL update is finished. The basic goal of this task is to prevent a restart of the Updater after successful programming of the new FBL version.

2.2.9 Erasure of the FBL Updater

Optionally, the Updater can erase itself from flash. This does not necessary mean that the complete FBL Updater memory area is erased, but only the relevant part of the memory which holds the reset safe information.

In case the Bootmanager is used, the header structure for the Bootmanager must be erased. Otherwise the Bootmanager always starts the FBL Updater again.

2.2.10 Reset

At last the Updater makes a reset to start the new Bootloader.

2.3 Reprogramming the Application

As the updater has invalidated itself, the FBL does not jump to the application. Therefore it is necessary to download the real application software to the ECU after the update process finished.

3 Integration

3.1 Scope of Delivery

The delivery of the FBL Updater contains the files, which are described in the following chapters.

3.1.1 Core Files

The first group contains components found in the BSW\FblUpd folder of your delivery. Do not modify the contents of the files in this group without prior written permission from Vector (modification of these files will void your warranty).

File	Description
upd_main.c/h	Updater main component, which provides necessary routine to update the FBL. Furthermore hooks are provided to add additional functionality to the update process.
upd_types.h	Defines public types of the FBL Updater.

Table 3-1 Core files

3.1.2 Application files

The second group consists of components that can be customized for your configuration. The files may be found in the BSW\FblUpd_Template folder. You should copy these files to your FBL Updater project folder and rename them, removing the leading underscore from the filename.

File	Description
_upd_cfg.h	Configuration file for the FBL Updater.
_upd_ap.c/h _upd_ap_cfg.h	Template files which implement the user callouts used in upd_main.c. They may be overwritten by the hardware or OEM specific files.

Table 3-2 General application files

3.1.2.1 OEM specific application files

If available, there are OEM specific FBL Updater files.

File	Description
_upd_oem_ap.c/h	OEM specific template files, which may contain: • OEM initialization • Updater invalidation process • ResponsePending handling
_upd_oem_cfg.h	OEM specific configuration file.

Table 3-3 Hardware specific application files

3.1.2.2 Hardware specific application files

If available, there are hardware specific FBL Updater files.

File	Description
_upd_hw_ap.c/h	Hardware specific template files, which may contain: • Hardware initialization (e.g. PLL) • Set of programming voltage • Activate transceiver • Reconfiguration of reset vector
_upd_hw_cfg.h	Hardware specific configuration file.

Table 3-4 OEM specific application files

3.1.3 Script files

The third group includes scripts to prepare the FBL hex-file to be compiled and linked with the Updater. The scripts can be found in the Misc\FblUpd folder. These scripts may have to be adapted to use the correct paths.

File	Description
FblUpd_Prepare.bat	Script to generate a C-Array file from a hex-file.
FblUpd_Prepare_Hw.bat (optional)	This script is called by the script FblUpd_Prepare.bat before the C-Array is generated to e.g. cut data, remap addresses.

Table 3-5 Script files

Note
It may be possible that there are additional scripts located in the Misc\FblUpd folder. These scripts are used to make the integration of the Updater easier.

3.2 Preparing new FBL version

As described in chapter 2.1, the new FBL version is not downloaded, but included in the Updater as a C-Array.

Practical Procedure

1. Compile and link the new FBL.
2. Convert new FBL into hex-file format.
3. Convert new FBL hex-file into a C-Array (FblUpd_Prepare.bat).
4. Add the C-Array to the Updater.
5. Compile and link the Updater.
6. (optionally) Generate OEM specific download container.

Caution

The byte array of the new FBL has to be aligned to the flash segment size of the particular flash device to achieve correct flash results.

3.3 Flash Driver

The flash driver is stored in the flash memory area of the FBL Updater. Before the update process begins, the flash driver is copied to RAM and executed from there.

The flash driver is encrypted, to avoid an unintentional activation of the flash driver. The decryption of the flash driver is performed, when the driver code is copied to RAM. After the update process has finished, the flash driver code is removed from RAM.

The flash driver code is provided in the files FlashRom.c/h. These files have to be compiled and linked with the FBL Updater. Normally these files are located in the demonstration project of the FBL Updater.

3.4 Updater configuration

The configuration of the FBL Updater with the GenTool (e.g. GENy or DaVinci Configurator) is the same as for the FBL configuration. The main difference is that the flash blocks of the FBL area are changed from “Protected” to the respective memory device (mostly “Flash”). So that the FBL memory area is known to the FBL Updater and can use it for reprogramming.

4 API Description

In this chapter only the configurable hooks and callouts are described, as the FBL Updater is a standalone program and is not called from another module.

The FBL Updater already provides a default implementation, which applies to most cases (upd_main). For more information about the default implementation, please refer to chapter 4.3. These functions can be overwritten by e.g. the hardware specific component (upd_hw_ap) by redefining the hook or callout to a custom function.

4.1 Hook

There are several so called “hooks”, which are called from the updater main component, to add additional functionality to the update process (e.g. send ResponsePending messages).

The hooks can be split into two groups. The first group contains the processing hooks and the second group can be used to adapt the segment handling.

4.1.1 Processing hooks

The processing hooks are used to influence the update process at specific update process steps.

The following table summarizes all update process steps, which are performed by the Updater in the sequence listed below. It specifies if there is a default implementation available (see chapter 4.3) and if there is a possibility to overwrite the step with a custom function. For more information about the overwrite hooks, please refer to the chapter 4.1.1.1.

Update Process Step[1]	Default	Overwrite Hook
Updater Initialization		FBL_UPD_HOOK_INIT
Initialization of Response Pending	■	FBL_UPD_HOOK_SEND_RESPONSE
Initialization of device drivers	■	
Pre-erase	■[2]	FBL_UPD_HOOK_PREPARE_ERASE
Erase	■	
Post-erase		FBL_UPD_HOOK_FINALIZE_ERASE
Pre-program		FBL_UPD_HOOK_PREPARE_PROGRAM
Program	■	
Post-program		FBL_UPD_HOOK_FINALIZE_PROGRAM
Verify	■	
Post-verify	■[2]	FBL_UPD_HOOK_FINALIZE_VERIFY

[1] All mentioned steps are described in detail in chapter 2.2.
[2] Only if validity range are enabled (see chapter 4.3.1)

Update Process Step[1]	Default	Overwrite Hook
Invalidate updater	■[3]	FBL_UPD_HOOK_INVALIDATE_UPDATER
Perform reset	■	FBL_UPD_HOOK_RESET

Table 4-1 Update process steps

4.1.1.1 Function API

Prototype	
`tFblResult FBL_UPD_HOOK_INIT( void )`	
Return code	
`tFblResult`	kFblOk on success, otherwise kFblFailed
Functional Description	
First hook function of the update process.	

Prototype	
`tFblResult FBL_UPD_HOOK_SEND_RESPONSE( void )`	
Return code	
`tFblResult`	kFblOk on success, otherwise kFblFailed
Functional Description	
Send initial response pending (RCR-RP) message.	
Particularities and Limitations	
> In case sending initial response pending failed it may be wiser to return kFblOk instead of kFblFailed. So that at least the update process can continue and will not be aborted.	

Prototype	
`tFblResult FBL_UPD_HOOK_PREPARE_ERASE( void )`	
Return code	
`tFblResult`	kFblOk on success, otherwise kFblFailed
Functional Description	
This is the first hook function with activated flash driver.	
Particularities and Limitations	
> The default hook implementation erases all flash blocks which are included in the validity range. For more information about the validity range, please refer to chapter 4.3.1.	

[3] Only available if respective OEM has this feature enabled.

Prototype
`tFblResult FBL_UPD_HOOK_FINALIZE_ERASE( void )`
Return code
`tFblResult`	kFblOk on success, otherwise kFblFailed
Functional Description
Hook function which is called after the Updater has erased the Bootloader memory area.

Prototype
`tFblResult FBL_UPD_HOOK_PREPARE_PROGRAM( void )`
Return code
`tFblResult`	kFblOk on success, otherwise kFblFailed
Functional Description
Hook function is called before the actual program happens.

Prototype
`tFblResult FBL_UPD_HOOK_FINALIZE_PROGRAM( void )`
Return code
`tFblResult`	kFblOk on success, otherwise kFblFailed
Functional Description
Hook function is called after the Bootloader has been programmed.

Prototype
`tFblResult FBL_UPD_HOOK_FINALIZE_VERIFY( void )`
Return code
`tFblResult`	kFblOk on success, otherwise kFblFailed
Functional Description
Hook function when the verification was successful.
Particularities and Limitations
> The default hook implementation programs the validity range at this point. For more information about the validity range, please refer to chapter 4.3.1.

Prototype
`tFblResult FBL_UPD_HOOK_INVALIDATE_UPDATER( void )`
Return code
`tFblResult`	kFblOk on success, otherwise kFblFailed

Functional Description
Hook function to invalidate the updater/application by clearing the valid flag/pattern.

Prototype	
`tFblResult FBL_UPD_HOOK_RESET( void )`	
Return code	
`tFblResult`	kFblOk on success, otherwise kFblFailed
Functional Description	
Last callout function in the update process. This hook function shall perform a reset to start the new FBL.	
Particularities and Limitations	
> This function shall not return.	

4.1.2 Segment handling hooks

The updated Bootloader may consist of multiple segments.

The segment handling hooks are used to iterate over those segments. Depending on the requirements of the programming/erasing/verification routine further modifications may be applied to the original segment range. An example for this could be exclusion of a specific memory range.

4.1.2.1 Function API

Prototype	
`tFblResult FBL_UPD_HOOK_GET_SEGMENT( vuintx index, V_MEMRAM1 tFblUpdSegmentInfo V_MEMRAM2 V_MEMRAM3 * pSegment )`	
Parameter	
`index`	Index of referenced segment
`pSegment`	Pointer resulting segment information
Return code	
`tFblResult`	kFblOk when referenced segment is available, otherwise kFblFailed
Functional Description	
Get segment of new FBL referenced by given index.	

Prototype	
`tFblResult FBL_UPD_HOOK_ADJUST_SEGMENT_PROGRAM( V_MEMRAM1 tFblUpdSegmentInfo V_MEMRAM2 V_MEMRAM3 * pSegmentList, V_MEMRAM1 vuintx V_MEMRAM2 V_MEMRAM3 * pSegmentCount )`	
Parameter	
`pSegmentList`	Pointer to segment list, first entry contains input segment Input: Size of segment list

pSegmetnCount	Output: Number of resulting segments
Return code	
tFblResult	kFblOk when segment adjustment was successful, otherwise kFblFailed
Functional Description	
Adjust segments for programming / verification operation. Skip the area occupied by the validity range.	
Particularities and Limitations	
> Validity range doesn't cross segment border.	

Prototype	
`tFblResult FBL_UPD_HOOK_ADJUST_SEGMENT_VALIDITY( V_MEMRAM1 tFblUpdSegmentInfo V_MEMRAM2 V_MEMRAM3 * pSegmentList, V_MEMRAM1 vuintx V_MEMRAM2 V_MEMRAM3 * pSegmentCount )`	
Parameter	
pSegmentList pSegmetnCount	Pointer to segment list, first entry contains input segment Input: Size of segment list Output: Number of resulting segments
Return code	
tFblResult	kFblOk when segment adjustment was successful, otherwise kFblFailed
Functional Description	
Adjust segments for validation operation. Only return a potential validity range lying within the segment bounds.	

4.2 Callout

Furthermore the updater provide callouts to handle hardware or OEM specific requirements.

Prototype
`void FBL_UPD_CALLOUT_INIT_POWER_ON( void )`
Functional Description
Initialization callout function.

Prototype
`void FBL_UPD_CALLOUT_INIT_POWER_ON_HW( void )`
Functional Description
Hardware specific initialization callout function.

Prototype
`void FBL_UPD_CALLOUT_INIT_POWER_ON_OEM( void )`
Functional Description
OEM specific initialization callout function.

Prototype
`void FBL_UPD_CALLOUT_SET_VFP( void )`
Functional Description
Callout to enable programming voltage.

Prototype
`void FBL_UPD_CALLOUT_RESET_VFP( void )`
Functional Description
Callout to disable programming voltage.

Prototype
`void FBL_UPD_CALLOUT_RESET( void )`
Functional Description
Callout to perform a reset.

Prototype	
`tFblResult FBL_UPD_CALLOUT_CHECK_RESPONSE_PENDING( void )`	
Return code	
`tFblResult`	kFblOk if FBL Updater should send RCR-RP, otherwise kFblFailed
Functional Description	
Callout function to check if response pending message is required.	

Prototype	
`void FBL_UPD_CALLOUT_PREPARE_RESPONSE_PENDING( V_MEMRAM1 vuint8 V_MEMRAM2 V_MEMRAM3 * pResponse )`	
Parameter	
`pResponse`	Pointer to response pending buffer.
Functional Description	
Callout function to prepare the response pending message buffer.	

Particularities and Limitations
> By default the size of the message buffer is 3 bytes.

4.3 Default Implementation

For some of the hook functions the Updater already provides default implementations. Those implementations are automatically used, if a specific define is set/used.

4.3.1 Validity Range

The validity range is used to mark a special range. This range is considered throughout the update process. The range is given with `FBL_UPD_VALIDITY_RANGE_ADDR` and `FBL_UPD_VALIDITY_RANGE_SIZE`.

Note
Only one validity range is supported by the default implementation.

The given range is erased first (see FBL_UPD_HOOK_PREPARE_ERASE) and programmed last after successful verification (see FBL_UPD_HOOK_FINALIZE_VERIFY).

This mechanism is used to make sure that data, which determines whether the Bootloader is started instead of the Updater, isn't programmed until the update was successfully finished. This prevents that in case of an unexpected reset an incomplete Bootloader is started. Instead the Updater is executed again and a retry of the update is performed.

4.3.2 Response Pending

The default implementation to send the response pending message checks if a response pending is needed (see FBL_UPD_CALLOUT_CHECK_RESPONSE_PENDING), initializes the communication stack and prefills the send message buffer (see FBL_UPD_CALLOUT_PREPARE_RESPONSE_PENDING).

4.3.3 Reset

After the Updater has invalidated itself, the Updater makes a reset to start the new Bootloader. To trigger a reset a callout function (see FBL_UPD_CALLOUT_RESET) is called.

Furthermore, it is possible to adapt the Updater, so that the Updater will erase itself before triggering a reset. This requires that all functions/constants, which are used to erase and trigger the reset, to be located in RAM.

5 Glossary and Abbreviations

5.1 Abbreviations

Abbreviation	Description
ECU	Electronic control unit
FBL	Flash Bootloader
OEM	Original Equipment Manufacturer (here: Car Manufacturer)
RCR-RP	Response Correctly Received – Response Pending
RCHW	Reset Configuration Half-Word
BMI	Boot Mode Index

6 Contact

Visit our website for more information on

- News
- Products
- Demo software
- Support
- Training data
- Addresses

www.vector.com