

Flash Bootloader OEM

Technical Reference

Vector UDS (SLP3) - CAN configuration

Version 3.1

Authors	Achim Strobelt, Marco Riedl
Status	Released

Document Information

History

Author	Date	Version	Remarks
Achim Strobelt	2017-08-14	3.0	Rework based on Version 2.8
Achim Strobelt	2017-10-28	3.1	Corrected Request Download service description, added new Cfg configuration options

Reference Documents

No.	Source	Title	Version
[1]	ISO	14229 Road Vehicles – Unified diagnostic services (UDS) Part 1: Specification and Requirements	2005
[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	1.9.3
[7]	Vector	Manual vFlash	3.0
[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	2.1.0
[11]	Vector	AN-ISC-8-1188 – Custom Flash Drivers	1.0
[12]	Vector	Technical Reference Communication Wrapper PDU Router	1.3
[13]	Vector	LIN Driver – Technical Reference	2.xx.xx



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
2.1.1	Pre-Programming Step	9
2.1.2	Programming Step	10
2.1.3	Post-Programming Step	11
2.2	Detailed Description of Diagnostic Services	12
2.2.1	Diagnostic Session Control	12
2.2.2	ECU Reset.....	12
2.2.3	Read Data By Identifier	13
2.2.4	Security Access.....	13
2.2.4.1	Request Seed	14
2.2.4.2	Send Key	14
2.2.5	Communication Control.....	15
2.2.6	Write Data By Identifier	15
2.2.7	Routine Control	16
2.2.7.1	Check Routine	16
2.2.7.2	Check Programming Preconditions.....	16
2.2.7.3	Erase Memory	17
2.2.7.4	Check Programming Dependencies.....	17
2.2.7.5	Force Boot Mode	18
2.2.8	Request Download.....	18
2.2.9	Transfer Data	19
2.2.10	Request Transfer Exit.....	19
2.2.11	Tester Present.....	19
2.2.12	Control DTC Setting	20
2.3	Negative Response Codes	21
3	Flash Bootloader Delivery.....	22
3.1	Bootloader Components	22
3.2	Bootloader Integration.....	22
4	Bootloader Installation.....	23
4.1	[#oem_files] - Bootloader package structure	23
4.2	Demonstration Bootloader	24
5	Generation Tool (GENy)	25
5.1	Project Setup	25

5.2	CAN configuration.....	29
5.3	CAN ID configuration	30
5.4	Bootloader Configuration	33
5.5	Memory Configuration.....	38
5.5.1	Flash Block Table	38
5.5.1.1	Logical Block Table	39
5.5.2	Memory Device Table.....	39
5.5.3	NV-Wrapper Configuration	40
5.5.4	Running the Generator.....	42
6	User callback files and hardware specific adaptations.....	43
6.1	Startup code	43
6.2	Hardware, Input/Output and miscellaneous Callbacks	44
6.3	Validation and non-volatile memory callbacks	52
6.4	59	
6.5	Diagnostic Service Callbacks.....	60
6.6	Watchdog Callbacks	61
7	Build your Bootloader	63
7.1	Bootloader specific linker requirements.....	63
8	Adapt your application	65
8.1	Memory Layout	65
8.2	Application Start.....	65
8.2.1	Controllers with a configurable interrupt base address	65
8.2.2	Controllers with interrupt jump tables	65
8.3	Shared files between FBL and application	66
8.4	Shared memory between FBL and application	66
8.5	Transition between Bootloader and Application.....	67
8.5.1	[#oem_start] [#oem_trans] - Programming Session Request	67
8.5.2	ECU Reset and Default Session Request	68
9	Memory Drivers	69
9.1	Flash Driver	69
9.2	Non-volatile Memory Driver.....	69
9.3	Memory driver requirements	70
10	Miscellaneous	71
10.1	[#oem_valid] - Application validation	71
10.1.1	Presence Patterns	72
10.1.1.1	Storage of application valid flag	73
10.1.2	[#oem_time] - Validation OK – Application faulty	74

10.1.3	Force Boot Mode	74
10.2	[#oem_sec] Seed / Key Mechanism	75
10.3	[#oem_seccomp] - Security module	76
10.4	Data Processing Support	76
10.4.1	tProcParam	77
10.5	[#oem_multi] - Multiple ECU Support	77
10.6	Configuration without NV-Memory driver	78
10.7	Pipelined Programming and Pipelined Verification	79
11	vFlash Configuration	80
12	Glossary and Abbreviations	82
12.1	Glossary	82
12.2	Abbreviations	82
13	Contact	83

Illustrations

Figure 1-1	Manual and References for the Flash Bootloader	8
Figure 2-1	Basic Programming Sequence	9
Figure 2-2	Pre-Programming Step	9
Figure 2-3	Programming Step	10
Figure 2-4	Software & Data Download Sequence	10
Figure 2-5	Post-Programming Step	11
Figure 4-1	Bootloader Folder Structure	23
Figure 5-1	GENy Setup Dialog	25
Figure 5-2	GENy main window after pre-configuration	26
Figure 5-3	GENy Channel Setup (CAN or LIN)	26
Figure 5-4	GENy main window after channel setup	27
Figure 5-5	GENy Components	28
Figure 5-6	GENy CAN configuration	29
Figure 5-7	FblWrapperCom_Can GENy module	31
Figure 5-8	CAN connection table	32
Figure 5-9	GENy – CAN message table	32
Figure 5-10	GENy General FBL configuration	34
Figure 5-11	OEM specific UDS configuration	37
Figure 5-12	Flash Block Table	38
Figure 5-13	Logical Block Table	39
Figure 5-14	Memory Device Table	40
Figure 5-15	NV-Wrapper configuration	41
Figure 5-16	Meta Data configuration	41
Figure 10-1	Presence Pattern	72
Figure 10-2	Presence Pattern Examples	73
Figure 10-3	Data processing sequence	76
Figure 10-4	Multiple ECU setup	77
Figure 11-1	vFlash Configuration	80

Tables

Table 2-1	NRC codes used by Bootloader	21
Table 4-1	Bootloader Folder Structure	24
Table 5-1	FblWrapperCom_Can main page GENy options	31
Table 5-2	GENy – CAN connection table	32
Table 5-3	GENy – CAN message table	32
Table 5-4	GENy General FBL Configuration	37
Table 5-5	OEM UDS Specific Configuration	37
Table 5-6	Flash Block Table Configuration	38
Table 5-7	Logical Block Table Configuration	39
Table 5-8	Memory Device Table Configuration	40
Table 5-9	Generated Files	42
Table 7-1	RAM/ROM linkage	64
Table 10-1	Type Definition - tProcParam	77
Table 11-1	vFlash Configuration	81

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

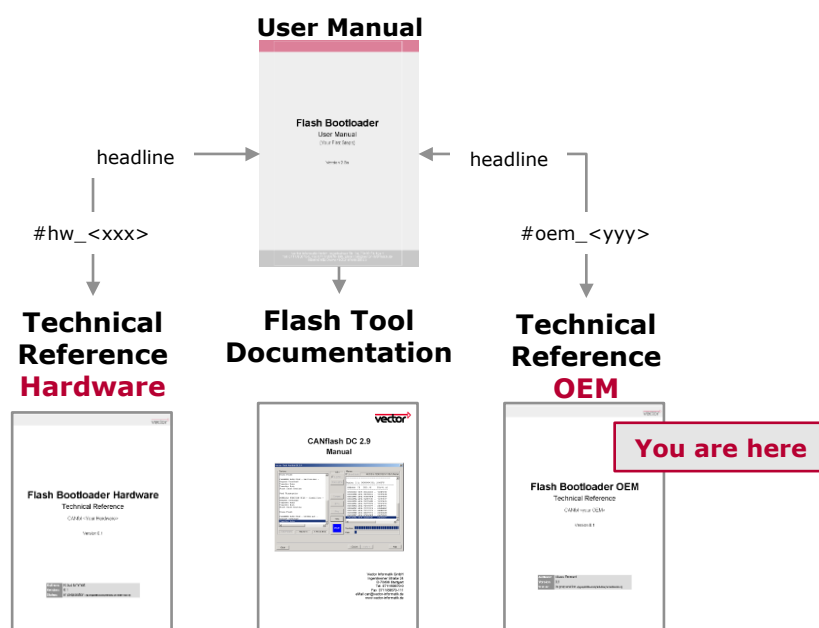


Figure 1-1 Manual and References for the Flash Bootloader

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 process can be divided into three basic steps:

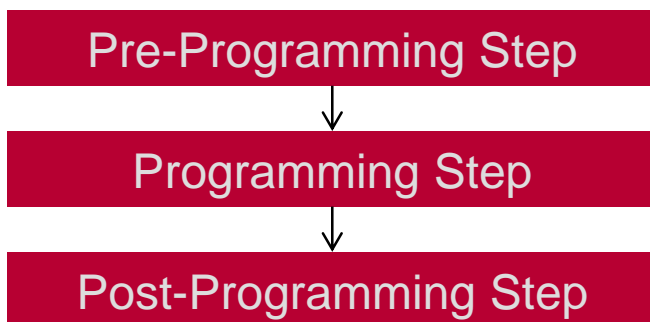


Figure 2-1 Basic Programming Sequence

The following chapters document the download sequence.

- ▶ Steps marked red are steps which are sent physically to the ECU which is flashed.
- ▶ Steps marked grey are sent functionally to all ECUs in the network.

2.1.1 Pre-Programming Step

The Pre-Programming Step prepares the vehicle's network for flashing. This includes disabling normal communication to increase the available bandwidth, stop logging of DTCs and a check to ensure the ECU which should be flashed is in a state which allows flashing.

The step "Routine Control – Check Programming Preconditions" is optional and can be deactivated in both bootloader and vFlash.

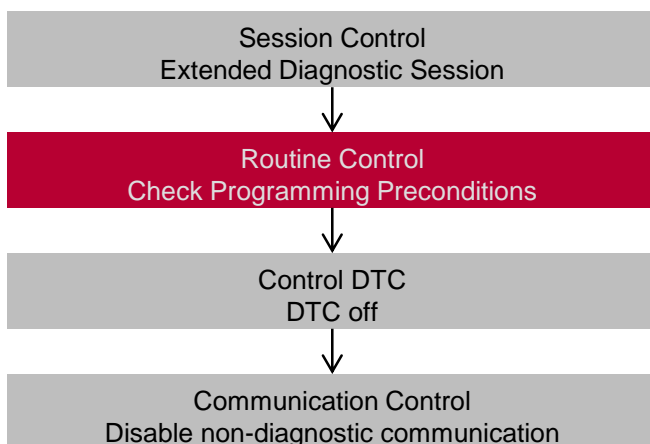


Figure 2-2 Pre-Programming Step

2.1.2 Programming Step

The Programming Step performs the actual software download to the ECU. The other ECUs in the network have to be kept in extended session by the tester while the ECU is flashed. This is achieved by sending Tester Present messages (\$3E) functionally addressed to all ECUs in the network.

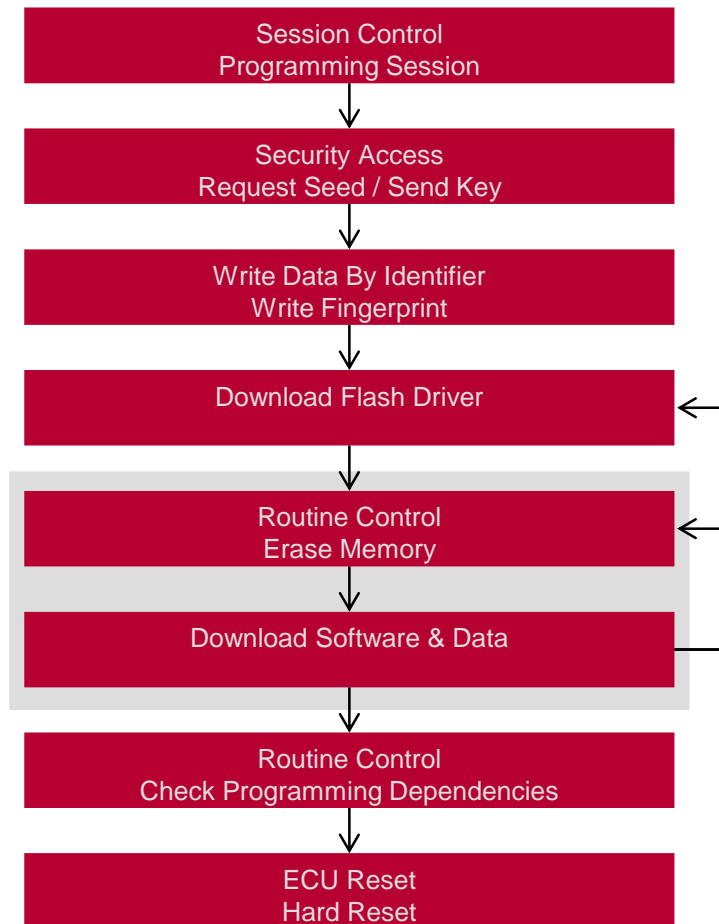


Figure 2-3 Programming Step

The download of flash driver, software and data use the same download sequence:

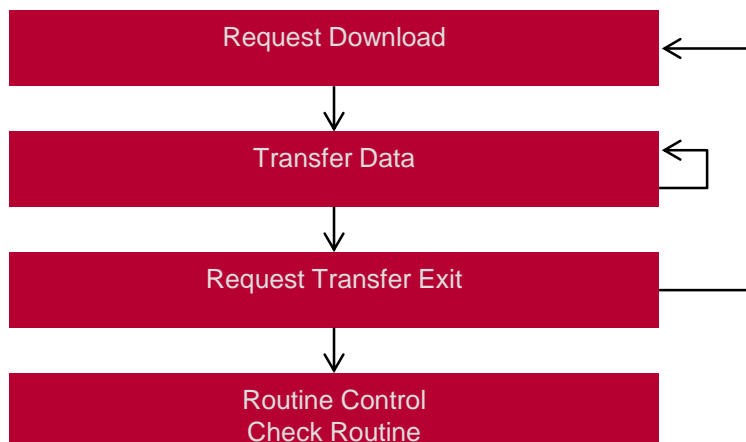


Figure 2-4 Software & Data Download Sequence

The flash driver normally is downloaded via CAN. It has to be downloaded before erasing the first logical block. After the first logical block, additional flash driver downloads are optional.

**Note**

The flash driver can be optionally stored in ROM. In this case, the flash driver download is no longer necessary.

Please note: This feature must be requested with the bootloader order.

Routine Control – Erase Memory and the Software & Data download sequence have to be executed for each logical block.

Depending on the bootloader configuration, one or more iterations of the services Request Download, Transfer Data and Request Transfer Exit are allowed for every logical block.

2.1.3 Post-Programming Step

The Post-Programming Step is used to reactivate the normal network behavior after flashing one ECU.

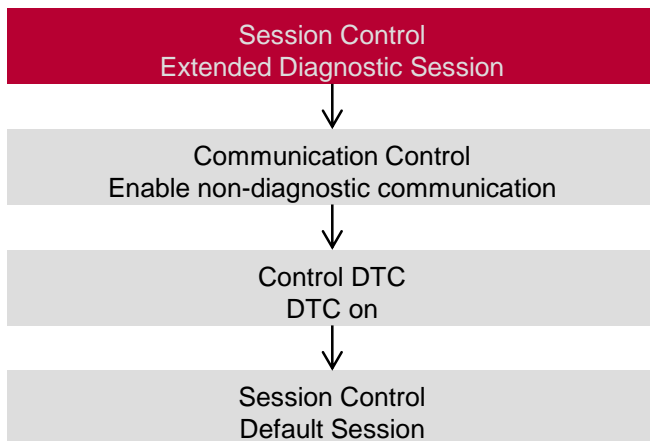


Figure 2-5 Post-Programming Step

2.2 Detailed Description of Diagnostic Services

The flash bootloader supports several diagnostic services. These services are relevant for the download sequence. A list and more detailed description of the diagnostic services can be found at [1].



Note

The service descriptions in [1] are generic definitions of UDS. Please note that this bootloader package specifies the usage and interpretation of some parameters if they are not specified in detail.

2.2.1 Diagnostic Session Control

This service is used to select the different diagnostic sessions in the ECU

Diagnostic Session Control \$10 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$10	Request Service ID
2	\$XX	Diagnostic Session Type <ul style="list-style-type: none"> ▶ \$01 Default Session ▶ \$02 Programming Session ▶ \$03 Extended Session
Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$50	Response Service ID
2	\$XX	Diagnostic Session Type (see byte #2 of request message)
3-4	\$XXXX	P2 _{CAN} _Server_Max
5-6	\$XXXX	P2* _{CAN} _Server_Max

2.2.2 ECU Reset

ECU Reset \$11 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$11	Request Service ID
2	\$01	Reset Type – Hard Reset
Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$51	Response Service ID
2	\$01	Reset Type – Hard Reset

2.2.3 Read Data By Identifier

Read Data By Identifier \$22 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$22	Request Service ID
2-3	\$XXXX	Data Identifier #1
...	\$XXXX	Data Identifier #n
Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$62	Response Service ID
2-3	\$XXXX	Data Identifier #1
4...x	\$XX	Requested data of DID #1
(x+1)-(x+2)	\$XXXX	Data Identifier #n
(x+3)...	\$XX	Requested data of DID #n

Several DIDs can be read with one request as long as they fit into the diagnostic buffer.

**Note**

This service is not necessary for the download sequence, but it can be used to obtain useful information from the ECU.

2.2.4 Security Access

This service is used to restrict access to certain services in the ECU, e.g. all services which write data to the ECU.

The seed/key mechanism is described in [4].

**Caution**

Please note that Vector SLP3 bootloaders include a demonstration seed/key algorithm. This algorithm does not provide safe access restriction in production ECUs and should be replaced by a safe algorithm.

2.2.4.1 Request Seed

Security Access; Request Seed \$27 \$11 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$27	Request Service ID
2	\$RS	Security Access Type – Request Seed
Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$67	Response Service ID
2	\$RS	Security Access Type – Request Seed
3..x	\$XX	Seed value

The security level (Request Seed sub-function) is configurable. It has to be configured to the same value in bootloader and vFlash. vFlash supports security levels \$01 - \$13.

2.2.4.2 Send Key

Security Access; Send Key \$27 \$12 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$27	Request Service ID
2	\$SK	Security Access Type – Send Key
3..x	\$XX	Key calculated by tester
Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$67	Response Service ID
2	\$SK	Security Access Type – Request Seed

The Send Key sub-function is always \$RS + 1 and depend on the Request Seed sub-function.

2.2.5 Communication Control

Transmission and/or reception of messages can be controlled using this service. This service is mainly used by application which uses non-diagnostic messages.

Communication Control \$28 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$28	Request Service ID
2	\$XX	Control Type <ul style="list-style-type: none"> ▶ \$00: Enable Rx and Tx ▶ \$01: Enable Rx and disable Tx
3	\$01	Communication Type – Normal communication
Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$68	Response Service ID
2	\$XX	Control Type <ul style="list-style-type: none"> ▶ \$00: Enable Rx and Tx ▶ Either \$01: Enable Rx and disable Tx ▶ or \$03: Disable Rx and Tx

One of the two possible sub-functions \$01 and \$03 is used in pre-programming sequence. It is configurable in bootloader and vFlash.

The post-programming sequence expects always sub-function \$00.

2.2.6 Write Data By Identifier

This service can be used to write information into the ECU.

Write Data By Identifier \$2E – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$2E	Request Service ID
2-3	\$XXXX	Data Identifier
4..x	\$XX	Data Record
Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$6E	Response Service ID
2-3	\$XXXX	Data Identifier

By default, the bootloader requires a write fingerprint service. The length of this service is configurable in vFlash and bootloader. If the DID should be changed from the default value of \$F15A to a different value, a project specific vFlash template has to be obtained.

2.2.7 Routine Control

This service is used to start a routine, stop a routine or request routine results. Only the start of routines is relevant for the FBL.

2.2.7.1 Check Routine

This routine is used to calculate a checksum or verify a signature of the downloaded data. The calculated checksum is compared to the downloaded checksum.

Routine Control; Check Routine \$31 \$01 \$02 \$02 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$31	Request Service ID
2	\$01	Routine Control Type – Start Routine
3-4	\$0202	Routine Identifier – Calculate Checksum
5..x	\$XX	Routine Control Option Record. These data bytes contain the checksum or signature which is compared to the calculated checksum or signature.

Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$71	Response Service ID
2	\$01	Routine Control Type – Start Routine
3-4	\$0202	Routine Identifier – Calculate Checksum
5	\$XX	Routine Status Record - \$00 indicates a successful checksum comparison, a different value indicates which kind of error occurred.

2.2.7.2 Check Programming Preconditions

This service is optional and can be de-activated in GENy.

Routine Control; Check Programming Preconditions \$31 \$01 \$02 \$03 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$31	Request Service ID
2	\$01	Routine Control Type – Start Routine
3-4	\$0203	Routine Identifier – Check Programming Preconditions

Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$71	Response Service ID
2	\$01	Routine Control Type – Start Routine
3-4	\$0203	Routine Identifier – Check Programming Preconditions
5..x	\$XX	Routine Status Record – contains the preconditions that are not fulfilled.

2.2.7.3 Erase Memory

Routine Control; Erase Memory \$31 \$01 \$FF \$00 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$31	Request Service ID
2	\$01	Routine Control Type – Start Routine
3-4	\$FF00	Routine Identifier – Erase Memory
5	\$nm	ALFI – supported values BCD coded between \$11 and \$44.
6..(6+n-1)	\$XX	Memory Address (n Bytes)
(6+n).. ((6+n)+m-1)	\$XX	Erase Length (m Bytes). Address based download only.

Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$71	Response Service ID
2	\$01	Routine Control Type – Start Routine
3-4	\$FF00	Routine Identifier – Erase Memory
5	\$XX	Routine Status Record - \$00 indicates a successful erase operation, a different value indicates which kind of error occurred.

2.2.7.4 Check Programming Dependencies

Routine Control; Check Routine \$31 \$01 \$FF \$01 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$31	Request Service ID
2	\$01	Routine Control Type – Start Routine
3-4	\$FF01	Routine Identifier – Check Programming Dependencies.

Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$71	Response Service ID
2	\$01	Routine Control Type – Start Routine
3-4	\$FF01	Routine Identifier – Check Programming Dependencies
5	\$XX	Routine Status Record - \$00 indicates a successful dependency check, a different value indicates which kind of error occurred.

2.2.7.5 Force Boot Mode

This service is optional and is available if “Stay in Boot” is enabled in GENy.

Routine Control; Check Routine \$31 \$01 \$F5 \$18 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$31	Request Service ID
2	\$01	Routine Control Type – Start Routine
3-4	\$F518	Routine Identifier – Stay In Boot
Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$71	Response Service ID
2	\$01	Routine Control Type – Start Routine
3-4	\$F518	Routine Identifier – Stay In Boot

2.2.8 Request Download

Request Download \$34 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$34	Request Service ID
2	\$xx	Data Format Identifier – Determines if a download is compressed or encrypted.
3	\$nm	ALFI – supported values BCD coded between \$11 and \$44.
4, m Bytes	\$xx	Memory Address. Length m Bytes.
4+m, n Bytes	\$xx	Memory Size. Length n Bytes
Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$74	Response Service ID
2	\$nm	Length Format Identifier – n determines the length of the maxNumberOfBlockLengthParameter..
3..x (n Bytes)	\$xx	Max Number Of Block Length. Contains the maximum number of bytes to be transmitted with each transfer data service.

2.2.9 Transfer Data

Transfer Data \$36 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$36	Request Service ID
2	\$XX	Block Sequence Counter. This value starts at \$01 and is incremented with each following transfer data service.
3..x	\$XX	Transfer Data Parameter Record. Contains the downloaded data.
Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$76	Response Service ID
2	\$XX	Block Sequence Counter

2.2.10 Request Transfer Exit

Request Transfer Exit \$37 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$37	Request Service ID
Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$77	Response Service ID

2.2.11 Tester Present

Tester Present \$3E – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$3E	Request Service ID
2	\$00	Zero Sub-function
Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$7E	Response Service ID
2	\$00	Zero Sub-function

2.2.12 Control DTC Setting

Control DTC Setting \$85 – Request Message Format		
Data Byte#	Data Value	Parameter Description
1	\$85	Request Service ID
2	\$01/\$02	Control DTC Setting Type: \$01 = on, \$02 = off
3..5	\$FFFFFF	Control DTC option record. Deactivate all DTCs (optional, can be configured in GENy and vFlash)
Positive Response Message Format		
Data Byte#	Data Value	Parameter Description
1	\$C5	Response Service ID
2	\$01/02	Control DTC Setting Type

2.3 Negative Response Codes

The bootloader uses negative responses as described in the UDS specification [1]. The following NRCs are used:

NRC	Description
\$10	General reject
\$11	Service not supported
\$12	Sub-function not supported
\$13	Incorrect message length or invalid format
\$14	Response too long
\$21	Busy – Repeat request
\$22	Conditions not correct
\$24	Request sequence error
\$31	Request out of range
\$33	Security Access denied
\$35	Invalid key
\$36	Exceeded number of attempts
\$37	Required time delay not expired
\$70	Upload/Download not accepted
\$71	Transfer data suspended
\$72	General Programming failure
\$73	Wrong Block Sequence counter
\$75	Illegal Byte count in block transfer
\$7E	Sub-function not supported in active session
\$7F	Service not supported in active session
\$92	Voltage too high
\$93	Voltage too low

Table 2-1 NRC codes used by Bootloader



Note

This NRC table includes an overall summary of the NRCs supported by the bootloader. Your actual implementation may not use all of them.

3 Flash Bootloader Delivery

3.1 Bootloader Components

The flash bootloader delivery contains the following components:

- ▶ Flash bootloader source code
- ▶ Configuration Tool GENy
- ▶ 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

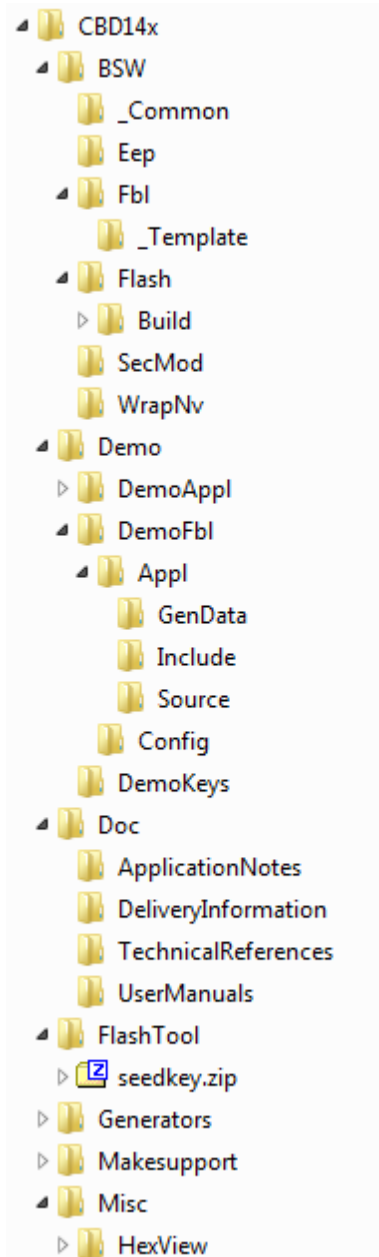
1. Install the bootloader package
2. Start your own integration project
3. Configure the bootloader using the configuration tool
4. Adjust the user callback files from template
5. Build the bootloader
6. Adapt your application software

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:



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.
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.
Flash	Flash driver. Includes its own make system and binary file in Build.
SecMod	SecMod contains the security module. Depending on the security class supported by the bootloader, a library and source code can be included.
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.

Figure 4-1 Bootloader Folder Structure

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 generation tool.

5 Generation Tool (GENy)

Standard configurations are done with the generation tool GENy. The generation tool is installed automatically during the delivery install. A link to GENy is added to your PC's start menu.

5.1 Project Setup

To create a new configuration, please select "New..." from the File menu. The following dialog will appear:

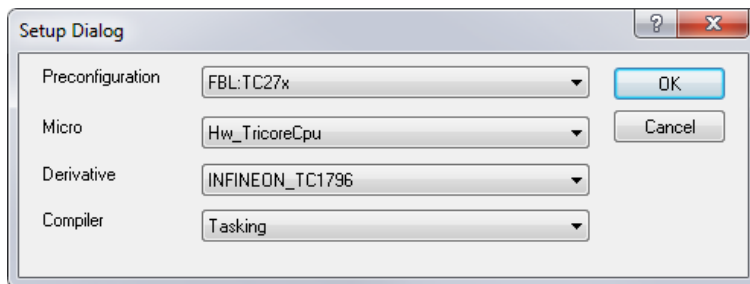


Figure 5-1 GENy Setup Dialog

The actual field values will vary, depending on your delivery license and hardware. Note that several pre-configurations can exist. Please check, which one fits the most to your configuration.

The setup dialog can include different pre-configuration variants if different bootloader addressing modes or other options which affect the communication configuration are included. The following options can be configured with different pre-configuration:

- ▶ NormalFixed Addressing or Extended Addressing
- ▶ XCP

The main GENy window will be updated as shown below once the initial setup has been completed.

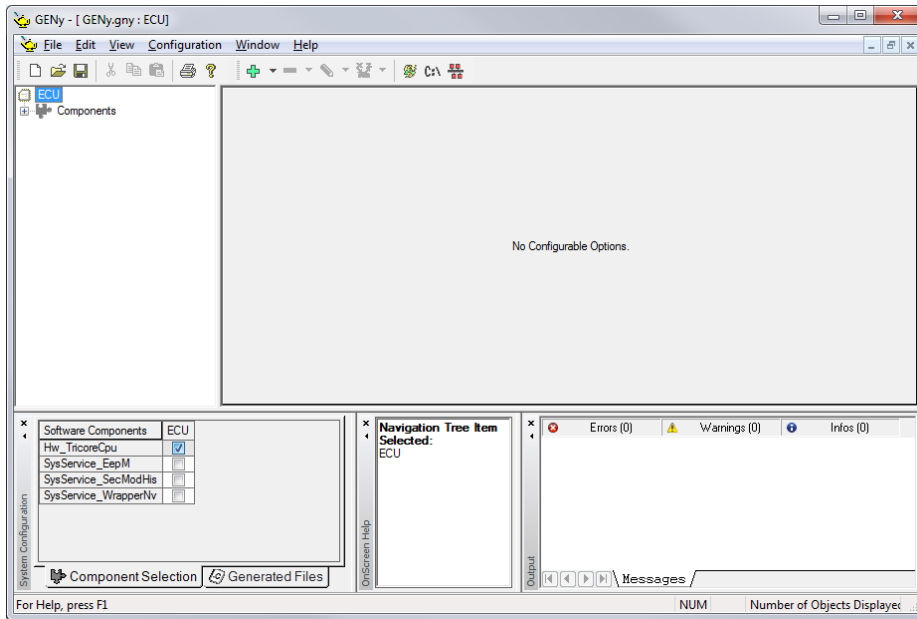


Figure 5-2 GENy main window after pre-configuration

You have to select the communication database (.dbc) in the next step. This file should be provided by your OEM. The database file is selected by using Configuration/Add Channel/CAN or the “+” icon.

Once the database has been selected, a list of ECU Nodes defined by the database will appear in the “Database Nodes” field. Please select the node which should be used by the FBL. Multiple nodes may be selected if your ECU is used multiple times in the same vehicle. You can choose the actual node during the bootloader’s startup.

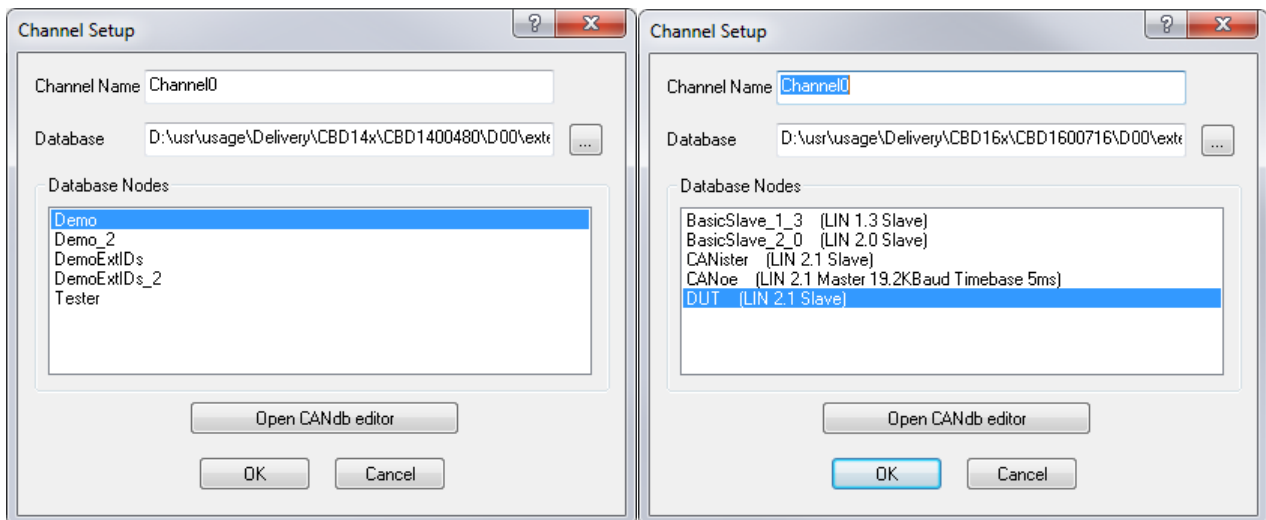


Figure 5-3 GENy Channel Setup (CAN or LIN)

After the node was selected, you can activate the GENy modules used to configure the bootloader.



Caution

The CAN ID settings configured in the database are automatically included into the bootloader configuration during the initial setup when the database is configured first. If the database of an already existing configuration is changed, the communication settings are not automatically reconfigured to prevent overwriting of manually adapted settings.

The changes have to be reloaded manually using GENy module FblWrapperCom_Can.

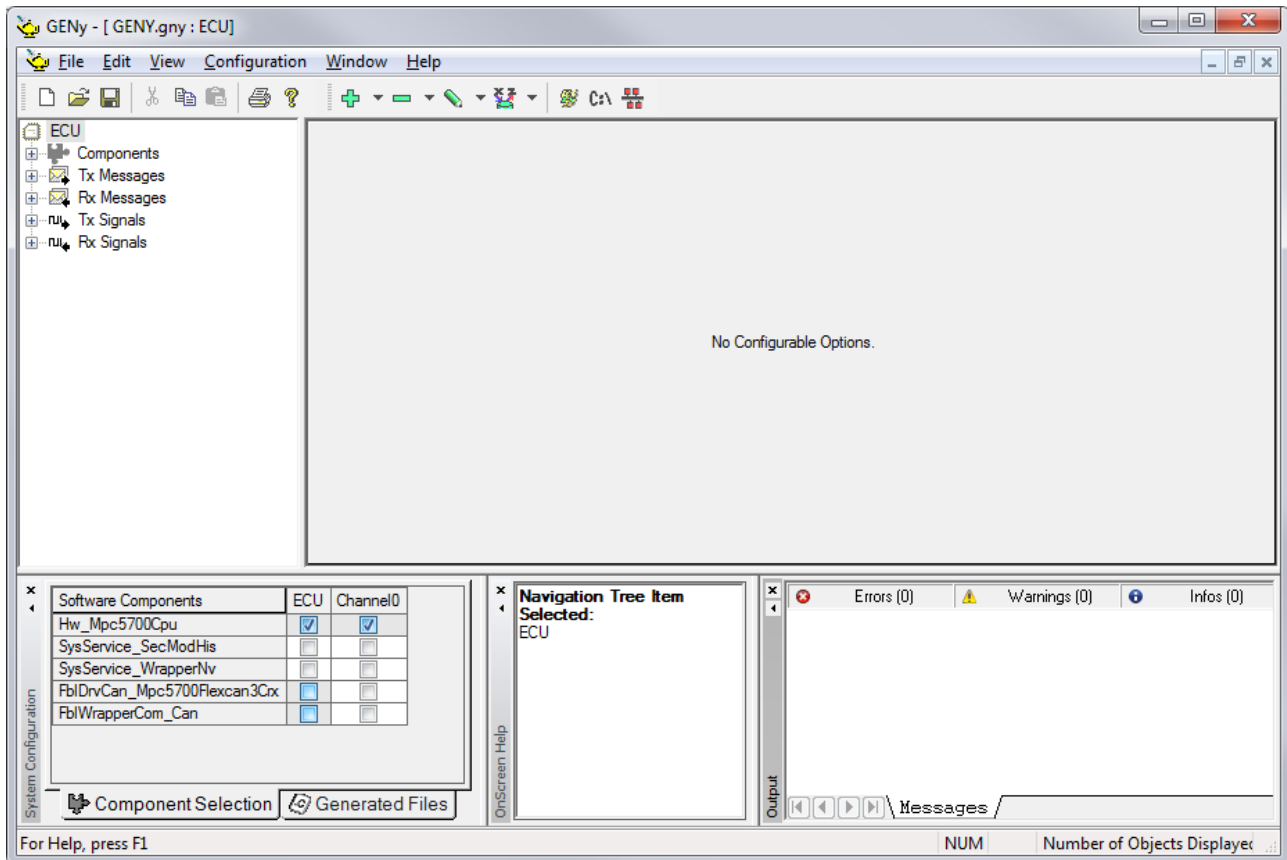


Figure 5-4 GENy main window after channel setup

At this point, you must add the Software Components for the FBL to the channel that was defined. Click on the check-boxes in the Channel 0 column for each component. Continue by adding the FblDrvCan_<hardware> and FblCan_14229_Vector components.

After the Software Components have been selected, you should expand the tree list in the leftmost window to select between the component-configuration windows. The tree is expanded by clicking on the '+' button, or by double-clicking on the "Components" label.



Note

Component selections for Tx Messages, Rx Messages, Tx Signals and Rx Signals are not relevant for standard bootloader configurations.

If your bootloader delivery uses a communication driver from CANbedded or Microsar, this information is relevant.

If all necessary components are activated, GENy should look like this:

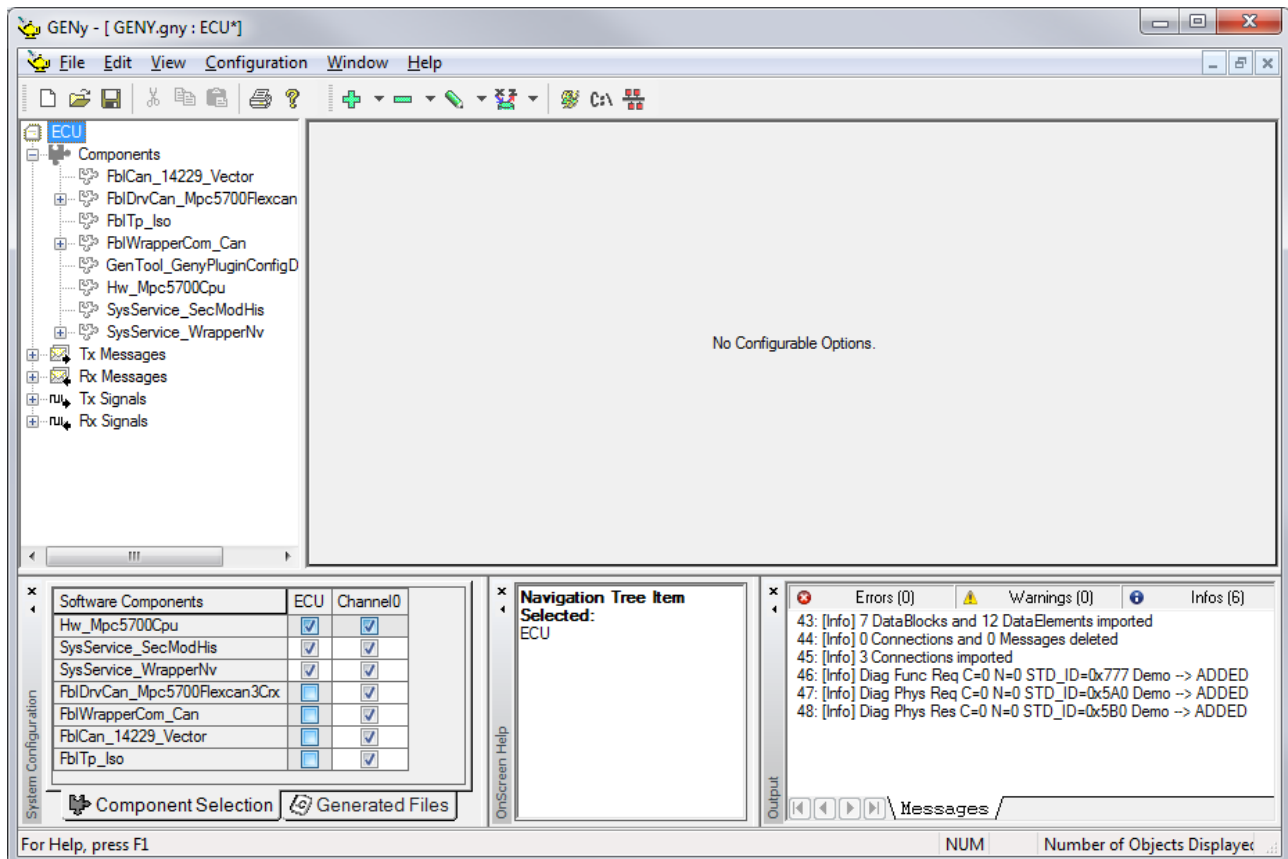


Figure 5-5 GENy Components

Before proceeding to component configuration, you should save the configuration. Once the configuration has been saved, you should select “Generation Paths...” from the Configuration menu.

By default, GENy will place the generated files into the Project Directory. You may enter a new path relative to the Project Directory, or enter a path relative to the GENy’s working-directory (the directory, where GENy is executed from), or you may enter an absolute path. After pressing OK, you are ready to configure the individual components.

5.2 CAN configuration

The CAN controller is configured by expanding the “Channels” component, and selecting the “Channel 0” component from the tree list. The contents of the window on the right will be highly dependent on the hardware that has been selected. Refer to the TechnicalReference_FBL_<hardware>.pdf that is also located in the documentation folder of your delivery.

An example of the CAN configuration window is shown below:



Note

Note that this bootloader supports only one channel if nothing else is mentioned in project specific documentation.

Configurable Options	Channel0
[-] General Settings	
Bus System Type	CAN
Manufacturer	Vector
[-] FbIDrv	
Physical Node	Node 0
IO Port	000
[-] FBL	
[-] Information	
NmBaseAddress	0x400*
[-] Initialization	
[-] Init Structures	Add
[-] Init Structure	Delete
Acceptance Filter Configuration	...
Bustiming Configuration	...
[-] TriCore MultiCAN (CPU)	
Number of CAN Objects	7*
+ CAN object usage	

Figure 5-6 GENy CAN configuration

By default, GENy initializes the CAN hardware as high speed CAN (500.0 KBPS). Use the “Bustiming Configuration” dialog to select the appropriate baud rate.

5.3 CAN ID configuration

In case a bootloader CAN stack is used, CAN IDs can be configured with the GENy module FbiWrapperCom_Can.

**Note**

Vector SLP3 bootloaders are delivered with support for Normal Addressing. If NormalFixed Addressing or Extended Addressing is required, support for these addressing modes has to be requested with the bootloader order.

The module can be delivered in two variants:

- ▶ **Standard CAN configuration:** The connections, CAN objects and handler functions are pre-configured and write protected. The CAN identifiers of the .dbc file provided to the bootloader are used and can be overwritten.
- ▶ **Extended CAN configuration:** The connections, CAN objects and handler functions can be changed by the user. This setting is used if a bootloader supports several addressing modes which can be configured in GENy or is delivered with several access options like UDS or XCP.

**Caution**

The bootloader connection configuration has to be verified carefully if the extended CAN configuration is changed. It is possible to create configurations which do not fulfill all bootloader requirements.

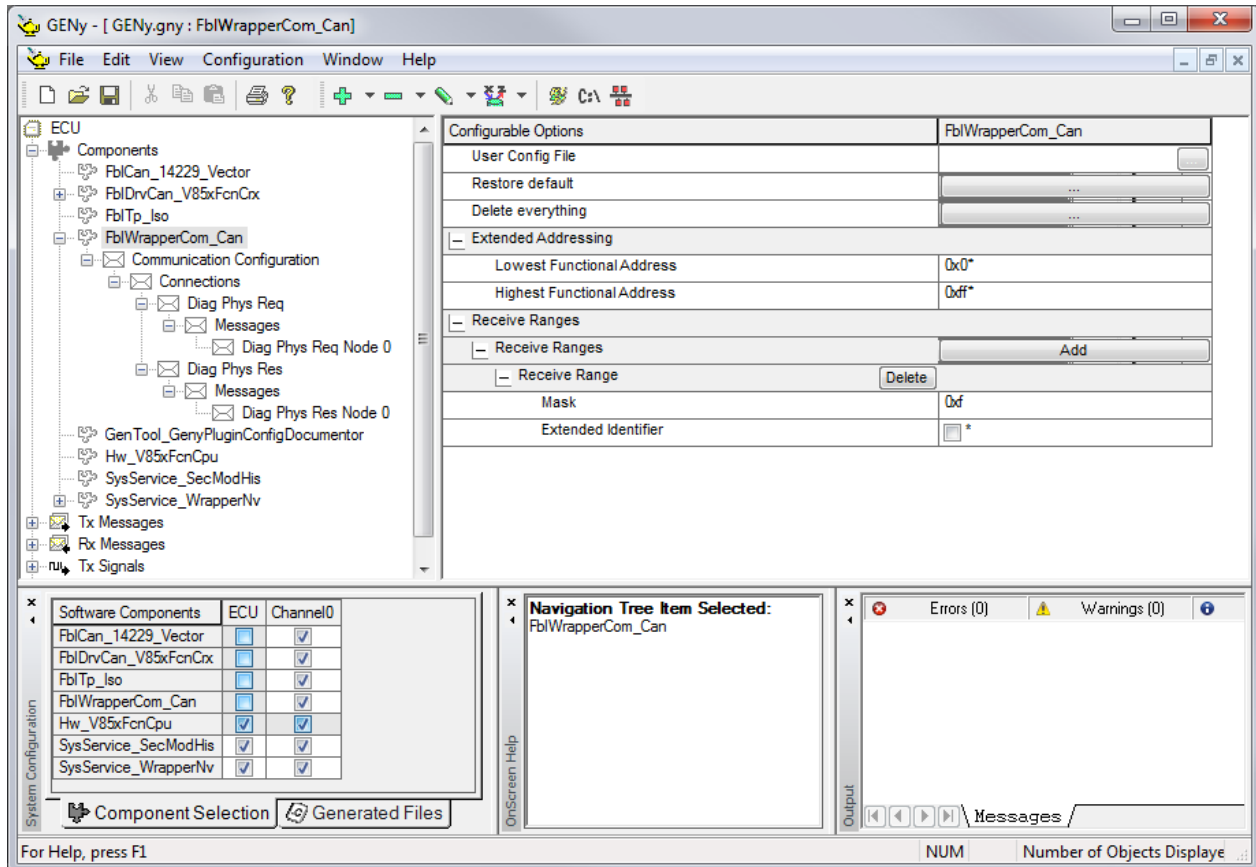


Figure 5-7 FblWrapperCom_Can GENy module

This module includes a tree structure to configure the bootloader CAN messages and some overall options on the main page.

Configuration Option	Description
User Config File	The path and name of a file to be included in the generated configuration file fbl_cw_cfg.h may be specified. The file may be used to activate features of the FBL that are not included in the GENy components. Normally, this field is blank.
Restore default	Restores the default configuration using pre-configured data and the data read from the communication database. Please note: Data which has been manually adapted is overwritten.
Delete everything	Deletes the complete CAN ID setup.
Lowest Functional Address	Specify the lowest functional address for Extended Addressing.
Highest Functional Address	Specify the highest functional address for Extended Addressing.
Receive Ranges	Add receive ranges for addressing modes like NormalFixed Addressing or Extended Addressing. This option is write protected.

Table 5-1 FblWrapperCom_Can main page GENy options

The CAN connections used by the bootloader are configured in the following table. This table is write-protected if a standard CAN configuration setup is used.

	CAN Message	Direction	Indication/Confirmation Function	Message Object	Nodes	Addressing Mode	Receive Handle
Diag Phys Req	DiagnosticPhysical	Request	FblCwProcessPhysicalRequest	0*	1*	Normal	FblRxCanMsg0Hdl
Diag Func Req	DiagnosticFunctional	Request	FblCwProcessFunctionalRequest	1	1*	Normal	FblRxCanMsg1Hdl
Diag Phys Res	DiagnosticPhysical	Response	FblTpConfirmation	0*	1*	Normal	V_NULL

Figure 5-8 CAN connection table

Configuration Option	Description
CAN Message	Documents the purpose of the used CAN message.
Direction	Request or response
Indication/Confirmation Function	Configures which function is called if a request is received or a response is successfully transmitted.
Message Object	Software receive object list or transmit object list index.
Nodes	Number of nodes used for this message. If there is more than one node available, it can be selected during the bootloader's startup.
Addressing Mode	Checks if a message is used for a Normal addressing, NormalFixed addressing or Extended addressing.
Receive Handle	Parameter which is handed over by the CAN driver. It identifies the receive object set by the CAN driver.

Table 5-2 GENy – CAN connection table

Every message can include several CAN IDs (in case of a multiple nodes setup). The CAN IDs for each node can be configured in this table:

	CAN ID	Extended Identifier	PGN	Base Address	Target Address	Source Address	Description
Diag Phys Req Node 0	0x5a0	<input type="checkbox"/> *	0x0*	0x0*	0x0*	0x0*	Demo
Diag Phys Req Node 1	0x6c0	<input type="checkbox"/> *	0x0*	0x0*	0x0*	0x0*	Demo_2

Figure 5-9 GENy – CAN message table

Configuration Option	Description
CAN ID	Value of the CAN ID which is sent or received.
Extended Identifier	If this checkbox is set, the CAN ID is configured as a 29-Bit CAN ID.
PGN	PGN used for NormalFixed addressing.
Base Address	Receive range base address for Extended addressing.
Target Address	Target address of a CAN message. Identifies the ECU in case a message is received and the tester in case a message is sent.
Source Address	Source Address of a CAN message. Identifies the ECU in case a message is sent and the tester in case a message is received.
Description	Comment to identify the CAN node and message.

Table 5-3 GENy – CAN message table

5.4 Bootloader Configuration

The features of the bootloader are configured in two separate components: FblDrvCan_<hardware>, and FblCan_14229_Vector. Configuration options for the selected component on the left will be shown on the right panel.

The FblDrvCan component contains both hardware-specific, and hardware independent selections. For details of the hardware-specific features, please refer to the document “Technical Reference Hardware”.

The figure shown below shows the hardware independent sections of the FblDrvCan component:

[- FBL	
User Config File	<input type="button" value="Browse"/>
Project State	Integration
Stay in Boot	<input checked="" type="checkbox"/>
Maximum Number of Segments	8*
Multiple Modules	<input checked="" type="checkbox"/> *
Sleep Mode	<input checked="" type="checkbox"/>
Sleep Time [ms]	300000*
Application Task	<input checked="" type="checkbox"/> *
Bootloader Header Address	0x0*
Diagnostic Buffer Size [B]	4095*
P2 Time [ms]	10*
P2* Time [ms]	5000*
Gap Filling	<input type="checkbox"/> *
Fill Code	0xff*
Internal Memory Copy	<input checked="" type="checkbox"/> *
Presence Pattern	<input type="checkbox"/> *
FblStart Function	<input checked="" type="checkbox"/>
Response After Reset	<input checked="" type="checkbox"/> *
User Subfunction	<input type="checkbox"/> *
User Service	<input type="checkbox"/> *
User Routine	<input type="checkbox"/> *
Flash Driver Usage	Download Driver
[- Data Processing	
Encryption Mode	<input type="checkbox"/> *
Compression Mode	<input type="checkbox"/> *
Data Processing Buffer Size [B]	256*
[- Download Handling	
Pipelined Verification	<input checked="" type="checkbox"/>
Output Verification	<input type="checkbox"/>
Signature Verification Length [B]	64*
Adaptive RCR-RP for TransferData	<input type="checkbox"/> *
Pipelined Programming	<input checked="" type="checkbox"/>
Write Segmentation [B]	256*
Unaligned Data Transfer	<input checked="" type="checkbox"/> *
[- Watchdog	
Watchdog Service	<input checked="" type="checkbox"/> *
Trigger Cycle [ms]	1*

Figure 5-10 GENy General FBL configuration

Each field is described below:

Configuration Option	Description
User Config File	The path and name of a file to be included in the generated configuration file <code>fbl_cfg.h</code> may be specified. The file may be used to activate features of the FBL that are not included in the GENy components. Normally, this field is blank.
Project State	Project state controls the number of internal checks done by the bootloader. There are 3 levels available: <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 to use the Production error reporting level.
Stay In Boot	This switch enables the “Force Boot Mode” feature. Please see chapter 10.1.3 - Force Boot Mode for further information.
Maximum Number of Segments	Maximum number of segments in a logical block. This value specifies how many sections are allowed for one logical block.
Sleep Mode	If this switch is enabled, <code>ApplFblBusSleep()</code> will be called after a configurable time without diagnostic communication has expired.
Sleep Time [ms]	Activation time of sleep mode.
Application Task	When selected, the FBL will periodically call <code>ApplFblTask()</code> while it is idle. This function may be customized by you to implement idle task operations.
Bootloader Header Address	The address of the constant <code>FblHeader</code> has to be entered here. The value has to match the address used by your linker to map the header to its proper location.
Diagnostic Buffer Size [B]	This is the amount of memory (in bytes) reserved for data in diagnostic request messages. The value may be up to 4095.
P2 Time [ms]	P2 diagnostic timeout. The bootloader will send a response or response pending message before this timeout expires.
P2* Time [ms]	P2* diagnostic timeout. This timeout will be used instead of the P2 timeout if a response pending message has been sent.
Gap Filling	This feature is used to fill unused areas of a logical block with a specific value.
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 on 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.
Internal Memory Copy	Activates a bootloader specific memory copy function. If deactivated, the memory copy function of the compiler library is

Configuration Option	Description
	used.
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.
FblStart 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.
Response After Reset	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. If activated, the bootloader will send a response pending message before issuing the reset.
User Subfunction	Enables a callback function which could be used to add additional sub-functions to existing services.
User Service	Enables a callback function which allows the implementation of additional diagnostic services.
User Routine	Enables a callback function which allows the implementation of additional Routine Identifiers.
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.
Encryption Mode	Optional feature. If activated, the bootloader an interface to decrypt data after download.
Compression Mode	Optional feature. If activated, the bootloader provides an interface to decompress data.
Data processing buffer size [B]	If encryption or compression mode is enabled, the buffer size used by these data processing functions can be configured here.
Pipelined Verification	Optional feature: Download is verified in background while additional data is received.
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.
Signature verification length [B]	Length of data blocks sent to security module for verification.
Pipelined Programming	Optional feature. If activated, the bootloader will process and write downloaded data while the following data is transferred to

Configuration Option	Description
	the bootloader. This feature is also known as “Double Buffered Download”.
Write segmentation [B]	Size of data packets handed over to the flash driver.
Unaligned data transfer	If activated, the bootloader will request to receive the full diagnostic buffer and handles flash segment alignment internally. Otherwise, the tester has to send data with the length aligned to the flash segment size.
Watchdog Service	Enables watchdog handling by the bootloader.
Trigger Cycle [ms]	Specifies the interval in milliseconds between the calls of the watchdog handling function ApplFblWDTTrigger().

Table 5-4 GENy General FBL Configuration

The FblCan_14229_Vector module includes the following options:

Configurable Options	FblCan_14229_Vector
[-] FblCan	
Access Delay Time [ms]	10000*
Communication Control Type	Enable Rx and Disable Tx
ControlDTC Option Record	<input checked="" type="checkbox"/> *
Check Programming Preconditions Service	<input checked="" type="checkbox"/> *
Application Validity Flag	<input checked="" type="checkbox"/> *

Figure 5-11 OEM specific UDS configuration

Configuration 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.
Communication Control Type	Configures sub-function of Communication Control service sent in pre-programming sequence.
ControlDTC Option Record	Activates or deactivates the usage of a ControlDTC Option Record with the ControlDTC service.
Check Programming Preconditions Service	If activated, the service Routine Control – Check Programming Preconditions is mandatory before the bootloader switches to programming session.
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).

Table 5-5 OEM UDS Specific Configuration

5.5 Memory Configuration

The memory configuration module is used to configure the memory layout used to store application software and calibration data. It can be found in the FbIDrvCan_<HW>-module.

The memory configuration is divided into 3 modules:

5.5.1 Flash Block Table

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

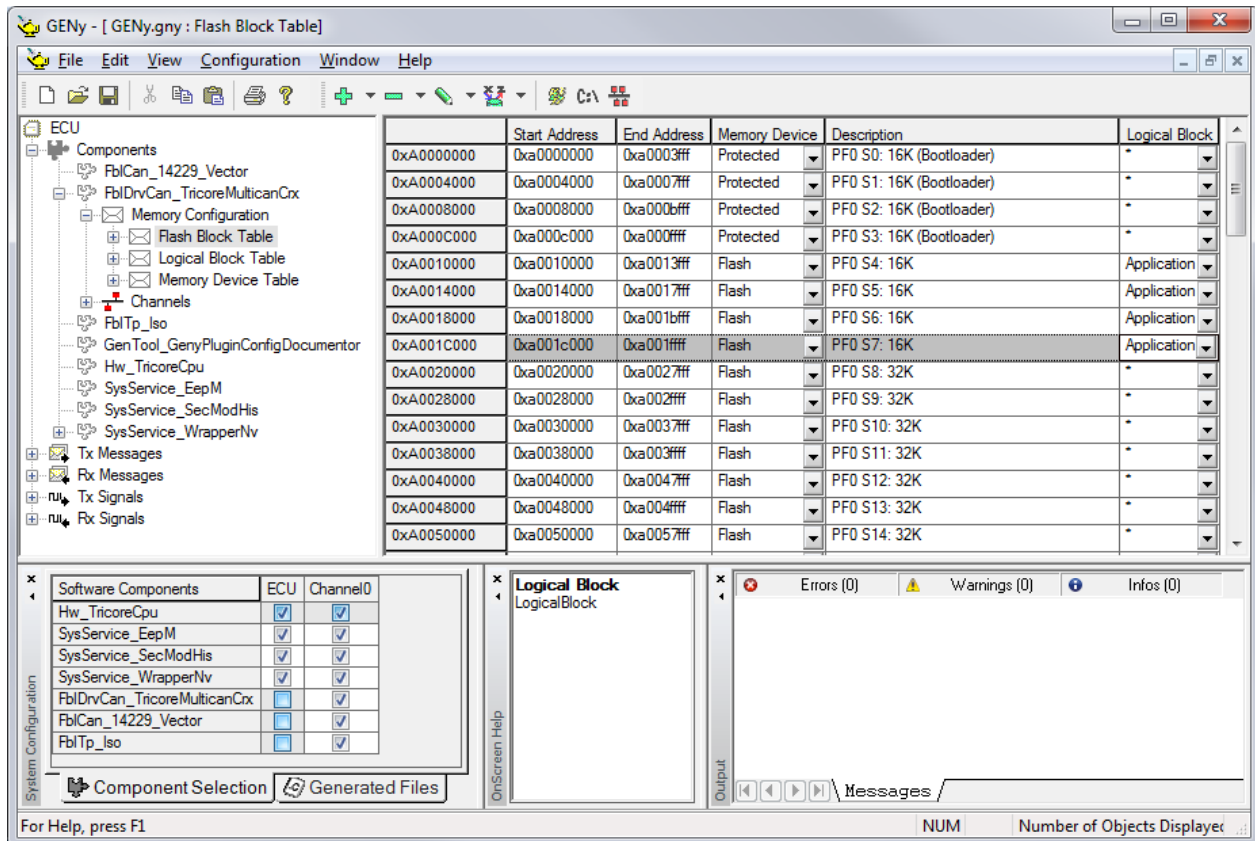


Figure 5-12 Flash Block Table

Configuration Option	Description
Start Address	Start address of the flash block. Has to be aligned to the start of an erasable flash block.
End Address	End Address. Has to be the end of an erasable flash block.
Memory Device	Memory device: Select the memory device used for this flash block. If "Protected" is selected, the block is not erasable/flashable.
Description	Comment to identify the block.
Logical Block	Used to map a logical block to a flash block.

Table 5-6 Flash Block Table Configuration

5.5.1.1 Logical Block Table

	Name	Disposability	Start Address	End Address	Max. reprogramming attempts	Pipelined Verification	Output Verification	Description
DemoAppl	DemoAppl	mandatory	0x30000	0x4fff	0x0*	SecM_VerifySignature*	SecM_Verification*	Demo Appl
DemoData_0	DemoData_0	mandatory	0x70000	0x8fff	0x0*	SecM_VerifySignature*	SecM_Verification*	Demo Data

Figure 5-13 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.

Configuration Option	Description
Name	Name used to identify the block in the flash block table.
Disposability	Is used to determine if a block is mandatory or optional during application validation.
Start Address	Set automatically by flash block table.
End Address	Set automatically by flash block table.
Max. reprogramming 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.
Pipelined Verification	If the bootloader is ordered with support of pipelined verification, the verification routine can be chosen here.
Output Verification	Optional setting: If needed, the verification routine can be chosen here.
Description	Comment to describe the block in generated code.

Table 5-7 Logical Block Table Configuration



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 configured to “protected” to ensure the bootloader cannot erase itself.

5.5.2 Memory Device Table

All available memory devices have to be configured in the memory device table. There are 2 default devices:

- ▶ Protected: Areas protected by the bootloader. All areas which must not be flashed by the bootloader have to be set to protected. This includes the bootloader itself and memories used by modules like EEPROM emulations.
- ▶ Flash: Internal flash memory delivered with the bootloader.

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].

The following settings have to be done for every added memory device:

Configuration Option	Description
Name	Name used to identify the device in the flash block table. It also is used as prefix to the driver interface functions.
Segment Size	Smallest writeable data segment.
Erased Value	The value which is set after a device has been erased.

Table 5-8 Memory Device Table Configuration

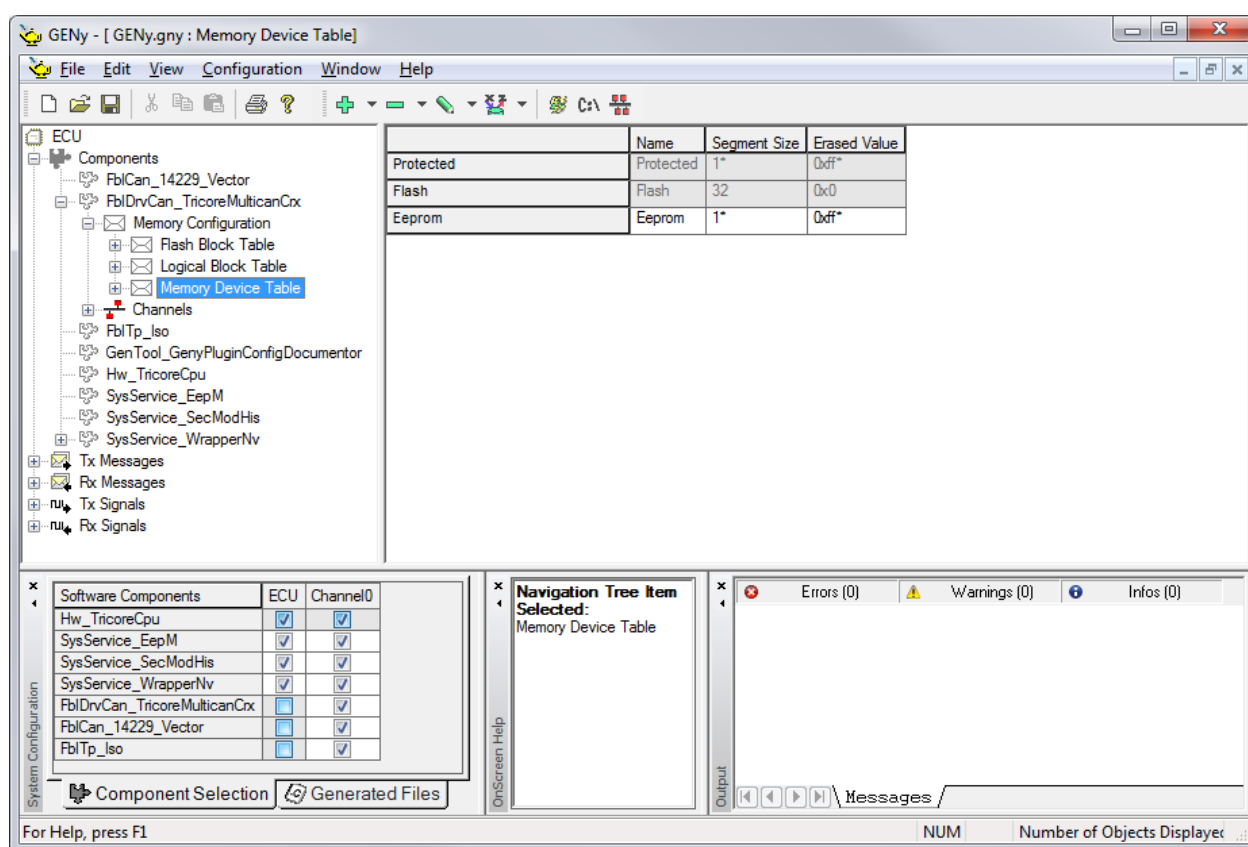


Figure 5-14 Memory Device Table

5.5.3 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 within your bootloader. Most of the information is stored once per bootloader:

	Type	Generate	Name	Count	Description
0 ProgReqFlag	Single	<input checked="" type="checkbox"/> *	ProgReqFlag	1*	Programming request flag
1 ResetResponseFlag	Single	<input checked="" type="checkbox"/> *	ResetResponseFlag	1*	*
2 AppValidity	Single	<input checked="" type="checkbox"/> *	AppValidity	1*	Application validity flag
3 ValidityFlags	Single	<input checked="" type="checkbox"/> *	ValidityFlags	1*	Logical block validity flags
4 SecAccessDelayFlag	Single	<input checked="" type="checkbox"/> *	SecAccessDelayFlag	1*	Security Access delay flag
5 SecAccessInvalidCount	Single	<input checked="" type="checkbox"/> *	SecAccessInvalidCount	1*	Security Access invalid count
6 Metadata	Table	<input checked="" type="checkbox"/> *	Metadata	2	Internal meta data (for each logical block)

Figure 5-15 NV-Wrapper configuration

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

	Generate	Name	Length	Default	Description
Fingerprint	<input checked="" type="checkbox"/> *	Fingerprint	9	*	Download fingerprint
ProgCounter	<input checked="" type="checkbox"/> *	ProgCounter	2	*	Successful reprogramming attempts
ProgAttempts	<input checked="" type="checkbox"/> *	ProgAttempts	2	*	Reprogramming attempts
CRCValue	<input checked="" type="checkbox"/> *	CRCValue	4	0x00	CRC total of logical block
CRCStart	<input checked="" type="checkbox"/> *	CRCStart	4	0x00	Start address of CRC total
CRCLength	<input checked="" type="checkbox"/> *	CRCLength	4	0x00	Length of CRC total

Figure 5-16 Meta Data configuration



Caution

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

5.5.4 Running the Generator

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

File Name	Contents
fbl_apfb.c	Contains the flash block table. If multiple memory devices are configured, this file also contains a table mapping the block entries to the appropriate device driver API functions.
fbl_apfb.h	Interface definition of fbl_apfb.c.
fbl_cfg.h	Contains macro definitions used to configure the bootloader. For the most part, the file defines switches in the form of FBL_ENABLE_<feature> or FBL_DISABLE_<feature>.
fbl_cw_cfg.c	Communication setup of CAN bootloaders.
fbl_cw_cfg.h	Communication setup of CAN bootloaders.
fbl_mtab.c	Contains the logical block table.
fbl_mtab.h	Interface definition of fbl_mtab.c.
ftp_cfg.h	This file is used to configure the transport protocol.
SecM_cfg.h	Configuration of the security module.
Secmpar.c	Additional constants used by the security module, e.g. the public key if security class CCC is used. This file isn't needed if security class DDD is used.
Secmpar.h	Interface definition of Secmpar.c.
v_cfg.h	Contains macro definitions specific to your hardware.
v_inc.h	Single include file to include v_cfg.h and v_par.h.
v_par.c, v_par.h	Defines to identify the bootloader delivery version.
WrapNv_cfg.c	Not needed currently. This file is generated for future use. Please do not include it into your project, it won't compile.
WrapNv_cfg.h	NvWrapper configuration. Defines a wrapper layer between ID and address based NV-memory stacks.

Table 5-9 Generated Files

6 User callback files and hardware specific adaptations

The bootloader includes a number 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 `fb_l_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	
<code>void ApplFblInit(void)</code>	
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	
<ul style="list-style-type: none"> > <code>ApplFblInit()</code> 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 <code>ApplFblInit()</code>, it will run if the application software is started. The bootloader will begin to handle the watchdog later during its initialization. 	

Prototype	
<code>void ApplFblStartup(vuint8 initposition)</code>	
Parameter	
<code>initposition</code>	The call context of this function is defined by this parameter.
Functional Description	
<p>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.</p> <p>The parameter of <code>ApplFblStartup()</code> determines the bootloader initialization step and if the function is called before or after the initialization step. The following parameters are supported:</p> <ul style="list-style-type: none"> ▶ (<code>kFblInitPreCallback</code> <code>kFblInitBaseInitialization</code>): First call. No initializations are done by the bootloader at this point ▶ (<code>kFblInitPreCallback</code> <code>kFblInitFblCommunication</code>): Call before the communication stack of the bootloader is initialized. ▶ (<code>kFblInitPreCallback</code> <code>kFblInitFblFinalize</code>): Call before the bootloader initialization is finalized. At this point, the bootloader decided not to start the application software independent of the startup configuration. ▶ (<code>kFblInitPostCallback</code> <code>kFblInitBaseInitialization</code>): Called after the hardware pre-initialization. At this point, <code>ApplFblInit()</code> normally is called to initialize the hardware for both bootloader and application. ▶ (<code>kFblInitPostCallback</code> <code>kFblInitFblCommunication</code>): Called after the communication stack is initialized. ▶ (<code>kFblInitPostCallback</code> <code>kFblInitFblFinalize</code>): Called after the bootloader initialization is finalized. At this point, all core modules are fully initialized. 	
Particularities and Limitations	
<ul style="list-style-type: none"> > 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 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	
<code>tFblResult ApplFblDataProcessing(tProcParam * procParam)</code>	
Parameter	
<code>procParam</code>	Pointer to data processing parameter structure
Return code	
<code>tFblResult</code>	<code>kFblOk/kFblFailed</code>
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 <code>procParam</code> 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	
<code>tFblResult ApplFblDeinitDataProcessing(tProcParam * procParam)</code>	
Parameter	
<code>procParam</code>	Pointer to data processing parameter structure
Return code	
<code>tFblResult</code>	<code>kFblOk/kFblFailed</code>
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

This function is called every millisecond 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 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 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

```
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	
<code>void ApplFblStartApplication(void)</code>	
Functional Description	
This function is used to start the application software.	
Particularities and Limitations	
<ul style="list-style-type: none"> > When this function is called, ApplFblInit() has been executed. > The non-volatile memory driver has most likely been initialized. 	

6.3 Validation and non-volatile memory callbacks

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	
<ul style="list-style-type: none"> > The programming request flag will be reset by this function. 	

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

Prototype	
<code>tFblResult ApplFblValidateApp(void)</code>	
Return code	
<code>tFblResult</code>	Validation success
Functional Description	
Returns OK if the application valid flag has been successfully written.	

Prototype	
<code>tFblResult ApplFblInvalidateApp(void)</code>	
Return code	
<code>tFblResult</code>	Invalidation success
Functional Description	
Returns OK if the application valid flag has been successfully removed.	

Prototype	
<code>tFblResult ApplFblValidateBlock(tBlockDescriptor blockDescriptor)</code>	
Parameter	
<code>blockDescriptor</code>	Block descriptor structure of logical block which is currently validated.
Return code	
<code>tFblResult</code>	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	
<code>tFblResult ApplFblInvalidateBlock(tBlockDescriptor blockDescriptor)</code>	
Parameter	
<code>blockDescriptor</code>	Block descriptor structure of logical block which is currently invalidated.
Return code	
<code>tFblResult</code>	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	
<code>static tFblResult ApplFblChgBlockValid(uint8 mode, tBlockDescriptor descriptor)</code>	
Parameter	
<code>Mode</code>	Determines if a block is validated or invalidated.
<code>tBlockDescriptor</code>	Block descriptor of logical block to be validated or invalidated.
Return code	
<code>tFblResult</code>	Validation information stored correctly.
Functional Description	
This function changes the validation flag of a logical block in NV-memory.	
Particularities and Limitations	
<ul style="list-style-type: none"> > The value is stored inverted. > Not used if presence patterns are activated. 	

Prototype	
<pre>static vsint16 ApplFblGetPresencePatternBaseAddress(vuint8 blockNr, IO_PositionType * pPresPtnAddr, IO_SizeType * pPresPtnLen)</pre>	
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	
<p>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.</p>	

Prototype	
<pre>static tFblResult ApplFblSetModulePresence(tBlockDescriptor * blockDescriptor)</pre>	
Parameter	
blockDescriptor	Pointer to logical block descriptor.
Return code	
tFblResult	kFblOk if operation was successful, otherwise kFblFailed.
Functional Description	
<p>This function sets the presence pattern for a specific logical block.</p>	
Particularities and Limitations	
<p>> Previous erase operation must have cleared this area.</p>	

Prototype	
<pre>static tFblResult ApplFblClrModulePresence(tBlockDescriptor * blockDescriptor)</pre>	
Parameter	
blockDescriptor	Pointer to logical block descriptor.
Return code	
tFblResult	kFblOk if operation was successful, otherwise kFblFailed.
Functional Description	
<p>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.</p>	

Prototype	
<code>static tFblResult ApplFblChkModulePresence(tBlockDescriptor * blockDescriptor)</code>	
Parameter	
<code>blockDescriptor</code>	Pointer to logical block descriptor.
Return code	
<code>tFblResult</code>	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	
<code>tFblResult ApplFblAdjustLbtBlockData(tBlockDescriptor * blockDescriptor)</code>	
Parameter	
<code>blockDescriptor</code>	Pointer to logical block descriptor.
Return code	
<code>tFblResult</code>	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 <code>ApplFblSetPresencePattern</code> 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>tFblResult ApplFblStoreFingerprint(vuInt8 * buffer)</code>	
Parameter	
<code>buffer</code>	Pointer to received fingerprint.
Return code	
<code>tFblResult</code>	Result of storage operation.
Functional Description	
This function is called by the <code>WriteDataByIdentifier</code> 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	
<code>void ApplFblErrorNotification(tFblErrorType errorType, tFblErrorCode errorCode)</code>	
Parameter	
<code>errorType</code>	Indicates the error type (see fbl_def.h for a list of error types)
<code>errorCode</code>	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	
<code>tFblResult ApplFblWriteCRCTotal(tBlockDescriptor * blockDescriptor, vuint32 crcStart, vuint32 crcLength, vuint32 crcValue)</code>	
Parameter	
<code>blockDescriptor</code>	Information about the logical block being downloaded.
<code>crcStart</code>	Start address of checksum calculation area
<code>crcLength</code>	Length of checksum calculation area
<code>crcValue</code>	Checksum
Return code	
<code>tFblResult</code>	Write access result.
Functional Description	
This function saves the CRC total to non-volatile memory.	

Prototype	
<code>vuint8 ApplFblRWSecurityAccessDelayFlag(vuint8 mode, vuint8 value)</code>	
Parameter	
<code>mode</code>	Indicates if security access delay flag should be written or read.
<code>value</code>	Value to be written
Return code	
<code>vuint8</code>	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	
<code>tFblResult ApplFblReadSecAccessInvalidCount(vuint8 * invalidCount)</code>	
Parameter	
<code>invalidCount</code>	Number of invalid security access attempts.
Return code	
<code>tFblResult</code>	Indicates if the NV-memory access was successful.
Functional Description	
Read number of invalid security access attempts.	
Particularities and Limitations	

Prototype	
<code>vuint16 ApplFblGetPromMaxProgAttempts(vuint8 blockNr)</code>	
Parameter	
<code>blockNr</code>	Number of logical block.
Return code	
<code>vuint16</code>	Maximum number of allowed programming attempts.
Functional Description	
This function returns the number of allowed programming attempts for a given logical block.	

Prototype	
<code>vuint8 ApplFblGetSecAccessDelayFlag(void)</code>	
Return code	
<code>vuint8</code>	0 if security access delay flag is not set, != 0 if set.
Functional Description	
Returns the state of the security access delay flag.	

Prototype	
<code>vuint8 ApplFblSetSecAccessDelayFlag(void)</code>	
Return code	
<code>vuint8</code>	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.	

6.4

6.5 Diagnostic Service Callbacks

Prototype	
<pre>void ApplDiagUserService(vuint8 * pbDiagData, tTpDataType diagReqDataLen) void ApplDiagUserRoutine(vuint8 * pbDiagData, tTpDataType diagReqDataLen) void ApplDiagUserSubFunction(vuint8 * pbDiagData, tTpDataType diagReqDataLen)</pre>	
Parameter	
pbDiagData	Pointer to diag service data (after SID!)
diagReqDataLen	Service data length (without SID!)
Functional Description	
<p>The flash bootloader supports a set of diagnostic services. All diagnostic services not supported by the bootloader are passed to three call-back functions.</p> <p>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..</p>	

Prototype	
<pre>void ApplFblReadDataByIdentifier(vuint8 * pbDiagData, tTpDataType diagReqDataLen)</pre>	
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	
<pre>vuint8 ApplFblWriteDataByIdentifier(vuint8 * pbDiagData, tTpDataType diagReqDataLen)</pre>	
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
<code>void ApplFblInitErrStatus(void)</code>
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“

6.6 Watchdog Callbacks

Prototype
<code>void ApplFblWDInit(void)</code>
Functional Description
<p>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()).</p> <p>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.</p> <p>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.</p>
Particularities and Limitations
<ul style="list-style-type: none"> > 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
<code>void ApplFblWDLong(void)</code>
Functional Description
<p>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.</p> <p>The function is called just before the bootloader starts your application.</p>
Particularities and Limitations
<ul style="list-style-type: none"> > 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:

Bootloader use case			
Module / function	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)			■

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 structur 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

Vector SLP3 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.

Vector SLP3 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 GENy (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 ApplFbInvalidateBlock() and ApplFbValidateBlock(). 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.

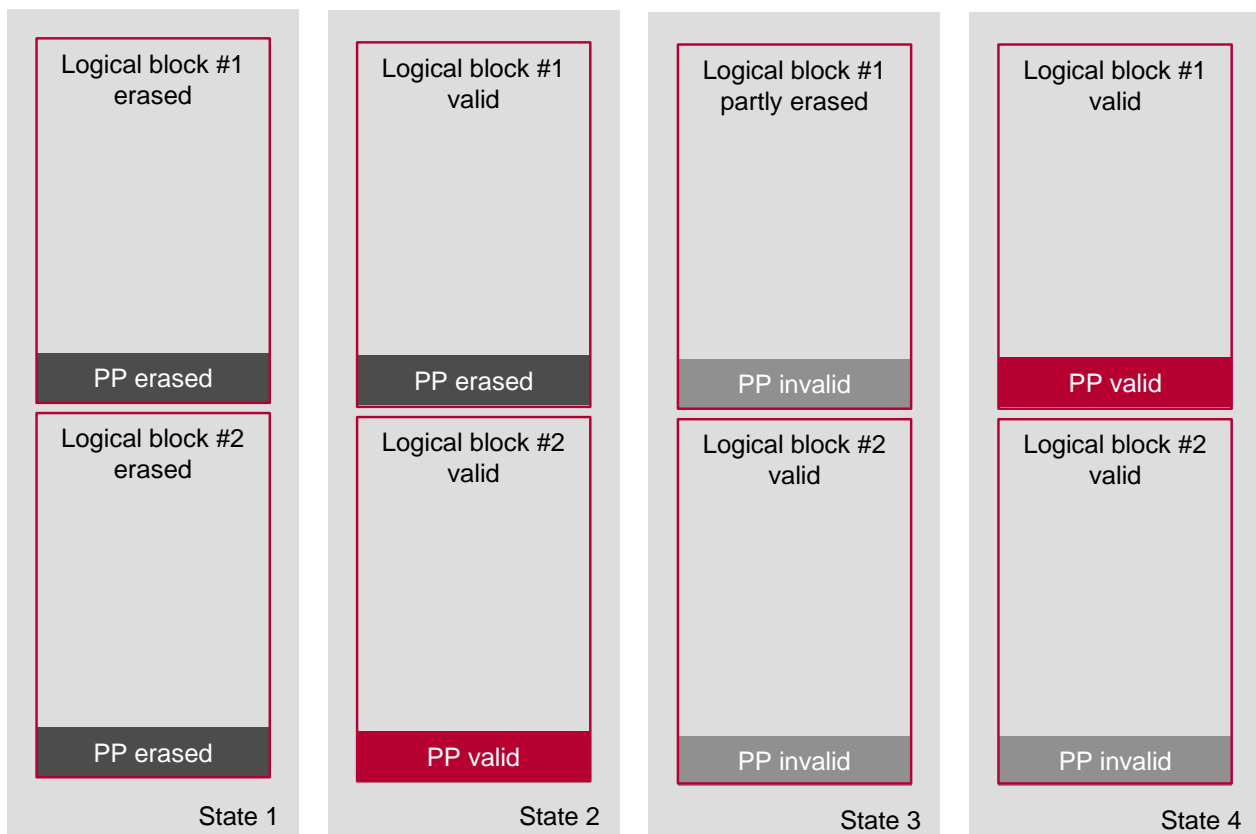


Figure 10-2 Presence Pattern Examples

- ▶ 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**.

10.1.3 Force Boot Mode

This bootloader supports the “Force Boot Mode” feature. If enabled, the bootloader can be prevented from starting the application if a special message is sent to the bootloader within a specified time.

The default wait time is configured to 20ms, but can be changed by a user configuration file:



Example

```
/* Wait 30 ms before jump into application */  
#define kFblStartMessageDelay 30u
```

Please note that the time slot has to be big enough to receive at least one message. The allowed range is from 0 to 255, but values smaller than 5ms may be hard to hit by a start message sent by the tester.



Caution

The “Force Boot Mode” feature should be used as a development feature because it increases the startup time of the ECU and allows a transition to programming session without using a preprogramming sequence.

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.

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

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

AppIFblDeinitDataProcessing 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.

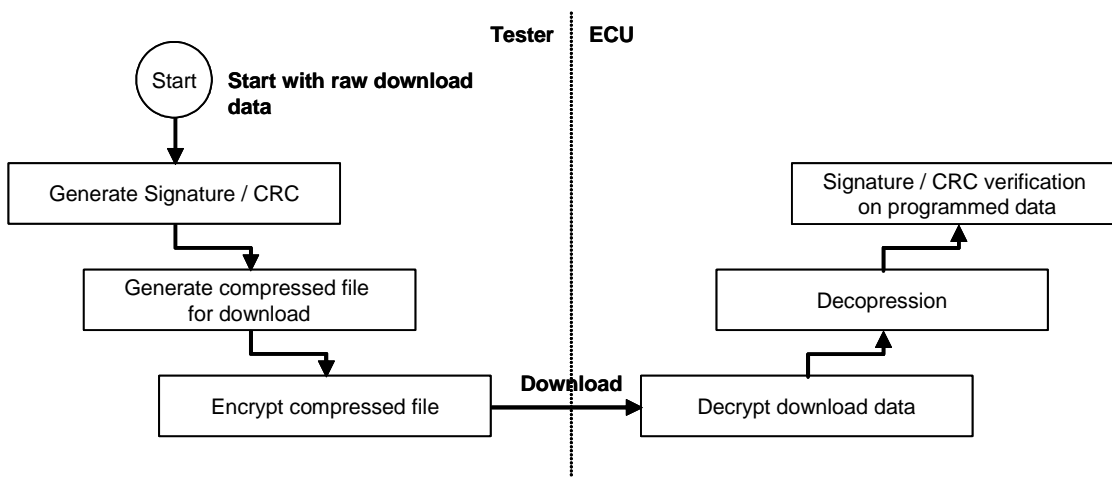


Figure 10-3 Data processing sequence

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:

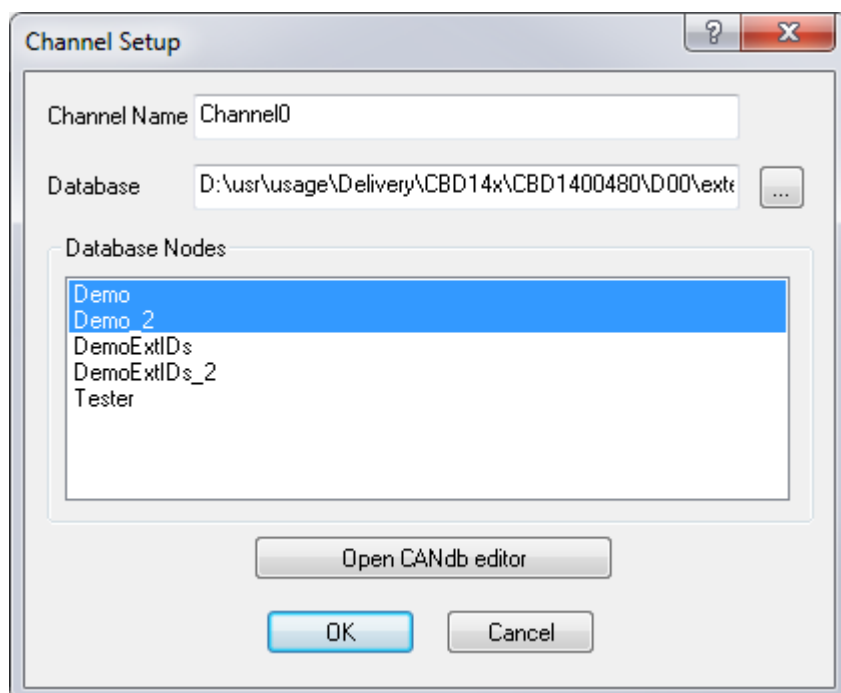


Figure 10-4 Multiple ECU setup

The GENy module FblWrapperCom_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 FblWrapperCom_Can.

**Caution**

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

10.6 Configuration without NV-Memory driver

The Vector SLP3 bootloader can be used without an additional NV-memory driver. This can be useful if there is no process data handling like fingerprint handling or reprogramming counters required for a specific use case.

The following steps have to be executed if no NV-memory driver should be used:

**Practical Procedure**

Configuration steps to remove NV-memory driver:

- ▶ Use block based validation with compatibility check on startup – see chapter 10.1.
- ▶ Remove any NV-memory drivers from your project and you project's make file.
- ▶ Change process handling functions and macros to empty macros which return valid results (kFbIOk).
- ▶ Use RAM for reset response flag and reprogramming request flag.
- ▶ If you keep the Dummy EEPROM driver used by the bootloader demonstration project, process data may be available until a power-on reset occurs.

**Caution**

Please note that there is no possibility to track testers, programming dates or a reprogramming count if no NV-driver is used in your bootloader.

The security access delay information also won't get tracked in a reset-safe way.

10.7 Pipelined Programming and Pipelined Verification

Optionally, Vector SLP3 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.

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:

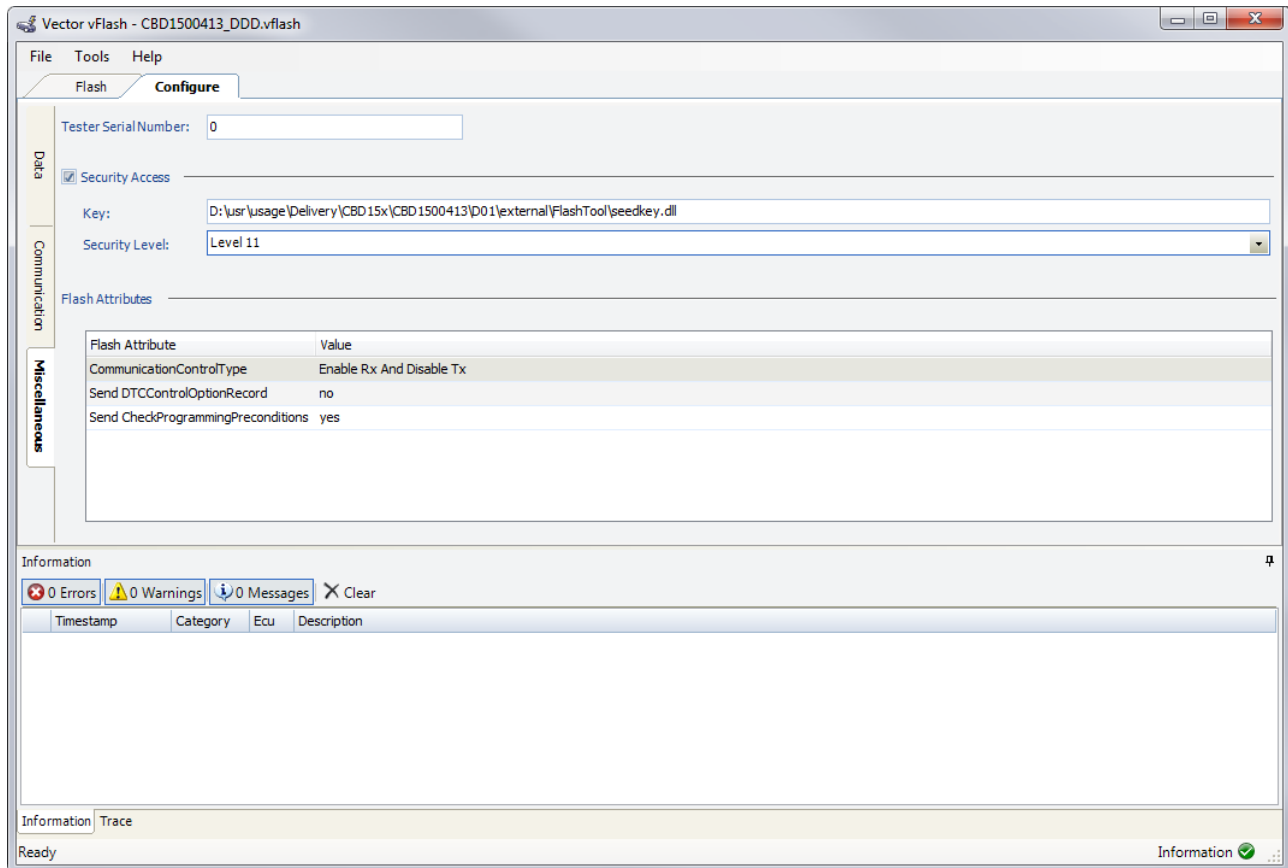


Figure 11-1 vFlash Configuration

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: #define kDiagRqIWriteDataByIdentifierFingerPrintParameter 0x09u
Security level	Selects the security level used for Seed/Key. Default is level 11. The configuration can be changed by a user configuration file: #define kDiagSubRequestSeed 0x11u #define kDiagSubSendKey 0x12u
Communication Control Type	Corresponds to the GENy setting "Communication Control Type".
Send DTCCControlOptionRecord	Corresponds to the GENy setting "Control DTC Option Record"

Configuration Option	Description
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
UDS	Unified Diagnostic Services

13 Contact

Visit our website for more information on

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

www.vector.com