Interface Application Bootloader

Version 1.1.1
2022-10-20
Support Note SN-IES-1-016

Author Marco Riedl, Andreas Dollinger, Mustafa Mohammed

Abstract Although Application and Bootloader are separate software parts, they require to exchange a minimum of information, for example whether the ECU shall enter and stay in FBL to wait for an update of the Application. For this reason, an Application-Bootloader-Interface must be implemented.

Table of Contents

1 Introduction

Automotive ECUs typically contain at least two separate software parts, the Application and the Flash Bootloader (Bootloader or also abbreviated via FBL). The Application implements the specific functionality of the ECU, e.g. controlling the engine or the airbags. The Bootloader allows to update the Application once the vehicle has been put on the market.

Although Application and Bootloader are separate software parts, they require to exchange a minimum of information, for example whether the ECU shall enter and stay in FBL to wait for an update of the Application. For this reason, an Application-Bootloader-Interface must be implemented.

This document gives an overview of the Application-Bootloader-Interface on both sides, the Application side and the Bootloader side and which information must be exchanged.

The focus of the description of the Application software is on AUTOSAR standard (R4.x) which is widely used for automotive ECUs nowadays. Concerning the Bootloader software, this document shows the Vector proprietary and the Vector AUTOSAR Bootloader.

2 ECU Boot Sequence

There are two conditions that decide whether the Application or the Bootloader are started after a Reset.

The first is a **reprogramming request** that is provoked by a diagnostic tester. The second is the check whether the Application is **valid** or not.

The Application is started, if there is no reprogramming request and the Application is valid. The Bootloader is started, if there is a reprogramming request or the Application is not valid.

Therefore, the reprograming request flag is one part of the information that has to be exchanged between the Application and the FBL.

Usually, the Boot Manager is the first instance which is started after the ECU powers up.

There are still systems without Boot Manager. Then the Bootloader is started first and performs the checks for valid Application.

2.1 Boot Sequence without Boot Manager

During the FBL initialization steps the FBL checks after each startup whether a reprogramming request is available. This is either realized by a RAM pattern or shared non-volatile memory. For reprogramming requests the FBL will finish its initialization, stays in the FBL and waits for requests from the Tester. In this case the FBL activates the programming session, sets the session timeout. When there won’t be a subsequent request from the tester the FBL will issue a timeout and preforms a reset.

When there is no reprogramming request the FBL will check whether the application validity is set or not. Depending on the check’s result the FBL will finish its initialization, stays in the FBL, and waits for requests from the Tester or jumps to the entry point of the application.

In case there is no valid application the FBL will stay in the FBL.

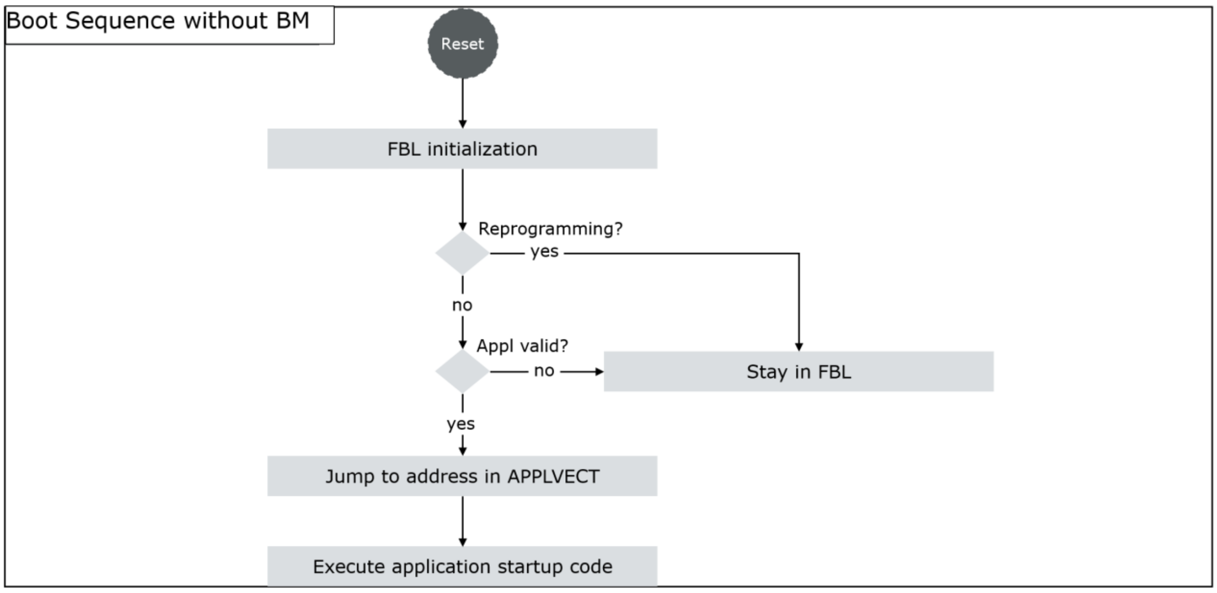


Figure 2--1 - Boot Sequence After Reset “Without BM”

2.2 Boot Sequence with Boot Manager

When a BM is part of the ECU. The BM’s startup code is executed quite after the reset. After the BM is initialized, it will check whether the FBL is valid or not. When no valid FBL is available on the ECU it will search for a Failsafe Updater.

With valid FBL the BM checks for a reprogramming flag. When the flag is set the FBL is started. If the flag is not set the BM will search for a valid executable target. In case a valid target is found the BM will start it otherwise the FBL will be started to make sure the ECU is able to be (re)programmed.

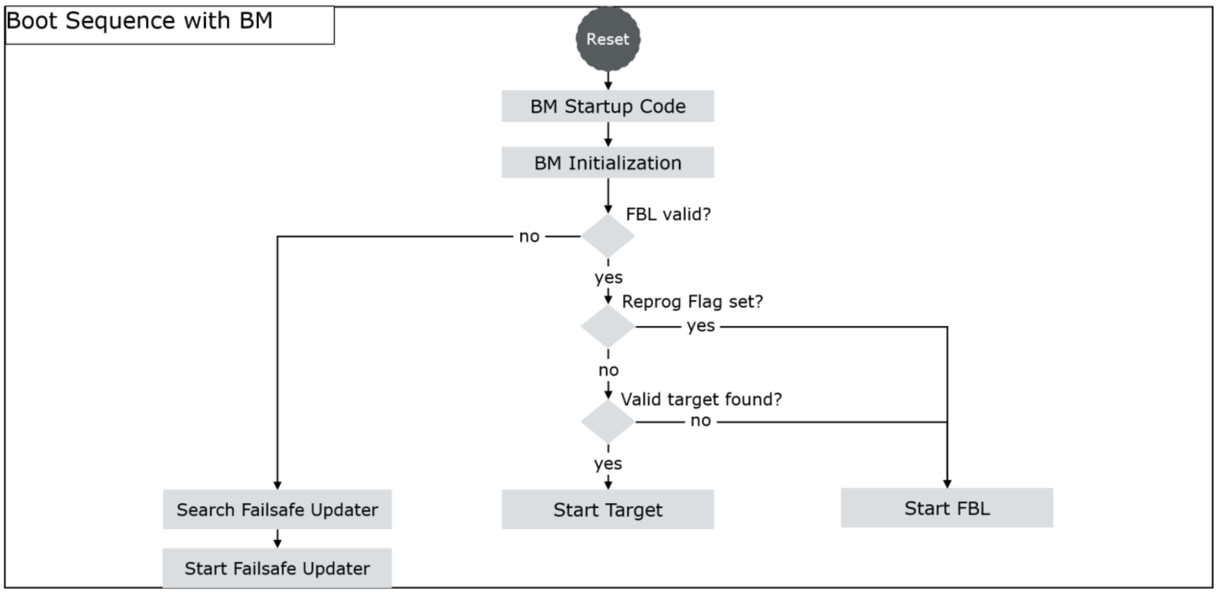


Figure 2-2 - Boot Sequence After Reset “With BM”

3 The Application

If there is no reprogramming request and the Application is valid, the ECU follows the above shown boot sequence and will stay in the Application.

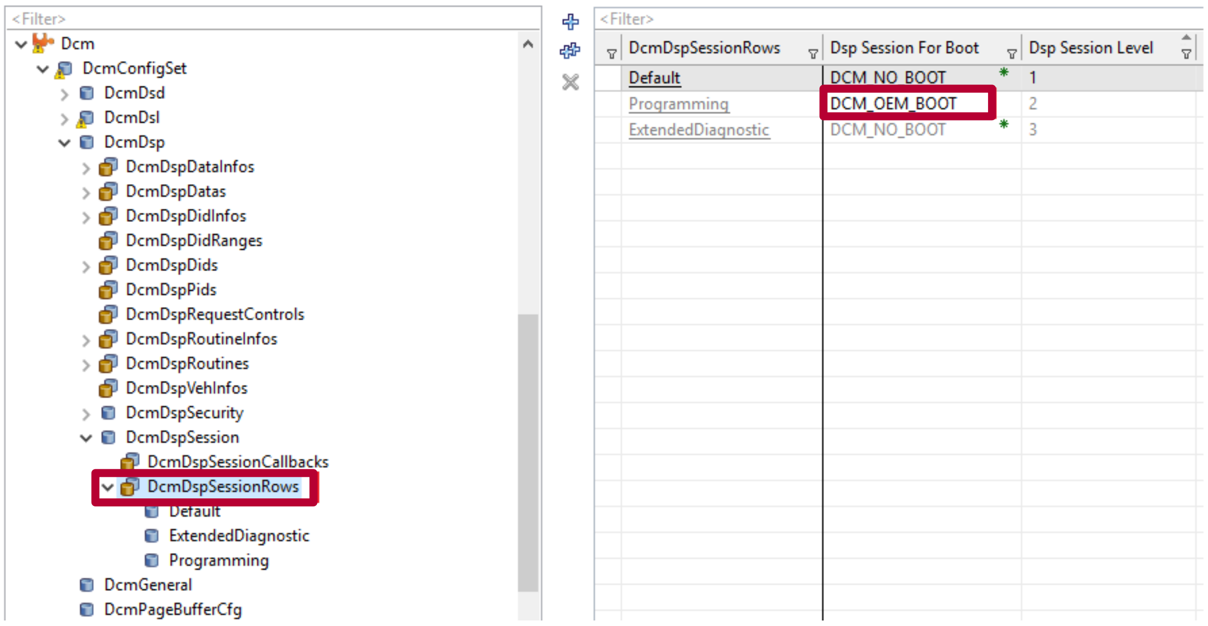
3.1 Jump From Application to Bootloader

If an external tester, e.g. a PC in a workshop, wants to update an Application, it can send a diagnostic request to the ECU that triggers a jump to the Bootloader. But before this jump, the Application must hand over all information that is relevant for the Bootloader.

In an AUTOSAR stack, the module responsible for the reception of the tester requests and triggering the jump to the Bootloader is the Diagnostic Communication Manager (Dcm), see [2].

The Dcm component provides two callout functions (`Dcm_SetProgConditions`, `Dcm_GetProgConditions`), where the Application can store and load information from non-volatile memory. Using these functions, both the Bootloader and the Application can be synchronized to work together.

To activate both functions, the programming session needs to be specified as a “session for boot” in the Dcm configuration as shown below.



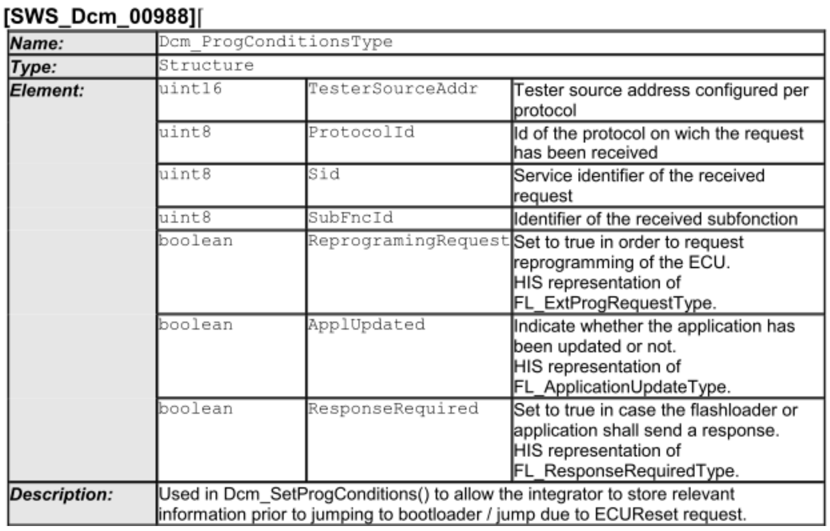
```
<Filter>
Dcm
  DcmConfigSet
    DcmDsd
    DcmDsl
    DcmDsp
      DcmDspDataInfos
      DcmDspDatas
      DcmDspDidInfos
      DcmDspDidRanges
      DcmDspDids
      DcmDspPids
      DcmDspRequestControls
      DcmDspRoutineInfos
      DcmDspRoutines
      DcmDspVehInfos
      DcmDspSecurity
      DcmDspSession
        DcmDspSessionCallbacks
        DcmDspSessionRows
          Default
          ExtendedDiagnostic
          Programming
    DcmGeneral
    DcmPageBufferCfg
```

DcmDspSessionRows	Dsp Session For Boot	Dsp Session Level
Default	DCM_NO_BOOT	1
Programming	DCM_OEM_BOOT	2
ExtendedDiagnostic	DCM_NO_BOOT	3

Figure 3-1 - Activate Support for Jumping to Bootloader on Programming Session

In order to store the information required by the Bootloader, the Dcm will execute the application callout `Dcm_SetProgConditions` (refer to [1] for detailed API description). This callout must be implemented by the user. The goal is to hand over the information in the parameter ProgConditions of the type `Dcm_ProgConditionsType` (see figure below) provided by Dcm and store it in a place and format, the Bootloader can access and understand.

Caution
Before storing the information for the Bootloader, the `Dcm` will notify the `BswM` via `ModeDeclarationGroupPrototype DcmEcuReset` by respectively sending mode `JUMPTOBootloader` or `JUMPTOSYSSUPPLIERBootloader`, so any activities for preparing the jump can be executed. These activities are optional.



[SWS_Dcm_00988]⌈

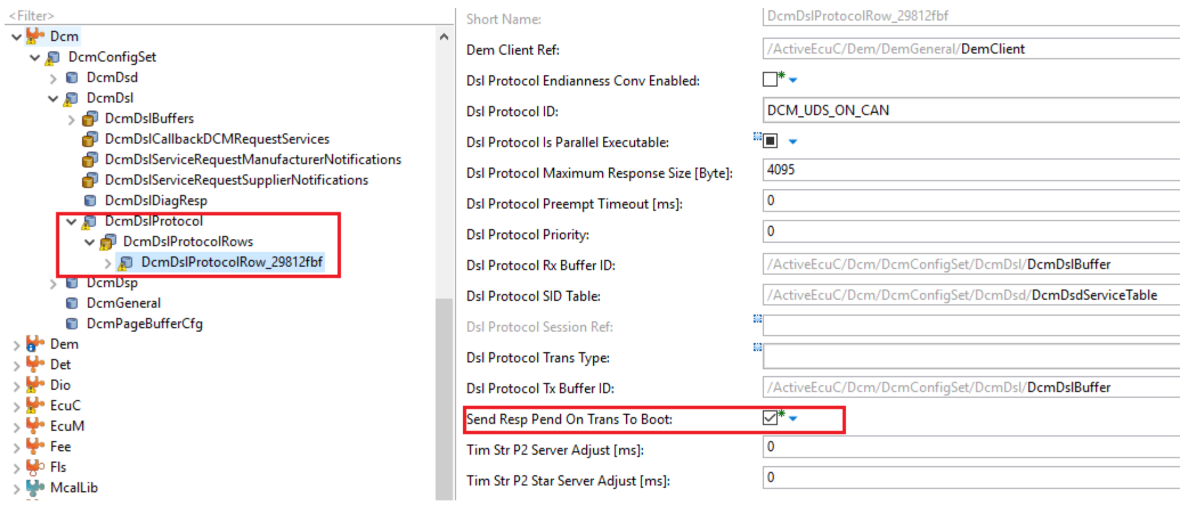
Name:	Dcm_ProgConditionsType		
Type:	Structure		
Element:	uint16	TesterSourceAddr	Tester source address configured per protocol
	uint8	ProtocolId	Id of the protocol on wich the request has been received
	uint8	Sid	Service identifier of the received request
	uint8	SubFncId	Identifier of the received subfonction
	boolean	ReprogramingRequest	Set to true in order to request reprogramming of the ECU. HIS representation of FL_ExtProgRequestType.
	boolean	ApplUpdated	Indicate whether the application has been updated or not. HIS representation of FL_ApplicationUpdateType.
	boolean	ResponseRequired	Set to true in case the flashloader or application shall send a response. HIS representation of FL_ResponseRequiredType.
Description:	Used in Dcm_SetProgConditions() to allow the integrator to store relevant information prior to jumping to bootloader / jump due to ECUReset request.		

Figure 3-2 - Specification of Dcm_ProgConditionsType from [2]

If an asynchronous processing is required to store the information, for instance by using an AUTOSAR memory stack, the callout `Dcm_SetProgConditions` allows the return value `DCM_E_PENDING` which results in the `Dcm_MainFunction` calling `Dcm_SetProgConditions` in each subsequent cycle until it returns `E_OK`, before the Dcm continues with the jump to Bootloader.

Caution
After storing the information for the Bootloader, the `Dcm` will notify the `BswM` via `ModeDeclarationGroupPrototype DcmEcuReset` by sending mode `EXECUTE`, so the project-specific implementation for jumping to bootloader, e.g. causing watchdog reset, is triggered. Implementing an appropriate reaction on mode `EXECUTE` is mandatory for jumping to the Bootloader.

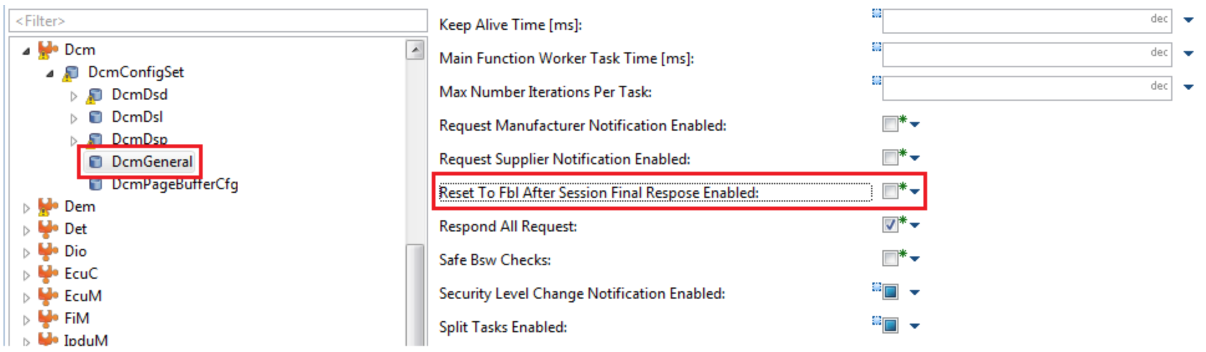
The option `Send Resp Pend On Trans To Boot` has to be activated in the DaVinci configurator (see Figure 3-3) to allow the application software to send a RCR-RP message (regardless of the posRspMsg-Indication-Bit) prior jumping to the Bootloader on the programming session request 0x10 0x02. The response pending message is needed to avoid a tester session timeout during the transition to Bootloader. After the reset, the Bootloader will send the positive response to the programming session request.



Short Name:	DcmDslProtocolRow_29812fbf
Dem Client Ref:	/ActiveEcuC/Dem/DemGeneral/DemClient
Dsl Protocol Endianness Conv Enabled:	☐
Dsl Protocol ID:	DCM_UDS_ON_CAN
Dsl Protocol Is Parallel Executable:	■
Dsl Protocol Maximum Response Size [Byte]:	4095
Dsl Protocol Preempt Timeout [ms]:	0
Dsl Protocol Priority:	0
Dsl Protocol Rx Buffer ID:	/ActiveEcuC/Dcm/DcmConfigSet/DcmDsl/DcmDslBuffer
Dsl Protocol SID Table:	/ActiveEcuC/Dcm/DcmConfigSet/DcmDsd/DcmDsdServiceTable
Dsl Protocol Session Ref:	
Dsl Protocol Trans Type:	
Dsl Protocol Tx Buffer ID:	/ActiveEcuC/Dcm/DcmConfigSet/DcmDsl/DcmDslBuffer
Send Resp Pend On Trans To Boot:	✓
Tim Str P2 Server Adjust [ms]:	0
Tim Str P2 Star Server Adjust [ms]:	0

Figure 3-3 - Activate Response Pending on Transition to Bootloader

In order to avoid that the Dcm sends a positive response to the services which cause the ECU reset (e.g. 0x10 and 0x11), the parameter `Reset to Fbl After Session Final Response Enabled` must be deactivated. When the parameter is deactivated, the Dcm sends only a response pending before issuing the reset and the Bootloader is the responsible to send the final positive response.


<Filter>
- Dcm
  - DcmConfigSet
    - DcmDsd
    - DcmDsl
    - DcmDsp
    - DcmGeneral
    - DcmPageBufferCfg
- Dem
- Det
- Dio
- EcuC
- EcuM
- FiM
- IpduM

Keep Alive Time [ms]:	dec
Main Function Worker Task Time [ms]:	dec
Max Number Iterations Per Task:	dec
Request Manufacturer Notification Enabled:	☐
Request Supplier Notification Enabled:	☐
Reset To Fbl After Session Final Respose Enabled:	☐
Respond All Request:	✓
Safe Bsw Checks:	☐
Security Level Change Notification Enabled:	■
Split Tasks Enabled:	■

Figure 3-4 - Deactivate Reset To Fbl After Session Final Response Enabled

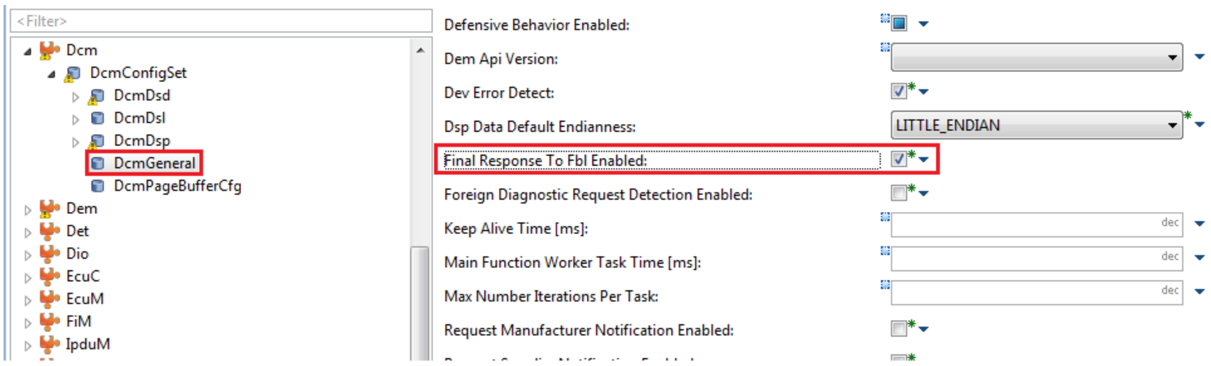
For more information see [4] (chapter 5.2 Transition from Application to Bootloader).

3.2 Jump From Bootloader to the Application

After all jobs have been finished in the Bootloader, usually after successfully programming an Application, the tester sends the diagnostic request for the jump to the Application.

Because the Application might have to send a correct response to the tester for this request and/or the Application must be informed that it has been updated, the FBL must store all required information before the jump can be executed (refer to chapter 4.2).

If a positive response has to be sent by the application, the `Final Response To Fbl Enabled` parameter has to be enabled (see Figure 3-5). When this parameter is active, the Dcm evaluates the `ResponseRequired` member of the "progConditions" input data structure of the `Dcm_GetProgConditions` to determine if a positive response has to be sent or not.

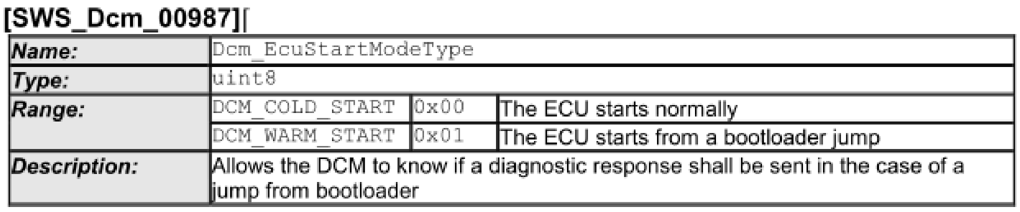

<Filter>
- Dcm
  - DcmConfigSet
    - DcmDsd
    - DcmDsl
    - DcmDsp
    - DcmGeneral
    - DcmPageBufferCfg
- Dem
- Det
- Dio
- EcuC
- EcuM
- FiM
- IpduM

Defensive Behavior Enabled:	■
Dem Api Version:	
Dev Error Detect:	✓
Dsp Data Default Endianness:	LITTLE_ENDIAN
Final Response To Fbl Enabled:	✓
Foreign Diagnostic Request Detection Enabled:	☐
Keep Alive Time [ms]:	dec
Main Function Worker Task Time [ms]:	dec
Max Number Iterations Per Task:	dec
Request Manufacturer Notification Enabled:	☐

Figure 3-5 - Activate Final Response to FBL Enabled

Caution
The user has to make sure that the information required from Bootloader is already available when `Dcm_Init` is called during start-up of the Application, for instance by restoring the data from non-volatile memory before.

During start-up (in scope of processing `Dcm_Init`) the Dcm will execute the application callout `Dcm_GetProgConditions` (refer to [1] for detailed API description). This callout must be implemented by the user. The goal is to detect if there was a normal start-up or a jump from the bootloader. This must be indicated to Dcm by using either the return value `DCM_COLD_START` or `DCM_WARM_START` of the type `Dcm_EcuStartModeType` (see figure below). In case of `DCM_COLD_START`, no further handling is required, because the Dcm will skip the respective evaluation of information and continue.



[SWS_Dcm_00987]⌈

Name:	`Dcm_EcuStartModeType`		
Type:	`uint8`		
Range:	`DCM_COLD_START`	`0x00`	The ECU starts normally
	`DCM_WARM_START`	`0x01`	The ECU starts from a bootloader jump
Description:	Allows the DCM to know if a diagnostic response shall be sent in the case of a jump from bootloader		

Figure 3-6 - Specification of Dcm_EcuStartModeType from [2]

In case of `DCM_WARM_START` the information stored by Bootloader must be read, translated to type `Dcm_ProgConditionsType` (see Figure 3-2) and provided to Dcm via parameter `ProgConditions`.

For more information see [4] (chapter 5.3 Transition from Bootloader to Application).

4 Bootloader

If a reprogramming request is present (set by application) or the application is not valid, then the Bootloader is started.

4.1 Jump From Application to Bootloader

During a specific service of the tester, the Application jumps to the Bootloader. To stay in the Bootloader and not to start the Application again, the Application needs to set a reprogramming request flag. There are two possibilities to share this information with the Bootloader.

Shared Memory: A shared memory location either in non-volatile memory (e.g. Eeprom or NvM) or in uninitialized RAM is used to hand over the programming request flag. This memory area must be written by the Application. After the Application has written the flag, it triggers a reset. For more information on how to share non-volatile data between Application and Bootloader using FEE Blocks see chapter 5.4.

FblStart: FblStart uses a RAM pattern called Magic Flag. Once the reprogramming request from the tester is received by the application the FblStart function is invoked by the application. This function simply writes a Magic Flag to RAM and performs a reset. After the reset the BM/FBL will not initialize the area where the Magic Flag has been written before. This enables the BM/FBL to check whether the Magic Flag has been written prior to the issued reset. Depending on the architecture (with or without BM) if it is written the BM would jump to the FBL or the FBL would stay in the FBL and will wait for further Tester requests.
Hint: The application does not have to consider the location of the Magic Flag because when the FblStart function is invoked the context is switched back to the BM/FBL. The Magic Flag is written to the RAM area which is linked in the BM/FBL and after the reset the BM/FBL will read from the RAM area which is linked in the BM/FBL.

In both variants the Boot Manager / Bootloader reads the flag (e.g. `ApplFblExtProgRequest()`) and clears it.

Caution
`CallFblStart()` does not return to the Application. Please make sure that all tasks have been finished, e.g. shutting down the Os, 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 start-up code).
2. The bootloader is accidentally started after power-up because the RAM location randomly contains 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 perform the call of `CallFblStart()`, you potentially overwrite the data. The ECU should be shutdown then immediately. This could otherwise cause unpredictable behavior of the ECU.

4.2 Jump From Bootloader to the Application

If the Bootloader receives a service request which leads to a reset (e.g. ECU Reset), there are two possibilities to handle the response:

The Bootloader sends the response and triggers the reset afterwards. This is the easier variant, as the Application does not need to send anything (see next variant), but the tester has to take into account that the reset takes some time.
The Bootloader sends a ResponsePending message, sets a flag and triggers a reset. The software which runs after reset (e.g. the Application) reads this flag, clears it and finally sends the response. This feature is called **Response After Reset**. In this case, the tester does not need to take anything into account, as the response is sent (e.g. by the Application), when everything is done. This flag is handed over via RAM or non-volatile memory (e.g. Eeprom or NvM).

4.3 Tester Source Address

In case the Bootloader supports multiple connections, the current connection (tester source address) needs to be handed over the reset.

The Bootloader already provides the possibility via FblCwPrepareResponseAddress() and FblCwSaveResponseAddress() to store/read this information.

4.4 Data exchange

The Bootloader stores/reads the required information via the component WrapperNv. The location of the data can be checked in WrapNv_Cfg.h.

5 Data Exchange Between Application and Bootloader

Not all data will be written and read by both, Application and Bootloader, and the format might differ. The next chapters explain which information is provided and/or consumed by which software part and how it will have to be formatted.

5.1 Data Exchange

The following table shows which software part will read and/or write which data.

Data / Software Part	Tester Source Address	Protocol ID	Service ID	Subfunction ID	Reprogramming Request Flag	Application Updated Flag	Response Required Flag
Application	r/w	-	r/w	r/w	w	r*/w	r/w
Bootloader	r/w	-	r/w		r	-**	r/w

Table 1 - Data Exchange Application/Bootloader

* Application Updated Flag will only be evaluated by Dcm, if the respective feature is used

**Bootloader does not handle Application Updated Flag. If Application is using this feature, the user has to handle this flag on bootloader side as described in 5.3.4.

5.2 Data Format and Usage (Application)

The following subchapters shall describe the handling of respective data by the Application and the expected format.

5.2.1 Tester Source Address

The Dcm uses the Tester Source Address to identify which tester has requested a diagnostic service. This value should be unique and corresponds to the AUTOSAR configuration parameter `DcmDslProtocolRxTesterSourceAddr` which must be configured for each `DcmDslMainConnection` in **| MICROSAR | Dcm | DcmConfigSet | DcmDsl | DcmDs l Protocol | DcmDslProtocolRow/DcmDslConnection |**.

For more information see [4] (chapter 5.1 Address Configuration).

5.2.2 Protocol ID

The Protocol ID is not used by Dcm.

5.2.3 Service ID

The Service ID is important for assembling the response to the tester. The Dcm uses the hexadecimal values as specified in the ISO14229-1.

5.2.4 Subfunction ID

The Subfunction ID is important for assembling the response to the tester. The Dcm uses the hexadecimal values as specified in the ISO14229-1.

5.2.5 Reprogramming Request Flag

The Reprogramming Request Flag is the indicator for the FBL whether to stay in bootloader and wait for reprogramming of the Application or to start-up the Application. The Dcm is setting/clearing this flag which is a Boolean value (**True** means reprogramming required, **False** means no reprogramming required), but not reading it.

5.2.6 Application Updated Flag

The Application Updated Flag is the indicator for the Dcm whether the Application was reprogrammed or not. The Dcm is only reading the flag (**True** means, the Application was updated, **False** means, the Application was not updated) if respective BSW mode port is used in BswM module, but always clearing it, regardless whether the feature is used.

5.2.7 Response Required Flag

The Response Required Flag is the indicator for the Dcm whether to send a response to a diagnostic request to the other software part or not. The Dcm has to read and set/clear this flag, that is a Boolean value (**True** means response required, **False** means no response required).

5.3 Data Format and Usage (Bootloader)

The following chapters describe the handling of the respective data within the Bootloader and the mapping to the Dcm data structure. Also check the OEM-related TechnicalReference (see [3]) within the Bootloader delivery.

5.3.1 Tester Source Address

The Bootloader has two variants for handling the tester source address. It can either be Dcm-based or index-based. For Dcm-based Bootloaders, the behavior is the same as in the Application (see chapter 5.2.1). For all other Bootloaders, where no Dcm module is available, the Bootloader will use an index-based value. The actual connection which is used, needs to be checked within the Bootloader configuration.

5.3.2 Reset Response Handling

In case the Bootloader is configured for Response After Reset, the starting software (e.g. the Application) needs to read out this flag (`ResetResponseFlag`) and depending on the value it sends a response to the tester by setting the Service ID and Subfunction ID to the specific value.

Value	Description
1	Response to the DiagnosticSessionControl-DefaultSession required.
2	Response to the EcuReset-HardReset required
3	Response to the EcuReset-KeyOffOnReset required
4	Response to the One-Step-Updater required
Others	No response required

Table 2 - ResetResponseFlag Definition

The actual location of this flag needs to be checked within the Bootloader.

5.3.3 Reprogramming Request Flag

To stay in the Bootloader and wait for a reprogramming, the Application needs to set the `ProgrammingRequestFlag` to a specific value. This flag is a Bootloader-specific flag and needs to be set when `Dcm_SetProgConditions::ReprogrammingRequest` is set to `True`.

Value	Desciption
0xB5	Normal reprogramming
0x5B	XCP reprogramming
0x5C	OTA reprogramming

Table 3 - ProgrammingRequestFlag Definition

The actual location of this flag needs to be checked within the Bootloader.

5.3.4 Application Update Flag

The Bootloader does not support such feature by default. If it is required, the customer can set a flag in a validation function of a block (e.g. **ApplFblValidateBlock**) that this block has been reprogrammed. This flag can then be checked within the Application.

5.4 Sharing non-volatile data between Application and Bootloader using FEE Blocks

This below sub chapters addresses the case that nonvolatile data should be shared between application and bootloader and how this can be achieved by using the AR memory stack within the bootloader

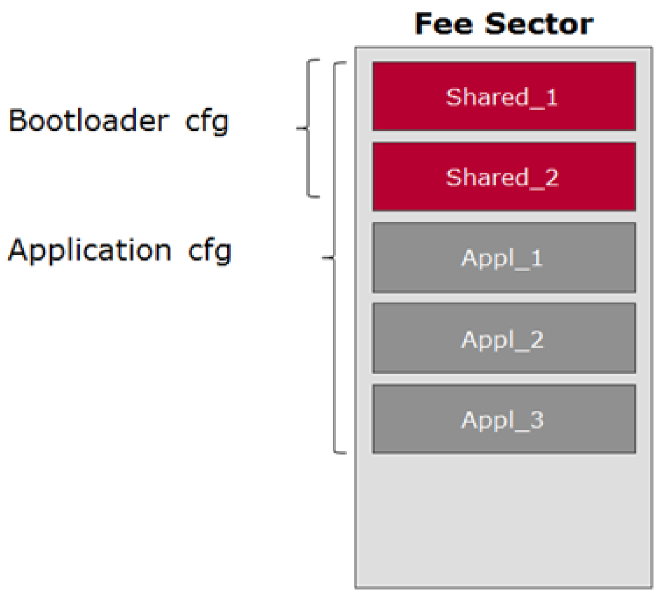


Figure 1-1 Shared FEE-Blocks between application and bootloader

Chapter 5.4.2 explains how the FEE works in the given use case and which are the necessary reconditions on the FEE layout.

Chapter 5.4.3 describes the necessary actions in the DaVinci Configurator Pro 5 step by step.

5.4.1 Preconditions on the FEE Layout

In general, only a subset of the nonvolatile blocks needs to be shared between the application and the bootloader. This leads to some preconditions on the FEE layout which will be described in the following sub chapter.

Note
Details on NvM and FEE configuration are not covered by this document.

5.4.1.1 MSR Module Precondition

At minimum the following AR Modules have to be used in bootloader:

1. AR MemIF
2. AR FEE
3. AR FLS
4. Stub for AR NVM or AR NVM (compare 5.4.1.2.1, 5.4.2.4)

In case the flash bootloader does not include ASR components the following is needed additionally

5. Stubs for AR DET and SCHM

This is necessary since the FEE uses a dynamic data layout. Therefore, it is not possible to address a memory block directly as it is in case of using EEPROM.

5.4.1.2 Config Limitations

This chapter describes the bootloader configuration limitations.

5.4.1.2.1 NvM Configuration in the Bootloader

Using the AR NvM module is optional and depends on the project specific requirements. If NvM features are necessary for the block to be shared (e.g. RAM Block CRC or NvBlockType redundant block) the NvM has to be integrated as well. Please note that this approach leads to higher resource consumptions of about 30-40 kByte ROM in the bootloader.

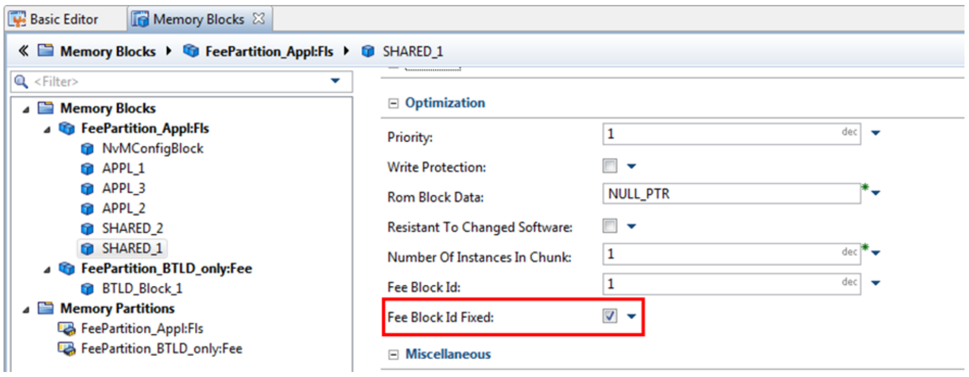
Real ASR Nvm module usage is not detailed within this document.

In any case, you must configure the `NvMDataSelectionBits` parameter in order to allow the FEE calculates valid block numbers. Currently it is therefore mandatory create an NvM stub in case no real NVM is present.

5.4.1.2.2 Block Limitation

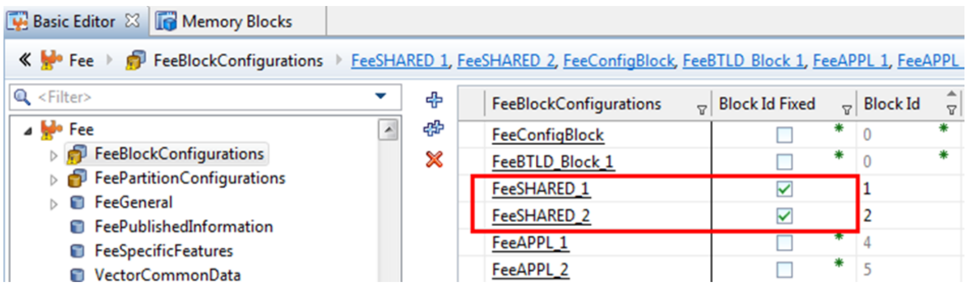
The integrator has to assure that the block ID of the given FEE block is the same in both configurations APPL and bootloader. It is therefore recommended that NvM/FEE-Blocks to be shared are located at the beginning of the FEE Layout to be robust against later layout changes.

Set the parameter `Fee Block Id Fixed` to **enabled** for at least all blocks which shall be accessed by the bootloader only and all blocks which can be accessed by bootloader and application. But it is also advisable to enable this parameter for application blocks to assure that a non-voluntary change of Fee Block Id is not possible.



Priority:	1
Write Protection:	☐
Rom Block Data:	NULL_PTR
Resistant To Changed Software:	☐
Number Of Instances In Chunk:	1
Fee Block Id:	1
Fee Block Id Fixed:	✓

Figure 2-1 Parameter Block Id Fixed in DaVinci Configurator Pro Comfort Editor Memory Blocks


Basic Editor
Memory Blocks
Fee ▸ FeeBlockConfigurations ▸ FeeSHARED_1, FeeSHARED_2, FeeConfigBlock, FeeBTLD_Block_1, FeeAPPL_1, FeeAPPL
<Filter>
Fee
FeeBlockConfigurations
FeePartitionConfigurations
FeeGeneral
FeePublishedInformation
FeeSpecificFeatures
VectorCommonData

FeeBlockConfigurations	Block Id Fixed	Block Id
FeeConfigBlock	☐	0
FeeBTLD_Block_1	☐	0
FeeSHARED_1	✓	1
FeeSHARED_2	✓	2
FeeAPPL_1	☐	4
FeeAPPL_2	☐	5

Figure 2-2 Parameter Fee Block Id Fixed in the DaVinci Configurator Pro Basic Editor

5.4.1.3 FEE Feature FeeFblConfig

The Vector FEE provides a feature called FeeFblConfig which sets the FEE into a special bootloader mode.

In this mode the FEE is able to detect and handle blocks which are not part of the bootloader configuration, but which need to be handled in case of a sector switch.

This chapter gives a short description about the issues non-Vector solutions may cause and how the Vector solution addresses this problem.

5.4.1.3.1 Potential problem with Regular AR FEE

The bootloader knows only its own blocks. But what happens in case of a sector switch in the bootloader? In case a regular FEE is used only the blocks known in the current configuration will be handled. This would lead to a loss of all data stored within the application blocks. To overcome this problem, you would have to use an exact copy of your application configuration within your bootloader. But this approach would lead to the fact that you have to update the bootloader every time you change the Nv-layout of some of the application blocks which is not feasible.

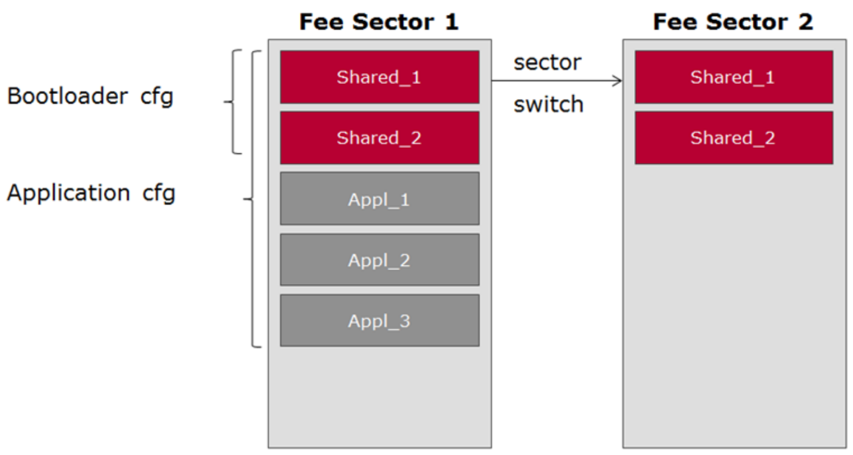


Figure 2-3 FEE Sector Switch with Regular AR FEE

5.4.1.3.2 Vector FEE Bootloader Configuration

Therefore, the Vector FEE can be switched in a special bootloader mode. This enables the FEE to scan the flash image and detect valid application blocks even if they are not part of the bootloader configuration. This way Nv-blocks can easily be shared between bootloader and application without the need to keep all the application blocks within your bootloader configuration.

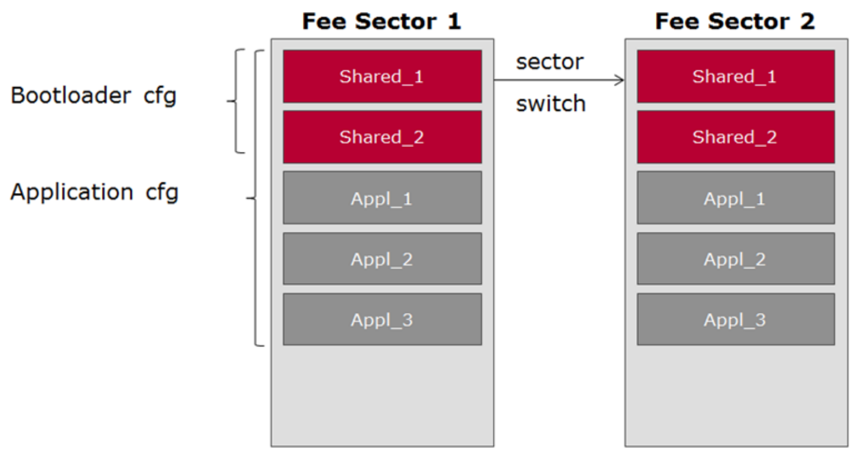


Figure 2-4 FEE Sector Switch with Vector FEE

.

Note
Please note that the background sector switch (BSS) is disabled in this mode. The FEE works according to a single sector principle.

5.4.2 FEE Configuration with the DaVinci Configurator Pro

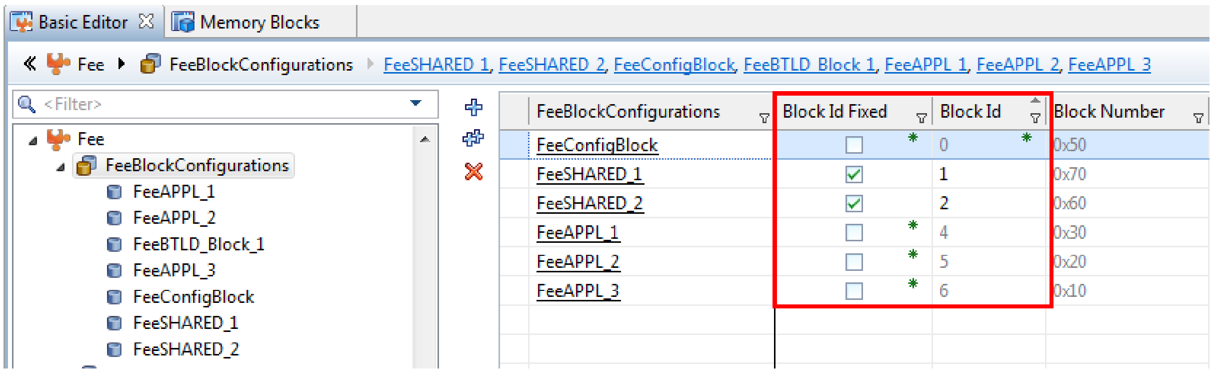
This chapter shows the necessary actions to be done in the DaVinci Configurator Pro in order to share Nv-blocks between application and bootloader. The described use case just uses FLS and FEE. If you plan to use the NvM in the bootloader you can skip step4 (5.4.2.4)

Note
Details on NvM and FEE configuration are not covered by this document.

Further information how to configure NvM and FEE can be found in the technical reference of these modules.

5.4.2.0 Step 1 Configure the NvM/FEE on Application Side.

Configure your NV layout as needed for your project and start with the blocks to be shared. Make sure that the block IDs are fixed as described in 5.4.1.2.2. Verify that the shared blocks do have the smallest block IDs (beside the FeeConfigBlock) in your FEE partition.

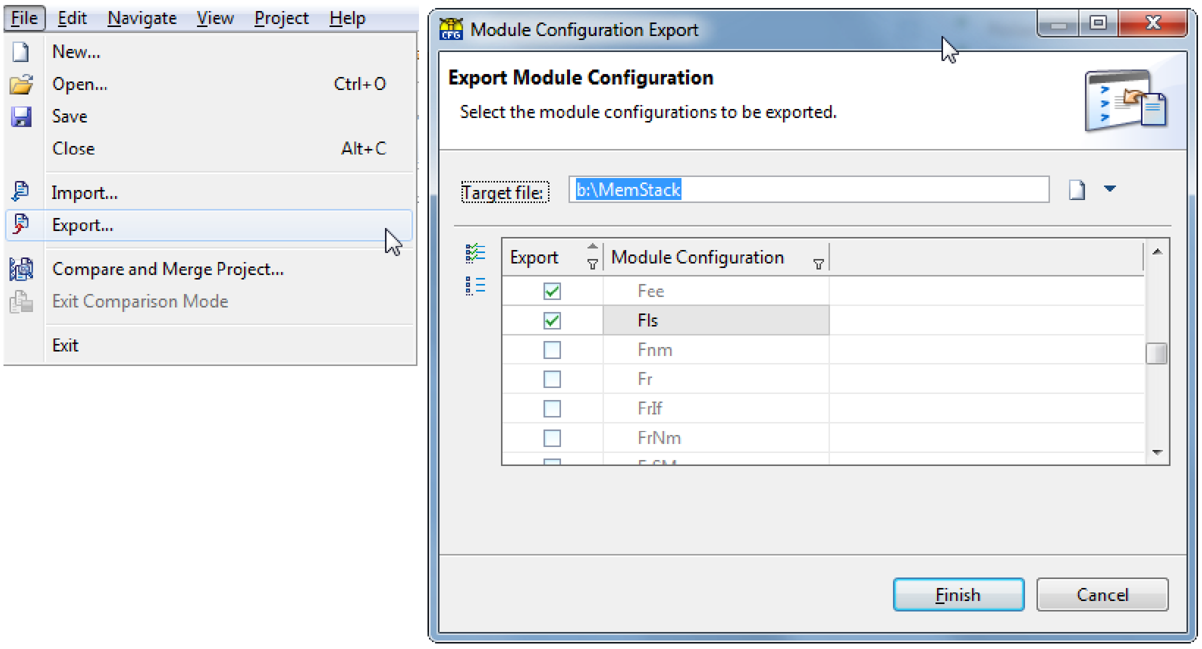


FeeBlockConfigurations	Block Id Fixed	Block Id	Block Number
FeeConfigBlock	☐	0	0x50
FeeSHARED_1	✓	1	0x70
FeeSHARED_2	✓	2	0x60
FeeAPPL_1	☐	4	0x30
FeeAPPL_2	☐	5	0x20
FeeAPPL_3	☐	6	0x10

Figure 3-1 Step1 Configure NvM/FEE

5.4.2.1 Step 2 Transfer the MEM Stack configuration to the Bootloader

Now you can export the MEM stack configuration from the application and import it into the bootloader. This gives you an easy start for the mem stack configuration of the bootloader.To export the configurations, use the export wizard of the DaVinci Configurator Pro.


File Edit Navigate View Project Help
New...
Open... Ctrl+O
Save
Close Alt+C
Import...
Export...
Compare and Merge Project...
Exit Comparison Mode
Exit
Module Configuration Export
Finish
Cancel

Export Module Configuration

Select the module configurations to be exported.

Target file: b:\MemStack

Export	Module Configuration
✓	Fee
✓	Fls
☐	Fnm
☐	Fr
☐	FrIf
☐	FrNm

Figure 3-2 Export Configurations with DaVinci Configurator Pro

Once this is done, you can import the configuration into your bootloader project.

Note
If your Fbl does not use DaVinci Configurator, but another tool (e.g. GENy), you have to create an empty configuration within the DaVinci Configurator first and import the configuration here in order to generate your Fee configuration (chapter **Fehler! Verweisquelle konnte nicht gefunden werden.** describes some more aspects for this situation).

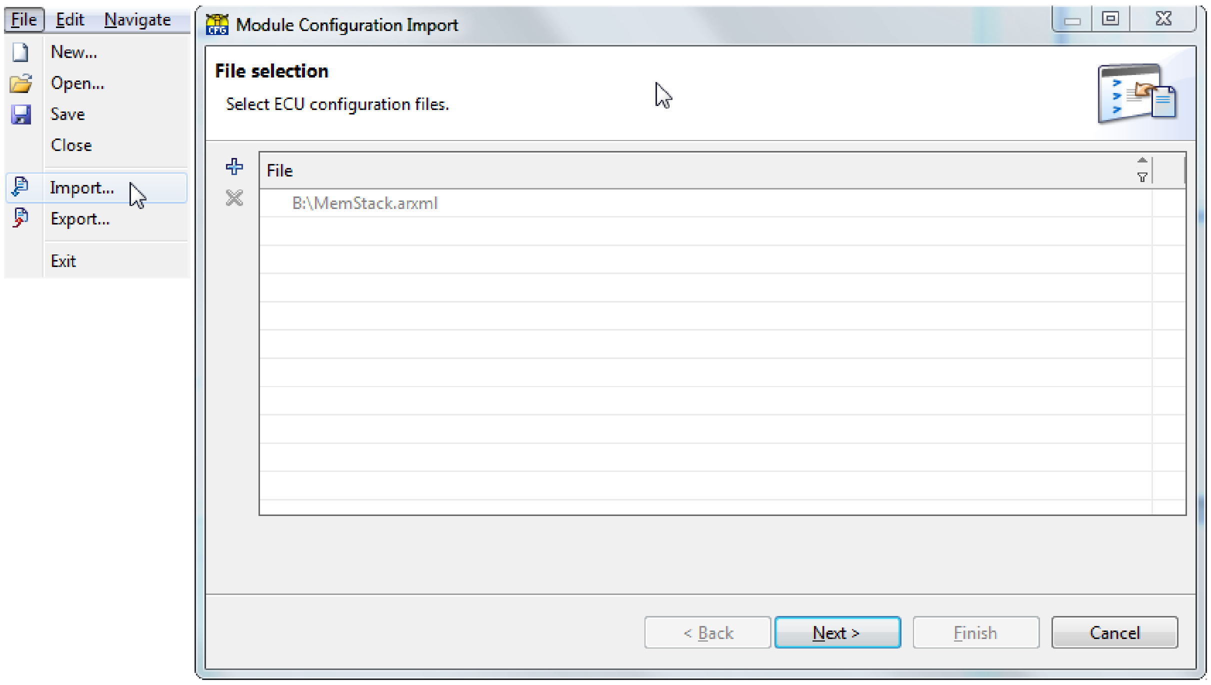


Figure 3-3 Import Configuration with DaVinci Configurator

The DaVinci Configurator automatically adds the new modules to your configuration.

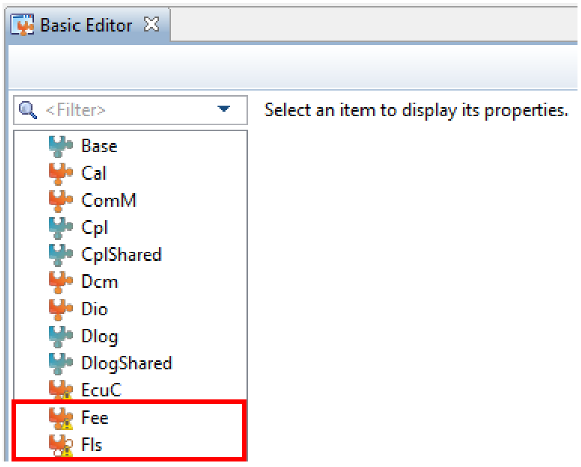
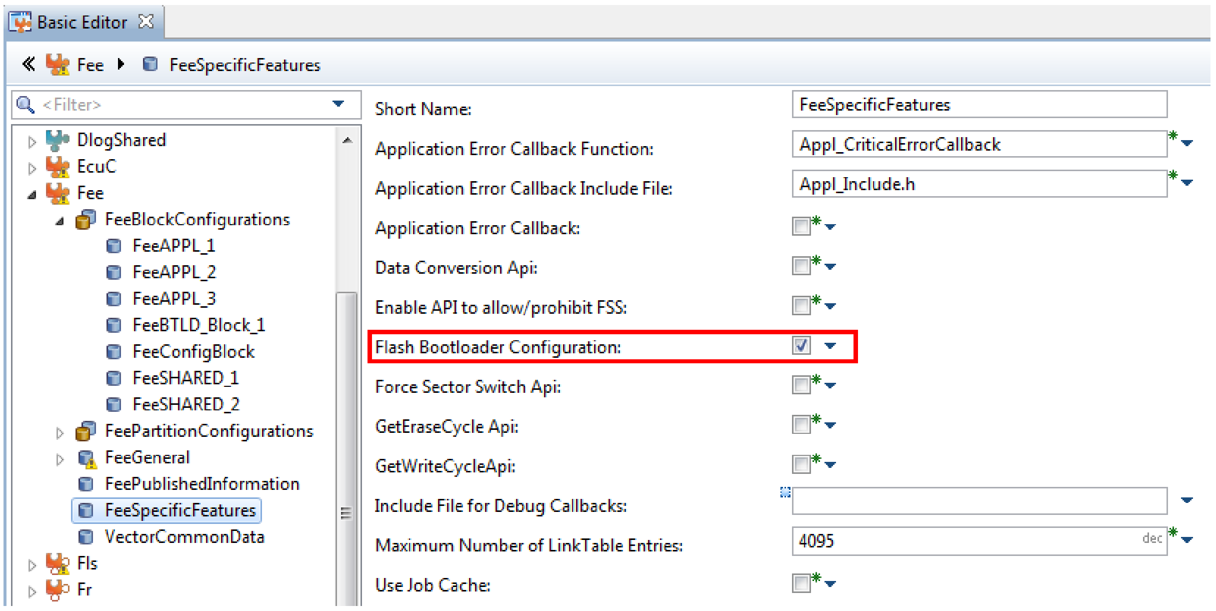


Figure 3-4 Imported MEM Stack Modules

5.4.2.2 Step 3 Adapt the imported Configuration to Your Needs

Once the import is done you can start to adapt the configuration to the needs of the bootloader. As we learned in chapter 5.4.1.3 we have to switch the FEE into the bootloader mode, this gives the FEE the information that it shall work in a special (bootloader) mode and there will be additional blocks in dataflash which are not known by the configuration (all blocks accessible only in application) but the FEE must take care for these blocks during internal management operations.



Short Name:	FeeSpecificFeatures
Application Error Callback Function:	Appl_CriticalErrorCallback
Application Error Callback Include File:	Appl_Include.h
Application Error Callback:	☐
Data Conversion Api:	☐
Enable API to allow/prohibit FSS:	☐
Flash Bootloader Configuration:	✓
Force Sector Switch Api:	☐
GetEraseCycle Api:	☐
GetWriteCycleApi:	☐
Include File for Debug Callbacks:	
Maximum Number of LinkTable Entries:	4095 dec
Use Job Cache:	☐

Figure 3-5 FEE Bootloader Mode

Note

It is possible to use FEE within bootloader context without using the bootloader mode.

In that case, the bootloader blocks must be in their own partition. That has the advantage that all services can be used without restrictions and the disadvantage that

potentially a lot of flash memory is needed

Now we can delete the blocks which are not needed in the bootloader. This can be easily done by the multi select feature of the DaVinci Configurator Pro. Mark the blocks you want to delete and hit the delete button.

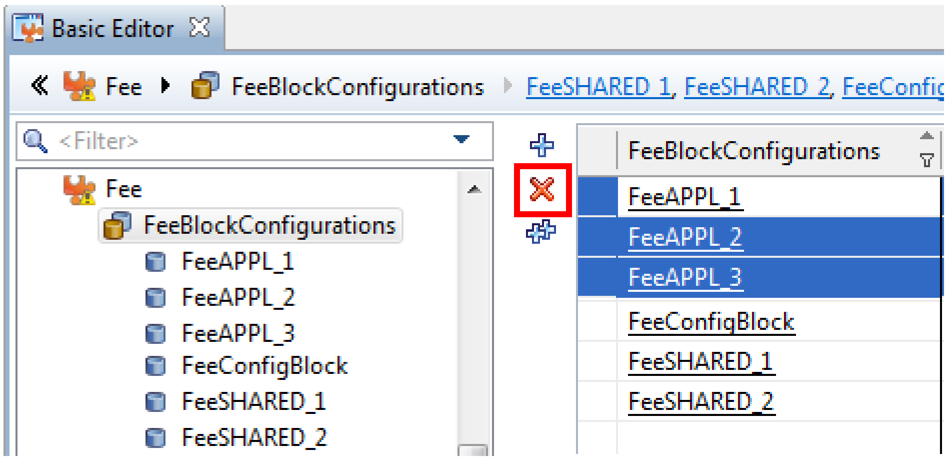
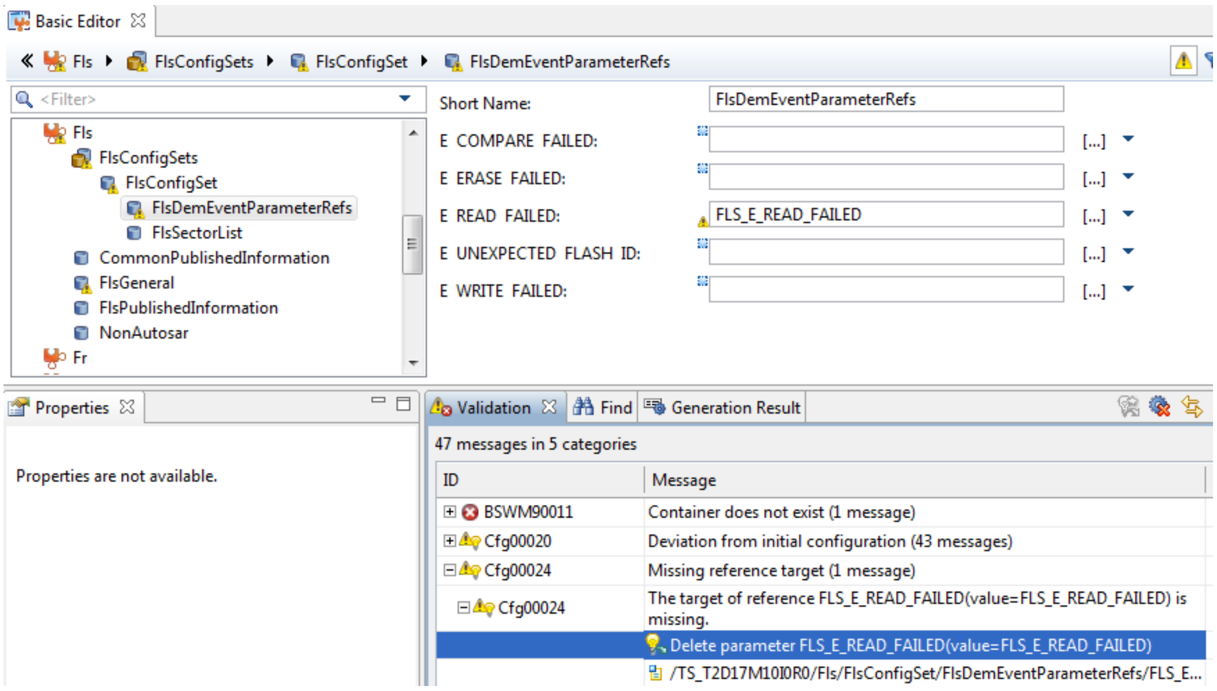


Figure 3-6 Delete Not Used FEE Blocks

Last step: Disable features which are enabled for the application configuration but not used in the bootloader. The development and production error detection for example. You can use the validation messages of DaVinci Configurator Pro. It will execute the necessary actions automatically.



Short Name:	FlsDemEventParameterRefs
E_COMPARE_FAILED:	
E_ERASE_FAILED:	
E_READ_FAILED:	FLS_E_READ_FAILED
E_UNEXPECTED_FLASH_ID:	
E_WRITE_FAILED:	

Properties are not available.

47 messages in 5 categories

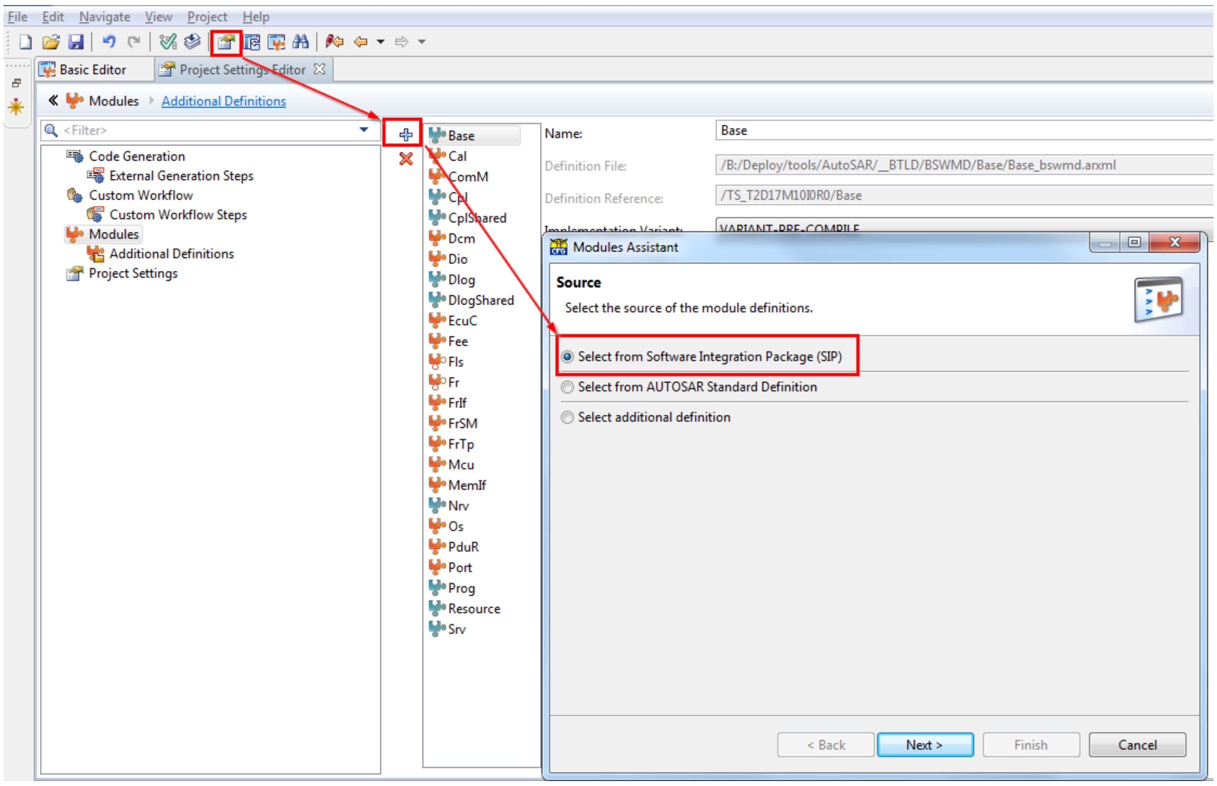
ID	Message
BSWM90011	Container does not exist (1 message)
Cfg00020	Deviation from initial configuration (43 messages)
Cfg00024	Missing reference target (1 message)
Cfg00024	The target of reference FLS_E_READ_FAILED(value=FLS_E_READ_FAILED) is missing.
	Delete parameter FLS_E_READ_FAILED(value=FLS_E_READ_FAILED)
	/TS_T2D17M10I0R0/Fls/FlsConfigSet/FlsDemEventParameterRefs/FLS_E...

Figure 3-7 DaVinci Configurator Validation Rules

5.4.2.3 Step 4 Add the Standard Definition of the NvM

To be able to configure the parameter `NvMDataSelectionBits` described in chapter 5.4.1.2.1 you have to add the AUTOSAR standard NvM definition to your configuration. This is only the case if you do not want to use the NvM in the bootlader.

To do so, add the NvM standard definition using the project settings editor of the DaVinci Configurator Pro.



Name:	Base
Definition File:	/B:/Deploy/tools/AutoSAR/_BTLD/BSWMD/Base/Base_bswmd.arxml
Definition Reference:	/TS_T2D17M10I0R0/Base
Implementation Variant:	VARIANT-PRE-COMPILE

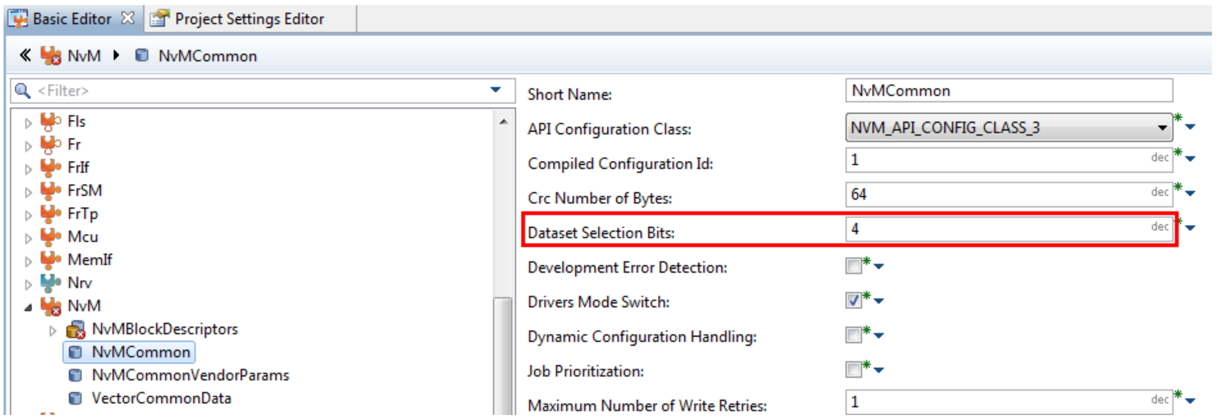
Source

Select the source of the module definitions.

[✓ Select from Software Integration Package (SIP)]
Select from AUTOSAR Standard Definition
Select additional definition

Figure 3-8 Add New Modules to Your Configuration

Now configure the parameter `NvMDataSelectionBits` to the same value as in the application configuration. All the other parameter errors and warnings can be ignored as we do not want to use nor generate the NvM module.



Short Name:	NvMCommon
API Configuration Class:	NVM_API_CONFIG_CLASS_3
Compiled Configuration Id:	1
Crc Number of Bytes:	64
Dataset Selection Bits:	4
Development Error Detection:	☐
Drivers Mode Switch:	✓
Dynamic Configuration Handling:	☐
Job Prioritization:	☐
Maximum Number of Write Retries:	1

Figure 3-9 NvMDataSelectionBits

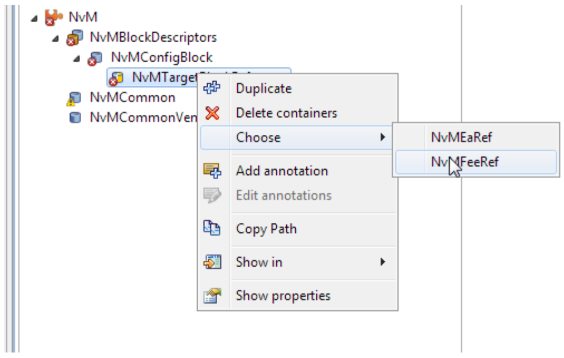
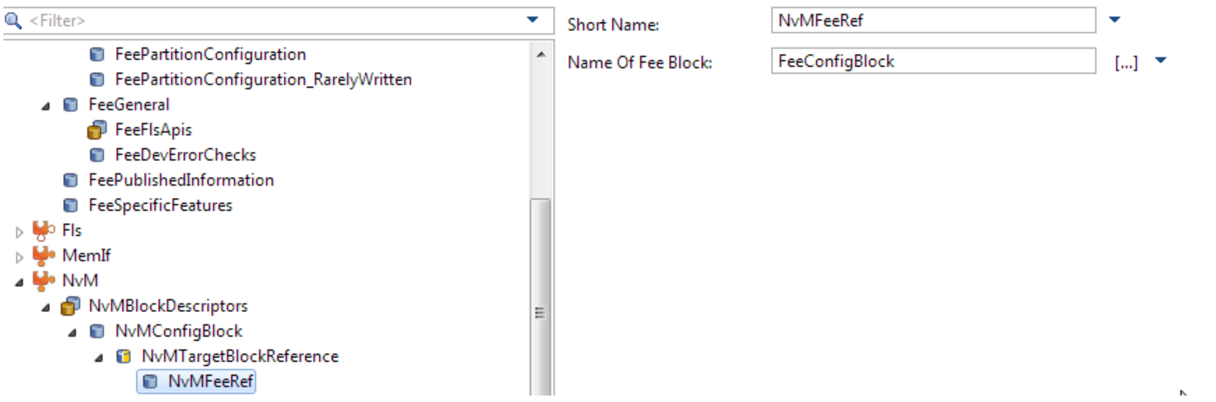


Figure 3-10 Choose Nvm Fee Ref



Short Name:	NvMFeeRef
Name Of Fee Block:	FeeConfigBlock

Figure 3-11 Nvm Stub Configuration to Allow Calculation of Valid Block Numbers

5.4.2.4 Configuration using MICROSAR Small Sector FEE

Compared to the MICROSAR Standard Fee, the MICROSAR SmallSector FEE works with static addresses for its blocks. This eases the handling of the blocks in the bootloader and the application. You can use the same block configuration for both application and bootloader.

The SmallSector FEE does not distinguish between Flash Bootloader and Application blocks in the configuration.

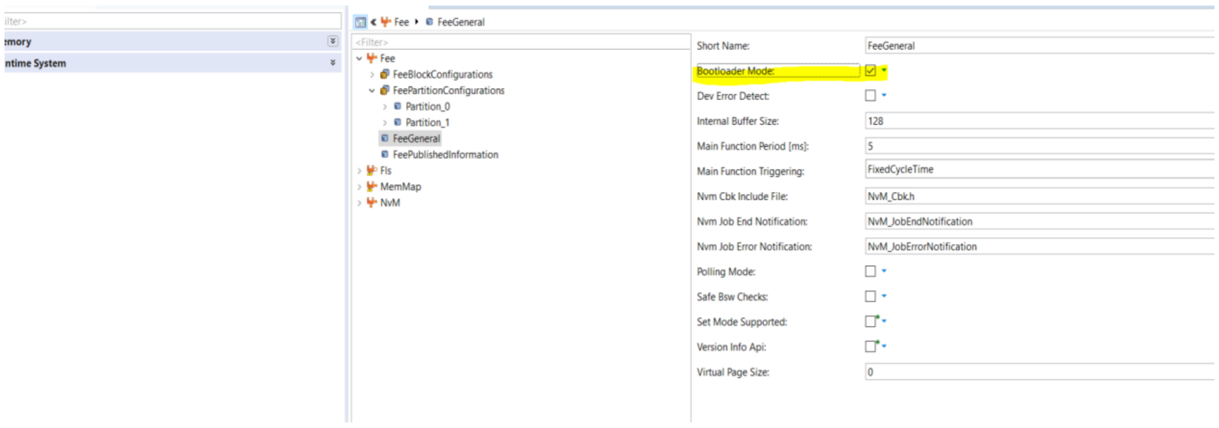
In this case there is no need to switch the FEE into the bootloader mode described in chapter 5.4.2.3. Please note that the parameters `Fee Block Id Fixed` and `Flash Bootloader Configuration` are not present in this configuration.

It is reasonable to use different FEE partitions to logically separate Flash Bootloader blocks from user data blocks. The Flash Bootloader partition should be the first partition in dataflash. Make sure the shared blocks have smallest block ID and same config between application and Flash Bootloader.

5.4.2.5 Configuration using MICROSAR FlexNor FEE

Using the new developed Vector FEE FlexNor component introduces a lot of new features, for more details check [5].

To use the FEE in bootloader mode the configuration parameter FeeGeneral/FeeBootloaderMode must be set to True.



Short Name:	FeeGeneral
Bootloader Mode:	✓
Dev Error Detect:	☐
Internal Buffer Size:	128
Main Function Period [ms]:	5
Main Function Triggering:	FixedCycleTime
Nvm Cbk Include File:	NvM_Cbk.h
Nvm Job End Notification:	NvM_JobEndNotification
Nvm Job Error Notification:	NvM_JobErrorNotification
Polling Mode:	☐
Safe Bsw Checks:	☐
Set Mode Supported:	☐
Version Info Api:	☐
Virtual Page Size:	0

In this mode the FEE provides services to read and write the configured blocks as usual but also has a few restrictions regarding other services.

Restrictions:

> The lookup table is not persisted in bootloader mode and hence the service to

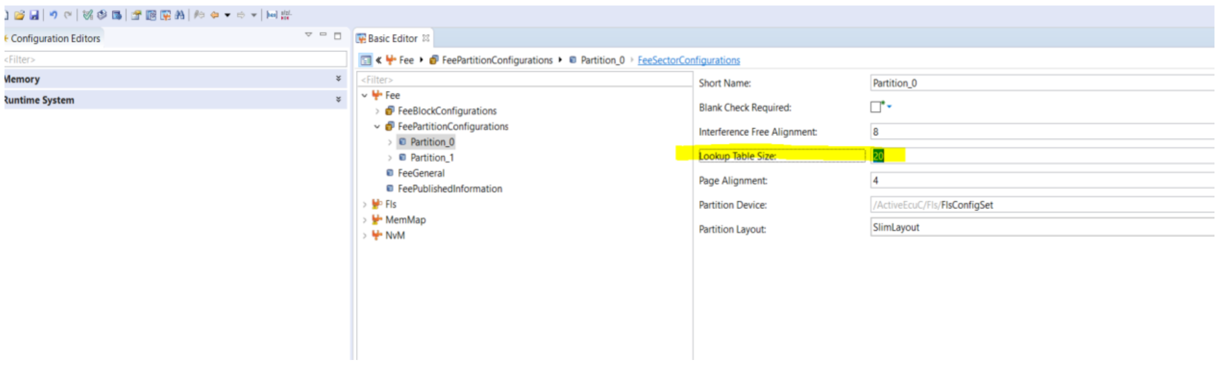
explicitly trigger the lookup table persistency is disabled. If called in bootloader

mode, the service will return E_NOT_OK.

> The size of the lookup table that is usually determined automatically must be

configured by the user for each partition.

User must configure the FeeLookupTableSize for every partition.



Short Name:	Partition_0
Blank Check Required:	☐
Interference Free Alignment:	8
Lookup Table Size:	20
Page Alignment:	4
Partition Device:	/ActiveEcuC/Fls/FlsConfigSet
Partition Layout:	SlimLayout

The parameter specifies the size of the lookup table which is used to reduce search effort for the latest valid instance of a block. If bootloader mode is enabled the lookup table size must be preconfigured by the user. The size should be at least the number of blocks configured for this partition plus one, because one index is reserved for the lookup table block.

ATTENTION: The size must be large enough to reserve space for future application updates without needing to update the bootloader.

It is possible to use the FEE within bootloader context without using the bootloader mode. In that case, the bootloader blocks must be in their own partition. That has the advantage that all services can be used without restrictions and the disadvantage that potentially more flash memory is needed.

If bootloader mode is disabled, the lookup table size is determined automatically from the number of blocks and cannot be configured by the user.

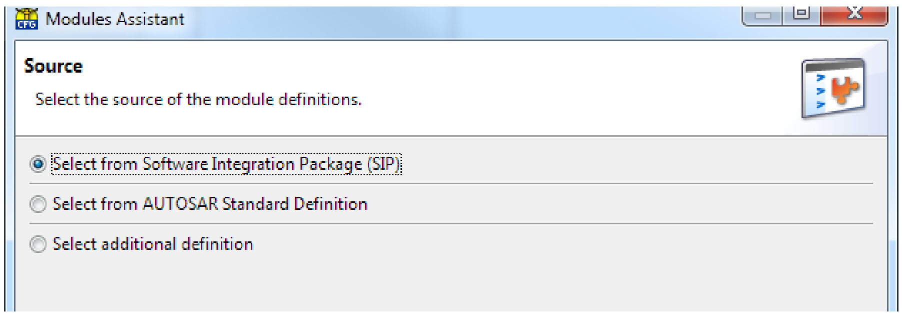
Please note that the parameters `Fee Block Id Fixed` is not present in this configuration.

It is reasonable to use different FEE partitions to logically separate Flash Bootloader blocks from user data blocks. The Flash Bootloader partition should be the first partition in dataflash.

5.4.2.6 Special configuration aspects in Non DaVinci Configurator Pro environments

In case your flash bootloader is not configured with DaVinci Configurator Pro some further configuration aspects need to be considered.

5.4.2.6.1 Further ASR Component Stubs required


Modules Assistant

Source

Select the source of the module definitions.

(●) Select from Software Integration Package (SIP)
( ) Select from AUTOSAR Standard Definition
( ) Select additional definition

Figure 3-12 Add Stubs for Det, EcuC, MemIf from Application Software Integration Package

After adding stubs for Det, EcuC and MemIf, some error may occur. Follow DaVinci Configurator Pro propositions to resolve these problems (e.g. add Fee reference to MemIf, resolve EcuC related errors (HW-dependent)).

5.4.2.6.2 Generation of the Fee Config for the Fbl

In case the configuration was created for Fbl Fee configuration only, only the minimum configuration Fee and Fls need to be generated

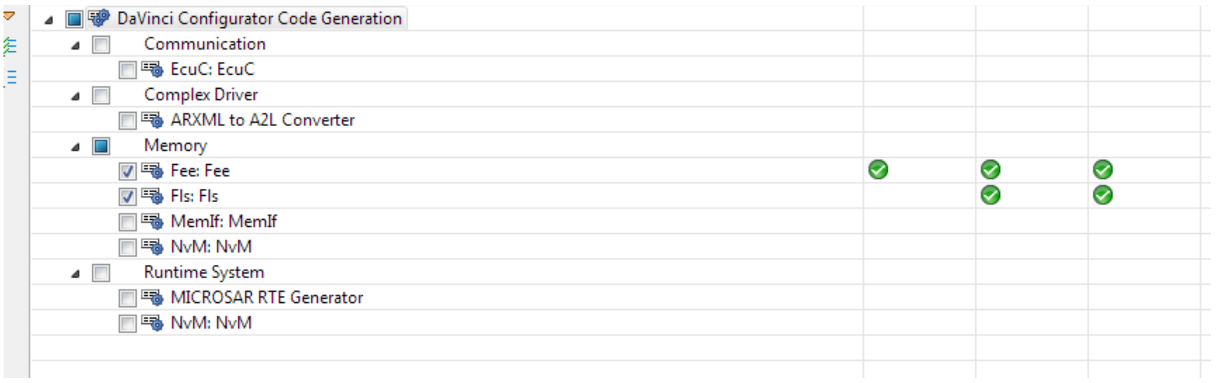


Figure 3-13 Generation of Fbl Config

5.4.3 Usage of Fee Configuration in Flash Bootloader

Once you imported the Fee configuration to the flash bootloader and are able to generate Fls/Fee configuration, the created output has to be added and compiled with the flash bootloader project. Additionally, static Asr modules and some static sub files have to be added to flash bootloader.

5.4.3.0 Modules Required in the Flash Bootloader

5.4.3.0.1 Asr BSW Modules to be added to Flash Bootloader

The following application core modules need to be added to your Fbl project:

Modules	Content
Fee	.\Fee > all content
Det	.\Det > Det.h
Fls	.\Fls > all content .\Mcal_<Hw> (Register definitions for Fls)
MemIf	.\MemIf > all header files
Other	._Common > all .\include of your project > compiler_cfg.h
Remove further includes within compiler_cfg.h (like Rte_Compiler_Cfg.h)	

Table 1 Asr BSW Modules

5.4.3.0.2 Asr Stub Modules to be used by the Fbl

The following stub modules are known to be required in the Fbl.

Modules	Content
Det	Det.c/Det_Cfg.h: Provide Prototype for function Det_ReportError
SchM	SchM_Fee.h :

	#define SchM_Enter_Fee_FEE_EXCLUSIVE_AREA_0() #define SchM_Exit_Fee_FEE_EXCLUSIVE_AREA_0()
	SchM_Fls.h: #define SchM_Enter_Fls(arg1, arg2) #define SchM_Exit_Fls(arg1, arg2) /* Function declaration is provided by SCHM */ FUNC(void, FLS_CODE) Fls_MainFunction(void);

Table 2 Stub modules

5.4.3.0.3 Generated Modules

The generated modules that should be added to the Fbl are

Modules	Content
Fee	Fee_Cfg.h
	Fee_Lcfg.c
	Fee_PrivateCfg.h
Fls	Fls_Cfg.h

Table 3 Generated modules

5.4.3.1 Initialization in Fbl

Depending on when the required Fee elements will be read, the initialization has to happen early enough to allow access to the required elements.

5.4.3.1.1 Initialization Required

Fls and Fee need to be initialized. In both initializations the following logic applies (replace Fxx with Fee/Fls)

Example

```
Call Fxx_Init()

while MEMIF_IDLE != Fxx_GetStatus() call Fxx_MainFunction()
```

Fee specific:

Example

Depending on configuration call Fee_EnableFss()

```
#if (FEE_FSS_CONTROL_API == STD_ON)
   Fee_EnableFss();
#endif
```

The location where the initialization needs to be done depends on the elements that need to be accessed via Fee. If elements need to be accessed before decision to jump to application (e.g. valid flag(s)), do the initialization within `ApplFblInit()`, otherwise within `ApplFblStartup()`, `initposition == kStartupPostInit`. If you need to do the initialization within `ApplFblInit()`, be sure to initialize watchdog and timer early in your configuration (set `FBL_ENABLE_PRE_WDINIT / FBL_ENABLE_PRE_TIMERINIT`)

5.4.3.2 Synchronized Calls to Fee

The bootloader usually requires synchronized calls to the Fee. The Fee API is usually asynchronous. A wrapper to embed the asynchronous API to synchronized calls should therefore be used.

5.4.3.2.1 Synchronized Read

Create a function to read from Fee in a synchronous way, e.g. with prototype

```
FblResult ApplFblNvRead ( vuint16 blockNumber, vuint16 blockOffset,
V_MEMRAM1 vuint8   V_MEMRAM2 V_MEMRAM3 * pBuffer, vuint16 length )
```

Example

Required implementation:

```
If  Fee_Read(blockNumber, blockOffset, pBuffer, length ) == E_OK
{
  While MEMIF_IDLE != Fee_GetStatus()
  {
    Call Fls_MainFunction()
    Call Fee_MainFunction()
       }
       //Check  Fee_GetJobResult() == MEMIF_JOB_OK /
MEMIF_BLOCK_INCONSISTENT /             MEMIF_BLOCK_INVALID
  If Fee_GetJobResult() == MEMIF_JOB_OK
    return OK
  Else If Fee_GetJobResult() == MEMIF_BLOCK_INCONSISTENT OR
          MEMIF_BLOCK_INVALID
    fill read buffer with FBL_FLASH_DELETED
    return OK
  Else
    return failed
}
Else return failed
```

5.4.3.2.2 Synchronized Write

Create a function to write to Fee in a synchronous way, e.g. with prototype

```
FblResult ApplFblNvWrite ( vuint16 blockNumber, V_MEMRAM1 vuint8
V_MEMRAM2 V_MEMRAM3 * pBuffer)
```

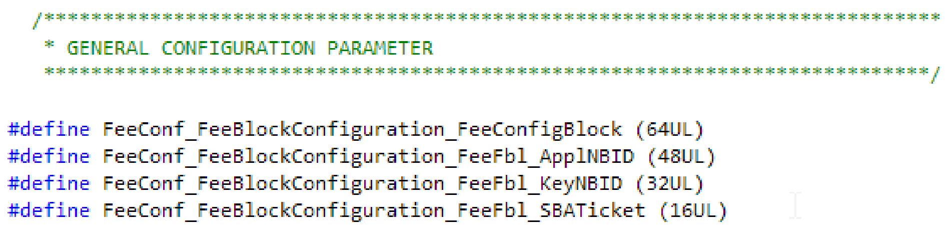
Example

Required implementation:

```
If  Fee_Write(blockNumber, pBuffer) == E_OK
{
  While MEMIF_IDLE != Fee_GetStatus()
  {
    Call Fls_MainFunction()
    Call Fee_MainFunction()
       }
  If  Fee_GetJobResult() == MEMIF_JOB_OK
    return Ok
  Else
    return failed
}
Else return failed
```

5.4.3.3 Data elements to be read and written by Fbl

In order to read and write from and to the Fee, the API requires the block numbers of elements. These block numbers can be found in `Fee_cfg.h`, macros starting with `FeeConf_FeeBlockConfiguration_Fee` e.g.:



```
  /**********************************************************************************
   * GENERAL CONFIGURATION PARAMETER
   *********************************************************************************/

#define FeeConf_FeeBlockConfiguration_FeeConfigBlock (64UL)
#define FeeConf_FeeBlockConfiguration_FeeFbl_ApplNBID (48UL)
#define FeeConf_FeeBlockConfiguration_FeeFbl_KeyNBID (32UL)
#define FeeConf_FeeBlockConfiguration_FeeFbl_SBATicket (16UL)
```

Figure 4-1 Example Fee Block Numbers Used in Flash Bootloader

The length information required for read information has to be defined somewhere in the flash bootloader

Example
```
#define FBL_NV_APPNBID_FEE_LENGTH          2u
#define FBL_NV_KEYNBID_FEE_LENGTH          2u
#define FBL_NV_SBAT_FEE_LENGTH           822u
```

5.4.3.4 Configure existing Flash Bootloader NVM Data for Fee

Existing NVM elements are read and written via the fbl_apnv.c/.h module. Usually functions and or macros are prepared to map to the generated WrapNv_cfg.h macros usable for either EEPROM or Eepm. For Fee configuration these elements now have to be mapped to synchronized Fee read/write functionality.

An example call that can be mapped to the corresponding fbl_apnv.c/.h function for reading the above mentioned element Fbl_SBATicket e.g. would be then:

```
ApplFblNvRead(FeeConf_FeeBlockConfiguration_FeeFbl_SBATicket, 0u,
(buffer),FBL_NV_SBAT_FEE_LENGTH)
```

6 Additional Resources

No.	Source	Title	Version
[1]	Vector	TechnicalReference_Dcm.pdf	14.1.0
[2]	AUTOSAR	AUTOSAR_SWS_DiagnosticCommunicationManager.pdf	4.2.2
[3]	Vector	TechnicalReference_FBL-<OEM>.pdf	
[4]	Vector	TechnicalReference_FblCw.pdf	3.03.01
[5]	Vector	TechincalReference_Fee_FlexNor.pdf	1.02.00

7 Contacts

For a full list with all Vector locations and addresses worldwide, please visit http://vector.com/contact/.