

Flash Bootloader OEM

Technical Reference

FCA UDS (SLP5)

Version 0.9

Authors	Christian Bäuerle
Status	Not Released

Document Information

History

Author	Date	Version	Remarks
Christian Bäuerle	2018-07-12	0.9	Creation

Reference Documents

No.	Source	Title	Version
[1]	ISO	14229 Road Vehicles – Unified diagnostic services (UDS) Part 1: Specification and Requirements	2013
[2]	ISO	15765 Road Vehicles – Diagnostics on CAN Part 3: Implementation of Unified Diagnostic Services	2004
[3]	Vector	AN-ISC-8-1188 – Custom Flash Drivers	1.0
[4]	HIS	Security Module Specification	1.0
[5]	Vector	User Manual Flash Bootloader	2.7
[6]	Vector	Technical Reference HexView	1.12.2
[7]	Vector	Manual vFlash	3.5
[8]	Vector	NV-Wrapper – Technical Reference	1.0
[9]	Vector	AN-ISC-8-1143 – Bootloader Validation Strategies	1.0
[10]	Vector	Technical Reference Security Module / Security Module Basic	2.2.0
[11]	Vector	AN-ISC-8-1188 – Custom Flash Drivers	1.0
[12]	Vector	Technical Reference Communication Wrapper PDU Router	3.2
[13]	FCA	CS.00100 Comprehensive Standard Unified Diagnostic Services (UDS) On CAN	2016-12-07
[14]	FCA	CS.00101 ECU Flash Reprogramming Requirements - UDS	2018-03-23
[15]	FCA	CS.00102 Standardized Diagnostic Data	2016-12-07



Caution

We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector's release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Contents

1	Introduction.....	8
2	Download concept.....	9
2.1	Overview.....	9
3	Flash Bootloader Delivery.....	10
3.1	Bootloader Components	10
3.2	Bootloader Integration.....	10
4	Bootloader Installation	12
4.1	[#oem_files] - Bootloader package structure	12
4.2	Demonstration Bootloader	14
5	Generation Tool (DaVinci Configurator 5).....	15
5.1	Project Setup and Communication Stack Configuration	15
5.2	Bootloader Configuration	16
5.2.1	General Bootloader Options.....	16
5.2.2	OEM Bootloader Options	21
5.2.3	Memory Configuration.....	22
5.2.3.1	Memory Device Table	22
5.2.3.2	Flash Block Table.....	22
5.2.3.3	Logical Block Table	23
5.2.3.4	NV-Wrapper Configuration	24
5.2.4	Security Module Configuration	26
5.2.4.1	Check Routine Without Additional CRC and Without Additional Signature.....	26
5.2.4.2	Check Routine with additional CRC32 and without additional Signature	29
5.2.4.3	Check Routine without additional CRC32 and with additional Signature	30
5.2.4.4	Check Routine with additional CRC32 and with additional Signature	32
5.3	Running the Generator	32
6	User callback files and hardware specific adaption.....	34
6.1	Startup code	34
6.2	Hardware, Input/Output and miscellaneous Callbacks	34
6.3	Validation and non-volatile memory callbacks	43
6.4	Diagnostic Service Callbacks	53
6.5	Watchdog Callbacks	56

7	Build your Bootloader	58
7.1	Bootloader specific linker requirements.....	58
8	Adapt your application	60
8.1	Memory Layout	60
8.2	Application Start.....	60
8.2.1	Controllers with a configurable interrupt base address	60
8.2.2	Controllers with interrupt jump tables	60
8.3	Shared files between FBL and application	60
8.4	Shared memory between FBL and application	61
8.5	Transition between Bootloader and Application.....	62
8.5.1	[#oem_start] [#oem_trans] - Programming Session Request	62
8.5.2	ECU Reset and Default Session Request	63
9	Memory Drivers	64
9.1	Flash Driver	64
9.2	Non-volatile Memory Driver.....	64
9.3	Memory driver requirements	64
10	Miscellaneous	66
10.1	[#oem_valid] - Application validation	66
10.1.1	Presence Patterns	66
10.1.1.1	Storage of application valid flag	67
10.1.2	[#oem_time] - Validation OK – Application faulty	69
10.2	[#oem_sec] Seed / Key Mechanism	69
10.3	[#oem_seccomp] - Security module	70
10.4	Data Processing Support	70
10.4.1	tProcParam.....	71
10.5	[#oem_multi] - Multiple ECU Support	71
10.6	Pipelined Programming and Pipelined Verification	72
10.7	Bootmanager	73
10.8	FCA-Specific Functionality	73
10.8.1	Download of SWIL / Flash Driver	73
10.8.2	Abnormal Shutdown.....	73
10.8.3	Flash Erase Detection.....	74
10.8.4	Update of Identification Data	74
10.8.5	Update of FCA Identification Data	74
10.8.6	Software Compatibility Information.....	74
11	vFlash Configuration.....	75
12	Glossary and Abbreviations	77

12.1 Glossary 77

12.2 Abbreviations 77

13 Contact..... 78

Illustrations

Figure 1-1	Manual and References for the Flash Bootloader	8
Figure 4-1	Bootloader Folder Structure	13
Figure 5-1	Bootloader Configuration in Basic Editor	16
Figure 5-2	General Bootloader Options Part 1	17
Figure 5-3	General Bootloader Options Part 2	19
Figure 5-4	OEM specific Bootloader options	21
Figure 5-5	Memory Device Table	22
Figure 5-6	Flash Block Table	22
Figure 5-7	Logical Block Table	23
Figure 5-8	NV-Wrapper configuration	25
Figure 5-9	Meta Data Configuration	25
Figure 5-10	Components of the security module	26
Figure 5-11	Security Class "Vendor"	27
Figure 5-12	Configuration of security class Vendor	29
Figure 5-13	Configuration of security class DDD	29
Figure 5-14	Configuration options for security class DDD	30
Figure 5-15	Security Class C or CCC	31
Figure 5-16	Configuration options for security class C/CCC	31
Figure 5-17	Configuration options for security class C/CCC with additional CRC32	32
Figure 10-1	Presence Pattern	67
Figure 10-2	Presence Pattern Examples	68
Figure 10-3	Data processing sequence	71
Figure 10-4	Multiple ECU setup	72
Figure 11-1	vFlash Configuration	75

Tables

Table 4-1	Bootloader Folder Structure	14
Table 5-1	General Bootloader Options Part 1	19
Table 5-2	General Bootloader Options Part 2	21
Table 5-3	OEM specific Bootloader Options	22
Table 5-4	Memory Device Table	22
Table 5-5	Flash Block Table	23
Table 5-6	Logical Block Table	24
Table 5-7	Generated Files	33
Table 7-1	RAM/ROM linkage	59
Table 10-1	Type Definition - tProcParam	71
Table 11-1	vFlash Configuration	76

1 Introduction

This document covers the OEM-specific particularities of the flash bootloader. It complements the explanations started in the user manual with OEM-specific details. All references there are resumed here in this document again and explained in detail.

The connection between a reference in the user manual and its specific description in this document is the headline. Both the reference and its explanation can be found below the same headline

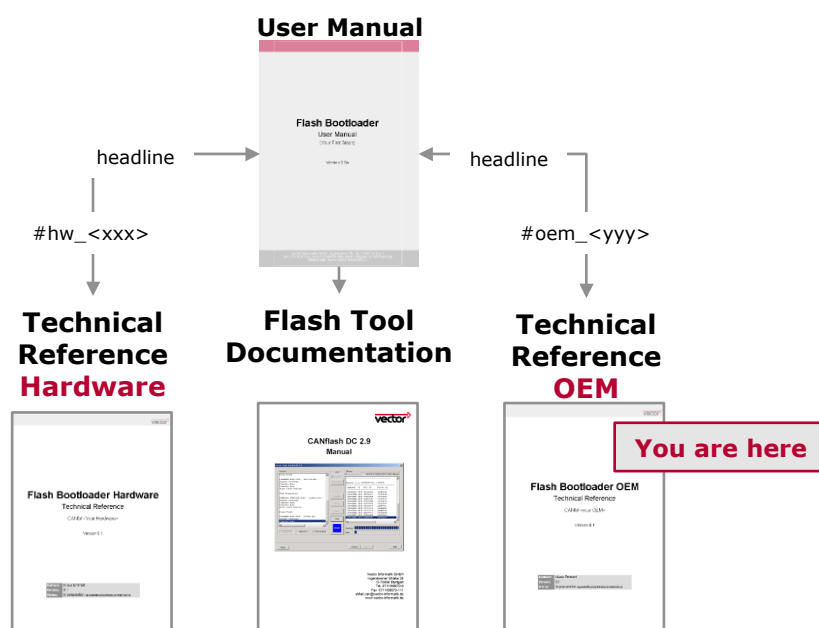


Figure 1-1 Manual and References for the Flash Bootloader

Additionally, this headline is marked with the ID of the reference from the User Manual. This ID looks like: **[#oem_<yyy>]**.

2 Download concept

2.1 Overview

The flash tool (Diagnostic Tester) uses the UDS protocol to communicate with the ECU. Inside the ECU the UDS communication is handled by the flash bootloader (aka FBL). The flash bootloader supports a subset of diagnostic services which are necessary for the flash download process.

The download sequence is implemented according to [14]. Diagnostic services are implemented according to [13] and [15]

3 Flash Bootloader Delivery

3.1 Bootloader Components

The flash bootloader delivery contains the following components:

- ▶ Flash bootloader source code
- ▶ Configuration Tool DaVinci Configurator 5

**Caution**

A separate license has to be obtained to run DaVinci Configurator 5. Please contact your Vector sales contact if you need to obtain a license.

- ▶ Flash Driver source code and executable
- ▶ Non-volatile memory module. Normally, a dummy EEPROM driver is delivered with the bootloader. Drivers for external EEPROMs or EEPROM emulations can be ordered as optional components.
- ▶ Security module (Security class DDD). Other security classes are optional.
- ▶ Optional decompression module.
- ▶ vFlash template

**Caution**

The vFlash framework is not licensed with the bootloader. A license of the vFlash framework has to be obtained separately. Please contact your Vector sales contact if you need to obtain a license.

- ▶ HexView. HexView can be used to show and change the contents of various binary formats. It can also be used for signature and checksum calculations
- ▶ MakeSupport: Make system which can be used to compile the bootloader.

3.2 Bootloader Integration

The integration of a flash bootloader project is described in this manual. If you are not familiar with Vector bootloaders, please see [5] for more information.

**Bootloader integration steps**

1. Install the bootloader package
2. Start your own integration project
3. Configure the bootloader using the configuration tool
4. Adjust the user callback files from template
5. Build the bootloader
6. Adapt your application software

4 Bootloader Installation

4.1 [#oem_files] - Bootloader package structure

For more general information about this see the UserManual_FlashBootloader in the chapter **Extract the files to a folder on your PC**.

You will get your software as an installer. After installation, a directory structure with several subfolders will be created:

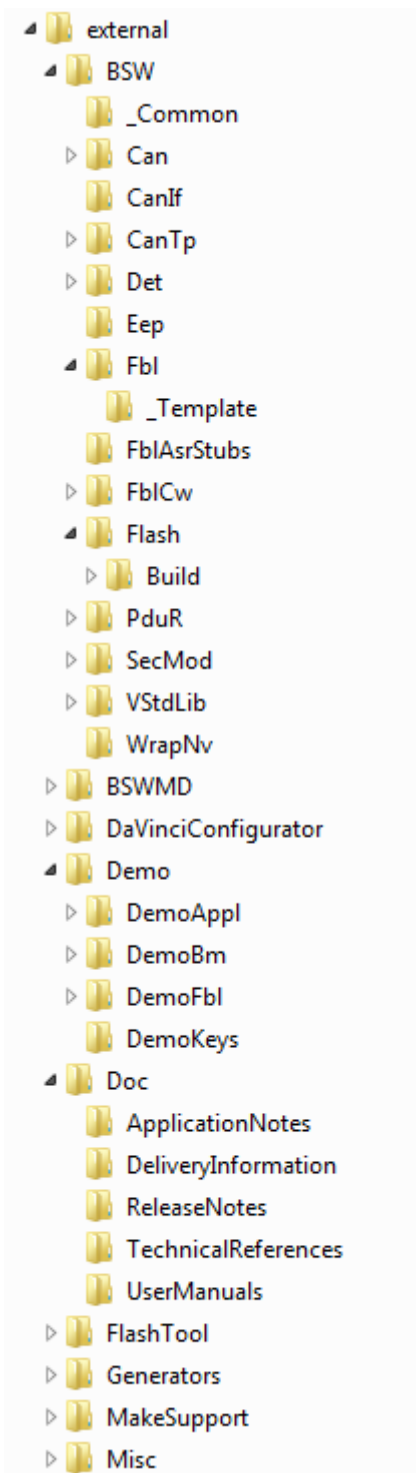


Figure 4-1 Bootloader Folder Structure

Directory	Description
BSW	Contains the bootloader source code (core files and template files). Files without leading underscore found in BSW and subfolders mustn't be changed.
_Common	Common modules, e.g. ECU- and compiler-specific type definitions.
Can, CanIf	MSR CAN-Driver
CanTp	MSR Transport Layer
Eep	EEPROM driver. If no EEPROM driver has been ordered, this directory contains the dummy EEPROM driver.
Fbl	Bootloader source files.
_Template	Contains the bootloader template files. You have to adapt these files for your project.
FblAsrStubs	Stub components for MSR components which are not used in the FBL.
FblBm	Boot manager core files
FblCw	Communication Wrapper
Flash	Flash driver. Includes its own make system and binary file in Build.
PduR	PDU Router
SecMod	SecMod contains the security module. Depending on the security class supported by the bootloader, a library and source code can be included.
VStdLib	Library with helper functions
WrapNv	Wrapper layer between non-volatile memory callback file and non-volatile memory driver.
DaVinci Configurator.	This folder contains the generation tool - DaVinciConfigurator
Demo	Root directory of demonstration application and demonstration bootloader.
DemoAppl	Root directory of demonstration application. This application is used to demonstrate the transition from application to bootloader and back.
DemoFbl	An example how the bootloader could be used. This example uses the bootloader source files from the Fbl directory and the adapted template files from "source" and "include" subdirectories.
DemoKeys	Demonstration keys for use with security class C or CCC. This directory is optional.

Config	Example bootloader configuration (either GENy or Cfg5)
Appl	This directory contains the bootloader demonstration project.
GenData	Generated files used in the demonstration bootloader.

Directory	Description
include	Adapted header templates of demonstration project.
source	Adapted Source templates of demonstration project.
Doc	Bootloader documentation, e.g. UserManual, this manual, Reference Manual Hardware, Reference Manual Security Module (optional) and project specific documentation.
FlashTool	vFlash template installer, example Seed/Key DLL and Visual Studio project to adapt the Seed/Key DLL.
Generators	Project specific part of the generation tool.
MakeSupport	Make system to recompile the demonstration bootloader.
Misc	Additional tools like HexView.

Table 4-1 Bootloader Folder Structure

4.2 Demonstration Bootloader

The .\Demo\DemoFbl\Appl directory contains an exemplary bootloader integration project. The bootloader template files from .\BSW\Fbl_Template have been copied to “source” and “include” subdirectories of the DemoFbl directory. The “GenData” subdirectory contains the necessary files for the configuration of the bootloader. These files are created by the DaVinci Configurator.

5 Generation Tool (DaVinci Configurator 5)

Bootloaders which are delivered with an Autosar 4 based communication stack from Vector use the generation tool DaVinci Configurator 5. This configuration is the standard configuration. GENy based configurations will be delivered if the bootloader configuration isn't supported by a Autosar based configuration.

The communication stack uses a standard Autosar 4 communication stack up to the PduR component. The bootloader diagnostics module is added as a CDD.

5.1 Project Setup and Communication Stack Configuration

The project setup is described in [12] Technical Reference Communication Wrapper PDU Router, chapter 4 – Configuration.

The list of components used to configure the bootloader is described in [12], Table 4-1 "Bootloader Component Selection".



Note

The list of components which are part of the SIP but have to be added manually is extended compared to the list in [12].

- ▶ Fbl
- ▶ FblHal
- ▶ Smh
- ▶ WrapNv

5.2 Bootloader Configuration

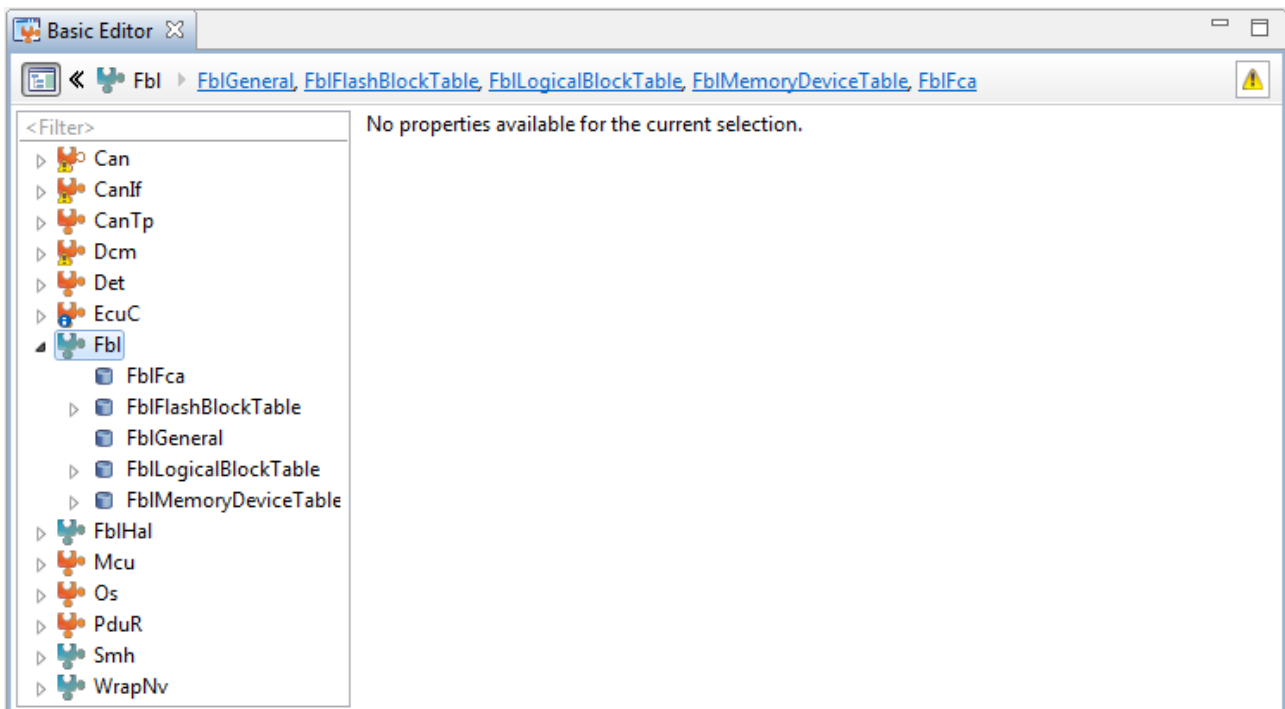


Figure 5-1 Bootloader Configuration in Basic Editor

The Bootloader configuration is done in the Fbl module. Additionally, a user configuration file can be used for project specific configuration items.

5.2.1 General Bootloader Options

The following options are available in the General Bootloader Options:

Short Name:	<input type="text" value="FblGeneral"/>
Adaptive Data Transfer Rccrp:	<input type="checkbox"/> *
Application State Task:	<input checked="" type="checkbox"/> *
Application Timer Task:	<input checked="" type="checkbox"/> *
Common Data:	<input checked="" type="checkbox"/> *
Compression Mode:	<input type="checkbox"/> *
Data Processing Buffer Size [Byte]:	<input type="text" value="0"/> dec*
Diagnostic Buffer Size [Byte]:	<input type="text" value="4095"/> dec*
Encryption Mode:	<input type="checkbox"/> *
Fill Code:	<input type="text" value="0x55"/> hex*
Flash Driver Usage:	<input type="text" value="USE_FLASH_DRIVER_FROM_ROM"/>
Gap Filling:	<input type="checkbox"/> *
Gateway Support:	<input type="checkbox"/> *
Header Address:	<input type="text" value="0xA0000100"/> hex*
Input Verification:	<input type="checkbox"/> *
Internal Memory Copy:	<input checked="" type="checkbox"/> *
Maximum Number Of Segments:	<input type="text" value="10"/> dec*
Output Verification:	<input checked="" type="checkbox"/> *
P2 Server [ms]:	<input type="text" value="50"/> dec*
P2* Server [ms]:	<input type="text" value="5000"/> dec*
Pipelined Programming:	<input type="checkbox"/> *

Figure 5-2 General Bootloader Options Part 1

Configuration Option	Description
Adaptive Data Transfer Rccrp	Not used by FCA SLP5 bootloaders
Application State Task	Application background function ApplFblStateTask() is called.
Application Timer Task	When selected, the FBL will periodically call ApplFblTask() while it is idle. This function may be customized to implement idle task operations.
Common Data	Enables a user-defined data structure which is referenced by FblHeader. This structure can be used to hand over constant data from the bootloader to the application software.
Compression Mode	Optional feature. If activated, the bootloader provides an interface to decompress data.

Configuration Option	Description
Data Processing Buffer Size [Byte]	If encryption or compression mode is enabled, the buffer size used by these data processing functions can be configured here.
Diagnostic Buffer Size [Byte]	This is the amount of memory (in bytes) reserved for data in diagnostic request messages. The maximum value is depending on the used communication stack.
Encryption Mode	Optional feature. If activated, the Bootloader provides a framework to decrypt data after download.
Fill Code	When writing data to a non-volatile memory device, the number of bytes written must be a multiple of the device's write-segment size. If an address-region does not end at a write-segment boundary, the FBL will fill the unused bytes with the value specified in this field. The fill code is also used if gap filling is activated.
Flash Driver Usage	Configures how the flash driver is copied to RAM <ul style="list-style-type: none"> ► Download Driver: Flash Driver is downloaded to ECU with the configured bus system. ► Use Flash Driver from ROM: Flash Driver is stored in an encrypted ROM image in the bootloader's flash memory and copied to RAM before initialization. Optional Flash Driver Download: Flash Driver can be downloaded to the ECU. If it isn't downloaded, the flash driver from an internal image will be used.
Gap Filling	This feature is used to fill unused areas of a logical block with a specific value.
Gateway Support	Not used by FCA SLP5 bootloaders
Header Address	The address of the constant FblHeader has to be entered here. The value has to match the address used by your linker to map the header to its proper location.
Input Verification	Not used by FCA SLP5 bootloaders
Internal Memory Copy	Activates a bootloader specific memory copy function. If deactivated, the memory copy function of the compiler library is used.
Maximum Number Of Segments	Maximum number of segments in a logical block. This value specifies how many sections are allowed for one logical block.
Output Verification	Standard verification method: Download is verified during the Routine Control - Check Memory service. Please note: Either Pipelined or Output Verification has to be configured.
P2 Server [ms]	P2 diagnostic timeout. The bootloader will send a response or response pending message before this timeout expires.
P2* Service [ms]	P2* diagnostic timeout. This timeout will be used instead of the P2 timeout if a response pending message has been sent.

Configuration Option	Description
Pipelined Programming	Optional feature. If activated, the bootloader will process and write downloaded data while the following data is transferred to the bootloader. This feature is also known as “Double Buffered Download” or “Early Acknowledge”.

Table 5-1 General Bootloader Options Part 1

Pipelined Verification:	<input type="checkbox"/> *
Presence Pattern:	<input type="checkbox"/> *
Processed Verification:	<input type="checkbox"/> *
Project State:	INTEGRATION*
Response After Reset:	<input type="checkbox"/> *
S3 Server Extended Session [ms]:	5000 dec*
S3 Server Programming Session [ms]:	5000 dec*
Sleep Mode:	<input checked="" type="checkbox"/> *
Sleep Time [ms]:	300000 dec*
Start Function:	<input type="checkbox"/> *
Stay In Boot:	<input type="checkbox"/> *
Suspend Driver Operation:	<input type="checkbox"/> *
Unaligned Data Transfer:	<input checked="" type="checkbox"/> *
User Config File:	\$(DpaProjectFolder)\Config\user_config.cfg*
User Routine:	<input type="checkbox"/> *
User Service:	<input checked="" type="checkbox"/> *
User Subfunction:	<input type="checkbox"/> *
Verification Function:	<input type="checkbox"/> *
Verification Segmentation [Byte]:	256 dec*
Watchdog Service:	<input checked="" type="checkbox"/> *
Watchdog Trigger Cycle [ms]:	1 dec*
Write Segmentation [Byte]:	256 dec*

Figure 5-3 General Bootloader Options Part 2

Configuration Option	Description
Pipelined Verification	Optional feature: Download is verified in background while additional data is received.
Presence Pattern	If this option is enabled, the bootloader will use flash memory at the end of each logical block to store the block validity information instead of using NV-memory.
Processed Verification	Not used by FCA SLP5 bootloaders.
Project State	<p>Project state controls the number of internal checks done by the bootloader. There are 3 levels available:</p> <ul style="list-style-type: none"> ► Integration: Detailed checks to find configuration errors. This level should be activated during the integration phase. ► Production: Checks for all errors which might occur during normal use of the bootloader. Should be used if the bootloader is ready for production. ► Production (reduced checks): Can be used to reduce code size. Please note that errors probably are detected later and that error messages like NRCs are more difficult to interpret if this setting is activated. We therefore recommend using the Production error reporting level.
Response After Reset	<p>This setting is used to decide if the Bootloader should send a positive response to default session or reset requests (deactivated) or the application software should send this response after a reset.</p> <p>If activated, the bootloader will send a response pending message before issuing the reset.</p>
S3 Server Extended Session	Not used by FCA SLP5 bootloaders.
S3 Server Programming Session	Not used by FCA SLP5 bootloaders.
Sleep Mode	If this switch is enabled, ApplFblBusSleep() will be called after a configurable time without diagnostic communication has expired.
Sleep Time [ms]	Activation time of sleep mode.
Start Function	The bootloader provides a function to start the programming session from the application (The function can be called using the FblHeader structure). If this feature is enabled, flags in shared RAM or non-volatile memory are not needed to invoke the bootloader.
Stay In Boot	Not used by FCA SLP5 bootloaders.
Suspend Driver Operation	Not used by FCA SLP5 bootloaders.
Unaligned Data Transfer	<p>If activated, the bootloader will request to receive the full diagnostic buffer and handles flash segment alignment internally.</p> <p>Otherwise, the tester has to send data with the length aligned to the flash segment size.</p>
User Config File	The path and name of a file to be included in the generated configuration file Fbl_Cfg.h may be specified. The file may be

Configuration Option	Description
	used to activate features of the FBL that are not included in the Cfg5 components. Normally, this field is blank.
User Routine	Enables a callback function which allows the implementation of additional routine identifiers.
User Service	Enables a callback function which allows the implementation of additional diagnostic services.
User Subfunction	Enables a callback function which could be used to add additional sub-functions to existing services.
Verification Function	Not used by FCA SLP5 bootloaders.
Verification Segmentation [Byte]	Length of data blocks sent to security module for verification.
Watchdog Service	Enables watchdog handling by the bootloader.
Watchdog Trigger Cycle [ms]	Specifies the interval in milliseconds between the calls of the watchdog handling function ApplFblWDTrigger().
Write Segmentation [Byte]	Size of data packets handed over to the flash driver.

Table 5-2 General Bootloader Options Part 2

5.2.2 OEM Bootloader Options

The following options are OEM-dependent settings:

Short Name:

Access Delay Time [ms]: dec *

Application Validity Flag: ☒ *

Check Programming Preconditions Service: ☒ *

Figure 5-4 OEM specific Bootloader options

Coinfiguration Option	Description
Access Delay Time [ms]	Delay time is active after three unsuccessful seed/key attempts. All further security access requests will be rejected during the delay time. Default value is 10 seconds. Setting this value to 0 will disable this feature.
Application Validity Flag	Configures, if the result of the Check Programming Dependencies service is used to determine if an application software is present or if the application validity is established on every startup (based on logical block validity information).

Check Programming Preconditions Service	If activated, the service Routine Control – Check Programming Preconditions is mandatory before the bootloader switches to programming session.
---	---

Table 5-3 OEM specific Bootloader Options

5.2.3 Memory Configuration

The memory configuration module is used to configure the memory layout used to store application software and calibration data.

The memory configuration is divided into 3 modules:

5.2.3.1 Memory Device Table

All available memory devices have to be configured in the memory table. The driver used for the internal flash memory is configured as default option.

Additional memory drivers like EEPROM drivers or drivers for external flash devices can be added here. These drivers must conform to the HIS memory driver interface. A detailed description of the flash driver interface can be found in [3].

FblMemoryDevices	01 Segment Size [Byte]	02 Erased Value
Flash	1 *	0xFF *

Figure 5-5 Memory Device Table

Configuration Option	Description
Short Name	Name used to identify the device in the flash block table. Also used as prefix to the driver interface functions.
01 Segment Size [Byte]	Smallest writeable data segment.
02 Erased Value	Device specific erase value.

Table 5-4 Memory Device Table

5.2.3.2 Flash Block Table

The flash block table is used to map memory devices and logical blocks to flash blocks. Blocks configured here should relate to physically erasable areas (1 or more physical flash blocks depending on the flash driver implementation and the flash block size).

FblFlashBlocks	01 Start Address	02 End Address	03 Memory Device Ref	04 Logical Block Ref	05 Description
FblFlashBlock_2	0x20000	0x2FFFF	Flash	DemoAppl	Flash Block 2 64 KB
FblFlashBlock_1	0x10000	0x1FFFF	Flash	DemoAppl	Flash Block 1 64 KB
FblFlashBlock_0	0x0 *	0xFFFF			

Figure 5-6 Flash Block Table

Configuration Option	Description
Short Name	Name of flash block.
01 Start Address	Start address of the flash block. Has to be aligned to the start address of an erasable flash block.
02 End Address	End Address. Has to be the end of an erasable flash block.
03 Memory Device Ref	Select the memory device used for this flash block.
04 Logical Block Ref	Select the logical block which uses this flash block.
05 Description	Comment to identify the block.

Table 5-5 Flash Block Table

5.2.3.3 Logical Block Table

The logical block table is used to divide the reprogrammable memory of the ECU into different logical blocks. They are writeable independently and can be used for e.g. an application and a calibration block.

Short Name:	<input type="text" value="Application"/>	▼
01 Block Index:	<input type="text" value="0x0"/>	hex * ▼
02 BlockType:	<input type="text" value="CODE"/>	* ▼
03 Disposability:	<input type="text" value="MANDATORY"/>	* ▼
04 Start Address:	<input type="text" value="0xA0040000"/>	hex ▼
05 End Address:	<input type="text" value="0xA005FFFF"/>	hex ▼
08 Max Reprog Attempts:	<input type="text" value="0"/>	dec * ▼
09 Input Verification:	<input type="text" value="SecM_VerifySignature"/>	* ▼
10 Processed Verification:	<input type="text" value="SecM_VerifySignature"/>	* ▼
11 Pipelined Verification:	<input type="text" value="SecM_VerifySignature"/>	* ▼
12 Output Verification:	<input type="text" value="SecM_Verification"/>	* ▼
13 Description:	<input type="text"/>	▼

Figure 5-7 Logical Block Table

Configuration Option	Description
Short Name	Name of logical block. The name is used as reference in the flash block table.
02 Block Type	Type of logical block. Supported block types are Boot, Code and Data.

03 Disposability	Used to determine if a block is mandatory or optional during application validation.
04 Start Address	Set automatically by flash block table.
05 End Address	Set automatically by flash block table.
08 Max Reprog Attempts	This value has to be set to the maximum allowed erase cycles of your hardware. If 0 is set, no reprogramming attempt check is done.
09 Input Verification	Not used by FCA SLP5 bootloaders.
10 Processed Verification	Not used by FCA SLP5 bootloaders.
11 Pipelined Verification	If ordered, the verification routine for pipelined verification is configured here.
12 Output Verification	The verification routine for output verification is configured here.
13 Description	Comment to describe the block in generated code.

Table 5-6 Logical Block Table


Caution

Be careful to set all needed blocks to “mandatory”. Otherwise they are not used to determine the application’s overall validity.

You also have to make sure the flash blocks used by the bootloader are not configured to any logical block to ensure the bootloader cannot erase itself.

5.2.3.4 NV-Wrapper Configuration

The bootloader stores a set of control information in non-volatile memory. This can be done using an address based memory like an external EEPROM or an ID based EEPROM emulation like the EepM module.

The NV-Wrapper provides an abstraction layer between these memory devices and the bootloader’s fbl_apnv layer. Please see [8] for more information.

The following NV-memory entries are pre-configured in your bootloader. The figure below shows the preconfigured NV-memory layout of the FCA bootloader.

<Filter>					
WrapNvDataBlocks	Count	Description	Type		
ProgReqFlag	1	Programming request flag	SINGLE	*	*
ResetResponseFlag	1	Reset response flag	SINGLE	*	*
ValidityFlags	1	Logical block validity flagss	SINGLE	*	*
ApplValidity	1	Application validity flag	SINGLE	*	*
ApplUpdate	1	Application update flag	SINGLE	*	*
SecAccessDelayFlag	1	Security Access Delay flag	SINGLE	*	*
SecAccessInvalidCount	1	Security Access Invalid count	SINGLE	*	*
DcmDslRxTesterSourceAddr	1	Connection for reset response	SINGLE	*	*
ProgrammingStatus	1	Programming Status Informationn	SINGLE	*	*
DiagnosticVersion	1	Diagnostic Version of DID F110h	SINGLE	*	*
VehicleManufEcuSwNum...	1	DID F188 - Vehicle Manufacturer ECU Software Number	SINGLE	*	*
EbomEcuPartNumber	1	DID F132 - EBOM ECU Part Number	SINGLE	*	*
CoDePEcuPartNumber	1	DID F187 - CoDeP ECU Part Number	SINGLE	*	*
ExhaustRegulationTan	1	DID F196 - Exhaust Regulation or Type Approval	SINGLE	*	*
Metadata	2	Metadata information for each logical block	TABLE	*	*
0 of 15 elements selected. Sorting by <Type>					

Figure 5-8 NV-Wrapper configuration

MetaData is a table which stores information needed for every logical block. Please see the following figure for details:

<Filter>			
WrapNvDataElements	Description	Length	
CRCLength	Length of CRC total	1	*
CRCStart	Start address of CRC total	1	*
CRCValue	CRC total of logical block	1	*
Fingerprint	Download fingerprint	13	*
MemoryStatus	Memory status; erased, programmed	1	*
ProgAttempts	Reprogramming attempts	2	*
ProgCounter	Successful reprogramming attempts	2	*
SoftwareIdentification	DID F181 or F182 - Application Software- or Application Data Identification	13	*
SwEbomPartNumber	DID F122 - Software EBOM Part Number	11	*
1 of 9 elements selected. Sorting by <WrapNvDataElements>			

Figure 5-9 Meta Data Configuration

**Caution**

The number of Meta Data entries must correspond to the configured number of logical blocks.

5.2.4 Security Module Configuration

This section gives an overview of the security module configuration. For more detailed information please see the technical reference of the Security Module see [4]

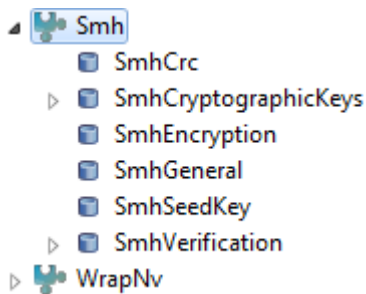


Figure 5-10 Components of the security module

FCA specifies several options for flash download verification (see [15]). The component SmhVerification provides the possibility to configure the desired variant. Depending on the configuration, the bootloader expects a specific format of the diagnostic requests RoutineControl – checkProgram (31h 01h F000h).

5.2.4.1 Check Routine Without Additional CRC and Without Additional Signature

This option provides a two-bytes checksum in the routine option record of the diagnostic request. The FCA bootloader supports this options with a customer-specific verification routine.

The standard bootloader delivery does not provide an implementation of this customer-specific verification function

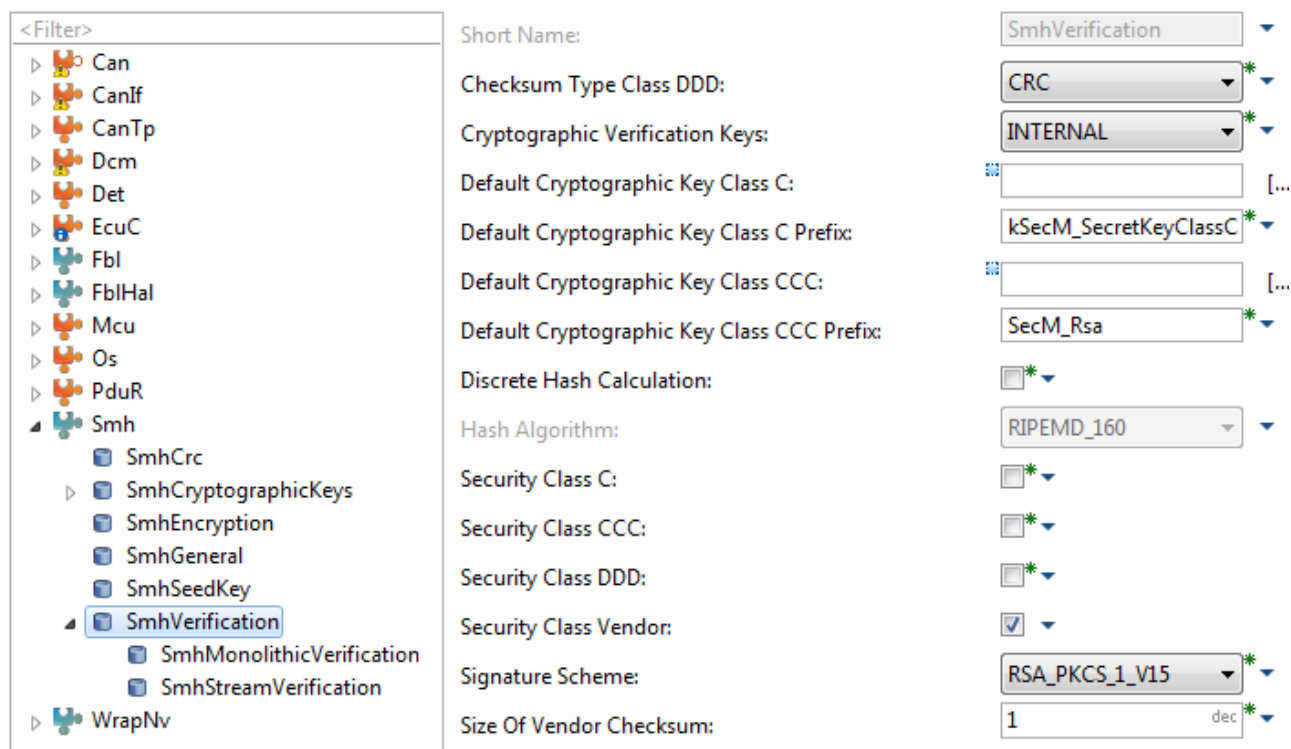


Figure 5-11 Security Class "Vendor"

The verification function is implemented with the following prototype. You find the documentation of the parameter type in [10] .

```
SecM_StatusType SecM_VerifyClassVendor( V_MEMRAM1 SecM_SignatureParamType V_MEMRAM2
V_MEMRAM3 * pVerifyParam )
```

Security class Vendor is enabled on the SmhVerification component with the corresponding check box. Additionally, on the SmhMonolithicVerification component, the security class "Vendor" has to be chosen in the combo box. Make sure that the CRC Offset is set to 0.



Note

The configurable CRC Offset and Signature Offset in bytes starts with the routine option record of the RoutineControl – checkProgram request. The routine option record is the parameter for the verification routine of the security module and is used to compare the internally computed checksum with the reference value.

The following example shows the implementation of a simple byte checksum:



Example

```
vuint16 checksum;
SecM_StatusType SecM_VerifyClassVendor( V_MEMRAM1 SecM_SignatureParamType V_MEMRAM2
V_MEMRAM3 * pVerifyParam )
{
    SecM_StatusType result;
    vuint32 idx;

    switch(pVerifyParam->sigState)
    {
        case SEC_HASH_INIT:
        {
            checksum = 0;
            result = SECM_OK;
            break;
        }

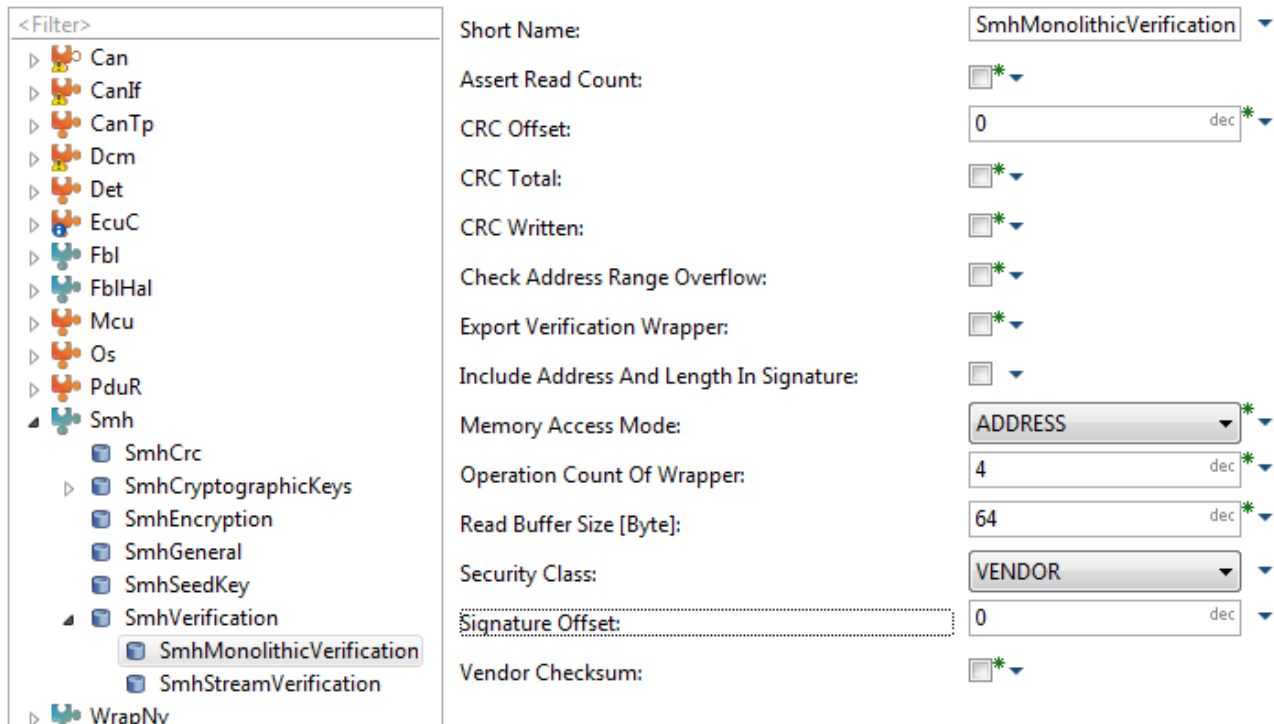
        case SEC_HASH_COMPUTE:
        {
            for(idx = 0; idx < pVerifyParam->sigByteCount; idx++)
            {
                checksum += pVerifyParam->sigSourceBuffer[idx];
            }
            result = SECM_OK;
            break;
        }

        case SEC_HASH_FINALIZE:
        {
            result = SECM_OK;
            break;
        }

        case SEC_SIG_VERIFY:
        {
            vuint16 checksumRef;
            checksumRef = pVerifyParam->sigSourceBuffer[0];
            checksumRef <<= 8u;
            checksumRef |= pVerifyParam->sigSourceBuffer[1];
            if (checksumRef == checksum )
            {
                result = SECM_OK;
            }
            else
            {
                result = SECM_NOT_OK;
            }
            break;
        }

        default:
        {
            result = SECM_NOT_OK;
            break;
        }
    }

    return result;
}
```

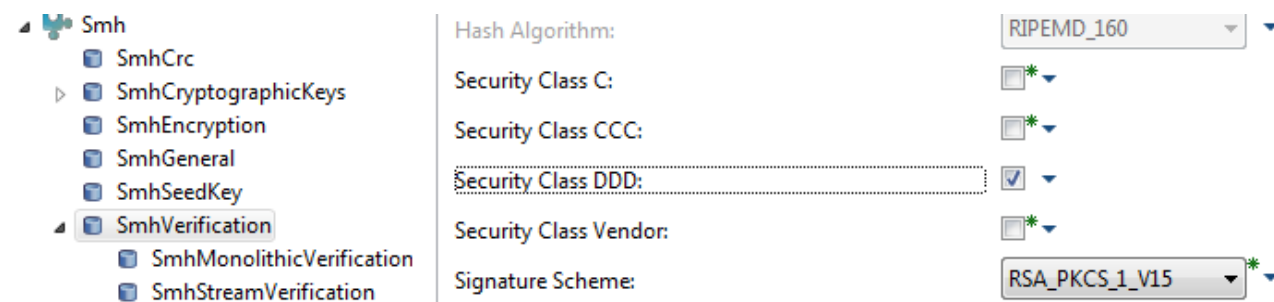


Property	Value
Short Name:	SmhMonolithicVerification
Assert Read Count:	<input type="checkbox"/>
CRC Offset:	0
CRC Total:	<input type="checkbox"/>
CRC Written:	<input type="checkbox"/>
Check Address Range Overflow:	<input type="checkbox"/>
Export Verification Wrapper:	<input type="checkbox"/>
Include Address And Length In Signature:	<input type="checkbox"/>
Memory Access Mode:	ADDRESS
Operation Count Of Wrapper:	4
Read Buffer Size [Byte]:	64
Security Class:	VENDOR
Signature Offset:	0
Vendor Checksum:	<input type="checkbox"/>

Figure 5-12 Configuration of security class Vendor

5.2.4.2 Check Routine with additional CRC32 and without additional Signature

This option provides a four bytes CRC32 value in the routine option record of the diagnostic service request.



Property	Value
Hash Algorithm:	RIPEMD_160
Security Class C:	<input type="checkbox"/>
Security Class CCC:	<input type="checkbox"/>
Security Class DDD:	<input checked="" type="checkbox"/>
Security Class Vendor:	<input type="checkbox"/>
Signature Scheme:	RSA_PKCS_1_V15

Figure 5-13 Configuration of security class DDD

Enable this option by setting the check box for security class DDD. Additionally, the CRC Offset must be set to 2 because the CRC32 value in the service request is preceded by a two-bytes length information.

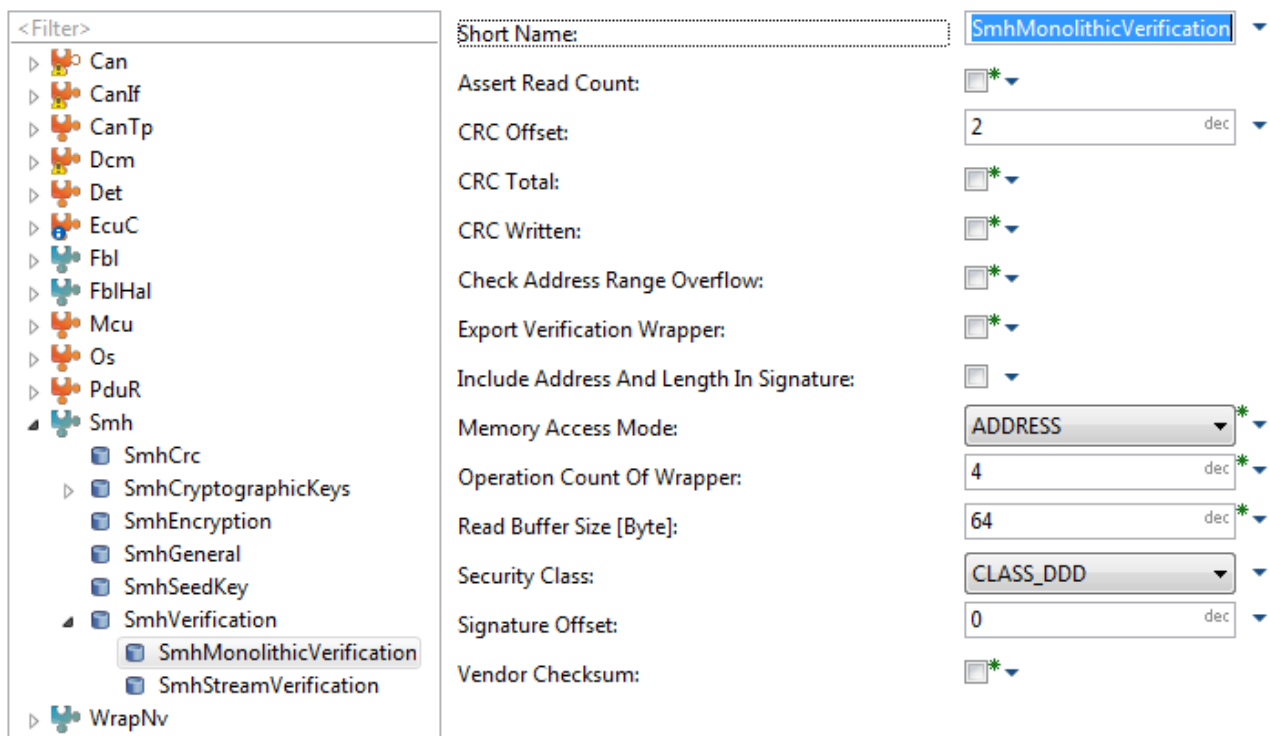


Figure 5-14 Configuration options for security class DDD

5.2.4.3 Check Routine without additional CRC32 and with additional Signature

This option provides a 20- or 128 bytes signature value in the routine option record of the diagnostic service request.

This option is enabled by setting the check box for security class C/CCC. The Signature Offset must be set to 2 because as with the CRC32 the signature value in the service request is preceded by a two-bytes length information.

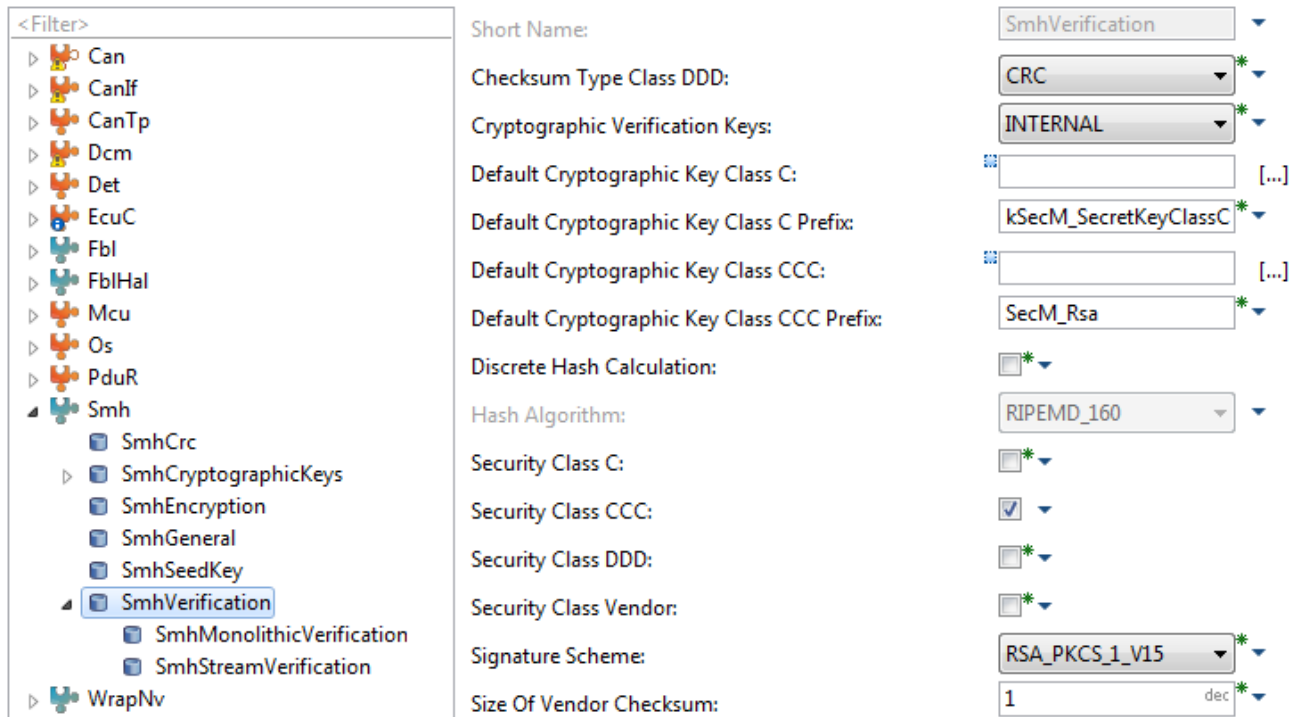


Figure 5-15 Security Class C or CCC

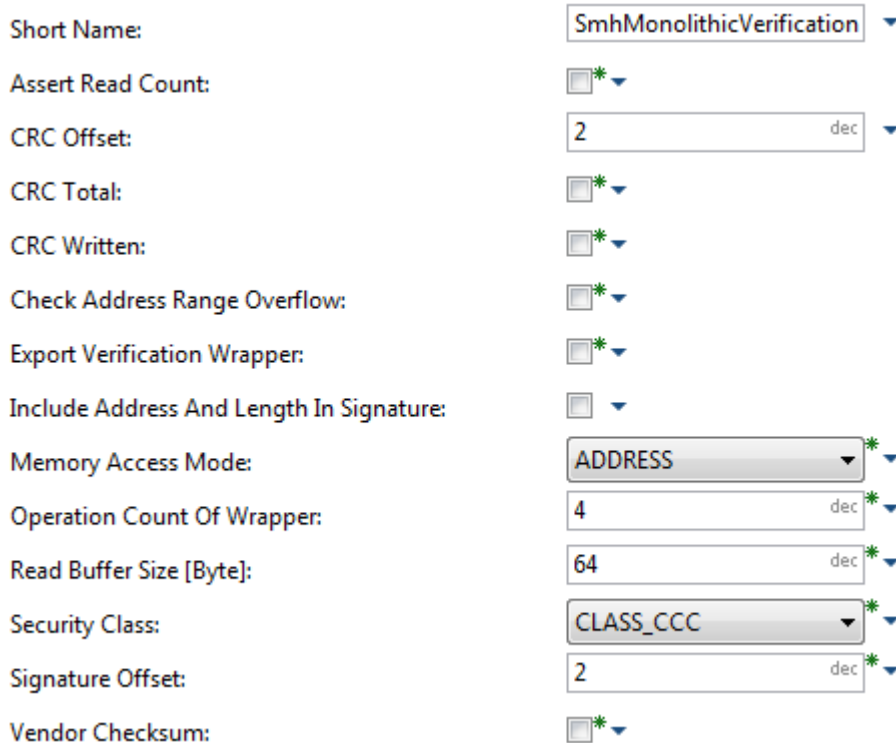


Figure 5-16 Configuration options for security class C/CCC

5.2.4.4 Check Routine with additional CRC32 and with additional Signature

This option provides the CRC32- and the signature value in the request. To configure this option, like for security class C/CCC the security class has to be activated on the SmhVerification component. Additionally, the “CRC Written” has to be enabled. The service request contains the CRC- and the signature value with the corresponding length information. Therefore, the offset for the CRC must be set to two and the offset for the signature must be set to eight bytes.

Short Name:	<input type="text" value="SmhMonolithicVerification"/>
Assert Read Count:	<input type="checkbox"/> *
CRC Offset:	<input type="text" value="2"/> dec
CRC Total:	<input type="checkbox"/> *
CRC Written:	<input checked="" type="checkbox"/>
Check Address Range Overflow:	<input type="checkbox"/> *
Export Verification Wrapper:	<input type="checkbox"/> *
Include Address And Length In Signature:	<input type="checkbox"/>
Memory Access Mode:	<input type="text" value="ADDRESS"/> *
Operation Count Of Wrapper:	<input type="text" value="4"/> dec*
Read Buffer Size [Byte]:	<input type="text" value="64"/> dec*
Security Class:	<input type="text" value="CLASS_CCC"/> *
Signature Offset:	<input type="text" value="8"/> dec
Vendor Checksum:	<input type="checkbox"/> *

Figure 5-17 Configuration options for security class C/CCC with additional CRC32

5.3 Running the Generator

When you have finished your configuration, you need to run the generator to produce the files needed to compile the FBL. To generate the files, click on the generate button on the toolbar, or select “Generate System” from the Project menu. The following table describes the contents of the generated files:

File Name	Contents
Can*.c/h	CAN communication stack
ComStack_Cfg.h	Global communication stack definitions

Det_Cfg.h	Development Error Tracer configuration
Fbl_Cfg.h	Global bootloader configuration
Fbl_Fbt.c/h	Flash block table
FblHal_Cfg.h	Hardware specific configuration.
Fbl_Lbt.c/h	Logical block table
FblCw_Cbk.h	Communication Wrapper callback definitions
PduR_*.c/h	PDU Router configuration
SecM*.c/h	Security module configuration
v_cfg.h	Vector platform definitions
WrapNv_Cfg.h	NvWrapper configuration. Defines a wrapper layer between ID and address based NV-memory stacks.

Table 5-7 Generated Files

6 User callback files and hardware specific adaptations

The bootloader includes a set of files that must be adapted and reviewed by you to make sure they fit the needs of your ECU and application. All files found at `.\BSW\Fbl_Template` require customization.

To use these files, please follow these steps:



User callback file setup

1. Copy the files from `.\BSW\Fbl_Template` to your project.
2. Remove the leading underscores from these files.
3. Adapt them to your needs.
4. Include them into your project setup and build the bootloader.

Please pay attention when adapting callback functions. If a routine (such as an EEPROM function) takes a long time to run, it is possible that the ECU's watchdog timer will force the ECU to reset. You should also be careful when using library routines such as `memcpy()`.

To avoid a reset, you must call `FblLookForWatchdog()` at least once per millisecond. More often is desirable. Failure to meet this requirement may lead to unexpected resets while programming the ECU. If the function is called during a service request before the response has been given, call the routine `FblRealTimeSupport()` instead of `FblLookForWatchdog()`. This will also maintain the response-pending of the service request.

6.1 Startup code



Caution

It is absolutely necessary that you adapt the startup code for the bootloader to your specific hardware platform and configuration.

Please be aware that there will be two startup codes executed subsequently (the startup code of the bootloader and the startup code of your application) and that there are write-once registers on several platforms.

6.2 Hardware, Input/Output and miscellaneous Callbacks

The file `fbl_ap.c` contains functions to handle hardware specific operations, e.g. input/output control.

The following pages describe each function. When building your bootloader, you should review the implementation of each function, and adapt them to conform with your ECUs requirements.

Prototype

```
void ApplFblInit( void )
```

Functional Description

Perform basic hardware and I/O initialization, e.g. PLL and CAN output PINs. If a non-volatile memory driver is needed to decide if the application should be started or not, the driver has to be initialized in this function as well.

Particularities and Limitations

- > ApplFblInit() is executed before the application software is started. Please keep in mind that it will be executed on every startup.
- > If the watchdog is initialized in ApplFblInit(), it will run if the application software is started. The bootloader will begin to handle the watchdog later during its initialization.

Prototype

```
void ApplFblStartup( uint8 initposition )
```

Parameter

initposition

The call context of this function is defined by this parameter.

Functional Description

This function is called during the bootloader initialization before and after each initialization step. User-specific initialization steps are added to the bootloader initialization using this function.

The parameter of ApplFblStartup() determines the bootloader initialization step and if the function is called before or after the initialization step. The following parameters are supported:

- > (kFblInitPreCallback | kFblInitBaseInitialization): First call. No initializations are done by the bootloader at this point
- > (kFblInitPreCallback | kFblInitFblCommunication): Call before the communication stack of the bootloader is initialized.
- > (kFblInitPreCallback | kFblInitFblFinalize): Call before the bootloader initialization is finalized. At this point, the bootloader decided not to start the application software independent of the startup configuration.
- > (kFblInitPostCallback | kFblInitBaseInitialization): Called after the hardware pre-initialization. At this point, ApplFblInit() normally is called to initialize the hardware for both bootloader and application.
- > (kFblInitPostCallback | kFblInitFblCommunication): Called after the communication stack is initialized.
- > (kFblInitPostCallback | kFblInitFblFinalize): Called after the bootloader initialization is finalized. At this point, all core modules are fully initialized.

Particularities and Limitations

- > This function is called several times during the bootloader initialization.
- > The second call of this function can be positioned before or after the decision if the application software should be started. This depends on the startup configuration, e.g. if Force Boot Mode is used or not.

Prototype

```
tFblResult ApplFblCheckProgConditions( void )
```

Return code

tFblResult

If all conditions are correct, the function returns kFblOk, otherwise kFblFailed.

Functional Description

This function is called after receiving the service request sessionControl ProgrammingSession to check the programming conditions like reprogramming counter, ambient temperature, programming voltage, etc.

Prototype

```
tFblResult ApplFblCheckConditions( vuint8 * pbDiagData, tTpDataType diagReqDataLen )
```

Parameter

pbDiagData	Pointer to diagnostic data buffer
diagReqDataLen	Request data length

Return code

tFblResult	kFblOk or kFblFailed
------------	----------------------

Functional Description

This is a general condition-check function. It is called for all diagnostic services and can be used to check conditions, which are necessary for the diagnostic service.

If a condition is not fulfilled, please trigger the respective NRC and return kFblFailed.

An example use case would be to check the ECU's voltage while the transfer data service is received and issue the NRCs \$92 and \$93.

Prototype

```
vuint8 ApplFblCheckHwSwCompatibility( vuint8 blockIdx )
```

Parameter

blockIdx	Index of logical block
----------	------------------------

Return code

Vuint8	Check result
	> kDiagDependencyCheckFailedSwHw
	> kDiagDependencyCheckOk

Functional Description

This function checks if the given logical block is compatible with the current hardware. The implementation example of this function uses a hard-coded hardware number to check the hardware compatibility. This version number of two bytes is compared with the hardware version of the application header of the logical block.

Prototype

```
vuint8 ApplFblCheckSwSwCompatibility( void )
```

Return code

Vuint8	Check result
	> kDiagDependencyCheckFailedSwSw
	> kDiagDependencyCheckOk

Functional Description

This function checks if the programmed logical blocks are compatible. This is done by comparing the software compatibility version information.

Prototype

```
vuInt8 ApplFblCheckProgDependencies( void )
```

Return code

VuInt8	If program dependencies are fulfilled, 0x00. Otherwise != 0x00
--------	--

Functional Description

This function is used to check the dependencies between bootloader and application software. The application software modules have to be verified against each other here as well.

There are 2 possibilities when this function is called:

- ▶ During application startup and compatibility check service if FBL_APPL_ENABLE_STARTUP_DEPENDENCY_CHECK is set. This setting can be controlled by the GENy option "Application Validity Flag".
- ▶ From compatibility check service

Prototype

```
void ApplTrcvrNormalMode( void )
```

Functional Description

This function is used to configure your CAN bus transceiver for normal CAN communication. You must implement this routine to perform the necessary Input/Output operations to control the transceiver state.

Prototype

```
void ApplTrcvrSleepMode( void )
```

Functional Description

This function is used to configure CAN bus transceiver for low-power (sleep) operation. You must implement this routine to perform the necessary Input/Output operations to control the transceiver state.

Prototype

```
void ApplFblSetVfp( void )
```

Functional Description

The purpose of this routine is to turn on the power supply required to program non-volatile memory. If your ECU does not require an external power source to erase and program memory, then this function may be left empty. It's been called shortly before the flashdriver will be initialized.

Particularities and Limitations

- > This function can remain empty on most hardware platforms. It is needed e.g. on some V850 controllers.

Prototype

```
void ApplFblResetVfp( void )
```

Functional Description

The purpose of this routine is to turn off the power supply required to program non-volatile memory. If your ECU does not require an external power source to erase and program memory, then this function may be left empty.

Particularities and Limitations

- > This function can remain empty on most hardware platforms. It is needed e.g. on some V850 controllers.

Prototype

```
void ApplFblReset( void )
```

Functional Description

This function is responsible for resetting the ECU. For example, you may execute a restart instruction (if available), jump to the reset vector, or use the watchdog. The choice is up to you, and depends on your ECU hardware. This function is called when it is necessary to reset the ECU.

Particularities and Limitations

- > Jumping directly to the ECU's reset vector should be considered a last-choice option. Some ECUs contain registers that may be accessed only once – it is possible that the FBL will be unable to re-initialize them when it is restarted via a jump to the reset vector.
- > Do not waste too much time in this function (e.g. through an endless loop waiting for a long-lasting watchdog trigger). This can cause unwanted delay on the network operation or service tester.

Prototype

```
vuInt8 ApplFblSecuritySeedInit( void )
```

Return code

vuInt8	Status of seed initialization
--------	-------------------------------

Functional Description

This function can be used to Initialize seed values. For example, a random number generator can be started here. This function will be called, when a TP message is received.

Particularities and Limitations

- > This function has to be linked to RAM if pipelined programming is configured.

Prototype

```
vuInt8 ApplFblSecuritySeed( void )
```

Return code

vuInt8	Status of seed generation
--------	---------------------------

Functional Description

This function is called to obtain the security-seed for the ECU. The function is called when a Security-Access (service \$27) request is received containing a request-seed sub-function.

Particularities and Limitations

- > See also ApplFblSecurityKey()
- > The default implementation calls the function SecM_GenerateSeed() of the security module.

Prototype

```
vuint8 ApplFblSecurityKey( void )
```

Return code

vuint8	Status of key verification
--------	----------------------------

Functional Description

The purpose of this function is to verify that the security-key supplied in the diagnostic request is equal to the expected security-key. The expected security-key is based upon the security-seed returned by ApplFblSecuritySeed().

This function is called when a Security-Access (service \$27) request containing the send-key sub-function is received.

Particularities and Limitations

- > See also ApplFblSecuritySeed()
- > The default implementation calls the function SecM_ComputeKey() of the security module.

Prototype

```
vuint8 ApplFblSecurityInit( void )
```

Return code

vuint8	Status of security module initialization
--------	--

Functional Description

This function can be used to initialize user specific functionality of the Seed/Key algorithm.

Prototype

```
tFblResult ApplFblInitDataProcessing( tProcParam * procParam )
```

Parameter

procParam	Pointer to data processing parameter structure
-----------	--

Return code

tFblResult	kFblOk/kFblFailed
------------	-------------------

Functional Description

ApplFblInitDataProcessing is called during processing of a RequestDownload service request with a data format identifier value != 0x00

The function should perform any necessary initialization required for application specific data processing functions (e.g. decryption/decompression algorithms).

Particularities and Limitations

- > This function can be used to process (decompress or decrypt) data. It includes an implementation in case an decompression or decryption module has been ordered with the bootloader.

Prototype

```
tFblResult ApplFblDataProcessing( tProcParam * procParam )
```

Parameter	
procParam	Pointer to data processing parameter structure
Return code	
tFblResult	kFbIOk/kFblFailed
Functional Description	
<p>This function is called to allow for user specific download data processing (e.g. data decompression and/or decryption).</p> <p>The bootloader will call this function in every Transfer-Data request if the preceding Request-Download request contained a non-zero data-format-identifier parameter. The input data corresponds to the data which has been received with the TransferData request, the output data will be written to the target memory.</p> <p>The function is responsible for decrypting and/or decompressing data. The decrypted/decompressed results must be placed in the output buffer procParam structure. The number of output bytes may be different from the number of input bytes so that this function also has to make sure that the output length variable is set to the correct value.</p>	
Particularities and Limitations	
<p>> This function is only provided if Vector data processing modules (decryption and/or decompression) are part of the bootloader package. In this case, the function will be used as a wrapper for the dedicated decryption/decompression module.</p>	

Prototype	
tFblResult ApplFblDeinitDataProcessing(tProcParam * procParam)	
Parameter	
procParam	Pointer to data processing parameter structure
Return code	
tFblResult	kFbIOk/kFblFailed
Functional Description	
<p>The FBL calls this function in order to conclude the data processing for the current contiguous address range. All remaining input data bytes have to be processed and passed to the output buffer.</p>	
Particularities and Limitations	
<p>> This function is only provided if Vector data processing modules (decryption and/or decompression) are part of the bootloader package. In this case, the function will be used as a wrapper for the dedicated decryption/decompression module.</p>	

Prototype	
void ApplFblTask(void)	
Functional Description	
<p>This function is called every millisecond if the bootloader is in idle state. It is not called during flash operations.</p>	

Particularities and Limitations

- > This function is called only if the configuration selects "Enable ApplTask".
- > This function is only called while the FBL is idle. Calling intervals exceeding the TpCallCycle can occur while the FBL handles diagnostic service requests. This normally occurs while erasing and writing non-volatile memory.

Prototype

```
void ApplFblStateTask( void )
```

Functional Description

This function is called continuously if the bootloader is in idle state. It is not called during flash operations.

Particularities and Limitations

- > This function is called only if the configuration selects "Enable ApplTask".
- > This function is only called while the FBL is idle. Calling intervals exceeding the TpCallCycle can occur while the FBL handles diagnostic service requests. This normally occurs while erasing and writing non-volatile memory.

Prototype

```
void ApplFblCanParamInit( void )
```

Functional Description

This function is used if the configuration selects "Enable Can Configuration", or if the configuration defines Multiple-ECUs (for example door ECUs (left door/ right door)). The routine is called as part of the FBL initialization sequence when started from both reset and from the application.

The purpose of the function is to allow the setting or modification of the CAN-ID and bus timing parameters. They can be set based upon runtime conditions instead of fixed ones at compile-time. Usually, only the CAN-IDs need to be adapted.

Particularities and Limitations

- > Two global variables are involved with the FBL communication initialization: fblCanIdTable, and CanInitTable.
- > fblCanIdTable contains the diagnostic message request CAN-IDs (the functionally-address ID, and the physically-addressed ID), as well as the CAN bus timing and other hardware initialization parameters.
- > At compile time, the constant kFblCanIdTable contains the fixed request and response message CAN-IDs and the bus timing parameters, based on the database and configuration settings (results from the generated files from "Geny"). The table is copied to fblCanIdTable just before the FBL calls this function.
- > The ApplFblCanParamInit() function in the fbl_ap.c file contains an exemplary implementation. It allows the user to overwrite the default settings before the CAN-controller is initialized.

Prototype

```
void ApplFblCanBusOff( void )
```

Functional Description

The FBL checks internally for communication errors via FblCanErrorTask() in the main loop. This function is called from FblCanErrorTask() while the CAN controller is in a bus-off state.

This is a notification that the ECU cannot transmit messages. A strategy to re-initialize the CAN cell and/or controller has to be implemented here to make sure the controller won't be stuck in a bus off state.

The example implementation resets the ECU. Please make sure that the reset complies with your ECU's requirements.

Prototype

```
void ApplFblBusSleep( void )
```

Functional Description

This function indicates transition to bus silence. By calling this function the FBL indicates the application that the CAN bus has to go to bus sleep.

Particularities and Limitations

- > If possible, the ECU must be set into sleep mode in this function.

Prototype

```
void ApplFblStartApplication( void )
```

Functional Description

This function is used to start the application software.

Particularities and Limitations

- > When this function is called, ApplFblInit() has been executed.
- > The non-volatile memory driver has most likely been initialized.

Prototype

```
void ApplFblSetShutdownStatus(tFblShutdownStatus status)
```

Parameter

status	Shutdown status, the supported values are kFblShutdownStatusNormal kFblShutdownStatusAbnormal
--------	---

Return code

tFblResult	kFblOk/kFblFailed
------------	-------------------

Functional Description

This function sets the shutdown status. Per default, the shutdown status is set to "abnormal" so that an abnormal shutdown can be detected at PowerOn/Reset. If a normal reset shall be issued, this function is called to set the shutdown status to "normal"

Particularities and Limitations

- > The status is stored in a global variable.
- > The global variable must be located in a common memory area of application and bootloader

Prototype

```
tFblShutdownStatus ApplFblGetShutdownStatus(void)
```

Return code

tFblShutdownStatus	Current shutdown status kFblShutdownStatusNormal kFblShutdownStatusAbnormal
--------------------	---

Functional Description

This function returns the shutdown status.

Particularities and Limitations

> None

Call context

> This function is called from ApplFbIsValidApp(). If an abnormal shutdown is detected the FBL does not start the application immediately.

Prototype

```
void ApplFbClrShutdownStatus(void)
```

Functional Description

This function sets the shutdown status to the default state "abnormal"

Particularities and Limitations

> None

Call context

> This function is called from ApplFbl

Prototype

```
void ApplFblFatalError( FBL_DECL_ASSERT_EXTENDED_INFO(vuint8 errorCode) )
```

Parameter

errorCode	Code number of the encountered assertion
module	Name of the affected module (Only if extended info is enabled)
line	Line number where the assertion occurred (configuration dependent)

Functional Description

If the bootloader assertions are enabled, this function will be called in case an invalid bootloader state is encountered.

Prototype

```
FUNC(Std_ReturnType, DET_CODE) ApplFbl_DetEntryHandler( uint16 ModuleId, uint8 InstanceId, uint8 ApiId, uint8 ErrorId )
```

Parameter

ModuleId	Code number of the encountered assertion
InstanceId	Name of the affected module (Only if extended info is enabled)
ApiId	Line number where the assertion occurred (configuration dependent)
ErrorId	Line number where the assertion occurred (configuration dependent)

Functional Description

This function is called by Det to filter error states which shall actually issue a DET error.

6.3 Validation and non-volatile memory callbacks

Prototype

```
static tFblResult ApplFblChgBlockValid( vuint8 mode, tBlockDescriptor descriptor )
```

Parameter

Mode	Determines if a block is validated or invalidated.
tBlockDescriptor	Block descriptor of logical block to be validated or invalidated.

Return code

tFblResult	Validation information stored correctly.
------------	--

Functional Description

This function changes the validation flag of a logical block in NV-memory.

Particularities and Limitations

- > The value is stored inverted.
- > Not used if presence patterns are activated.

Prototype

```
static vsint16 ApplFblGetPresencePatternBaseAddress( vuint8 blockNr,
                                                    IO_PositionType * pPresPtnAddr,
                                                    IO_SizeType * pPresPtnLen )
```

Parameter

blockNr	Logical block number
pPresPtnAddr	Presence pattern base address (output value)
pPresPtnLen	Presence pattern length (output value)

Return code

memSegment	Returns the index of the selected flash block. If no flash block is found, a negative value is returned.
------------	--

Functional Description

This function calculates the start address of the presence patterns of a specific logical block. It uses the logical block table to calculate the address, which is normally at the very end of a logical block.

Prototype

```
static tFblResult ApplFblSetModulePresence( tBlockDescriptor * blockDescriptor )
```

Parameter

blockDescriptor	Pointer to logical block descriptor.
-----------------	--------------------------------------

Return code

tFblResult	kFblOk if operation was successful, otherwise kFblFailed.
------------	---

Functional Description

This function sets the presence pattern for a specific logical block.

Particularities and Limitations

- > Previous erase operation must have cleared this area.

Prototype

```
static tFblResult ApplFblClrModulePresence( tBlockDescriptor * blockDescriptor )
```

Parameter

blockDescriptor	Pointer to logical block descriptor.
-----------------	--------------------------------------

Return code

tFblResult	kFblOk if operation was successful, otherwise kFblFailed.
------------	---

Functional Description

This function writes a value into the presence mask. This invalidates the block. Note that the presence pattern area must be cleared by a following erase operation.

Prototype

```
static tFblResult ApplFblChkModulePresence( tBlockDescriptor * blockDescriptor )
```

Parameter

blockDescriptor	Pointer to logical block descriptor.
-----------------	--------------------------------------

Return code

tFblResult	Returns if a logical block is valid.
------------	--------------------------------------

Functional Description

This function checks the presence pattern and mask if it indicates together a valid application.

Prototype

```
tFblResult ApplFblAdjustLbtBlockData( tBlockDescriptor * blockDescriptor )
```

Parameter

blockDescriptor	Pointer to logical block descriptor.
-----------------	--------------------------------------

Return code

tFblResult	Returns if a logical block is valid.
------------	--------------------------------------

Functional Description

This function is called by the diagnostic layer after the logical block has been erased according to the definition of the logical block table. Now in this function the logical block size can be re-adjusted to the flashable area. This gives the possibility to disallow the download of data into a particular area. With the use of presence pattern this function is used to spare the presence pattern and mask area from the download, so that no data can be downloaded into this section. Only after successful validation of the logical block the function ApplFblSetPresencePattern is allowed to write into this area. Otherwise, an invalid download could lead to valid presence pattern and accidental start of an invalid application.

Particularities and Limitations

> The block descriptor handed over by the diagnostics module is changed by this function.

Prototype	
<code>tFblAddress ApplFblGetModuleHeaderAddress(uint8 blockNr)</code>	
Parameter	
<code>blockNr</code>	Logical block number to get module header address from.
Return code	
<code>tFblResult</code>	Start address of the module header information
Functional Description	
<p>This function is used to determine the start address of the application's module header information which contains the software version information</p> <p>In the exemplary implementation, the module header information is expected at the start address of each Logical Block and this function returns the start address of the given Logical Block.</p>	

Prototype	
<code>tFblProgStatus ApplFblExtProgRequest(void)</code>	
Return code	
<code>tFblProgStatus</code>	External programming request set or not
Functional Description	
Function is called on power-on reset to check if an external programming request exists.	
Particularities and Limitations	
<p>> The programming request flag will be reset by this function.</p>	

Prototype	
<code>tApplStatus ApplFblGetApplStatus(void)</code>	
Return code	
<code>tApplStatus</code>	Validty status of the application kApplValid – Application is valid kApplInvalid – Application is not valid
Functional Description	
Returns kApplValid if application is valid (all mandatory blocks available) and application start is allowed. Otherwise it returns kApplInvalid.	

Prototype	
<code>tApplStatus ApplFblIsValidApp(void)</code>	
Return code	
<code>tApplStatus</code>	Application validity status
Functional Description	
Returns kApplValid if application is valid (all mandatory blocks available) and application start is allowed. Otherwise it returns kApplInvalid.	

Particularities and Limitations

- > This function checks if an abnormal shutdown occurred. If an abnormal shutdown is detected in this function, the return code is kApplInvalid.

Prototype

```
tFblResult ApplFblValidateApp( void )
```

Return code

tFblResult	Validation success
------------	--------------------

Functional Description

Returns OK if the application valid flag has been successfully written.

Prototype

```
tFblResult ApplFblInvalidateApp( void )
```

Return code

tFblResult	Invalidation success
------------	----------------------

Functional Description

Returns OK if the application valid flag has been successfully removed.

Prototype

```
tFblResult ApplFblValidateBlock( tBlockDescriptor blockDescriptor )
```

Parameter

blockDescriptor	Block descriptor structure of logical block which is currently validated.
-----------------	---

Return code

tFblResult	Validation information has been written correctly.
------------	--

Functional Description

Function is called after a successful download (CRC check was successful) to validate the logical block.

Prototype

```
tFblResult ApplFblInvalidateBlock( tBlockDescriptor blockDescriptor )
```

Parameter

blockDescriptor	Block descriptor structure of logical block which is currently invalidated.
-----------------	---

Return code

tFblResult	Validation information has been deleted correctly.
------------	--

Functional Description

Whenever the bootloader needs to delete data, this function is called to invalidate the current logical block.

Prototype

```
tFblResult ApplFblStoreFingerprint( vuint8 * buffer )
```

Parameter

buffer	Pointer to received fingerprint.
--------	----------------------------------

Return code

tFblResult	Result of storage operation.
------------	------------------------------

Functional Description

This function is called by the WriteDataByIdentifier service to store the received fingerprint in a temporary RAM buffer. The fingerprint is written into a non-volatile memory when the corresponding logical block is invalidated.

Prototype

```
void ApplFblErrorNotification( tFblErrorType errorType, tFblErrorCode errorCode )
```

Parameter

errorType	Indicates the error type (see fbl_def.h for a list of error types)
errorCode	Error code returned by the memory driver routine

Functional Description

Call-back function for diagnostic trouble code entries. This function is called whenever a programming error occurs.

Prototype

```
tFblResult ApplFblWriteCRCTotal( tBlockDescriptor * blockDescriptor,
                                vuint32 crcStart,
                                vuint32 crcLength,
                                vuint32 crcValue )
```

Parameter

blockDescriptor	Information about the logical block being downloaded.
crcStart	Start address of checksum calculation area
crcLength	Length of checksum calculation area
crcValue	Checksum

Return code

tFblResult	Write access result.
------------	----------------------

Functional Description

This function saves the CRC total to non-volatile memory.

Prototype

```
vuint8 ApplFblRWSecurityAccessDelayFlag( vuint8 mode, vuint8 value )
```


Parameter	
mode	Indicates if security access delay flag should be written or read.
value	Value to be written
Return code	
vuInt8	Status of security access delay flag.
Functional Description	
This function handles the security access delay flag. If the delay flag is set, a security access won't be possible (programming won't be possible). The delay flag is set after three unlock attempts. After a certain time period (default value: 10 minutes) the flag is cleared and the next attempt is possible.	

Prototype	
<pre>tFblResult ApplFblIncProgCounts(tBlockDescriptor blockDescriptor) tFblResult ApplFblIncProgAttempts(tBlockDescriptor blockDescriptor)</pre>	
Parameter	
blockDescriptor	Handle of current logical block.
Return code	
tFblResult	Indicates if the NV-memory access was successful.
Functional Description	
These functions are used to increment the programming counter and programming attempt counter for each logical block.	
Particularities and Limitations	
> Values are stored inverted.	

Prototype	
<pre>tFblResult ApplFblGetProgCounts(tBlockDescriptor blockDescriptor, vuInt16 * progCounts) tFblResult ApplFblGetProgAttempts(tBlockDescriptor blockDescriptor, vuInt16 * progAttempts)</pre>	
Parameter	
blockDescriptor	Handle of current logical block.
progCounts/Attempts	Count of successful program updates or programming attempts.
Return code	
tFblResult	Indicates if the NV-memory access was successful.
Functional Description	
Both functions are used to read the programming and programming attempt counters from NV-memory.	

Prototype	
<pre>tFblResult ApplFblWriteSecAccessInvalidCount(vuInt8 * invalidCount)</pre>	
Parameter	
invalidCount	Number of invalid security access attempts.

Return code

tFblResult	Indicates if the NV-memory access was successful.
------------	---

Functional Description

Write number of invalid security access attempts.

Particularities and Limitations

> Value is stored inverted.

Prototype

```
tFblResult ApplFblReadSecAccessInvalidCount(vuint8 * invalidCount)
```

Parameter

invalidCount	Number of invalid security access attempts.
--------------	---

Return code

tFblResult	Indicates if the NV-memory access was successful.
------------	---

Functional Description

Read number of invalid security access attempts.

Particularities and Limitations

Prototype

```
vuint16 ApplFblGetPromMaxProgAttempts(vuint8 blockNr)
```

Parameter

blockNr	Number of logical block.
---------	--------------------------

Return code

vuint16	Maximum number of allowed programming attempts.
---------	---

Functional Description

This function returns the number of allowed programming attempts for a given logical block.

Prototype

```
vuint8 ApplFblGetSecAccessDelayFlag( void )
```

Return code

vuint8	0 if security access delay flag is not set, != 0 if set.
--------	--

Functional Description

Returns the state of the security access delay flag.

Prototype

```
vuint8 ApplFblSetSecAccessDelayFlag( void )
```

Return code

vuint8	State of non-volatile memory access.
--------	--------------------------------------

Functional Description

Sets the security access delay flag in NV-memory.

Prototype

```
vuint8 ApplFblClrSecAccessDelayFlag( void )
```

Return code

vuint8	State of non-volatile memory access.
--------	--------------------------------------

Functional Description

Clears the security access delay flag in NV-memory.

Prototype

```
tFblResult ApplFblReadResetResponseFlag( vuint8* buffer );
```

Parameter

buffer	Read buffer
--------	-------------

Return code

tFblResult	Returns if NV-memory access was successful.
------------	---

Functional Description

Reads the reset response flag from NV-memory.

Particularities and Limitations

> Has to be evaluated in both application and bootloader.

Prototype

```
tFblResult ApplFblWriteResetResponseFlag( vuint8* buffer )
```

Parameter

buffer	Buffer with reset response flag.
--------	----------------------------------

Return code

tFblResult	Returns if NV-memory access was successful.
------------	---

Functional Description

Write reset response flag to NV-memory before reset.

Prototype

```
tFblResult ApplFblGetBlockHash( vuint8 blockNr, vuint8 *targetBuffer )
```

Parameter

blockNr	Number of logical block
targetBuffer	Buffer for logical block hash

Return code	
tFblResult	Return kFblOk if logical block is successfully copied into the buffer referenced by targetBuffer
Functional Description	
This function is called when the RoutineControl – getLogicalBlockHash is received. It has to provide the logical block hash of the requested logical block for the diagnostic response.	

Prototype	
tFblResult ApplFblProgStatusUpdate(vuint8 progStatSet, vuint8 progStatClr)	
Parameter	
progStatSet	Status bits to be set
progStatClr	Status bits to be cleared
Return code	
tFblResult	Return kFblOk if the programming status is updated successfully.
Functional Description	
Update the programming status in NVM	

Prototype	
eraseVer	
Parameter	
eraseVer	Erase Verification Status
Return code	
tFblResult	Return kFblOk if the programming status is updated successfully.
Functional Description	
Update the programming status in NVM	

Prototype	
tFlashStatus ApplFblGetBlockErased(tBlockDescriptor blockDescriptor)	
Parameter	
blockDescriptor	The blockDescriptor provides information about the logical block in question, like the block number, the block start address and the block length.
Return code	
kFlashErased	Flash area of logical block is actually erased and can be programmed without further flash erase operation
kFlashNotErased	Flash area of logical block is not erased.

Functional Description

- > The Bootloader maintains the memory status information in the fingerprint status byte for each logical block. After the erase routine is successfully passed for a logical block the memory status is set to state "Memory Erased". Furthermore, with the first flash write access to a logical block, the memory status information is set to "Memory Programmed".
- > Function `ApplFblGetBlockErased` is called from the `RoutineControl – eraseMemory` service implementation if the memory status information is set to "Memory Erased" in order to make sure that the flash erase operation for the current logical block can be safely omitted. Therefore, this function should return `kFlashErased` only if it is absolutely sure that the flash erase operation is not necessary..
- > If this function returns "kFlashErased", no flash erase operation is performed for the current logical block.
- > This function may be also called for other flash device types than the internal flash.

Particularities and Limitations

It depends on the applied flash memory hardware if a flash erase operation needs to be performed just before a programming operation because of e.g. the charge level of the flash cells.

The data retention time can be seriously violated if un-erased flash memory is programmed.

Flash devices can be damaged if un-erased flash memory is programmed.

6.4 Diagnostic Service Callbacks

Prototype

```
static tFblResult ApplFblReadProgAttemptCounter(vuint8 *targetBuffer, vuint32 blockType)
```

Parameter

targetBuffer	The provided data is copied into the targetBuffer.
blockType	Type of logical block

Return code

tFblResult	Returns if NV-memory access was successful.
------------	---

Functional Description

The programming attempt counter and the maximum number of programming attempts are copied into the target buffer for all logical blocks of the given block type.

Prototype

```
static tFblResult ApplFblReadEcuDiagnosticIdentification(vuint8 *targetBuffer)
```

Parameter

targetBuffer	The diagnostic identification is copied into this buffer.
--------------	---

Return code

tFblResult	Returns if NV-memory access was successful.
------------	---

Functional Description

This function copies the requested ECU Diagnostic Identification into the target buffer.

Prototype

```
static tFblResult ApplFblReadVehicleManufEcuHwNumber(vuint8 *targetBuffer)
```

Parameter

targetBuffer	Buffer for ECU Hardware Number
--------------	--------------------------------

Return code

tFblResult	Returns if NV-memory access was successful.
------------	---

Functional Description

This function copies the requested DID data into the target buffer.

Prototype

```
void ApplDiagUserService( vuint8 * pbDiagData, tTpDataType diagReqDataLen )
void ApplDiagUserRoutine( vuint8 * pbDiagData, tTpDataType diagReqDataLen )
void ApplDiagUserSubFunction( vuint8 * pbDiagData, tTpDataType diagReqDataLen )
```

Parameter

pbDiagData	Pointer to diag service data (after SID!)
diagReqDataLen	Service data length (without SID!)

Functional Description

The flash bootloader supports a set of diagnostic services. All diagnostic services not supported by the bootloader are passed to three call-back functions.

The call-back function ApplDiagUserService is called whenever the bootloader receives an unknown service identifier, ApplDiagUserSubFunction is called whenever the bootloader receives an unknown sub-function identifier and ApplDiagUserRoutine is called whenever the bootloader receives an unknown routine identifier..

Prototype

```
tFblResult ApplFblCheckProgPreconditions( vuint8 * pbDiagData, tTpDataType diagReqDataLen)
```

Parameter

pbDiagData	Pointer to diag service data (after SID!)
diagReqDataLen	Service data length (without SID!)

Functional Description

Check programming preconditions. This function is called when the corresponding diagnostic service is received.

Prototype

```
void ApplFblReadDataByIdentifier( vuint8 * pbDiagData, tTpDataType diagReqDataLen )
```

Parameter	
pbDiagData	Pointer to diag service data (after SID!)
diagReqDataLen	Service data length (without SID!)
Functional Description	
<p>This function is called to handle Read-Data-By-Identifier requests.</p> <p>The routine must retrieve the data-identifier (DID) from the message buffer, fill the message buffer with the appropriate response, and set the global variable DiagDataLength to the number of bytes added to the buffer.</p>	

Prototype	
vuint8 ApplFblWriteDataByIdentifier(vuint8 * pbDiagData, tTpDataType diagReqDataLen)	
Parameter	
pbDiagData	Pointer to diag service data (after SID!)
diagReqDataLen	Service data length (without SID!)
Return code	
vuint8	Success of NV-memory access.
Functional Description	
<p>This function is called to handle Write-Data-By-Identifier requests.</p> <p>The routine must retrieve the data-identifier (DID) from the message buffer.</p>	
Particularities and Limitations	
<p>> This function supports a special return value besides kFblOk and kFblFailed to signal that a fingerprint has been received: kDiagReturnValidationOk.</p>	

Prototype	
tFblResult ApplFblReportDTCByStatusMask(vuint8* pbDiagData, tTpDataType diagReqDataLen)	
Parameter	
pbDiagData	Pointer to diag service data (after SID!)
diagReqDataLen	Service data length (without SID!)
Return code	
tFblResult	Success of NV-memory access.
Functional Description	
Service function for diagnostic service ReportDTCByStatus	
Particularities and Limitations	
<p>> This function sends the positive response</p>	

Prototype	
void ApplFblInitErrStatus(void)	

Functional Description

This routine is called to initialize the global variables used to save the FBL state when an error occurs. The state may be retrieved by sending a Read-Data-By-Identifier request with a data-identifier of \$7F.

Particularities and Limitations

- > FBL error state is available only if project state is set to „Integration“

6.5 Watchdog Callbacks

Prototype

```
void ApplFblWDInit( void )
```

Functional Description

The purpose of this function is to start the watchdog function of the ECU. The function is called only after the FBL has determined that it will not start the application (It is called shortly after the first call of ApplFblStartup()).

If the watchdog is initialized in this routine, then the FBL may start the application without starting the watchdog. In this case, the application will be responsible for starting the watchdog itself.

The FBL uses a hardware timer to determine when to reset the watchdog (via ApplFblWDTrigger()). The timer is initialized shortly after this routine is called. You must ensure that the watchdog timer will not reset the ECU before the FBL can call the trigger function for the first time.

Particularities and Limitations

- > In addition to initializing the hardware responsible for the watchdog, the global variable WDTimer must be initialized to the number of “ticks” that define the interval between calls to the trigger function. The interval is determined from your configuration, and is defined by the macro FBL_WATCHDOG_TIME.
- > You may decide to initialize the watchdog hardware in the ECU startup code or in ApplFblInit().

Prototype

```
void ApplFblWDLONG( void )
```

Functional Description

The purpose of this function is to synchronize the start of the application with the watchdog. The call gives you the opportunity to ensure that the watchdog will not interrupt the application’s startup.

The function is called just before the bootloader starts your application.

Particularities and Limitations

- > The function is not called when the FBL jumps to the application directly after power-on. The call is made when the FBL is ready to start the O.S. after a download. Care should be taken if the watchdog (WD) is initialized in ApplFblInit(), as no synchronization is performed.

Prototype

```
void ApplFblWDShort( void )
```

Particularities and Limitations

- > Currently not used in bootloader.

Prototype

```
void V_CALLBACK_NEAR ApplFblWDTrigger( void )
```


Functional Description

This function is called by `FblLookForWatchdog()` and contains the actual watchdog trigger code. Depending on the configuration, please note the following points:

- ▶ If the function is copied by `FblCopyWatchdog()`, it has to be relocatable and normally no function calls out of this function are allowed.
- ▶ If the function is placed in RAM by the linker, it has to be present before the first call.
- ▶ If any function calls are done, the called functions have to be placed in RAM as well.

7 Build your Bootloader

The bootloader demonstration is delivered with a make system which can be used to build the bootloader. The following files control how your bootloader is built:

- **Makefile:** Compiler dependent settings like the path to your compiler installation and compiler options. To rebuild the demonstrations bootloader, you have to adapt the compiler path in this file.



Example

```
#-----  
#----- MUST be filled out -----  
# Define Compiler path  
# E.g.: COMPILER_BASE = C:\Uti\HC08\HIWARE  
#       COMPILER_BIN  = $(COMPILER_BASE)\prog  
#       COMPILER_INC   = $(COMPILER_BASE)\lib\hc08c\include  
#       COMPILER_LIB   = $(COMPILER_BASE)\lib\hc08c\lib  
#-----  
COMPILER_BASE = <path to your compiler>
```

- **Makefile.config:** Project specific settings.
- **Makefile.<platform>.<compiler>.ALL.make:** This file is used to generate the linker control file.
- **Makefile.project.part.defines:** Files and modules included in your bootloader project are configured in this file.

7.1 Bootloader specific linker requirements

The bootloader has to execute parts of its code from RAM because flash read accesses are normally not permitted while flash memory is written or erased.



Example

Please see the linker control file and the MemMap.h file of the bootloader demonstration project as an example.

These files also can be used to get a detailed list of functions which have to be linked to RAM.

Please make sure the following parts of the bootloader are linked to RAM:

Module / function	Bootloader use case		
	Standard Bootloader	Pipelined Verification	Pipelined Programming
FblLookForWatchdog()	■	■	■
FblLookForWatchdogVoid()	■	■	■
ApplFblWdTrigger()	■	■	■
FblMemRxNotification()			■
FblDiagRx*()			■
FblDiagTx*()			■
FblDiagDefaultPostHandler, other post handler functions			■
fbl_hw.c (Code and constants)			■
fbl_cw.c (Code and constants)			■
fbl_tp.c (Code and constants)			■
Communication Stack (Code and constants)	■	■	■

Table 7-1 RAM/ROM linkage



Caution

Depending on your hardware, additional functions or modules may have to be linked to RAM.

8 Adapt your application

8.1 Memory Layout

Bootloader and application mostly occupy different memory areas. There is only one shared area which is linked to the application memory by the bootloader and used to start the application software.

The bootloader has to be linked to a memory area which can be started by your microcontroller, e.g. by the reset vector or a reset data structure. Please see also [\[#hw_mem\]](#).

8.2 Application Start

The application software is started using the so-called application vector table. This data structure is included in the bootloader but linked to the application memory. It is overwritten by the application during flashing.

There are 2 variants of this table. Controllers with a configurable interrupt vector base address include a smaller table; controllers without a configurable interrupt vector base address include a complete interrupt vector table in bootloader and application.

8.2.1 Controllers with a configurable interrupt base address

Controllers with a configurable interrupt vector base address need to include a data structure which includes the start address of the application software. This structure is included in the file `fbl_applvect.c` and normally consists of a magic word and the address. A similar structure has to be added to the application software and linked to the exactly same address than the bootloader uses.

8.2.2 Controllers with interrupt jump tables

If the interrupt vector base address is not configurable, it is normally located in the bootloader's address space. The bootloader includes a interrupt vector table with jump opcodes. These jump opcodes point to a second vector table which is placed in the application memory and can be used like the original interrupt vector table. This table also is linked with the bootloader (mostly `fbl_applvect.c`, but an assembler version may be used depending on the compiler). The application variant can be created from the file `.\BSW\Fbl\Template_applvect.c`.



Note

Please note that the position of the interrupt vector table is fixed by the bootloader setup and cannot be changed independently by the application software.

8.3 Shared files between FBL and application

Bootloader and application do not have to share files. However, if some files are shared data exchange between both is made easier.

The FblHeader struct is defined in fbl_main.h. Access macros which can be used in the application software are also defined in this file. The fbl_main.h header file depends on fbl_def.h and fbl_cfg.h.

**Caution**

If “fbl_main.h”, “fbl_def.h” and “fbl_cfg.h” are used in the application software, you have to make sure the files are consistent between bootloader and application.

8.4 Shared memory between FBL and application

There is no memory, which is shared between the FBL and the application. Each project (FBL and application) uses its own stack, RAM and ROM.

Besides this, bootloader and application have to share some information:

- ▶ The transition between from bootloader to application and back has to be coordinated. In general, a reset response flag and a programming request flag has to be passed between both.
- ▶ Depending on your ECU's requirements, process data like fingerprints or version numbers have to be exchanged between bootloader and application.

User specific extensions can be added using the so-called common data structure. This data structure is added to FblHeader and can be defined according to the data which has to be shared between bootloader and application.

**Note**

The common data structure can be activated by the switch FBL_ENABLE_COMMON_DATA in a user configuration file.

The type definition and constant which contains the shared data should be added to a user modifiable file like fbl_ap.c.

**Caution**

The variable name of this structure has to be fblCommonData.

8.5 Transition between Bootloader and Application

8.5.1 [#oem_start] [#oem_trans] - Programming Session Request

There are two possibilities to handle the programming session request:

- ▶ Shared memory: A shared memory location either in non-volatile memory or uninitialized RAM is used to hand over the programming session request. This memory area has to be written by the application software. After a reset has been issued, the bootloader has to read the programming session request flag in `ApplFblExtProgRequest()` and clear the flag.
- ▶ FblStart is used. The FblStart-feature provides a possibility to store the programming session request flag completely in bootloader context. The `FblHeader`-structure provides a function pointer to a bootloader function called “`FblStart()`”. This function can be called using `CallFblStart()`. The function in bootloader context is started, a flag will be set in RAM and the bootloader issues a reset. After this reset, the flag is read by `ApplFblExtProgRequest()`.



Caution

`CallFblStart()` does not return to the application software. Please make sure all necessary download tasks, e.g. shutting down the OS and running NVM tasks, are finished before executing `CallFblStart()`



Caution

Consider the usage of RAM pattern carefully. It can cause issues, e.g. as follows:

1. The bootloader cannot be started, because the RAM contents get lost after reset (e.g. by a short loss of power on the micro or initialization through the startup code).
2. The bootloader is accidentally started after power-up, because the RAM location contains by coincidence the specified pattern named above.
3. The RAM pattern will be allocated by the bootloader anywhere in RAM. This may overlap with data inside the application. If you write into `fblStartMagicFlag[]` either by call of `CallFblStart()` or `ApplSetStartMagicFlag()`, you potentially overwrite your data with these. The ECU should be shutdown then immediately. This could otherwise cause unpredictable behavior of the ECU.

If additional data should be transmitted with the `FblStart` function call, a parameter can be used to transmit this data. It is copied to a memory area which is placed together with the magic flag.

**Caution**

Vector SLP3 bootloaders do not support a FblStart() parameter by default. A parameter can be added using the switch FBL_MAIN_ENABLE_FBLSTART_PARAM, but the respective data structures have to be defined in user callback files.

To define a parameter for the FblStart() function, the following settings have to be made:

- ▶ FBL_MAIN_ENABLE_FBLSTART_PARAM has to be set.
- ▶ The parameter type tFblStartParamter has to be defined.
- ▶ A pointer to this parameter (as void pointer) and the size of the parameter have to be added to the FblStart() function call.

8.5.2 ECU Reset and Default Session Request

If an ECU Reset request or a Default Session Request is received by the bootloader, there are two possibilities to handle the response:

- ▶ The response is sent by the bootloader and the reset is issued after the response was sent. This is the easier variant, but the tester has to take into account that the reset takes some time.
- ▶ The bootloader sends a Response Pending message, sets a flag and issues a reset. The application software has to read this flag and send the response. This feature is called "Response After Reset". In this case, the next request can be sent immediately, but the flag has to be handed over via RAM or non-volatile memory.

9 Memory Drivers

9.1 Flash Driver

The flash driver uses the HIS interface as described in [3]. The flash driver is downloaded to the ECU. It is copied and executed into RAM during runtime.

**Note**

There is usually no need to compile, link, or modify the flash driver in any way.

Relocatable flash drivers should always be located at address 0x00000000.

Additional flash drivers can be added to the bootloader configuration by yourself.

9.2 Non-volatile Memory Driver

The bootloader needs to use a non-volatile memory driver like an EEPROM driver or EEPROM emulation to store bootloader process data. The standard bootloader does not include such a driver.

**Caution**

The dummy EEPROM driver included in every bootloader delivery must not be used for production ECUs because it does not store data reset safe.

Non-volatile memory accesses can be configured by NV-Wrapper (see [8]) and are encapsulated in fbl_apnv.c/h.

Either module based EEPROM emulations or HIS interface (see [3]) EEPROM drivers can be used.

**Note**

If you don't want to use your own driver, various EEPROM drivers or EEPROM emulations can be ordered with the bootloader.

9.3 Memory driver requirements

Additional memory drivers have to fulfill a few requirements to fit into the bootloader environment. These requirements are valid to flash driver, EEPROM drivers and EEPROM emulations.

- ▶ The drivers have to provide a synchronous interface.

- ▶ Drivers mustn't use interrupts.
- ▶ The drivers have to call `FblLookForWatchdog()` at least once a millisecond. It is required to call it more often.

A detailed description of memory driver requirements can be found in [11] AN-ISC-8-1188 – Custom Flash Drivers.

10 Miscellaneous

10.1 [#oem_valid] - Application validation

For more general information about this see the UserManual_FlashBootloader in the chapter **Proposals for Handling the Validation Area**.

The bootloader has to decide if a valid application software is present or not. If the application is valid, the bootloader will start the application on the ECU's startup.

There are several variants to handle the application validation. Please see [9] for a short description of the available variants.



Note

FCA SLP5 bootloaders use the following validation strategy by default:

- ▶ The application's overall validity is established in service CheckProgrammingDependencies.
- ▶ During startup, the result of this check is evaluated.
- ▶ Either non-volatile memory or presence patterns can be used to store the validity information.

FCA SLP5 bootloaders use the functions ApplFbIsValidApp(), ApplFbIValidateApp(), ApplFbIInvalidateApp(), ApplFbIInvalidateBlock() and ApplFbIValidateBlock() to control the application valid status. An application wide compatibility check should be implemented in ApplFbICheckProgDependencies().

A second validation strategy without application wide validation flag can be selected in a user configuration file in DaVinci (deactivation of "Application Validity Flag"):

This configuration switch deactivates the application wide validation flag and uses the logical block validity flags combined with a compatibility check during the bootloader startup phase to establish the application validity.



Caution

Please note that using memory with ECC functionality may be critical to determine the application validity. If such a memory is used, the read function has to be able to handle errors.

Otherwise, the application and bootloader may get stuck in an endless reset loop because ECC errors trigger a reset or exception on every startup.

10.1.1 Presence Patterns

Instead of using flags in EEPROM, the validation status can be written into the flash memory of the respective logical block. Due to the fact that you cannot program a flash cell twice,

two segments are used to mark the logical block as valid or invalid. The handling of presence patterns is optional and can be activated in GENy.

The validation and invalidation is handled with two patterns at the end of each logical block: the mask value and the presence pattern. 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 values, the pattern 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 10-1 Presence Pattern

Invalidate: The mask value is written to its inverse of the erased state (the erased state is stored in the LogicalBlockTable). This makes the logical block invalid without re-writing to the presence pattern location.

Validate: The presence pattern is written to its location. Since both, the Pattern values and mask has been erased, it is possible to re-write and the mask contains the required erased state.

IsValid: The logical block is treaded as present, if the presence pattern has been written and the mask value contains its erased state.

The erasure will happen between the call to `ApplFblInvalidateBlock()` and `ApplFblValidateBlock()`. This ensures that the mask and value have been erased before.

The whole procedure guarantees that in any case the logical block is valid or not depending on the actual state. Even if the erasure occurs from top to bottom or vice versa or the erasure is interrupted, the logical block is invalid.



Note

Presence patterns either are used to store the application overall validity (if an application valid flag is used) or the logical block validity.

If the application valid flag is used, the block validity information is stored in non-volatile memory.

10.1.1.1 Storage of application valid flag

The application valid flag is shared over all logical blocks. To provide flexibility regarding the sequence of logical blocks, the presence patterns are stored according to the following rules:

1. Every logical block contains one application validity pattern. One valid application pattern will be interpreted as application valid pattern even if all other patterns are undefined or invalidated.

2. If one logical block is erased, the application validity patterns in all logical blocks are invalidated.
3. After a successful compatibility check, the presence pattern in the last logical block which has been flashed is validated. This ensures that only one block contains the application valid flag
4. During startup, all logical blocks are checked for an application valid flag. If the application valid flag is found, the application is started by the bootloader.

Figure 10-2 - Presence Pattern Examples demonstrates four states of an ECU using presence patterns to store the application valid flag.

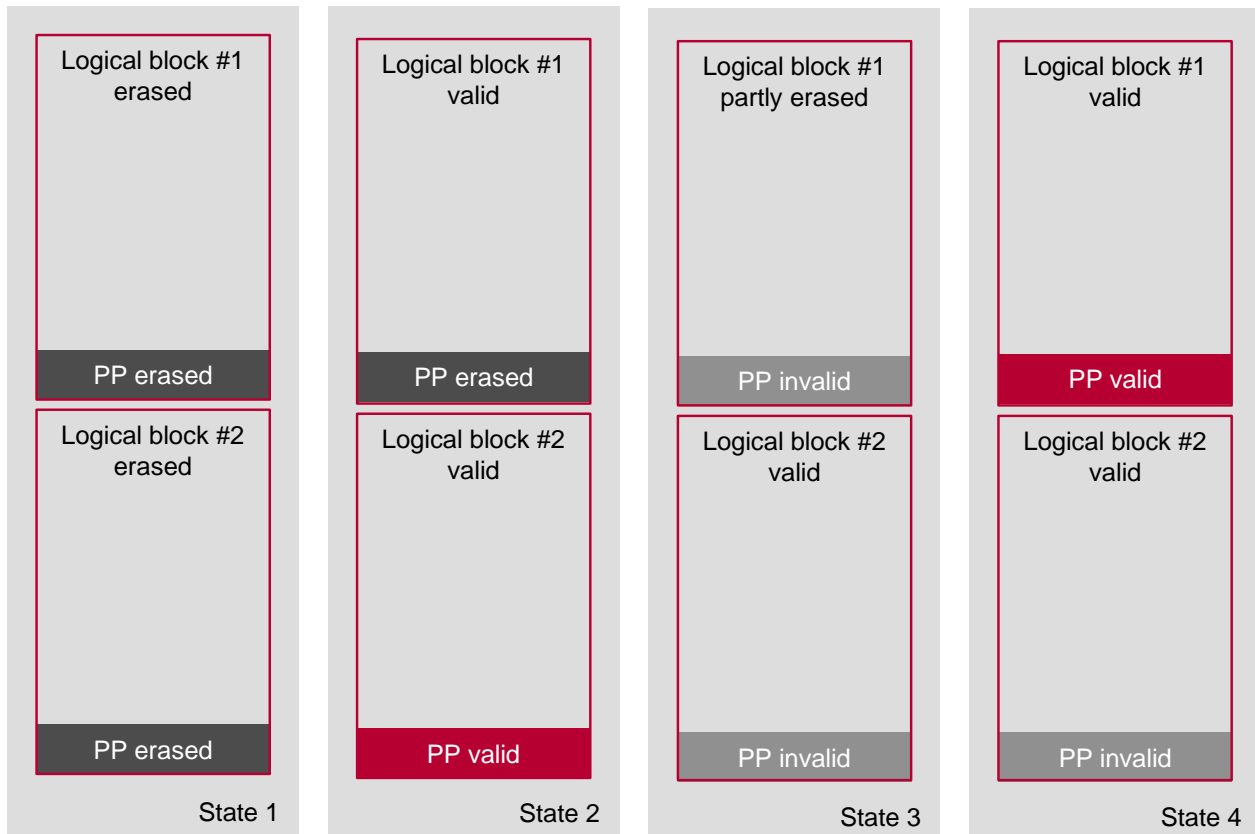


Figure 10-2 Presence Pattern Examples

- ▶ State 1: ECU completely erased. Both presence patterns are recognized as invalid and the application is not started.
- ▶ State 2: Logical block #1 and logical block #2 have been flashed successfully. Both blocks are compatible and the presence pattern is placed at the end of the logical block which has been flashed at last.
- ▶ State 3: Logical block #1 shall be re-programmed, but erasing the first logical block failed. Both patterns have been invalidated before erasing block #1, then erasing of block #1 started and a reset occurred. After this reset, the application is not started.
- ▶ State 4: Logical block #1 has been re-programmed and the compatibility check succeeded. The presence pattern is placed into block #1, the application is started after reset.

10.1.2 [#oem_time] - Validation OK – Application faulty

For more general information about this see the UserManual_FlashBootloader in the chapter **Validation OK – Application faulty**.

10.2 [#oem_sec] Seed / Key Mechanism

The Seed/Key mechanism is implemented in user callback files. The diagnostic layer uses the functions ApplFblSecuritySeed() and ApplFblSecurityKey() to create seed values and verify the keys. The actual calculation is done in the module Sec_SeedKeyVendor.c/Sec_SeedKey_Cfg.h which can be seen as a template for a Seed/Key module. Length of Seed and Key and the used algorithm can be configured in there.

The ApplFblSecuritySeed() function is called to obtain a (pseudo) random value to be sent to the tester. The function is called when a security access service request is received containing a request seed sub-function.



Caution

The request seed service should return a random value. Please check if your ECU is set up correctly so that a random value is returned.



Caution

There is no safe key calculation algorithm included in this bootloader by default. Please use the specified callbacks to implement your own, safe algorithm.

The example implementation is based on a simple XOR operation with the key constant configured in GENy.

Please note that the example DLL delivered with the bootloader implements a non-configurable constant of 0xFFFFFFFF.

The following settings can be done in the security access template. The settings are documented in detail by comments in the template files.

- ▶ Size of seed and key: SEC_SEED_LENGTH, SEC_KEY_LENGTH
- ▶ Format of seed and key (Byte arrays or word arrays).
- ▶ Security access algorithm implementation: SecM_ComputeKeyVendor()

**Note**

The Seed/Key calculation DLL used by vFlash has to be adapted as well if the Seed/Key algorithm is changed.

10.3 [#oem_seccomp] - Security module

The security module configuration is described in a separate document – see [10] for more information.

10.4 Data Processing Support

In order to support encrypted and/or compressed files, the FBL provides three interface functions that process the received data before programming.

ApplFblInitDataProcessing is called to initialize the data processing functionality with each RequestDownload service request.

ApplFblDataProcessing is called to perform data processing for data that is received with each TransferData service request.

ApplFblDeinitDataProcessing is called to conclude the data processing when a RequestTransferExit service request is received.

The order of data processing operations must be considered when the download files are prepared. The standard sequence is depicted in the figure below. There may be deviations from this sequence for special use cases.

In the first step, the signature or the CRC checksum is generated upon the raw download data in order to perform the counterpart operation on the ECU after the downloaded data have been programmed into NV-memory. If downloading of compressed data shall be supported, the compressed download file is computed next. The compression step should be performed before a potential encryption of the download data because the compression algorithm normally does not have much effect on encrypted data. Therefore, the last step is the encryption. During download, the reverse operations are performed on the ECU.

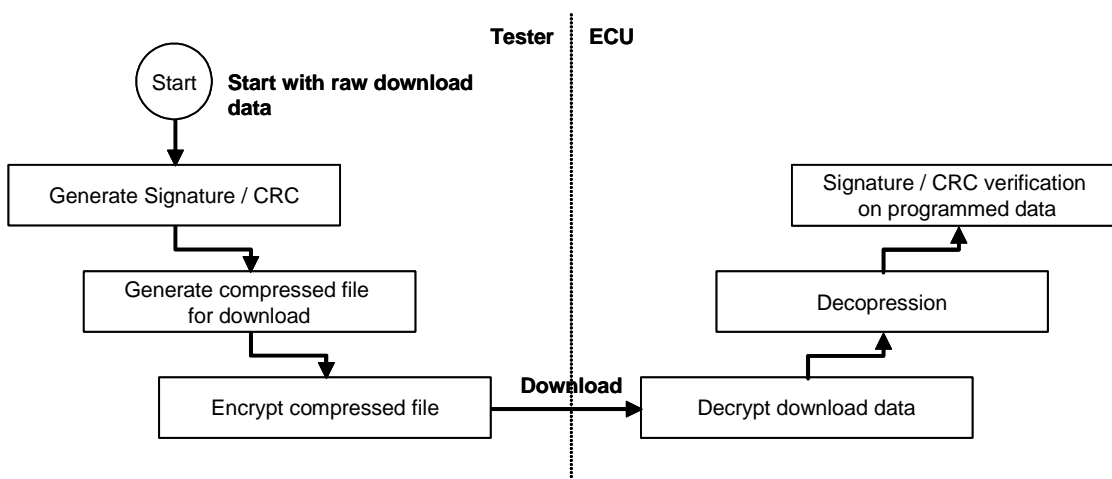


Figure 10-3 Data processing sequence

10.4.1 tProcParam

The data processing interface uses the tProcParam data structure in order to handle the input and output parameters.

tProcParam		
Element	Type	Description
dataBuffer	vuint8*	Pointer to input data buffer
dataLength	vuint16	Size of input data
dataOutBuffer	vuint8*	Pointer to output data buffer
dataOutLength	vuint16	Size of output data
dataOutMaxLength	vuint16	Size of output data buffer
wdTriggerFct	vuint8 (*wdTriggerFct)(void)	Pointer to watchdog service function
mode	vuint8	Data processing mode; equals the DFI which is passed with RequestDownload

Table 10-1 Type Definition - tProcParam

10.5 [#oem_multi] - Multiple ECU Support

This bootloader supports a so-called multiple ECU support. If this option is activated, several CAN identifiers are prepared to be used by the bootloader. During the bootloader startup, the function ApplFblCanParamInit() can be used to select the required identifier. This is useful for configuration which include several ECUs of the same kind, e.g. door control ECUs, in one vehicle.

Multiple ECUs can be activated in GENy by selecting more than one node in the channel setup dialog:

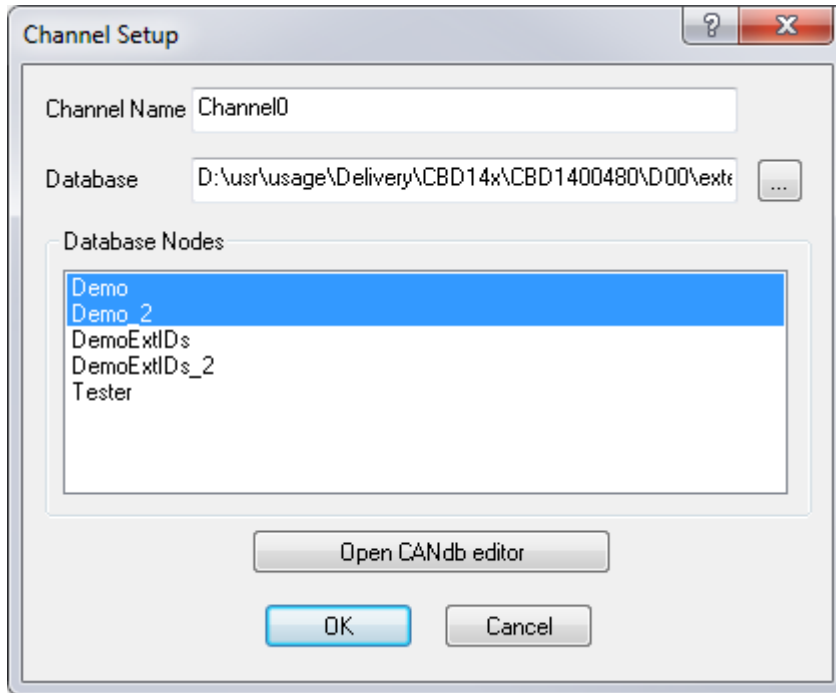


Figure 10-4 Multiple ECU setup

The GENy module FbiWrapperCom_Can allows manual adaptations of the settings which are read from the network database. To keep these manual adaptations in case a new database is selected, the configuration has to be updated manually in a second step using the “Restore default” button in the GENy module FbiWrapperCom_Can.



Caution

Manually adapted configuration settings will be overwritten in case the communication setup is reset.

10.6 Pipelined Programming and Pipelined Verification

Optionally, FCA SLP5 bootloaders can use the “Pipelined Programming” and/or “Pipelined Verification” feature to speed up the programming process. If one of these features is enabled, some tasks are executed in parallel to speed up the download process:

- ▶ **Pipelined Verification:** Written data is verified while additional data is received.
- ▶ **Pipelined Programming:** Received data is processed and written to flash while data is received.

**Caution**

If Pipelined Programming is activated, errors message from the bootloader may be delayed.

**Note**

The possible speedup is hardware dependent.

10.7 Bootmanager

The FCA SLP5 Flash-Bootloader provides a configurable and adaptable boot manager. The boot manager sources can be found in folder Bsw\FblBm. The basic operation of the boot manager is to perform a set of boot condition checks. If a boot condition is fulfilled, a corresponding callout function is called. Both the condition check and the corresponding action are performed in callout functions. The template implementations of the callout functions can be found in Bsw\Fbl\Template\fbl_apbm.c.

The boot condition checks can be freely parameterized by a checklist. The boot manager processes the checklist subsequently. Basically, the boot manager can check e.g. if the Flash-Bootloader, a Flash-Bootloader-Updater or the application is valid. To perform the check, the corresponding application has to provide the information structure tBmInfoHeader which is defined in fbl_bmtypes.h. With this information structure, the boot manager can determine the validity status of the corresponding application can perform a verification step and can call it if the check is successful.

10.8 FCA-Specific Functionality

10.8.1 Download of SWIL / Flash Driver

Per default the flash driver is part of the bootloader code in ROM and is copied into RAM when the download sequence is started. Optionally, the flash driver can be downloaded as SWIL in the download sequence. The FBL can be configured accordingly with the configuration option “Flash Driver Usage” on the FblGeneral component of the DaVinci Configurator.

After an application download has been completed the flash driver code is removed from its RAM buffer.

10.8.2 Abnormal Shutdown

The FCA SLP5 FBL permits to detect an abnormal shutdown. In this case, the FBL does not start the application immediately after PowerOn/Reset if a valid application is available. Instead a configurable time window in bootloader is activated that can be used to perform a download. This can be helpful if the application has got some fault that leads to an abnormal shutdown

10.8.3 Flash Erase Detection

The FBL provides an option to determine if the flash memory of a logical block has been previously erased.

If this option is enabled, the Bootloader checks the memory status flag after a RoutineControl – eraseMemory request is received. In case that the memory status flag indicates “flash erased”, additionally the callback function ApplFblGetBlockErased () is called to ensure that the flash erase routine can be omitted. The flash erase routine is not carried out if the callback function returns the code “kFlashErased” in order to speed-up the download.

10.8.4 Update of Identification Data

After a successful download, the FBL copies identification data from the downloaded application into NV-memory. This is done in function ApplFblValidateBlock() and in ApplFblValidateApp().

**Note**

Not all DIDs have to be supported for a specific ECU. Please adapt the functions ApplFblValidateBlock() and ApplFblValidateApp() to copy the required information.

10.8.5 Update of FCA Identification Data

Several part number DIDs have to be updated in the download process and since it is not permitted to use the WriteDataByIdentifier service, the new part numbers must be part of the downloaded data. The FBL delivery provides a template structure which contains the new DID data that can be linked to the application. The definition of this structure can be found in the files applfbl.c/applfbl.h. This application header structure is used to update the identification information in NV-memory after a logical block respectively after the application is set as valid in the functions ApplFblValidateBlock() and ApplFblValidateApp(). Additionally, this structure provides compatibility versions for the dependency check.

10.8.6 Software Compatibility Information

To perform the checks for Hardware-Software and Software-Software compatibility, the application header structure also contains compatibility version information that is used for the RoutineControl -checkProgrammingDependencies service. The dependency check is implemented in function ApplFblCheckProgDependencies().

11 vFlash Configuration

The address based vFlash template delivered with the bootloader offers some configuration options to synchronize the download sequence between bootloader and vFlash:

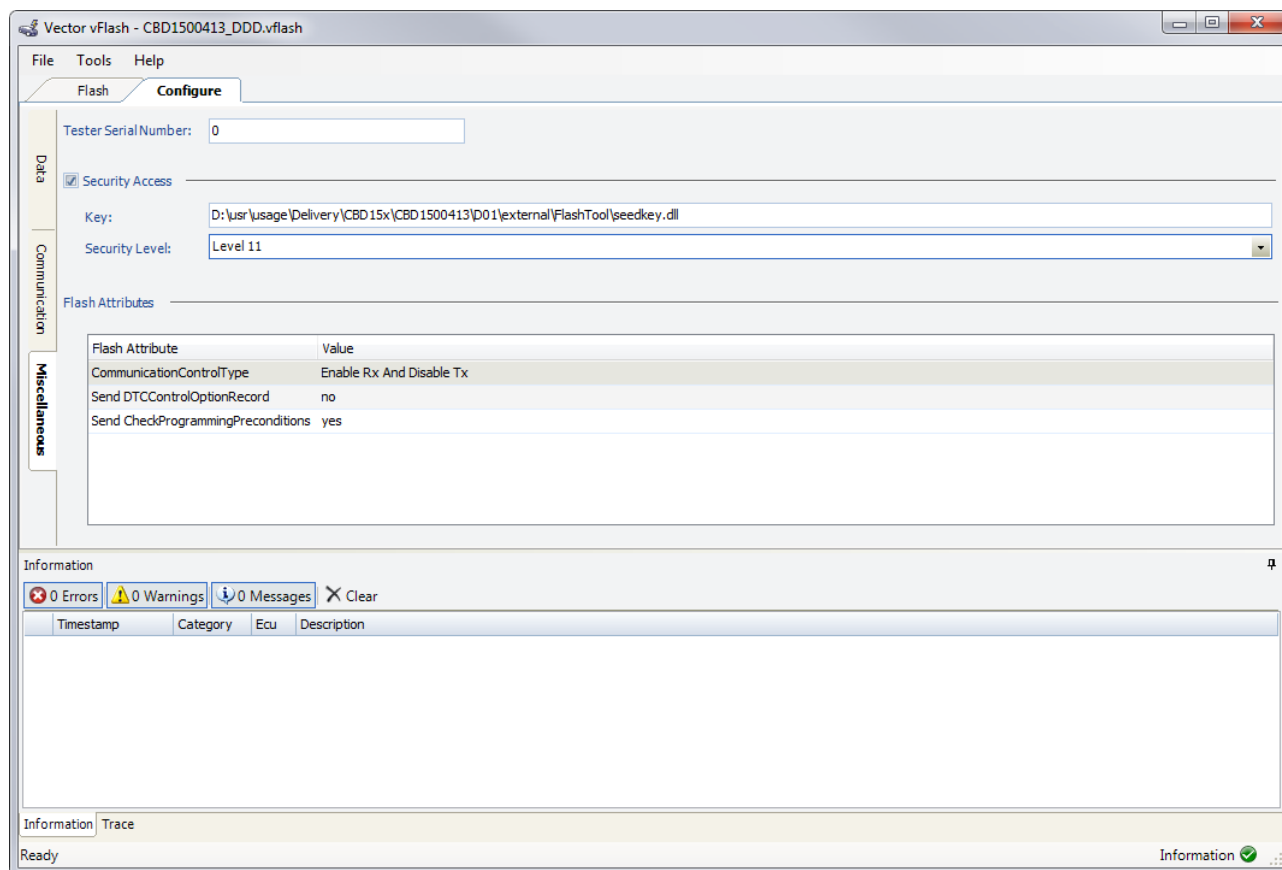


Figure 11-1 vFlash Configuration

Configuration Option	Description
Tester Serial Number	ASCII string which is appended to the current date (3 Bytes BCD, YYMMDD) and sent in the write fingerprint request. The length of this request is (3 + length of TesterSerialNumber) Bytes. Default value is 6 Bytes. If the length should be changed in the bootloader, please add the following define to a user config file: <code>#define kDiagRqIWriteDataByIdentifierFingerPrintParameter 0x09u</code>
Security level	Selects the security level used for Seed/Key. Default is level 11. The configuration can be changed by a user configuration file: <code>#define kDiagSubRequestSeed 0x11u</code> <code>#define kDiagSubSendKey 0x12u</code>
Communication Control Type	Corresponds to the GENy setting "Communication Control Type".

Configuration Option	Description
Send DTCCControlOptionRecord	Corresponds to the GENy setting “Control DTC Option Record”
Send CheckProgramming- Dependencies	Corresponds to the GENy setting “Check Programming Preconditions Service.”

Table 11-1 vFlash Configuration

12 Glossary and Abbreviations

12.1 Glossary

Term	Description
NV-Memory	Non-volatile memory which can be used to store small amounts of data during the bootloader's runtime.

12.2 Abbreviations

Abbreviation	Description
ALFI	Address/Length Format Identifier
BCD	Binary Coded Digit
CAN	Controller Area Network
Cfg5	Vector DaVinci Configurator 5
DID	Data Identifier
DFI	Data Format Identifier
FBL	Flash Bootloader
ECU	Electronic Control Unit
HIS	Herstellerinitiative Software
ISO	International Organization for Standardization
LIN	Local Interconnect Network
SWIL	Software Interlock
UDS	Unified Diagnostic Services

13 Contact

Visit our website for more information on

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

www.vector.com