VECTOR

Flash Bootloader Validation Strategies

2023-08-31
Support Note SN-IES-1-007

Author(s) Stolz, Thomas

Table of Contents

1 Overview

This Support Note describes different validation strategies in the field of application and logical blocks.

2 Introduction

In general, the validation process takes place after a logical block has been verified successfully by the Flash Bootloader.

Whereas the verification ensures data integrity and data authenticity the validation stores the logical result of the verification process. In simple terms when the verification is successful the logical block is set to valid. If not, the logical block stays invalid.

There are several approaches to handle the logical block and application validation. It is up to the Flash Bootloader and Boot Manager architecture and specification which approach shall be used. Two common options are available:

1. Presence Patterns
2. Valid Flags

Whereas Presence Patterns are usually written to the end of each logical block the valid flags for block, and application validity are programmed to data flash or EEPROM by means of having EA or FEE involved. Both approaches have in common that the information must be stored in a non-volatile memory to persist the validity permanently and reset safe.

The Flash Bootloader uses application functions to control and check the block and application validity no matter which validation process is used. The Boot Manager uses application functions as well but only to check the block and application validity. An overview is listed in chapter 3.3.

The application functions contain a default implementation which can be adapted to be compliant to OEM specifications accordingly.

3 Block and Application Validity

This chapter distinguishes between the validity of a single logical block and the application validity, also known as the disposability of the application. This does not necessarily mean that all OEMs use both block and application validity. Some may use just one or the other.

The validity information only has two states, either it is valid or invalid.

3.1 Block Validity

The block validity is used to mark a single logical block to either valid or invalid.

To change the validity information of a logical block the function `ApplFblValidateBlock()` and `ApplFblInvalidateBlock()` are called in defined steps during the download procedure. A descriptor of the logical block of which the validity information shall be changed is passed to the functions. To check whether a logical block is valid or not the function `ApplFblIsValidBlock()` can be called. A detailed function description can be retrieved from the document `TechnicalReference_FBL_<OEM>_SLP<x>.pdf` normally shipped with the FBL delivery.

3.2 Application Validity

In order to find out whether the application of an ECU can be started or not the application validity information is used. The application validity depends on the validation status of the mandatory logical blocks and is only set to valid when all mandatory logical blocks are valid. Logical blocks which are

marked as optional are not considered for the application validity. When one mandatory logical block is invalid the application validity is invalid and the start of the application is prohibited.

The handling of the application validity information is highly OEM dependent. Specified normally is, in which step of the download procedure the application validity is set to valid or invalid, e.g. in a ‘check verification request’ or ‘check programming dependency request’.

Even the location of the application validity may be specified by the OEM or pre-implemented by the Flash Bootloaders.

All variants and OEMs follow the same validation rule: The application validation is set to valid when all mandatory blocks are downloaded and verified successfully.

The following application validation strategies exist when presence patterns are enabled and may vary from OEM to OEM:

- The application validity is stored to the last valid downloaded logical block
- The application validity is stored to the last valid downloaded mandatory logical block
- The application validity is not stored. Instead each time the application validation status is requested, e.g. when the Boot Manager wants to start the application, the block validity status of all mandatory blocks is checked.

Figure 1 Application Validity States depicts different block and application validity states of a Flash Bootloader example configuration, which consists of two logical blocks. The LB#1’s disposability is set to **mandatory** and LB#2’s is **optional** with enabled presence pattern (PP). The states do not necessarily follow one after the other. In other words, the state order does not depict the download procedure.

Figure 1 Application Validity States

State 1: This is the initial state of the logical blocks when no data has been downloaded to any of the available logical blocks. Block and application presence patterns are erased, the application cannot be started.

State 2: The data for the first logical block LB#1 is downloaded and the verification has been performed successfully. The block presence pattern is set to valid for this block. Because the block

validity information for all mandatory blocks is valid the application validity is set to valid as well. This can be seen in the application presence pattern area of LB#1. LB#2 is untouched and therefore contains the erased pattern state for block and application presence pattern.

State 3: Only the LB#2 is downloaded, the block presence pattern is set to valid whereas the application presence pattern remains erased because not all mandatory logical blocks are downloaded.

State 4: Having **State 3** as base, LB#1 will be downloaded afterwards in a separate downloaded procedure. After the successful verification of LB#1 the block and application presence pattern are set to valid.

State 5: LB#1 and LB#2 are downloaded in sequence. At first LB#1 and then LB#2. The Flash Bootloader is implemented in a way that the application validity information is always written to the last successfully downloaded logical block. Therefore, the application presence pattern of LB#2 is set to valid although it is not the mandatorily defined block.

State 6: Depending on the OEM specification it may happen that whenever a (mandatory) logical block is downloaded all configured logical blocks will be erased before new data is downloaded to the ECU. Therefore, the logical blocks must be invalidated before the erase routine starts the erasure of the flash memory. State 6 depicts the state before the erase routine erases the flash memory. All presence patterns which were set to valid beforehand are set to invalid. Presence patterns which contain the erased value remain erased.

3.3 Overview of Involved Application Functions

Functions which are used by the Flash Bootloader in order to control and check the validity:

- `ApplFblValidateBlock()`
  - Will set the block validity to valid
- `ApplFblInvalidateBlock()`
  - Will set the block validity to invalid
- `ApplFblIsValidBlock()`
  - Checks whether the passed logical block is either valid or invalid
- `ApplFblValidateApp()`
  - Will set the application validity to valid
- `ApplFblInvalidateApp()`
  - Will set the application validity to invalid
- `ApplFblIsValidApp()`
  - Checks whether the application validity is valid or invalid

Functions which are used by the Boot Manager in order to check the validity information:

- Macro `FBLBM_CALLOUT_IS_VALIDBLOCK / ApplFblBmIsValidBlock()`
  - Like `ApplFblIsValidBlock`, checks whether the block validity is valid or invalid.
  - Macro can be redefined in bm_ap_cfg.h to another function
- Macro `FBLBM_CALLOUT_CHECK_TARGET_VALIDITY / ApplFblBmCheckTargetValidity()`
  - Like `ApplFblIsValidApp`, checks whether the application validity is valid or invalid.
  - Macro can be redefined in `bm_ap_cfg.h` to another function

4 Validation Mechanisms

4.1 Presence Pattern

Instead of using flags in EEPROM or FEE, the validation status can be written into the flash memory of the respective logical block. Due to the fact that a flash cell cannot be programmed twice, two segments are used to mark the logical block as valid or invalid. The handling of presence patterns is often optional and can be activated in the configuration tool.

The validation and invalidation is handled with two patterns at the end of each logical block: the presence pattern value and the presence pattern mask, see Figure 2 Presence Pattern Structure. The mask and the pattern have a size of at least two bytes.

The location of these values is reserved in the logical block to avoid that the application overwrites these locations.

The reason for having two pages for each presence pattern, the value and the mask, is because most flash manufacturers do not allow writing to a flash cell two times without a preceding erase. The validation/invalidation concept is based on a conjunction of the erased status of the mask and the presence pattern value.

Figure 2 Presence Pattern Structure

4.1.1 Workflow of a Presence Pattern

The example in Figure 3 depicts the validation and invalidation flow during a typical download procedure. Be aware that the exact procedure may vary between OEMs.

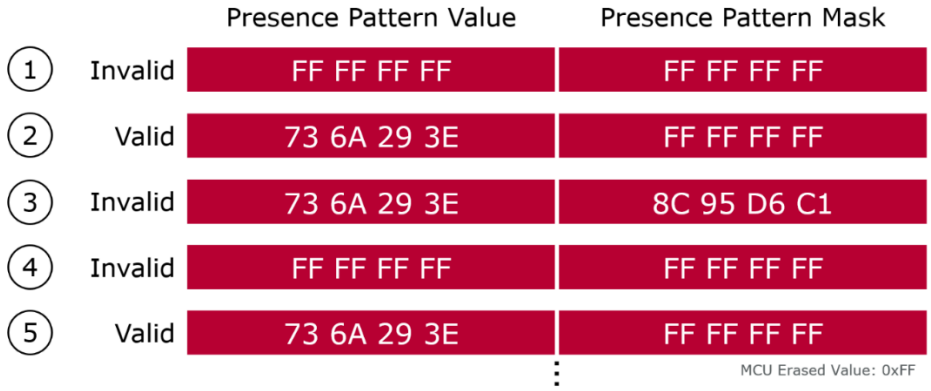

Figure 3 Block Validity Presence Pattern Example

1. Initially the logical block is erased therefore the presence pattern area is also erased.
2. Once the logical block has been downloaded and verified successfully the function `ApplFblValidateBlock` is invoked in order to write the presence pattern value to the logical block and check the define `kFblNvMarkerValue` in the implementation.
3. When the tester or rather the Flash Bootloader flashes a logical block again it will invalidate the logical block before the logical block is erased. The purpose of the invalidation is to mitigate issues that may occur during erasure, e.g. a power drop which keeps the presence pattern valid although half of the logical block was erased before the issue occurred. To invalidate a logical block the function ApplFblInvalidateBlock is invoked and the presence pattern mask is written, check the define `kFblNvMaskValue`.
4. During the erase of the complete logical block the presence pattern and mask are erased again like in step 1.

5. Like in step 2 the presence pattern value can be written again due to the prior erase of the presence pattern area.

4.1.2 Presence Pattern Area in Detail

When the presence pattern feature is configured in the Flash Bootloader more than one presence pattern can be used for each logical block. As described in chapter 3 there are usually two sets of validity information stored in dedicated presence patterns in the logical block validation area.

Depending on the Flash Bootloader's OEM specification there may be more than the block and application validity presence pattern available for each logical block. Additionally, there can be a presence pattern available to indicate whether a logical block is erased or not, as can be seen in Figure 4 ErasedState.

To check which presence patterns are available in the Flash Bootloader the structure `tFblNvPatternId` in the header file `fbl_nvpattern_oem.h` should be checked. Be aware that this structure is *only* available when the Flash Bootloader supports presence patterns and the component NvPattern is available in the SIP.

In the following section the MCU RH850 F1L is used as an example to show where the different presence pattern values and masks are located in detail. This will help to find the presence patterns accordingly.

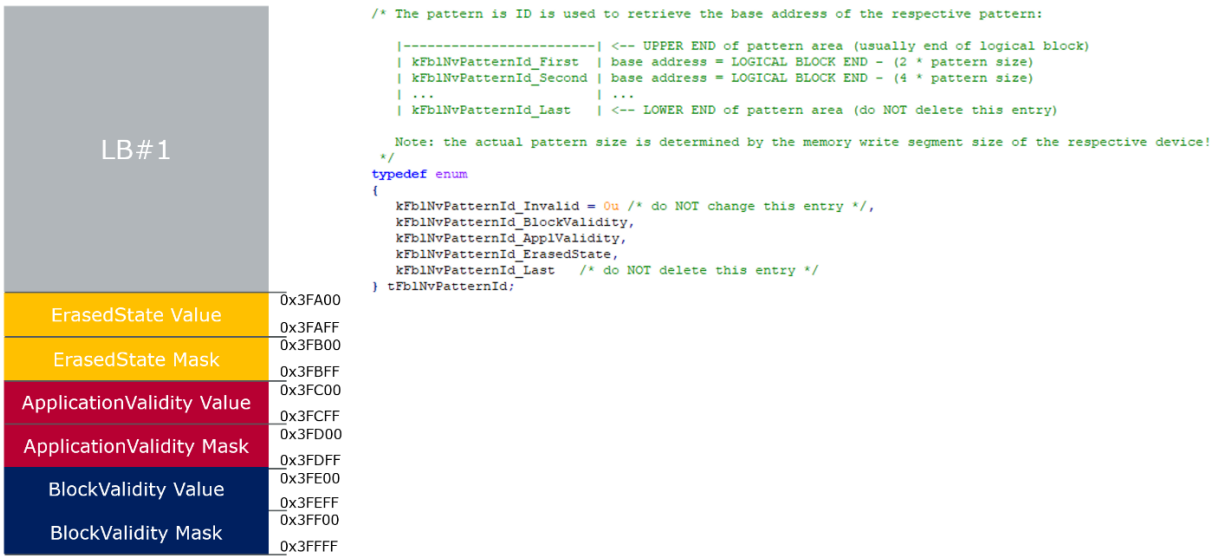

Figure 4 Presence Patterns Example

As shown in Figure 4 three different presence patterns are placed in the Flash Bootloader's logical block which correspond to the example enumeration `tFblNvPatternId`. The presence pattern length is calculated by 2 times the flash page size of the memory device but at least 2 or rather 4 bytes depending on the Flash Bootloader's presence pattern size, see define `kFblNvPatternSize` or `kFblPresencePatternSize` in the implementation.

Since the RH850 F1L platform has a flash page size[1] of 0x100 bytes the minimum length of one presence pattern is 2 times 0x100 bytes which results in a presence pattern length of 0x200 bytes.

[1] The RH850 controller is an extreme example. Usually the program flash page size of controllers varies between 0x4 to 0x20 bytes. When the flash page size is smaller than 0x4 bytes, e.g. for external NOR flash memory devices then the presence pattern value size is as large as the configured presence pattern size (kFblNvPatternSize).

Because three different presence patterns are used in the example's Flash Bootloader the complete memory which is occupied by the presence patterns is 0x600 bytes.

In the example the application validity presence pattern value which is depicted in red in Figure 4 starts at address 0x3FC00 and ends at 0x3FCFF. The application validity presence pattern mask starts at address 0x3FD00 and ends at 0x3FDFF. In HexView the application validity presence pattern of the logical block in state invalid will look like in Figure 5.

```
Block  0  Starts at: 0x3FC00  Ends at: 0x3FDFF (Length: 0x200=512)
0003FC00:  73 6A 29 3E  FF FF FF FF   FF FF FF FF  FF FF FF FF  sj)>............
0003FC10:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FC20:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FC30:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FC40:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FC50:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FC60:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FC70:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FC80:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FC90:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FCA0:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FCB0:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FCC0:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FCD0:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FCE0:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FCF0:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FD00:  8C 95 D6 C1  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FD10:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FD20:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FD30:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FD40:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FD50:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FD60:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FD70:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FD80:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FD90:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FDA0:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FDB0:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FDC0:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FDD0:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FDE0:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
0003FDF0:  FF FF FF FF  FF FF FF FF   FF FF FF FF  FF FF FF FF  ................
```

Figure 5 Invalid Block Validity State

Caution

If a memory area with error recognition or correction (e.g. ECC) is used to store the presence pattern, special care must be taken to make sure that the software can handle the error. If the ECU reacts e.g. with a reset if an error occurs during reading the pattern, the ECU will be locked because the presence pattern will be read during every start-up.

4.1.3 Memory Consumption of Presence Patterns

The next tables show an example calculation of the different controllers and memory devices and Flash Bootloader setups.

Controller	Renesas RH850 F1L
Flash Page Size	0x100
Presence Patterns	2 (block and application validity)
Calculation	$(\text{amount of presence patterns}) * 2 * (\text{flash page size})$ $= 2 * 2 * 0x100$ $= 0x400$
Result	The presence patterns consume 0x400 (1024 bytes) to store the block and application validity.

Controller	Infineon TriCore TC3xx
Flash Page Size	0x20
Presence Patterns	3 (block and application validity and erase state information)
Calculation	$(\text{amount of presence patterns}) * 2 * (\text{flash page size})$ $= 3 * 2 * 0x20$ $= 0xC0$
Result	The presence patterns consume 0xC0 (192 bytes) to store the block and application validity and erase state information.

Memory Device	Cypress S25FL208K
Flash Page Size	0x1
Presence Patterns	1 (block validity only)
Calculation	$(\text{amount of presence patterns}) * 2 * (\sout{\text{flash page size}}\ \text{presence pattern size})$ $= 1 * 2 * 0x4$ $= 0x8$
Result	The presence pattern consumes 0x8 bytes to store the block validity. Be aware that the presence pattern size is the inflating value and not the flash page size. This is because the flash page size of the external memory device is smaller than the presence pattern size used by the Flash Bootloader.

4.1.4 Configuration

To be able to use presence pattern in the configuration tool `/MICROSAR/Fbl/FblGeneral/FblPresencePattern` needs to be enabled. In the mentioned application functions, e.g. `ApplFblValidateBlock` the code for presence pattern handling will be enabled.

4.1.5 Debug and Integration Hints

Download of Flashware Fails

Does the flashware overlap the presence pattern area? Be aware that the presence pattern area will often cover more than just the last two page sizes of each logical block. When more than just one presence pattern is used, the presence pattern area is expanded and occupies more memory. Check the address and length of the flashware and check if an overlap takes place.

Note

The flashware must omit the presence pattern area.

Application Did Not Start after Successful Download of all Mandatory Blocks

The presence pattern value and mask can easily be checked in the memory view of the debugger when the flashware is downloaded to internal program or data flash. Please check Figure 4 where the presence pattern is expected and Figure 5 for how it shall look. When the application is not started most likely the presence pattern is written to a wrong location or it is not valid. Also check whether Boot Manager and Flash Bootloader are configured equally in that case. Otherwise the Boot Manager may not check the same validity information as the Flash Bootloader.

Check the functions `ApplFblBmIsValidBlock()` in the Boot Manager and `ApplFblValidateBlock()` and `ApplFblValidateApp()` in the Flash Bootloader. Is `ApplFblValidateApp()` even called?

4.1.6 Advantages and Disadvantages

The usage of the presence pattern feature has advantages and disadvantages. Presence pattern can be easily used on flash memory devices due to the pre-implementation in the application. This applies for internal program and data flashes and for external NOR flash memory devices. A certain amount of memory is necessary to store the presence patterns depending on the controller's page size and the quantity of presence patterns to be used.

Brief summary in which advantages are marked by '+' and disadvantages are marked by '-':

+ Easy to use due to pre-implementation in most Flash Bootloaders
+ Easy to read in memory view
+ Support for internal and external flash memory devices which support address-based access
+ Fast startup time since memory stack does not need to be initialized to check validity information
- Memory consumption may be high depending on the controller's page size
- Loop over all logical blocks necessary in order to find application validity

4.2 Validity Flags Stored in Non-Volatile Memory

When presence pattern shall not be used to store the validity information of the logical blocks and the application any other NVM can be used to store this information.

Usually the underlying NV-Memory is a flash memory or EEPROM and will be accessed via the memory abstraction layers Fee or Ea. When the storage of validity information in NV-Memory is desired normally the so called WrapNv module comes with the Flash Bootloader. The module handles the access to NvM, Fee, Ea or EEPROM directly, as can be seen in Figure 6. Detailed information can be obtained from the Technical Reference of the WrapNv module shipped with the FBL SIP when it is part of the delivery.

When supported by the Flash Bootloader (and Boot Manager) the validity flag handling is pre-implemented in the functions listed in chapter 3.3.

All those functions will then access the NV-Memory with the macros `ApplFblNvRead*` or `ApplFblNvWrite*`, defined in the header file `WrapNv_Cfg.h`. Be aware that the NV-Memory accesses are synchronous. Be aware also that the functions `FblRealTimeSupport()` or `FblLookForWatchdog()` will be called during NV-Memory access.

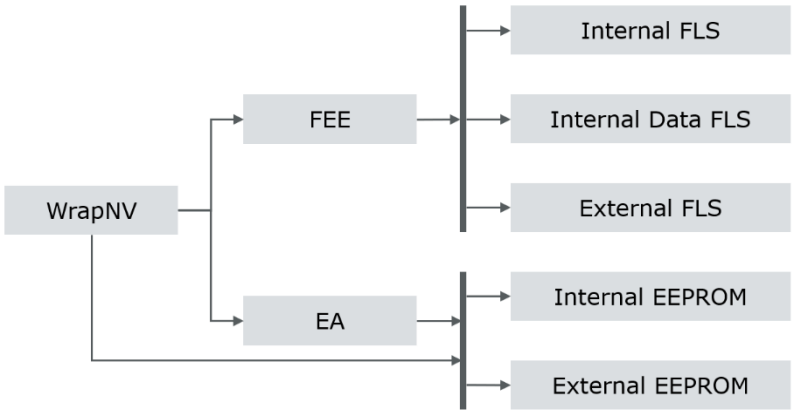

Figure 6 Flash Bootloader NV-Memory Stack

4.2.1 Configuration

To be able to use the storage of validity flags in non-volatile memory the presence pattern has to be disabled in the configuration tool `/MICROSAR/Fbl/FblGeneral/FblPresencePattern`. Additionally, the memory stack has to be integrated and configured accordingly. The WrapNv module needs to be able to access either FEE, EA or EEPROM directly and synchronously.

For more information please refer to the document `TechnicalReference_NvWrapper.pdf` if available in the FBL delivery.

4.2.2 Advantages and Disadvantages

Brief summary in which advantages are marked by '+' and disadvantages are marked by '-':

+ Application validity is only stored in one place, no loop over all logical blocks necessary
+ Support for any memory device which can be attached to the memory stack
- Integration of memory stack necessary (if not already available)
- Boot Manager also needs the memory stack when it is not compiled with the Flash Bootloader[2]
- Depending on the hardware platform an external memory device may be necessary
- Startup time is increased because the memory stack needs to be initialized and running in order to check validity information
- Not easy to read for integrator when external memory stores validity information or FEE/EA is used

4.3 Customer Specifics

Customers of Flash Bootloaders are totally free to adapt the functions listed in chapter 3.3 to their needs in order to implement their own validation strategies or to extend the pre-implemented validation.

[2] The Boot Manager needs the memory stack in order to read out the validity information of the Flash Bootloader and the targets in order to decide which instance shall be started.

5 Validity Handling with the Boot Manager

The Boot Manager has to check the validity of the Flash Bootloader and all other targets depending on the configured validity handling. It is recommended that the Boot Manager and the Flash Bootloader are configured in the same way with respect to the validity handling.

Be aware that when the validity information is stored in non-volatile memory like FEE or EA the memory stack has to be integrated with the Boot Manager.

The Boot Manager will never change the validity status of the logical blocks!

5.1 Boot Manager and Presence Pattern

One of the first tasks of the Boot Manager is to check whether the Flash Bootloader is available or not. If the Boot Manager is configured to use the presence pattern it will check whether the Flash Bootloader's block validity presence pattern is valid or not.

Therefore, the Flash Bootloader which is downloaded by the debugger needs a valid block presence pattern at the end of the logical block. It is important that the mask area remains empty otherwise the invalidation of the Flash Bootloader may fail and cause issues during the processing of the Flash Bootloader Updater.

In contrary the Flash Bootloader which comes with the Flash Bootloader Updater does not need the block presence pattern to be valid or even existent. The Flash Bootloader Updater will validate the new version of the Flash Bootloader by itself.

5.2 Boot Manager and Validity Flags

The pre-implementation of the Boot Manager's function `ApplFblBmIsValidBlock()` will not work for all Flash Bootloaders when presence pattern is disabled. The reason is that the Boot Manager expects the Flash Bootloader in a dummy logical block with block number 0 which is not always applicable.

One workaround is to consider the parameter `targetHandle` in function `ApplFblBmIsValidBlock()` and store the validity information of the Flash Bootloader separately in non-volatile memory.

Another workaround would be to still use the presence pattern of the Flash Bootloader although in the Boot Manager the presence pattern feature is disabled. Then chapter 5.1 shall be considered.

For all other targets the configured validity handling happens in the usual pre-implemented way.

6 Appendix

6.1 Glossary

Error Checking and Correction (ECC)	Detects bit errors in the memory and may correct them.
EEPROM Abstraction (EA)	Memory abstraction layer for EEPROM
Flash EEPROM Emulation (Fee)	Memory abstraction layer which operates emulates EEPROM accesses on flash memory.
Flashware	Any downloadable content.
Presence Pattern (PP)	Validity Pattern in NV-Memory

7 Contacts

For support related questions please address to the support contact for your country https://www.vector.com/int/en/company/contacts/support-contact/.