

# Flash Bootloader GM

TechnicalReference

CANfbl GM SLP5

Version 1.0

<b>Authors:</b>	Andreas Wenckebach / Dennis O'Donnell
<b>Version:</b>	1.0
<b>Status:</b>	Released

## 1 History

Author	Date	Version	Remarks
Andreas Wenckebach	2013-08-07	0.9	General Motors FBL SLP5
Dennis O'Donnell	2013-01-31	1.0	Updates for initial release

## Contents

<b>1</b>	<b>History .....</b>	<b>2</b>
<b>2</b>	<b>Introduction .....</b>	<b>8</b>
<b>3</b>	<b>FBL Software Architecture .....</b>	<b>9</b>
3.1	Components.....	9
3.2	Memory Layout .....	12
3.2.1	The GM File-Container.....	12
3.2.2	Operating S/W Interrupt Vector Table .....	14
3.2.3	Logical Block table .....	14
3.2.4	Partitions .....	14
3.2.5	Presence Patterns / Programmed State Indicator .....	14
3.3	Run-Time Details.....	15
3.3.1	Power-On Reset.....	16
3.3.2	Start from Operating Software .....	16
3.3.3	Download Sequence .....	17
<b>4</b>	<b>Configuration of the GM Flash Bootloader .....</b>	<b>20</b>
4.1	Overview .....	20
4.2	Starting with GENy .....	20
4.3	CAN Configuration .....	24
4.3.1	Single wire ECU (Body bus) .....	25
4.4	Bootloader Configuration.....	25
4.4.1	FblDrvCan component .....	26
4.4.2	FblCan_14230_Gm configuration.....	28
4.5	Memory Configuration .....	35
4.5.1	Device Types.....	36
4.5.2	Flash Block Definition.....	38
4.5.3	Logical Block Definition .....	39
4.6	Mandatory Delivery Preconfig .....	40
4.7	NV-Wrapper Configuration .....	41
4.8	Running the Generator.....	42
<b>5</b>	<b>Adapting the FBL Implementation .....</b>	<b>44</b>
5.1	Hardware, Input/Output and miscellaneous Callbacks.....	45
5.2	Diagnostic Service Callbacks .....	59
5.3	Module Validation Callbacks.....	69
5.4	Watchdog Callbacks.....	80
5.4.1	Start of Watchdog.....	81

5.4.2	Synchronize Watchdog with application .....	81
5.4.3	Window Watchdogs.....	81
5.4.4	Watchdog triggering during Memory operations .....	82
5.5	Callback configuration summary.....	85
5.5.1	Required callback configuration .....	85
5.6	Application Vector Table .....	86
5.7	Transport-Layer Configuration.....	86
5.8	[#hw_wd] – Compiling the Watchdog components .....	87
5.9	[#oem_valfunc] – Flashing After A Reset .....	87
5.10	[#oem_valid] – Proposals for Handling The Validation Area.....	87
5.11	[#oem_start] Startup .....	89
5.12	[#oem_ref] – Label Reference File .....	90
<b>6</b>	<b>Adapting the Operating Software.....</b>	<b>91</b>
<b>7</b>	<b>Device Driver .....</b>	<b>95</b>
7.1	General Information.....	95
7.2	High-Level Device-Driver Functions .....	95
<b>8</b>	<b>Using the Flash Tool for GM .....</b>	<b>101</b>
8.1	Preparing the File-Header .....	101
8.2	Configuring vFlash .....	101
8.2.1	vFlash Communication Tab .....	101
8.2.2	vFlash Miscellaneous Tab .....	103
8.2.3	vFlash Data Tab .....	103
8.3	Starting the flash sequence with vFlash.....	104
<b>9</b>	<b>Miscellaneous.....</b>	<b>105</b>
9.1	[#oem_multi] – Multiple-Identity-Modules .....	105
9.2	Multiple Processor Support .....	105
9.3	Request-Data-By-Identifier (Debug-Status) .....	106
9.4	[#oem_time] – Stay-In-Boot mode .....	107
9.5	User-Callable Support Functions.....	107
9.6	Low Power Mode in the bootloader .....	116
9.6.1	Integrated sleep mode enabled .....	116
9.6.2	Integrated sleep mode enabled with wakeup interrupt.....	116
9.6.3	Integrated Sleep mode handling disabled.....	116
9.7	Example / hints to prepare containers for Type A Ecus .....	117
9.8	Security Requirements .....	117
9.8.1	Digital Signature .....	118
9.8.2	Signature-Bypass Authorization (SBA) ticket.....	119

9.8.3	Message Digest .....	120
9.8.4	Signer Info.....	121
9.8.5	Application Software – Not Before Identifier (App-NBID) .....	122
9.8.6	Security Key – Not Before Identifier (Key-NBID).....	122
9.9	Programming of Unused Flash Space / Gap Fill .....	122
<b>10</b>	<b>Limitations.....</b>	<b>124</b>
10.1	Implementation-specific limitations .....	124
10.2	GMW3110 V1.4 V.1.5 V1.6.....	124
10.3	GMW3110 V 1.6.....	124
10.4	Global-A Secure Bootloader Specification .....	124
10.5	CG3532 ECU Security Requirements Ver. 1.0.....	124
<b>11</b>	<b>Terminology.....</b>	<b>126</b>
<b>12</b>	<b>References.....</b>	<b>130</b>
<b>13</b>	<b>Contacts.....</b>	<b>131</b>

## Illustrations

Figure 2-1	Manuals and References for the Flash Bootloader .....	8
Figure 3-1	Software component relationships .....	9
Figure	Example Memory Layout .....	12
Figure 4-1	Initial GENy main window .....	20
Figure 4-2	GENy Setup Dialog.....	21
Figure 4-3	GENy main window after pre-configuration .....	21
Figure 4-4	GENy Channel Setup .....	22
Figure 4-5	GENy main window after channel setup .....	22
Figure 4-6	GENy Components.....	23
Figure 4-7	GENy directory selection .....	24
Figure 4-8	GENy CAN Configuration .....	25
Figure 4-9	FblDrvCan configuration example.....	26
Figure 4-10	GM Fbl configuration Settings and GM Diagnostic service options .....	29
Figure 4-11	GM Modules and Boot Info Block configuration .....	33
Figure 4-12	Memory Configuration .....	35
Figure 4-13	Typical Logical Block partition .....	36
Figure 4-14	GENy Device Types.....	37
Figure 4-15	Example Device Type .....	38
Figure 4-16	GENy Flash Block Table .....	38
Figure 4-17	GENy Logical Block Table.....	39
Figure 4-18	NV-Wrapper configuration in GENy .....	42
Figure 8-1	Example vFlash Communication Configuration Dialogue.....	102
Figure 8-2	Example vFlash Miscellaneous Configuration dialog .....	103
Figure 8-3	Example vFlash Data Configuration Dialogue.....	104
Figure 9-1	Multi-Processor logical block table configuration.....	105
Figure 9-2	Application and Calibration signed header structures and signature calculation. "Signature" represents the digital signature. ....	119
Figure 9-3	Signature-Bypass Authorization Header structure and signature calculation. ....	119
Figure 9-4	Application and Calibration Message Digest.....	120
Figure 9-5	Signer Info Structure and signature calculation. ....	121

## Tables

Table 3-1	FBL Files .....	10
Table 3-2	User-modifiable Files .....	11
Table 3-3	Generated Files .....	11
Table 3-4	Boot Info Block configuration. ....	13
Table 3-5	Callback functions used by diagnostic service handlers.....	19
Table 4-1	Bootloader Configuration .....	28
Table 4-2	GM-Specific Configuration .....	32
Table 4-3	GM Modules and Boot Info Block Detail Configuration.....	35
Table 4-4	GENy configuration of the Logical Block Table.....	40
Table 4-5	Generated File contents.....	43
Table 5-1	User-Modifiable file contents.....	44
Table 5-2	Miscellaneous Callback functions .....	45
Table 5-3	Diagnostic Callback Functions .....	59
Table 5-4	Module Validation Callbacks .....	69
Table 5-5	Watchdog Callbacks .....	82

Table 6-1      Parameters passed to FBL by Operating Software ..... 93

Table 8-1      Parameter description of the communication tab in vFlash. .... 103

Table 8-2      Parameter description of the miscellaneous tab in vFlash..... 103

Table 9-1      Response for debug-status request ..... 106

## 2 Introduction

This document covers the GM-specific particularities of the Flash Bootloader. The bootloader is designed to comply with all requirements defined in references [1] and [2]. The documentation 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.

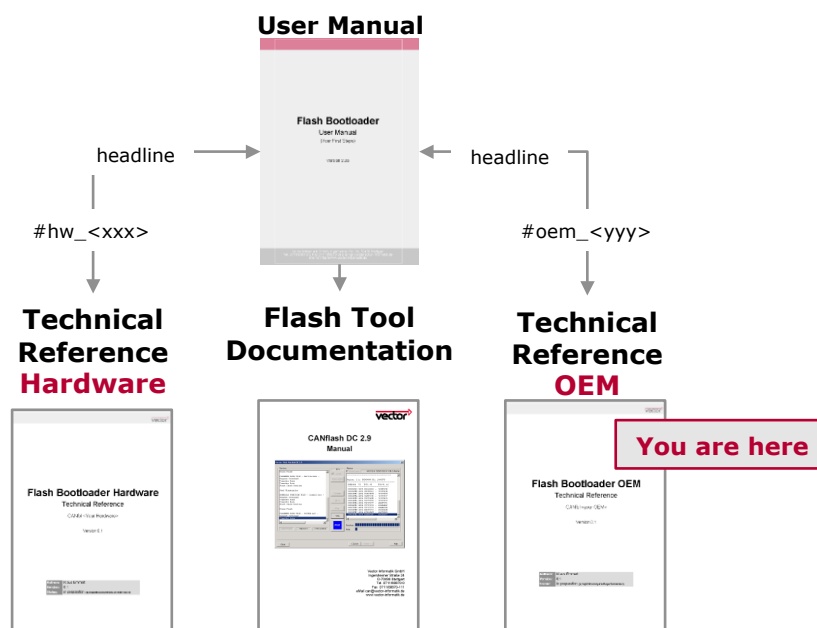


Figure 2-1 Manuals and References for the Flash Bootloader

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



### 3 FBL Software Architecture

#### 3.1 Components

The Flash Bootloader (FBL) is a complete, self-contained application made up of several software components. Each component is contained in individual .c and .h files. The components interact as shown below:

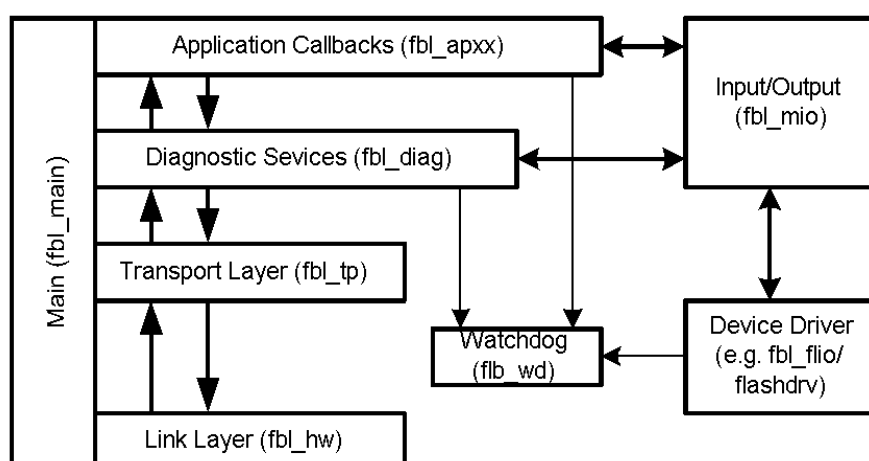


Figure 3-1 Software component relationships

**[#oem\_files]** A high-level description of each component follows. The components are grouped in three categories. The first group contains components found in the FBL, Flash, SecMod, Eep, and \_Common folders of your delivery (inside BSW if you have a new layout delivery). Do not modify the contents of the files in this group without prior written permission from Vector (modification of these files will void your warranty).



#### Note

There may be additional folders if non-standard components were also ordered (e.g. data flash driver, EEPROM manager, etc.).

Component/File	Description
applvect.h	Definition of Application-Vector-Table data structure
fbl_can.h	CAN parameter macros
fbl_def.h	Common data-type and structure definitions used by the FBL.
fbl_diag	Module implements diagnosis service functions. The implementation

	is specific to requirements defined by GM (reference [1]).
fbl_flio	Flash I/O routines provide interface to flash driver
fbl_hdr	Module for Gm header parsing. The implementation is specific to requirements defined by GM (reference [2]).
fbl_hw	Hardware dependent code (CAN communication and H/W Timer)
fbl_main	Main module with main loop
fbl_mem	Module for Diagnostic buffer handling and programming. Upon request the module can be delivered in order to support: Encryption / Compression / Pipelined Programming (interleaved data transfers). In standard configuration none of these features are supported.
fbl_mio	General purpose API to device-drivers (MIO == Memory Input/Output)
fbl_tp	Transport Layer – Combines (and splits) diagnostics requests (and responses) across multiple CAN message frames.
fbl_vect	FBL vector table – defines all Interrupt Service vectors (usually this cannot be modified – See Section 5.5).
fbl_wd	Watchdog support module
flashdrv, eepdrv, EepIO, etc.	Non-volatile memory device-driver. You should not modify any source found in the device-driver folder(s).
flashrom	C-array of the flash driver executable. This is linked to ROM and then copied to RAM at the appropriate time.
lotypes	Type definitions used by the memory Input/Output component
Sec_*	Security module. Used for required signature/hash calculations. The object files in Secmod/obj must be set to link with the rest of the bootloader source files.
v_def	ECU and compiler-specific type-definitions and macros used to abstract fundamental data types, pointers, and subroutine declarations.
v_ver.h	Version information of all delivered/licensed components.
WrapNv.h	Structures and macros used by the non-volatile wrapper.

Table 3-1 FBL Files

The second group consists of components that must be customized for your hardware and application. The files may be found in the FBL\\_Template folder. You should copy these files to your bootloader project folder, and rename them, removing the leading underscore from the filename.

Component/File	Description
_fbl_applvect.c	Application vector table – Jump table to all Interrupt Service Routines in Operating Software.
_fbl_ap	Hardware specific callback routines
_fbl_apdi	Application specific diagnostic routines
_fbl_apnv	Nonvolatile memory access routines, e.g. for presence-pattern handling.
_fbl_apwd	Application specific watchdog routines

_fbl_inc.h	Include file for including all FBL related include files
_memmap.h	Memory map configuration. Allows for custom memory placement of specific bootloader sections.
_wrapnv_inc.h	Include file for including any files necessary to the WrapNv component.
Application File	Description
_applvect.c	Application vector table configured to link to Operating Software. This will contain the calls to your application interrupt service routines.

Table 3-2 User-modifiable Files

The third set of components is generated by the configuration tool, GENy. You should not modify these files by hand.

Component/File	Description
fbl_apfb.c/.h	Contains the Flash-Block Table. The table defines the regions of memory that may be programmed by the bootloader.
fbl_cfg.h	Bootloader feature switches and parameters.
fbl_mtab.c	Logical Block table configuration file. Contains the Application module address range configuration.
ftp_cfg.h	Transport Layer configuration switches and parameters
v_cfg.h	Hardware and compiler specific switches and parameters.
v_inc.h	Include file for including all version-tracking include files. Not needed to compile FBL.
v_par	Version information of the Generation Tool (GENy). Not needed to compile FBL.
WrapNv_cfg	Contains macros for accessing non-volatile data.

Table 3-3 Generated Files

The delivery also includes a demo sba-ticket “dummySba.c/.h” that can be used for test/development purpose together with the Demo.

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

## 3.2 Memory Layout

The Flash Bootloader and the Operating Software must share a number of data-structures. The address of these structures must be known to both the Operating Software and the FBL. Since the FBL cannot be changed once released, the starting location in memory of the shared data must not change after release. The figure to the right shows one possible arrangement of the software elements that may be stored in non-volatile memory. Except as noted below, the actual order and location of the elements is unimportant.

### 3.2.1 The GM File-Container

Every module downloaded by the FBL must contain a File-Container (known as a data structure in [2]). The file containers (also referred to as file-headers in this doc) are used to identify the type of the module, and may contain information about where data in the module is written to. The container format will depend on the type of the module. A single module may have 1 or more container layers (known as envelopes in [2]). A complete description of the containers may be found in “Programmable Data Files” (Chapter 10 in reference [2]).

#### 3.2.1.1 The Operating S/W File-Container

Currently the bootloader supports an operating S/W download as a signed application S/W file. In the future, the bootloader will support a signed and compressed application S/W file. Further information can be found in “Supported Data Files” (section 10.2 of [2]).

The operating S/W file container contains information on the location of where to store the operating S/W into flash memory as well as information on the calibration partitions (see section 3.2.4 Partitions).

The Module-ID (MID) field of the header must correspond to one of the values reserved for Operating Software (0x0001, 0x0021, 0x0031, or 0x0041).

#### 3.2.1.2 Calibration Module File-Container

Currently the bootloader supports a calibration download as a signed calibration file. In the future, the bootloader will support a signed and compressed calibration file. Further information can be found in “Supported Data Files” (section 10.2 of [2]).

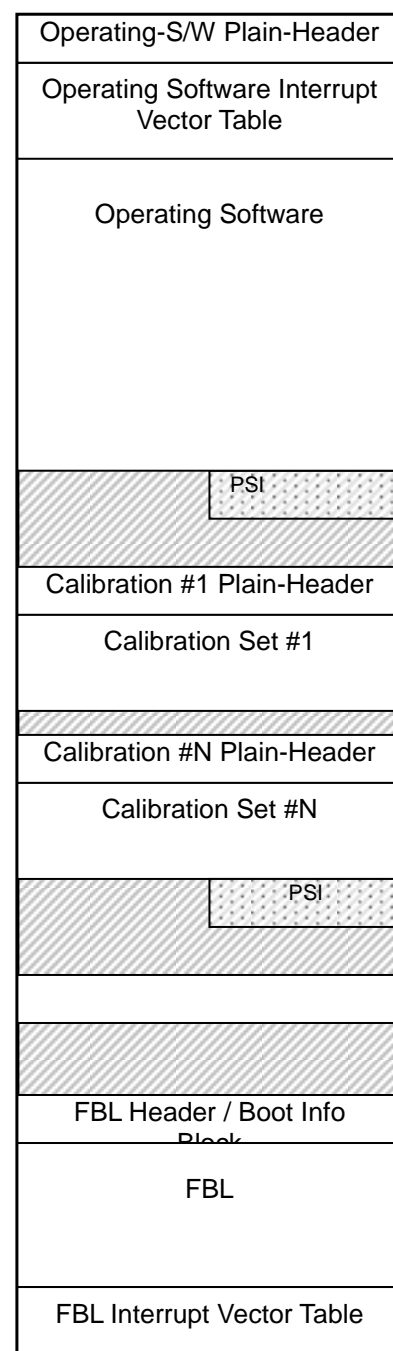


Figure 3-2 Example Memory Layout

The Module-ID (MID) field must correspond to the range of values associated with the Operating Software module that determines where the Calibration module is located. For example, use 0x0002 – 0x0014 for modules associated with Operating Software MID 0x0001. The Calibration MIDs must be in sequential order, starting with the value following the Operating Software's MID. For example, the first Calibration module associated with Operating Software MID 0x0021 must be 0x0022. In this example, additional Calibration MIDs are numbered 0x0023, 0x0024, etc.

### 3.2.1.3 Boot Info Block

The Boot Info Block is stored in the protected memory of the bootloader ROM. The values stored are configured as described in the table below.

Parameter	Configuration
Security Public Key	Input in GENy. This key is used to validate the root signature of the signer info.
Subject Name	Input in GENy. Hexadecimal value that identifies the group of ECUs for which the signer info is applicable.
ECU Name	Input in GENy. ASCII representation of the ECU name.
BCID	Input in GENy. Bootloader compatibility identifier. This is used to establish if the operating software is compatible with the bootloader.
Application Space	Defined by logical block configuration in GENy. A single logical block defines the combined Application and Calibration space of a single operating software and all of its corresponding calibration modules.
Calibration Space	Defined by logical block configuration in GENy. A single logical block defines the combined Application and Calibration space of a single operating software and all of its corresponding calibration modules.
DLS	Input in GENy. Design Level Suffix. Each bootloader software revision shall have a unique DLS.
Hex Part Number	Input in GENy. Bootloader software part number represented in hexadecimal.
ASCII Part Number	ASCII representation of the part number.

Table 3-4 Boot Info Block configuration.

The Module-ID (MID) field should be set to 0x0047.

If the FBL has been configured with the ROM Integrity check enabled, you will need to calculate the value of the CheckSum (CS) field (e.g. using hexview). The tool should read the FBL container file and update the NOAR, PMA/NOB field(s), and the CS field. The algorithm used must match that implemented in the FBL (see `ApplFblRomIntegrityCheck ()`).

### 3.2.2 Operating S/W Interrupt Vector Table

The Operating Software Interrupt Vector Table is a mirror for the ECU's actual vector table. The FBL distribution usually contains two tables, contained in the files `fbl_applvect.c` and `applvect.c`.<sup>1</sup> The first file, `fbl_applvect.c`, is linked with the FBL, and is used to support ECU sleep/wakeup modes (when available). The latter file, `applvect.c`, is linked with the Operating S/W. This file must be tailored to contain jumps to the application's interrupt service routines (ISR's). The tables may be located anywhere within the segments used for the Operating Software, but the address must be the same for the Operating S/W and FBL (the Op. S/W table replaces the FBL's table). In most cases, the location is established by encapsulating the table in a named section (usually 'APPLVECT'). The linker command directives then define an absolute address for the section. When setting the address of the tables, the linker directives for both the FBL and Operating S/W must be edited.

### 3.2.3 Logical Block table

The Logical block table has to be configured in GENy to contain all modules programmed to flash. Calibration files are not configured separately; they belong to the corresponding application area configured. Multiple application files can be configured (if applicable).

**Note**

The Logical block table contains the Application Space and Calibration Space parameters of the Boot Info Block.

### 3.2.4 Partitions

The memory generally is split into 3 or more partitions (bootloader, operational S/W, and calibration partitions). The application and calibration partitions are determined by the contents of the application plain header. Each calibration module is placed into a calibration partition and a calibration partition may contain multiple calibration modules. See [2] for more information on partitions.

### 3.2.5 Presence Patterns / Programmed State Indicator

Multiple presence-patterns/PSIs are needed if multiple modules (Operating S/W and Calibration) must be downloaded to your ECU. The requirements found in "Programmed State Indicator (PSI)" (Section 12.5.10 of reference [2]) identify how to manage the PSI(s) (known as presence-patterns) (the presence-patterns shown in the memory-layout figure above is an example of patterns that comply with the requirements).

The presence-patterns are managed in the implementation of the functions `ApplFblValidateBlock()`, `ApplFblInvalidateBlock()`, and `ApplFblIsValidApp()` (all found in `fbl_apnv.c`). Please read section 5.10 ([#oem\_valid] – Proposals for Handling The Validation Area) very carefully for details of how presence-patterns are implemented in the FBL.

<sup>1</sup> In many deliveries, the vector tables are supplied as assembly-language source code. See also the FBL Release Notes for your delivery.

Downloaded modules are not allowed to overlay the regions containing the presence-patterns. The bootloader will return an error if this is detected.

### **3.3 Run-Time Details**

A general description of the order that application and hardware-specific routines are called in may be found in reference [4].

The following sections summarize when callback functions are used during the power-on reset sequence, start from Operating-Software, and Download sequences.

### 3.3.1 Power-On Reset

Upon reset, the FBL will execute the compiler-specific start-up code, followed by a call to `main()`. The sequence of steps performed by the FBL is shown below:

```
< main >
  < ApplFblInit() >  < ApplFblExtProgRequest() >-----Start-by-
Op/SW-->>-----|
  < ApplFblIsValidApp() >                                     |
  |                                     -----Op/SW not ready-->>-----+
  < ApplFblWDLong() >                                         |
  < FblTimerStopp() >                                         |
  < ApplFblStartApplication >                                 |
  | < JSR_APPL() >                                           /* Start Operating S/W */             |
  |                                                         |
  < ApplFblStartup(kStartupPreInit) >-----<<-----+ <
FblInit() >
  | < FblInitWatchdog() > /* Copy trigger routine to RAM */
  | < ApplFblWDInit() >
  | < ApplFblCanParamInit() >
  | < FblHardwareInit() > /* start timer, CAN, etc */
  | | < ApplTrcvrNormalMode() >
  | < FblDiagInit() >
  | | < ApplFblResetVfp() >
  | | < ApplFblInitErrStatus() >
  < ApplFblStartup(kStartupPostInit) >
  | < ApplFblRomIntegrityCheck() >
  | | < ApplFblVerifyChecksum() >
  | < ApplFblRamIntegrityCheck() >
  < FblRepeat() > /* Wait for requests */
```

### 3.3.2 Start from Operating Software

The Operating Software should start the bootloader by initializing the CAN-Init structure, and then using the macro `CallFblStart()` (found in `fbl_def.h`). Details of the CAN-Init structure may be found in “STEP 4:” of chapter 6. The start-up macro will retrieve the address of the start-function (`fbl_main.c::FblStart()`) from the FBL header, and invoke it. In turn, `FblStart()` will save the parameters provided by the Operating Software, and then set the “startup magic flags”. The startup magic flags may be used in `ApplFblExtProgRequest()` to determine if the bootloader is being started by the Operating Software. Finally, `FblStart()` will use the macro `APPL_FBL_RESET()` to call `ApplFblReset()`. The ECU should reset at this point, and execution continues as described in section 3.3.1.

The FBL determines if it is started by the Operating Software by calling the function `ApplFblExtProgRequest()`. If this function returns `kProgRequest`, then the application-validation call (`ApplFblIsValidApp()`) is skipped, and the mode-flag `START_FROM_APPL` is set. The macro `GetFblMode()` returns the FBL mode; the



returned value may be compared to `START_FROM_APPL` (a bitwise compare) to determine if the FBL was started by the Operating Software.

When the FBL is started by the Operating Software, the implementation of `ApplFblCanParamInit()` should not normally modify the CAN parameters. In this case, the initialization structure, `CanInitTable` should contain the necessary data (The table is initialized in `FblStart()` from data provided by the Operating Software). The FBL extracts this information in the function `FblCanParamInit()`, which is called immediately after `ApplFblCanParamInit()` if the FBL is started by the Operating System.

The FBL will execute one additional series of steps following the call to `FblDiagInit()`:

```
< FblInit() >
...
< FblDiagInit() >
< FblDiagInitStartFromAppl() >
| < ApplTrcvrHighSpeedMode() >
| < ApplFblCheckProgConditions() >
| < FblEnterProgrammingMode() >
| | < ApplFblSetVfp() >
| < FblEnterRequestDownload() >
| | < ApplFblRequestDownload() >
...
```

The addition of `FblDiagInitStartFromAppl()` handles all the initialization needed to prepare the FBL to receive Transfer-Data requests.

### 3.3.3 Download Sequence

To download a module, the download tool (such as vFlash) will send a sequence of diagnostics requests. The initial requests are sent to all ECUs on the CAN bus, so that all ECUs are operating at the same baud rate, and to ensure that the maximum bus bandwidth is available. Complete details of the sequence of requests may be found in “ECU Programming Process” (Section 9.4 of reference [1]). A summary of the diagnostics requests (in order of transmission) is shown below:

#### Requests sent to all ECUs

```
[ Read Data By Identifier ($1A) ]
[ Disable Normal Communication ($28) ]
[ Report Programmed State ($A2) ]
[ Read Data By Identifier ($1A) ]
[ Request Programming Mode ($A5) ]
[ Enable Programming Mode ($A5) ]
```

#### Requests sent only to target ECU

```
[ Request Security Seed ($27) ]
```

```
[ Send Security Key ($27) ]
[ Request Download ($34) ]
[ Transfer Data ($36) ]
[ Report Programmed State ($A2) ]
[ To all ECUs: Return To Normal Mode ($20) ]
```

Although not shown, the download tool is required to send Tester-Present (\$3E) requests periodically after the Disable-Normal-Communication (service \$28) request.

If the Operating Software is running, then it must handle all requests up to Request-Download (service \$34). When service \$34 is received, the Operating Software should invoke the FBL without sending a response (if necessary, it may send the response-pending response). The FBL will send the response after it has completely initialized.

Associated with each diagnostic request (except Disable-Normal-Communication and Tester-Present) are one or more callback functions. The following table identifies the functions used by each service handler (where more than one is called, the functions are listed in the order actually used by the FBL). Complete details of each function may be found in section 5.

Diagnostic Service	Callback Functions
Read Data By Identifier (\$1A)	ApplFblReadDataByIdentifier
Disable Normal Communication (\$28)	-
Report Programmed State (\$A2)	ApplFblReportProgrammedState
Programming Mode (\$A5)	ApplFblRequestPrgMode ; For subfunction \$01 and \$02 ApplTrcvrHighSpeedMode ; For subfunction \$03 if \$02 ; previously requested ApplFblSetVfp ; For subfunction \$03 ApplFblEnablePrgMode ; For subfunction \$03
Security Access (\$27)	ApplFblSecuritySeed ;For subfunction \$01 ApplFblSecurityKey ;For subfunction \$02 ApplFblSecurityAccess ;For all other subfunctions ApplFblCheckMecAndVFlag
Request Download (\$34)	ApplFblCheckDataFormatIdentifier ApplFblRequestDownload
Transfer Data (\$36)	ApplFblTransferData ApplFblInvalidateBlock ; after request-download ApplFblFillGaps ApplFblValidateBlock
Return To Normal Mode (\$20)	ApplFblResetVfp ApplTrcvrNormalMode ApplFblWDLONG ; If in programming-mode

	ApplFblReset	; If in programming-mode
--	--------------	--------------------------

Table 3-5 Callback functions used by diagnostic service handlers

## 4 Configuration of the GM Flash Bootloader

### 4.1 Overview

The Fbl is configured using the Generation tool GENy.

A complete description of GENy is beyond the scope of this document. The sections that follow provide a quick tutorial of GENy and details that are specific to the GM Bootloader. Please refer to the on-line help in GENy for complete details.

An example configuration is included with the demonstration FBL included with your delivery.

### 4.2 Starting with GENy

When installing GENy, a link to GENy is added to your PC's Start menu (by default, a project-specific link to start GENy is in Programs/Vector/CANfbl/<project>).

When started, the following window will appear:

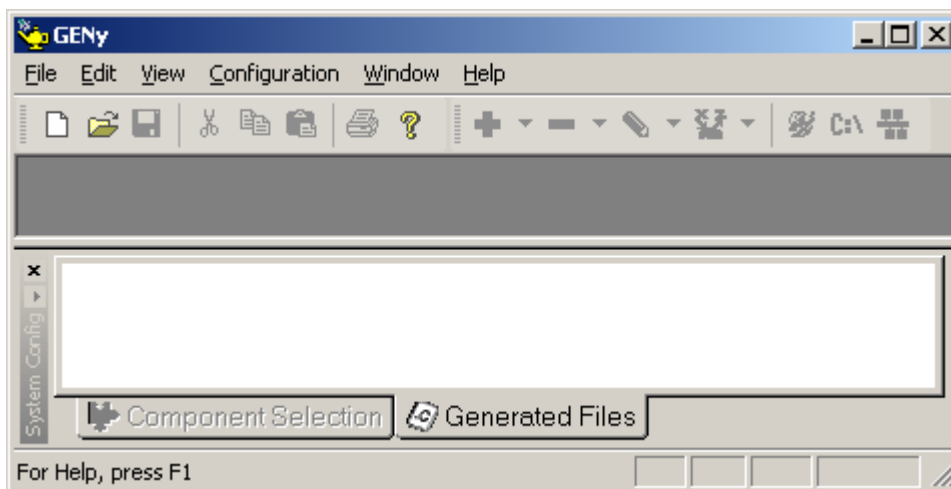


Figure 4-1 Initial GENy main window

To create a new configuration, you should select **New...** from the File menu. The following dialog will appear:

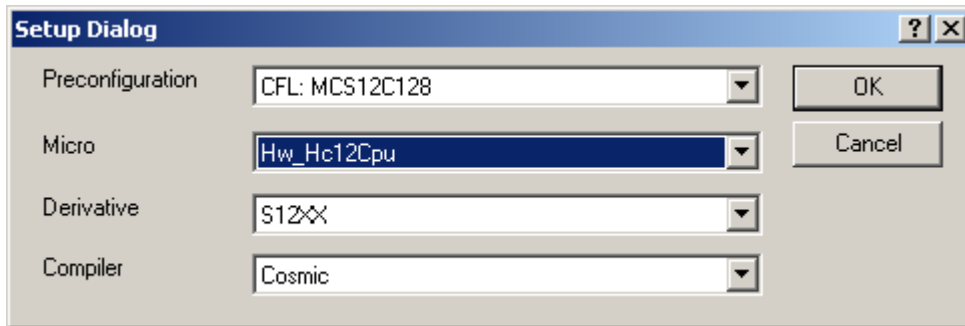


Figure 4-2 GENy Setup Dialog

The actual field values will vary, depending on your delivery license and hardware. In most cases, the default values will be appropriate, and you need only click on **[OK]** to proceed.

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

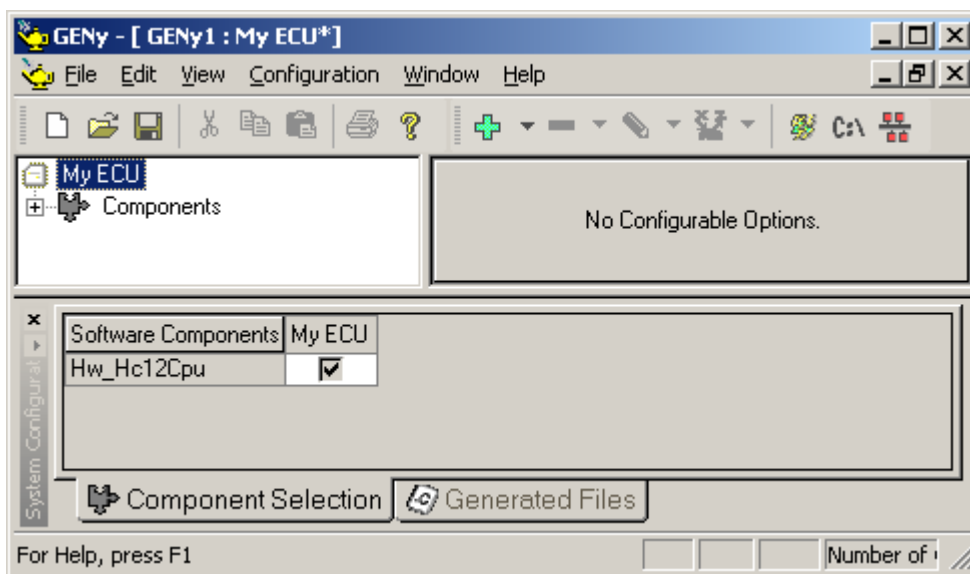


Figure 4-3 GENy main window after pre-configuration

The next step is to define a channel. You may select **Add Channel** from the **Configuration** menu, or you may click on the Plus icon from the tool bar. A dialog like that shown below will appear.

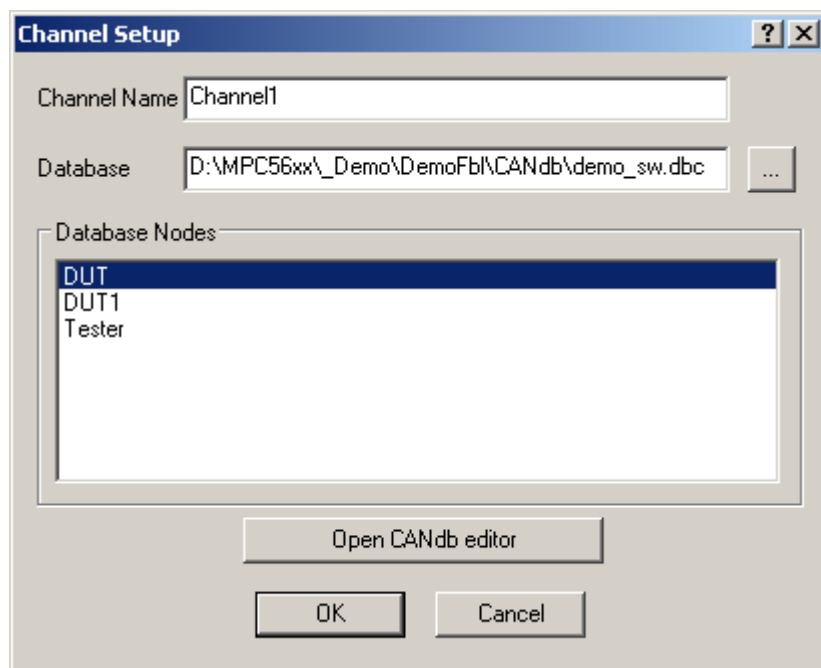


Figure 4-4 GENy Channel Setup

The first step is to select the database (.DBC file) provided by your OEM (e.g. General Motors). Once the database has been selected, a list of the ECU Nodes defined by the database will appear in the **Database Nodes** field.

You should select the node that your FBL will be running in. Multiple nodes may be selected if your ECU is used multiple times in the same vehicle. This enables the Multiple-Identity-Module (MIM) feature of the FBL. See also section 9.2.

When you click on **[OK]**, the GENy main window will appear as shown below:

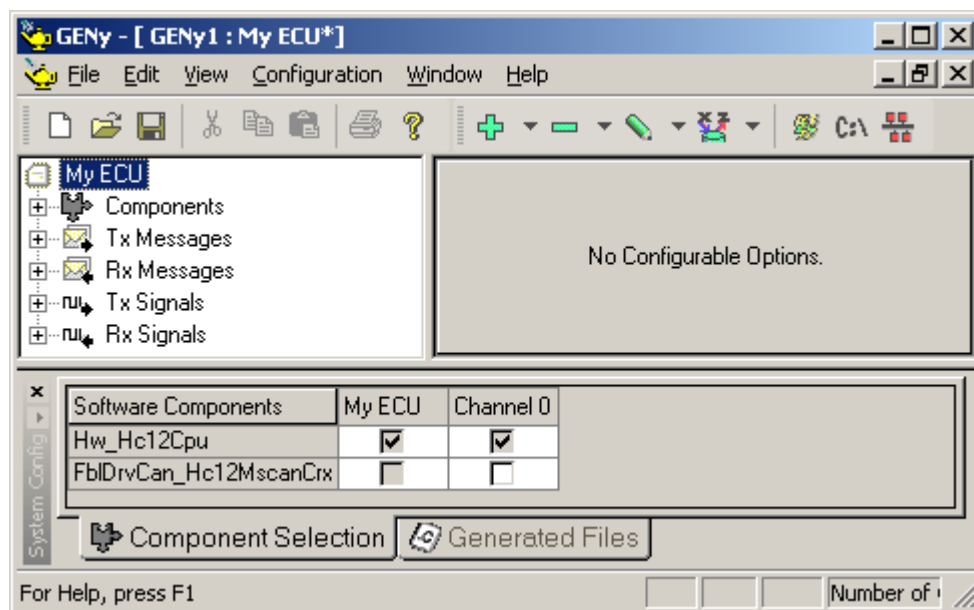


Figure 4-5 GENy main window after channel setup

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. You must add the FblDrvCan\_<hardware>, FblTpIso and FblCan\_14230\_GM components.

After the Software Components have been selected, you should expand the list-tree in the left-most 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. The expanded tree is shown below:

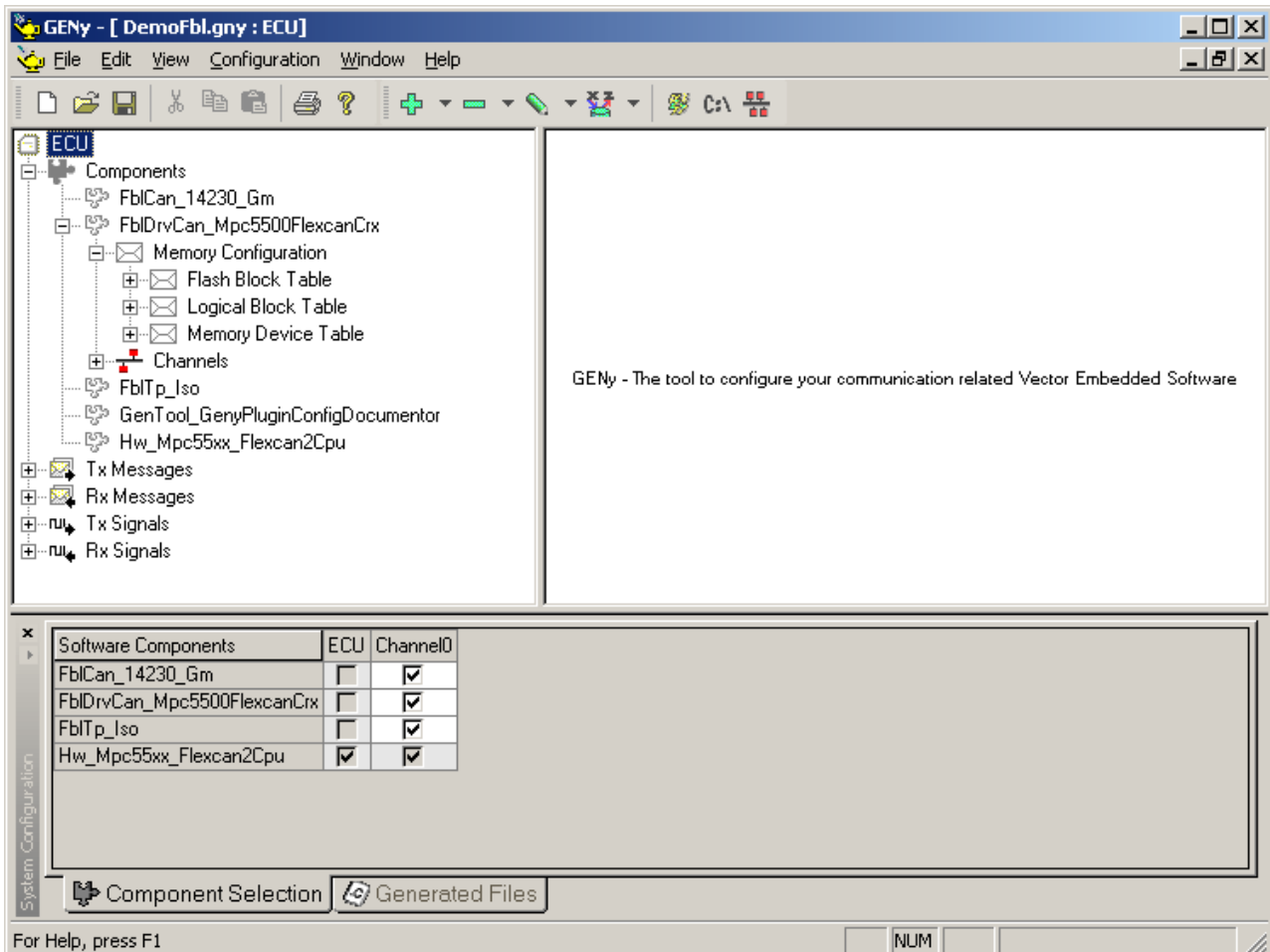


Figure 4-6 GENy Components

Before proceeding to component configuration, you should save the configuration. Select **Save** from the File Menu, or press the floppy-disk icon on the tool bar. GENy will display a standard file-selection dialog box, with the default path set to the directory the database is located in. You may change the path and set the filename as desired. GENy will create a file with a .GNy extension. The folder the file is saved in will be used as the Project Directory.

Once the Project Directory has been established, you should select **Generation Paths...** from the Configuration menu. GENy will display the dialog shown below:

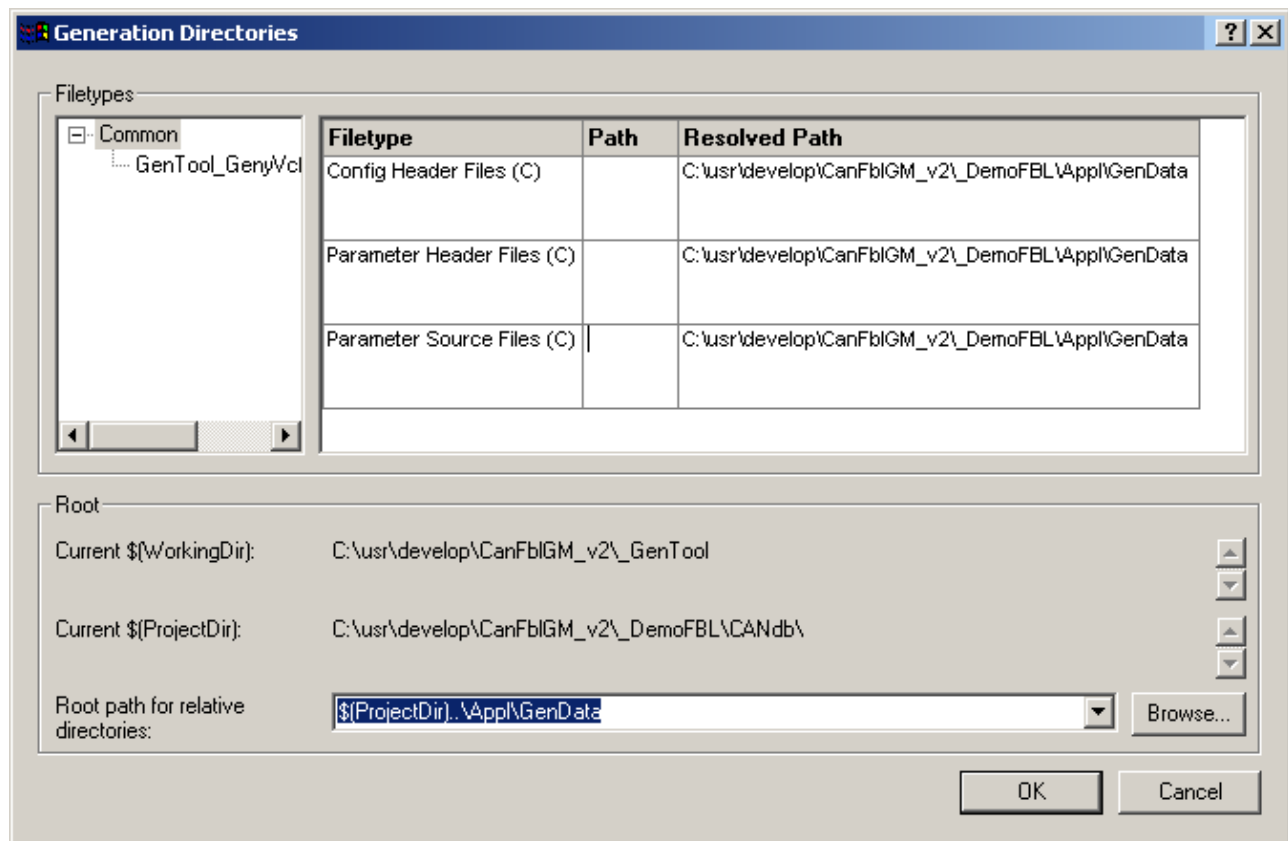


Figure 4-7 GENy directory selection

By default, GENy will place the files it generates into the Project Directory. You may enter a new path relative to the Project Directory (as shown above), enter a path relative to the GENy's working-directory (the directory GENy is executed from), or you may enter an absolute path.

After pressing **[OK]**, you are ready to configure the individual components.

### 4.3 CAN Configuration

The CAN controller is configured by expanding the **Channels** component, and selecting the **Channel 0** component from the list-tree. The contents of the window that appears on the right will be highly dependent on the hardware that has been selected. An example of the CAN configuration window is shown below:



#### Info

Note that the GM Bootloader supports only one channel.



Configurable Options		Channel 0
Type of bussystem	CAN	
Manufacturer	GM	
242A0CD9-284B-4e9a-BA17-D19EF24CA6E8	*	
+ FBL		
- Initialization		
- Init Structures	Add	
- Init Structure	Delete	
Module Control Register 0	0x4	
Module Control Register 1	0x84	
Bus Timing Register 0	0x7b	
Bus Timing Register 1	0x14	
Receiver Interrupt Enable Register	0xc5	
Identifier Acceptance Control Register	0x20	
Acceptance Filter Configuration	...	
Bustiming Configuration	...	
- CAN Controller (HC12)		
Register block offset	0x180	

Figure 4-8 GENy CAN Configuration

For details regarding this window for your hardware, please refer to the document **Technical Reference\_<Hardware>** provided with your delivery.

By default, GENy initializes the CAN hardware as dual-wire (500.0 KBPS). Use the bus timing Configuration dialog to select the appropriate baud-rate.

#### 4.3.1 Single wire ECU (Body bus)

If the ECU is connected to a single-wire (Body-Bus) CAN bus, the correct baud rate should be 33.333 KBPS. In addition, a second Init Structure is required. Press the **[Add]** button, and select the bus timing Configuration button [...] for the second structure. The baud-rate for the second structure should be set to 83.333 KBPS.

The FBL will accept the Request-Programming-Mode-High-Speed sub-function in Programming-Mode (service \$A5) requests in this configuration. The FBL will call `ApplTrcvrHighSpeedMode()` upon receipt of the high-speed programming-mode request; the same callback is also called when started by the Operating Software if high-speed programming is requested.

## 4.4 Bootloader Configuration

The features of the bootloader are configured in two separate components: `FblDrvCan_<hardware>` and `FblCan_14230_GM` (see Figure 4-6). Features specific to the component is shown in the right-hand side of the window when the component is selected in the left-hand side of the window.

#### 4.4.1 FblDrvCan component

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>**.

Configurable Options	FblDrvCan_Mpc5500FlexcanCrx
[- FblDrvCan_Mpc5500FlexcanCrx	
Flash code buffer size (Byte)	0x400*
Watchdog function size	0x200*
<i>Additional controller specific configuration may be available..</i>	
[- FBL	
User Config File	D:\usr\usage\Delivery\CBD13x\CBD1300343\B01\external\Demo\...
Project State	Integration
Stay in Boot	<input type="checkbox"/> *
Maximum Number of Segments	10*
Sleep Mode	<input type="checkbox"/> *
Application Task	<input checked="" type="checkbox"/> *
Bootloader Header Address	0x400*
Diagnostic Buffer Size [B]	4095*
Fill Code	0xc3*
Internal Memory Copy	<input checked="" type="checkbox"/> *
FblStart Function	<input checked="" type="checkbox"/> *
[- Download	
Download	...
[- General Timer Algorithm	
Timer Clock [kHz]	64000*
[- Download Handling	
Data processing buffer size [B]	512*
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 4-9 FblDrvCan configuration example

Each configuration field of FblDrvCan component is described below:

Field Name	Description
Flash Code Buffer Size (Byte)	This configuration is found in the hw specific configuration of every hw platform available. Configure the size of the array to hold the downloaded flashdrv. Recommendation is to add a surplus of 20% or more to allow for larger flashdrv through updates in future.

Watchdog function size	Array size reserved to hold code for the watchdog trigger functionality (FblLookForWatchdog() and ApplFblWDTrigger() ). This configuration may not be offered for your compiler dependent configuration if a linker based approach to copy watchdog code is used. Check <b>Technical Reference_&lt;Hardware&gt;</b> for details.
User Config File	The path and name of a file to be included in the generated configuration file fbl_cfg.h may be specified. The file may be used to activate features of the FBL that are not included in the GENy components. Normally, this field is blank.
Project State	<p>When configured to “Integration”, the bootloader helps you find configuration errors and sends out useful information in case of errors (see 9.3). This setting is recommended when starting a configuration or when having issues in the bootloader to support you in finding the root cause. Please use “Integration” when you send us a communication log as it contains useful information.</p> <p>Please make sure you select “Production” when you do not need any Integration support any more, at the latest when you prepare for testing/production. If you do not change to “Production” the bootloader will continue send out data unexpected by GM on 1A 7F service.</p> <p>See details on this configuration in GENy Onscreen Help box.</p>
Stay in Boot	Development feature: Allow forcing bootmode. This is done by checking for a defined message during startup within a given time window. For details compare [#oem_time] – Stay-In-Boot mode, 9.4
Maximum number of Segments	Maximum number of address regions allowed in any download module.
Sleep mode	<p>When selected, the FBL will use the CAN-controller’s wake-up interrupt to start the ECU after entering sleep-mode. Not all CAN-controllers support this feature.</p> <p>See the descriptions for  <code>ApplFblSleepModeAllowed()</code>, <code>ApplTrcvrSleepMode()</code>,  <code>ApplFblEnterStopMode()</code>, and <code>ApplFblCanWakeUp()</code>.</p>
Application task	When selected, the FBL will periodically call <code>ApplFblTask()</code> . This function may be customized by you to implement any background operations.
Bootloader Header address	The logical address of the File-Header for the bootloader must be entered here. The value <b>must</b> match the address used by your linker to map the header to its proper location (in most cases, the mapping is resolved by the address associated with a named section, usually FBLSTART).
Diag 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
Fill code	<p>If the Gap-Fill feature is enabled, the FBL will also use this value to fill all unused regions of memory. The value must be between 0x00 and 0xFF (multi-byte fill pattern is not supported).</p> <p>When writing data to a non-volatile 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</p>

	will fill the unused bytes with the value specified in this field.
FblStart Function	<p>This allows the user to disable standard functionality provided to transition from application to boot mode. Disabling is only required if you want to save code size.</p> <p>The switch can e.g. be disabled if an Eeprom is used instead of a Ram pattern, or if the application already prepares the ram pattern and copies all shared Ram required by the bootloader to transition.</p>
Watchdog Enable	<p>When enabled, the FBL will use the functions <code>ApplFblWDInit()</code>, <code>ApplFblWDLONG()</code>, and <code>ApplFblWDTrigger()</code>, to manage the ECU reset-timer.</p> <p>If not selected, the FBL will assume that the reset timer is not used.</p>
Watchdog time	<p>This specifies the interval, in milliseconds, between calls to the watchdog trigger function, <code>ApplFblWDTrigger()</code>. The value only has meaning if the Watchdog Enable switch is selected. The value must be less-than the time-out period of the watchdog. The value must be in the range 1 – 65535 and must be an integer multiple of <code>FBL_REPEAT_CALL_CYCLE</code>.</p> <p>Note: Earlier versions of the FBL are limited to a 255 ms watchdog trigger period.</p>

Table 4-1 Bootloader Configuration

#### 4.4.2 FblCan\_14230\_Gm configuration

The following tables describe the FblCan\_14230\_GM specific switches that can be configured. Two tables and screenshots separate

- GM Fbl configuration Settings and GM Diagnostic service options and
- GM Modules configuration

Configurable Options		FblCan_14230_Gm
[- GM FBL configuration Settings		
ECU SPS Type	TYPE_B	
RAM Integrity Check	<input type="checkbox"/>	
RAM Integrity Check Start Address	0x0*	
RAM Integrity Check End Address	0x0*	
ROM Integrity Check	<input type="checkbox"/>	
Gap Fill	<input checked="" type="checkbox"/> *	
Reset delay	30*	
[- NodeTable		Add
[- NodeTable Node		Delete
Node Name	DUT	
Node Address	0x45	
[- GM Diagnostic Service Options		
Initiate-Diagnostic-Operation	<input type="checkbox"/> *	
Read-Memory-By-Address	<input type="checkbox"/> *	
Write-Data-By-Identifier	<input type="checkbox"/> *	
Security-Access	<input checked="" type="checkbox"/>	
Read-Memory-By-Address Address Length	4 Bytes	
Request-Download Memory-Size Length	4 Bytes	
Transfer-Data Address Length	4 Bytes	
Seed and Key Size	5*	

Figure 4-10 GM Fbl configuration Settings and GM Diagnostic service options

Field Name	Description
ECU SPS Type	<p>Consult your CTS which type your ECU shall be.</p> <p>TYPE_B:</p> <p>The FBL will initialize with CAN-ID for diagnostic messages found in the database (permanent IDs), and start with responses enabled.</p> <p>TYPE_C:</p> <p>The FBL will initialize with CAN-IDs set based on the Diagnostics Address obtained from the database, and start with responses disabled.</p> <p>Note that any TYPE_A Ecu falls back to either TYPE_B or TYPE_C as long as it remains in boot mode (Operating Software and/or Calibration data is missing); in boot mode hence every ECU will be either TYPE_B or TYPE_C.</p> <p>Compare GMW3110 v1.6 Section 9.: ECU Programming Requirements and Process.</p> <p>Note that in case of the TYPE_B configuration, the FBL will invoke</p>


	<p>ApplFblCanParamInit() to do the TYPE_B initialization (FBL_ENABLE_CAN_CONFIGURATION switch).</p> <p>For additional information, see also "SPS service Types and Identifiers".</p>
Enable RAM Integrity Check	<p>When selected, the FBL may invoke ApplFblRamIntegrityCheck() from ApplFblStartup(). The function is responsible for verifying that Random-Access-Memory is functioning properly, and updating any flags used by ApplFblReportProgrammedState() to indicate the result.</p>
RAM Integrity Check Start Address	<p>Start address of RAM region to be checked. Checked Ram should include all RAM used by the bootloader (stack, global variables, flashCode buffer)</p> <p>Note: If several regions are required, only the first region can be configured in GENy, any further regions will have to be configured inside fbl_ap.c ApplFblRamIntegrityCheck() address table.</p>
RAM Integrity Check End Address	<p>End address of RAM region to be checked. Checked Ram should include all RAM used by the bootloader (stack, global variables, flashCode buffer)</p> <p>Note: If several regions are required, only the first region can be configured in GENy, any further regions will have to be configured inside fbl_ap.c ApplFblRamIntegrityCheck() address table.</p>
ROM Integrity Check	<p>The GM "Global Bootloader Specification" requires that the FBL perform an integrity check on FBL ROM, and report the result to the \$A2-Report ProgrammedState requests. When enabled, the FBL will calculate the checksum based on the values in its GM File Header. If disabled, the FBL will not perform the ROM check.</p> <p>Note :</p> <p>The check region will have to be configured inside GENy "GM Bootloader Header" address region. A single region is specified. If an additional region is required, it will have to be manually added inside fbl_ap.c ApplFblRomIntegrityCheck() to the segment list.</p>
Gap Fill	<p>_____</p> <p>_____</p> <p>If enabled, the FBL will fill all unused regions of the non-volatile device(s) with "Fill Code" specified in FblDrvCan_XX configuration. The fill happens at the end of the download. The drop box below allows you to configure detailed behavior for this selection.</p> <p>If disabled, unused regions will contain the value resulting from the device's erase operation.</p> <p>Compare [2] section 7.5, Unused Flash Space: "GM shall determine with the ECU supplier if the Pad bytes will be programmed and what the Pad and Fill bytes value should be."</p>

	<p>Note that there is the possibility to disable this feature in order to reduce code size and fill your download container with the intended fill value instead.</p> <p>There is also the option to disable this feature and implement a custom gap fill algorithm in the function <code>ApplFblFillGaps</code>. This may allow to reduce code size and decrease execution time of the built in gap fill function. See also section 5.3 and 9.9.</p>
Reset Delay	<p>Configuration relevant for Single Wire ECUs only.</p> <p>If you want to reduce reset delay time, measure your applications reset to communication (first message on the bus) timing and reduce this value by the time measured. (e.g. reset to communication in 20ms at minimum: configure 10; reset to communication in 40ms at minimum: configure 0 )</p> <p>Note: A default value of 30 (ms) will always configure your ECU correctly but may lead to unwanted, unnecessary delay.</p> <p>Background (GMW3110 v.1.6, 8.5 ReturnToNormalMode (\$20) Service ): Ecus shall not do a transition from \$20 ReturnToNormal mode to normal communication in less than 30ms. This is in order to prevent communication starts while bus participants have different baud rates activated. For fast initializing ECUs the reset shall be delayed the required time to allow this condition to be fulfilled.</p>
NodeName	<p>This non-editable field identifies the name of the node to be configured. If multiple nodes were selected in the Channel Setup window, there will be multiple Node-Name and Node-Address pairs displayed in GENy.</p>
Node Address	<p>This field should be set to the value of the node's Diagnostic Node Address. The correct value is specified in the CTS document provided by GM, and is often defined in your project database (dbc file).</p> <p>The value of this field will determine the CAN message identifiers used for diagnostic requests and responses if "Enable CAN Configuration" is not selected.</p> <p>Two fields are available in a multiple node configuration.</p>
Enable Initiate-Diagnostic-Operation	<p>When selected, the FBL will accept Initiate-Diagnostic-Operation (service \$10) requests. When a request is received, the FBL will call the function <code>ApplFblInitiateDiagnosticOperation()</code>.</p> <p>If not selected, the FBL will respond with a negative response indicating that the service is not supported.</p>
Enable Read-Memory-By-Address	<p>When selected, the FBL will accept Read-Memory-By-Address (service \$23) requests. When a request is received, the FBL will call the function <code>ApplFblReadMemoryByAddress()</code>.</p> <p>If not selected, the FBL will respond with a negative response indicating that the service is not supported.</p>



Enable Write-Data-By-Identifier	<p>When selected, the FBL will accept Write-Data-By-Identifier (service \$3B) requests. When a request is received, the FBL will call the function <code>ApplFblWriteDataByIdentifier()</code>.</p> <p>If not selected, the FBL will respond with a negative response indicating that the service is not supported.</p>
Enable Security-Access	<p>When selected, the FBL will accept Security-Access (service \$27) requests. If a request containing the SPS-Request-Seed (\$01) subfunction is received, the FBL will call <code>ApplFblSecuritySeed()</code>. If the request contains the SPS-Send-Key subfunction, the FBL will call <code>ApplFblSecurityKey()</code>. The FBL will call <code>ApplFblSecurityAccess()</code> if any other subfunction is requested.</p> <p>The FBL will also maintain the security locked/unlocked state, verify that a security-seed request precedes a security-key request, and manage the bad-key counter and delay timer.</p> <p>If not selected, the FBL will respond to Security-Access requests with a negative response, indicating that the service is not supported. The FBL will be in an unlocked state with respect to secured services (such as Request-Download).</p>
Read-Memory-By-Address Address Length	<p>“ReadMemoryByAddress (\$23) Service” (Section 10.7 in reference [1]) specifies that the memory-address parameter may be 2, 3, or 4 bytes long.</p> <p>The value of this field configures the FBL to accept the appropriate address parameter width.</p> <p>Note that this is relevant only if “Enable Read-Memory-By-Address” is selected.</p>
Request-Download Memory-Size Length	<p>“RequestDownload (\$34) Service” (Section 10.12 in reference [1]) specifies that the uncompressed-memory-size parameter may be 2, 3, or 4 bytes long.</p> <p>The value of this field configures the FBL to accept the appropriate width of the memory size parameter.</p> <p>Note that this needs to be configured to the same size in the flash tool.</p>
Transfer-Data Address Length	<p>“TransferData (\$36) Service” (Section 10.13 in reference [1]) specifies that the starting-address parameter may be 2, 3, or 4 bytes long.</p> <p>The value of this field configures the FBL to accept the appropriate address parameter width.</p> <p>Note that this needs to be configured to the same size in the flash tool.</p>
Seed and Key Size	<p>Number of seed/key bytes used by the ECU. Normally 2 or 5, depending on requirements from GM.</p>

Table 4-2 GM-Specific Configuration

	<p><b>Note</b></p> <p>The “RAM Integrity Check” requires that the check be performed after I/O and CAN initialization has been completed. However, <code>ApplFblRamIntegrityCheck()</code> (via <code>ApplFblStartup()</code>) is called before the CAN initialization (I/O initialization requirement is satisfied if performed in <code>ApplFblInit()</code> or in <code>ApplFblStartup()</code>)</p>
---	---




before the check).

### Gm Modules configuration:

[- GM Modules configuration	
[- GM Download Modules	
Calibration module support	<input checked="" type="checkbox"/> *
Multiple application modules support	<input type="checkbox"/> *
[- Boot Info Block Parameters	
Checksum	0x1812
FBL Module ID	0x47*
FBL SWMI	0x11223344*
DLS	0x0*
DCID	0x8000*
Ecu Name	DemoFbl1
Ecu Id (Byte 1 to 4)	0x0*
Ecu Id (Byte 5 to 8)	0x0*
Ecu Id (Byte 9 to 12)	0x0*
Ecu Id (Byte 13 to 16)	0x1*
Subject Name	Engine*
[- FblCan_GM_PMAAddressLengthPool	<input type="button" value="Add"/>
[- FblCan_GM_PMAAddressLength	<input type="button" value="Delete"/>
PMAAddress	0x400
PMA Length	0x10

Figure 4-11 GM Modules and Boot Info Block configuration

Field Name	Description
Calibration module support	Can be disabled if no calibration files are required in order to reduce code size.
Multiple application module support	<div>  <div> <b>Caution</b>  Multiple applications are not supported for this release. </div> </div> <p>Bootloader supports multiple application module downloads if enabled. Disable this option in order to reduce code size.</p> <p>Check chapter 9.2 for further configuration aspects.</p>

Boot Info Block Parameters	
Checksum	<p>This field defines the compiled-in value of the Checksum (CS) field for the bootloader's File-Header.</p> <p>It is likely that the required value of this field will be different each time the FBL is changed. You will probably need to develop software tools that may be run after the FBL is compiled that will update the File-Header fields.</p>
FBL Module ID	<p>This field defines the compiled-in value of the Module-Id (MID) field for the bootloader's File-Header.</p> <p>Per "Module ID Value" (Section 10.4.14.1 in reference [2]), the value of this field should always be 0x47.</p>
FBL SWMI (HexPart Number, Boot Info Block Parameter)	<p>This field defines the compiled-in value of the Software Module Identifier (SWMI) field for the bootloader's File-Header.</p> <p>Per "Boot Info Block" (Section 6.10 in reference [2]), the value of this field shall be your ECUs bootloader part number coded as a 4-byte hexadecimal number. The number is normally determined by agreement between you and General Motors.</p>
DLS (DLS Boot Info Block, parameter)	<p>This field defines the compiled-in value of the Design-Level-Suffix (DLS) field (also known as the Alpha Code) for the bootloader's File-Header.</p> <p>Per "Boot Info Block" (Section 6.10 in reference [2]), the value of this field shall contain two ASCII characters.</p>
DCID (BCID Boot Info Block, parameter)	<p>This field defines the compiled-in value of the Data Compatibility Identifier (DCID) field for the bootloader's File-Header.</p> <p>The value is a two-byte number that may be compared to the DCID field in the Operating Software's File-Header. The value determines that the Operating Software and bootloader interfaces are compatible.</p> <p>The check is not performed if the value of this field is 0xFFFF.</p> <p>The legal range for this field is 0x8000 – 0xFFFF.</p> <p>For additional information, see "Boot Info Block" (Section 6.10 in reference [2]). Note that reference [2] often refers to this field as either BCID (Bootloader Compatibility ID) or CCID (Calibration Compatibility ID).</p>
ECU Name	ECU name in ASCII. Maximum 8 byte length.
Ecu Id (Byte 1 to 4)	Byte 1 to 4 of the ECU ID. Each unique ECU will have a unique ECU ID. Value relevant for development, a process must be in place to override this value uniquely for each ECU.
Ecu Id (Byte 5 to 8)	Byte 5 to 8 of the ECU ID. Each unique ECU will have a unique ECU ID. Compare comment Byte 1-4
Ecu Id (Byte 9 to 12)	Byte 9 to 12 of the ECU ID. Each unique ECU will have a unique ECU ID. Compare comment Byte 1-4
Ecu Id (Byte 13 to 16)	Byte 13 to 16 of the ECU ID. Each unique ECU will have a unique ECU ID. Compare comment Byte 1-4
Subject Name	ECU family name as specified by GM. Input as ASCII. Maximum 16 byte length.
PMA Address	This field defines the logical address of the address-region used by the

	bootloader.
PMA Length	This field defines the length (in bytes) of the address-region occupied by the bootloader.

Table 4-3 GM Modules and Boot Info Block Detail Configuration

## 4.5 Memory Configuration

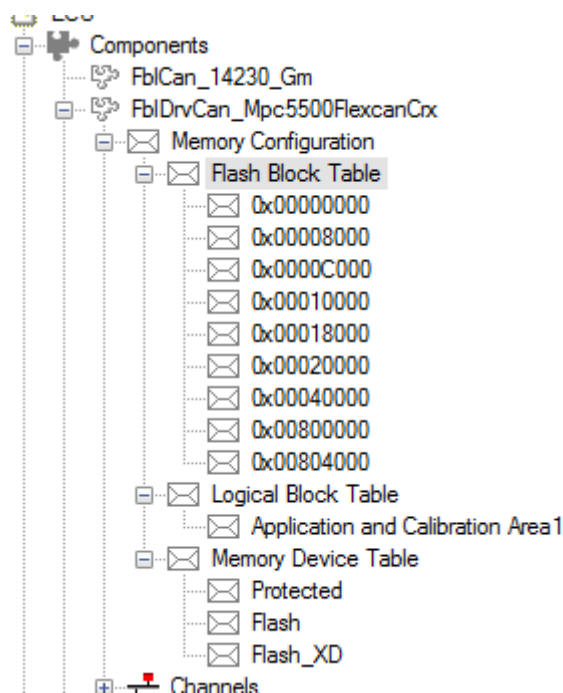


Figure 4-12 Memory Configuration

The Flash Block Table identifies the regions of memory that may be written to by the bootloader. The table also defines the addresses sent to the device-driver(s) when erasing memory.

The Logical Block Table Describes independently erased Module sets. These are

- ▶ Primary Operating Software (+ Calibration modules)<sup>2</sup>
- ▶ Secondary Operating Software (+ Calibration modules)<sup>3</sup>

The Logical Block table hence contains only one entry in standard use case: Primary Operating software (+ calibration files). It contains several entries if reserved modules or Multi-processor environment is required, also compare 9.2.

<sup>2</sup> Remember: All calibration files are erased upon application download; these are hence not independently erasable.

<sup>3</sup> Not supported in the current release.

The typical configuration is one logical block with optional calibration file memory reserved.

Application Header Offset (relative to Logical Block table start) and Application Presence Pattern offset (relative to application header) decide on the internal partition for application / calibration file areas:

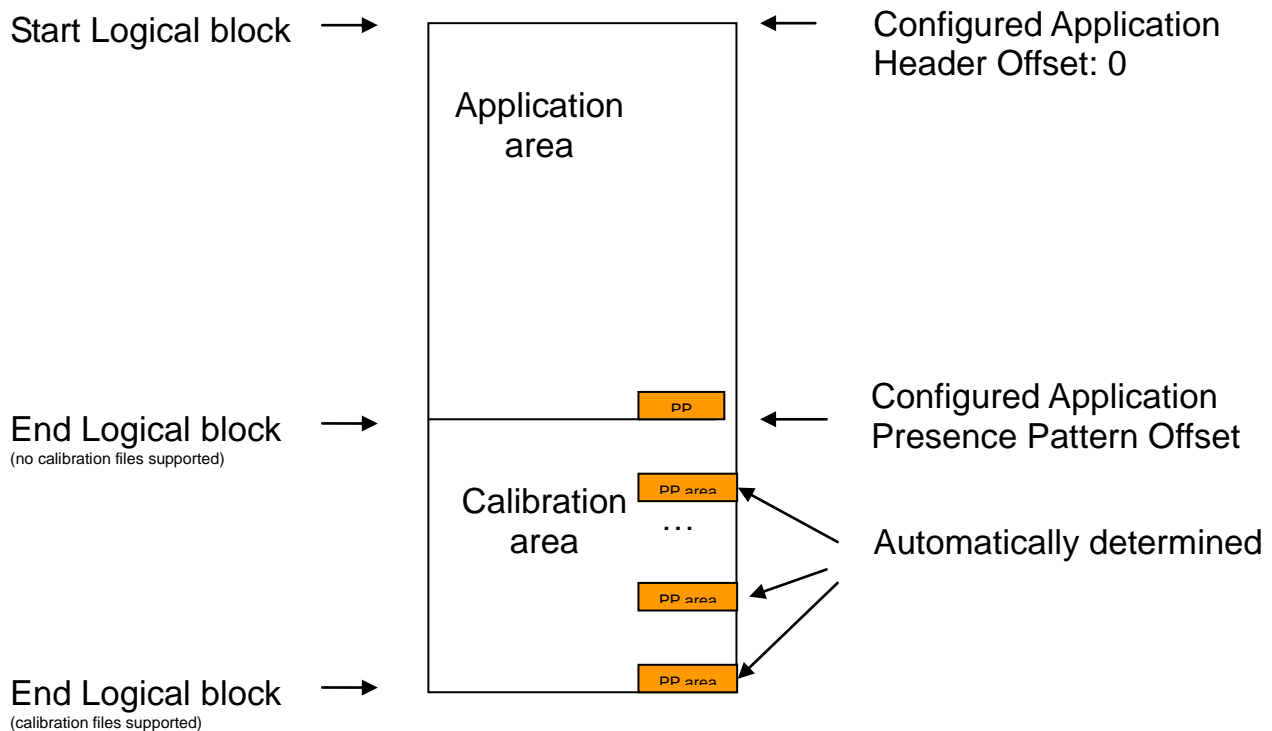


Figure 4-13 Typical Logical Block partition

#### 4.5.1 Device Types

The bootloader is capable of programming multiple non-volatile memory devices, such as Flash and EEPROM memory. GENy predefines one non-volatile memory type, Flash. If your ECU supports additional devices, then they should be defined before defining the flash-block table. Devices are added in the FbIDrvCan\_<XX>->Memory Configuration->Memory Device Table

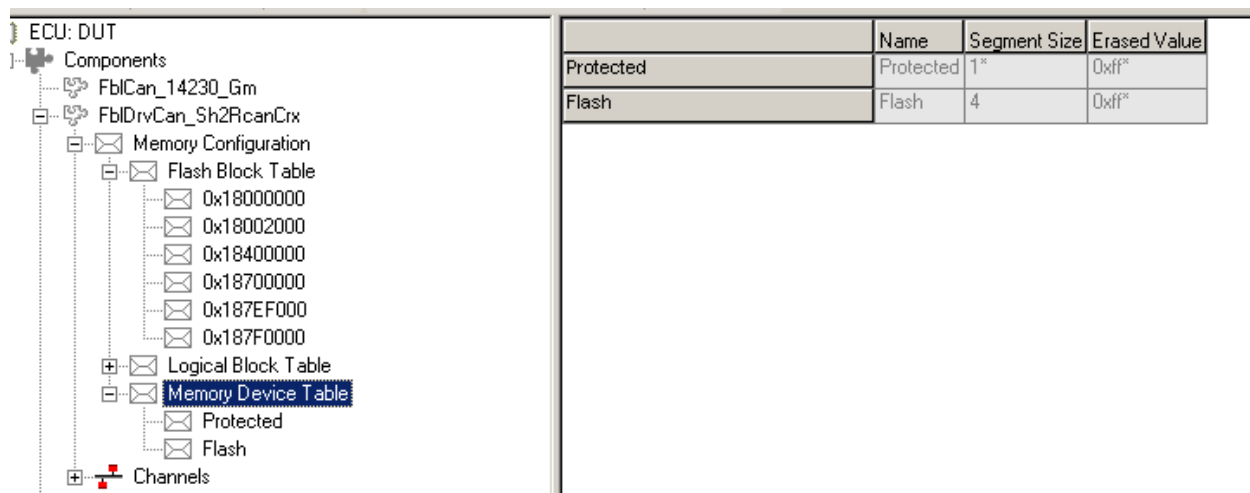


Figure 4-14 GENy Device Types

Two sub-fields will be available: **Device type** and **Segment Size**. For information regarding values for these fields, please refer to the hardware-specific documentation provided with your delivery (Technical Reference Hardware).

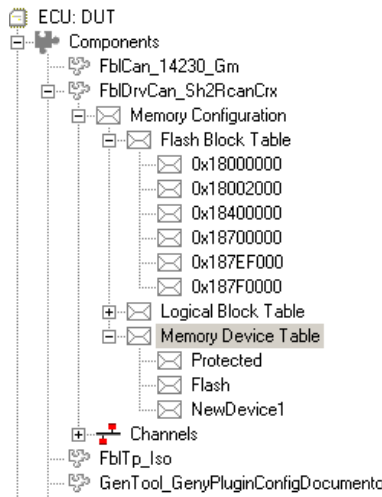
The **Device Type** field determines the name of the device, as it will appear in the flash-block table. The name is required to match the device-driver API. The API functions are declared in the device-driver I/O file (For example, fbl\_flio.h or eepIO.h). The file contains a prototypes section defining the API routines for the device. The function names will look like <prefix>Driver\_InitSync, <prefix>Driver\_DeinitSync, <prefix>Driver\_RReadSync, etc. Enter the function prefix into GENy exactly as typed in the header file.

**Info**

There is no need to add the Flash device to the device type table. This device-type is predefined by GENy.

The **Segment Size** field defines the minimum number of bytes that must be written to the device. You should enter the value appropriate for your hardware.

A completed example is shown below:

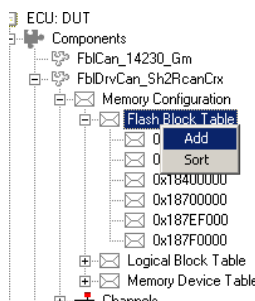


	Name	Segment Size	Erased Value
Protected	Protected	1*	0xff*
Flash	Flash	4	0xff*
NewDevice1	Eeprom	1*	0xff*

Figure 4-15 Example Device Type

## 4.5.2 Flash Block Definition

The Flash-Block table defines the regions of memory that may be written to by the bootloader. The table consists of 5 columns, and an arbitrary number of rows. Each row defines a block of memory on your ECU. An example is shown below:



	Start Address	End Address	Memory Device	Description	Logical Block
0x18000000	0x18000000	0x18001fff	Protected	Renesas Loader Program	*
0x18002000	0x18002000	0x183fffff	Flash	Application	Application and calibration
0x18400000	0x18400000	0x186fffff	Flash	Application	Application and calibration
0x18700000	0x18700000	0x187effff	Flash	Application	Application and calibration
0x187ef000	0x187ef000	0x187effff	Flash	4K Calibration area	Application and calibration
0x187f0000	0x187f0000	0x187fffff	Protected	CANFBI Vector BootLoader	*

Figure 4-16 GENy Flash Block Table

The **StartAddress** field defines the starting address of a block of memory.

The **EndAddress** field defines the ending address of a block of memory.

The **Memory Device** field identifies the device-driver responsible for reading, writing, and erasing the block of memory. The reserved value **Protected** defines regions that cannot be accessed by the bootloader. At a minimum, you should define the blocks that are occupied by the bootloader as protected.

The **Description** field allows you to enter text that may indicate the purpose of the block. The contents of this field are not passed to the generated files.

There are some constraints on the address boundaries defined by the block table:

1. A block must begin and end on an erase-sector boundary. Hence, a block may not be smaller in size than one sector.
2. A block may span multiple sectors of the device. However, some devices define boundaries that constrain write commands (addresses that cannot be crossed in a single write command). A block definition may not cross such boundaries.

3. If Calibration modules are to be downloaded, you must be careful to partition the blocks such that the calibration modules do not occupy any block occupied by the Operating Software.

The **Logical Block** field associates the FlashBlock to a configured Logical Block (“Logical Block Table” element). Be sure to add all Flashblocks you want to program for a given set of Application + calibration files to one single Logical Block as shown above in Figure 4-16 GENy Flash Block Table.

### 4.5.3 Logical Block Definition

At least one logical block is required to describe the programmable space of the main application and all of its calibrations.



#### Info

The logical blocks represent the Application Space and Calibration Space parameters of the Boot Info Block.

	Name	Block Index	Disposability	Start Address	End Address
Application and Calibration Area1	Application and Calibration Area1	0x1*	mandatory	0xc000	0x807fff

Header Address	Presence Pattern Address	Verification RAM	Verification ROM	Description
0x0*	0x0*	FblHdrPipelinedVerifyIntegrity*	FblHdrVerifyIntegrity	*

Figure 4-17 GENy Logical Block Table

Logical Block Configuration	
Name	Arbitrary name describing what the logical block represents. This is not generated into source code.
Block Index	Index of the logical block. Must be equivalent to the module ID of the operating software that it represents (i.e. 0x1 for main application, 0x15 (21) for second application, 0x1F (31) for third application, 0x29 (41) for fourth application).
Disposability	This field is not used.
Start Address	Start address of logical block. Inherited from the flash block table configuration.
End Address	End address of the logical block. Inherited from the flash block table configuration.
Header Address	Address of the plain header of the application module that uses this logical block. This must be within the address region of the logical block. This value is access by the software through ApplFblGetModuleHeaderAddress.
Presence Pattern Address	Start address of the presence pattern (PSI) location. This must be 2 times the flash segment size less than the end address of any flash block. This value is accessed by the software through ApplFblGetBaseModulePPRegion, which

Logical Block Configuration	
	verifies the above restriction in Project state Integration
Verification RAM	Verification function to verify the message digest. Normally this should not be changed. This should only be changed in case verification needs to be done on an external device (e.g. second microprocessor connected over SPI). In this case contact Vector.
Verification ROM	Verification to verify the integrity word (if enabled). Normally this should not be changed. This should only be changed in case verification needs to be done on an external device (e.g. second microprocessor connected over SPI). In this case contact Vector.

Table 4-4 GENy configuration of the Logical Block Table

Additional Logical Blocks need only be configured for Multi-processor configurations or when reserved modules are to be used (compare chapter 9.2 for how to add Logical block table entries).

Upon module download the bootloader will check the Logical block table defined region against the addresses found in the header. The header address of Application modules/Reserved modules has to be configured inside `AppIFblGetModuleHeaderAddress()`, which shall define an offset relative to either Logical Block area start / end address or an absolute address. This allows for later access to header information, e.g. for software module information for application and for calibration file location information.

## 4.6 Mandatory Delivery Preconfig

A user config file is required for this release. The file `MandatoryDeliveryPreconfig.cfg` is located in the delivery under `_Demo/DemoFbl/CANdb`. This file must be included in the GENy configuration as a user config file (see 4.4.1). This file must be configured manually using a source editor.

The following settings must be configured using this file:

```
#if defined ( FBL_ENABLE_VERIFY_INTEGRITY_WORD )
# define SEC_ENABLE_SECURITY_CLASS_VENDOR
#endif
#define SEC_ENABLE_WORKSPACE_INTERNAL
#define SEC_ENABLE_VERIFICATION_KEY_EXTERNAL
#define SEC_ENABLE_KEY_LOCATION_RAM
```



```
#define SEC_SIZE_CHECKSUM_VENDOR      2u
#define SEC_SECURITY_CLASS_VERIFY     SEC_CLASS_CCC
#define SEC_SECURITY_CLASS_VERIFICATION SEC_CLASS_DDD
```

**Note**

These settings will be configured through GENy in future releases.

The following settings are optionally configured using this file:

**Caution**

These macros are for test purposes only.

Macro	Description
FBL_TEST_SBA_TICKET	When this is defined the bootloader will write SBA ticket from flash (sbaBlk0) to NVM on startup.
FBL_TEST_ECU_ID	When this is defined the ECU ID stored in the bootloader header will be copied to NVM on startup.
FBL_TEST_KEY_NBID	When this is defined the bootloader will write 0x0000 key NBID into NVM on startup.
FBL_TEST_APP_NBID	When this is defined the bootloader will write 0x0000 app NBID into NVM on startup.
FBL_ENABLE_TRANSFER_VERIFICATION_CHECK	Optional feature: Required for Integrity Word verification feature. This feature is intended for development to verify the Plain header integrity word. This is not required by GM. Note: configuring this will cause the Bootloader reading the programmed data twice, therefore it is recommended only for development to verify header integrity word configuration.

## 4.7 NV-Wrapper Configuration

The bootloader accesses a set of non-volatile information in EEPROM or other non-volatile memory. Therefore, the NV-Wrapper is provided as an abstraction layer for non-volatile memory access. This permits the usage of different types of NV-memory with the same interface. The configuration of the NV-information handling is supported by GENy.

This section describes the specific settings for the GM Flash-Bootloader. For detailed information, please see [11].

All required NV-memory items are already preconfigured. What remains to be done is determine the address of the information block if an address-based NV-memory driver like an EEPROM-driver is used. If a handle-based driver like the EEPROM-Manager or FEE is used, the first handle has to be specified.

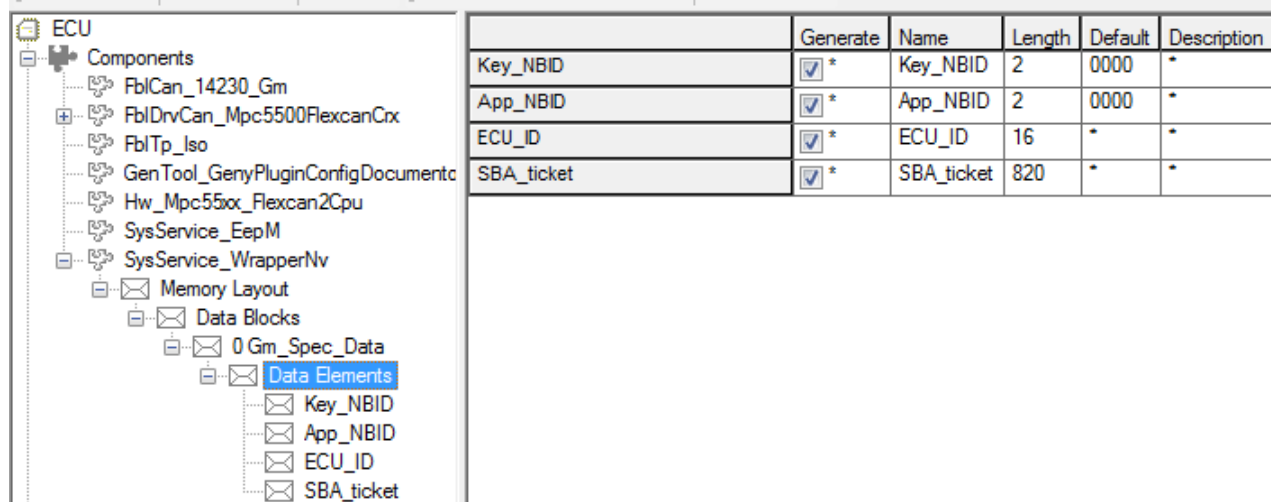


Figure 4-18 NV-Wrapper configuration in GENy



### Caution

Unless explicitly ordered, a real NVM driver is not delivered. In its place, a dummy NVM driver is delivered that emulates EEPROM in RAM. This must be replaced by a real NVM driver.

## 4.8 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	Description
fbl_apfb.c/.h	Contains the Flash-Block table. If multiple devices are supported, this file also contains a table mapping the block entries to the appropriate device-driver API functions.
Fbl_mtab.c/.h	Logical Block Table definition file
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>.
V_cfg.h	Contains macro definitions specific to your hardware.
V_inc.h	Includes the generated headers, so that they may be obtained from a single source. This file is not required to compile the FBL.
V_par.c, v_par.h	Contains information about your license. These files are not required to compile the FBL.
WrapNv_cfg	Contains macros for accessing non-volatile data.

Table 4-5 Generated File contents

## 5 Adapting the FBL Implementation

The bootloader includes a number of files that you must review and adapt to fit the needs of your ECU and application. All files found in the FBL\\_Template folder of your delivery may require customization. You should copy all files from FBL\\_Template to your project directory, rename them (removing the leading underscore character), and adapt them for your application. The FBL demo included with your delivery contains examples of these files (in the \_Demo\DemoFbl\App\Source and \_Demo\DemoFbl\App\Include folders). A brief description of the files is shown below:

File Name	Description
_fbl_ap.c, _fbl_ap.h	Contains hardware-specific, Input/Output, and miscellaneous callback functions.
_fbl_apdi.c, _fbl_apdi.h	Contains callbacks for diagnostic service requests.
_fbl_apnv.c, _fbl_apnv.h	Contains non-volatile operation callbacks. These are used to handle the module (presence-pattern) validation.
_fbl_apwd.c, _fbl_apwd.h	Contains watch-dog callback functions
_fbl_aplvect.c	Contains application-vector-table used by FBL. This file should be adapted if Sleep-Mode is enabled (The interrupt vector used to wake-up the ECU must be defined).
_fbl_inc.h	Contains references to all include files used by the FBL.

Table 5-1 User-Modifiable file contents

Special attention must be paid in all of the 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.

For callback functions that are invoked in response to a diagnostics request, you may also need to determine if you need to send a Response-Pending response message. **ECU Timing Parameter P2<sub>CE</sub>** (Section 6.2.1.1 in reference [1]) requires that a response be sent by the ECU within 100 milliseconds (ms). If your callback implementation may execute longer than this period, you should call the function `FblRealTimeSupport()`. This may be used in place of calls to `FblLookForWatchdog()`, and should be called at least once per millisecond. The function will send a response-pending message at the appropriate interval.

## 5.1 Hardware, Input/Output and miscellaneous Callbacks

The file `fbl_ap.c` contains functions to handle hardware-specific operations, such as input/output control, and several miscellaneous callback functions. A complete list of the functions in this component is shown below:

<code>ApplFblCanBusOff</code>	<code>ApplFblCanParamInit *</code>	<code>ApplFblCanWakeUp</code>
<code>ApplFblCheckProgConditions</code>	<code>ApplFblFatalError</code>	<code>ApplFblCheckDataFormatIdentifier</code>
<code>ApplFblEnterStopMode</code>	<code>ApplFblInit *</code>	<code>ApplFblStartApplication</code>
<code>ApplFblInitErrStatus</code>	<code>ApplTrcvrSleepMode</code>	<code>ApplFblRamIntegrityCheck</code>
<code>ApplFblReset</code>	<code>ApplFblResetVfp *</code>	<code>ApplFblRomIntegrityCheck</code>
<code>ApplFblSetVfp *</code>	<code>ApplFblSleepModeAllowed</code>	<code>ApplFblStartup *</code>
<code>ApplFblTask *</code>	<code>ApplFblTpErrorInd</code>	<code>ApplFblInitDataProcessing</code>
<code>ApplTrcvrHighSpeedMode</code>	<code>ApplTrcvrNormalMode *</code>	<code>ApplFblDataProcessing</code>
<code>ApplFblDeinitDataProcessing</code>		

Table 5-2 Miscellaneous Callback functions

\* These routines are also described in reference [4].

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.

### ApplFblCanBusOff

#### Prototype

```
void ApplFblCanBusOff( void )
```

#### Parameter

-	-
---	---

#### Return code

-	-
---	---

#### Functional Description

The FBL internally checks 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. No action is required in order to recover.

#### Particularities and Limitations

None

### ApplFblCanParamInit

#### Prototype

```
void ApplFblCanParamInit( void )
```

Parameter	
-	-
Return code	
-	-
Functional Description	
<p>This function is used if the configuration selects ECU SPS Type „Enable Can Configuration”, or if the configuration defines multiple-identities (MIM). The routine is called as part of the FBL initialization sequence when started from both reset and from the Operating Software.</p> <p>The purpose of the function is to allow the CAN-ID and bus timing parameters to be set based upon runtime conditions instead of fixed at compile-time. Usually, only the CAN-Ids are of concern.</p>	

## Particularities and Limitations

Three global variables are involved with the FBL communication initialization: `fblCanIdTable`, `CanInitTable`, and `diagNodeAddr`.

`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.

`CanInitTable` contains the CAN identifiers for diagnostic requests and responses, CAN bus timing, and other hardware initialization parameters. The CAN-ID for response messages is always obtained from `CanInitTable`; the remaining fields are only used when the FBL is started by the Operating Software.

The variable `diagNodeAddr` is used by the bootloader to determine if a functionally-addressed diagnostic request should be processed. The FBL will respond to functionally-addressed messages containing the “all-node” (0xFE) address, and to messages containing the Diagnostics Node Address. In all cases, this variable must be initialized by this function.

At compile time, the constant `kFblCanIdTable` is fixed with the request message CAN-IDs and bus timing parameters based on the database and configuration settings. The table is copied to `fblCanIdTable` just before the FBL calls this function.

The contents of `fblCanIdTable` are altered by the FBL *after* this function is called if the FBL is started by the Operating Software. In this case, the FBL will copy the physically-addressed request CAN-ID and bus-timing parameters from `CanInitTable` into `fblCanIdTable` after this function returns (the functionally-addressed request CAN-ID is not modified). Since `CanInitTable` is passed to the FBL by the Operating Software, there is usually no need to change either `fblCanIdTable` or `CanInitTable` in this case. The macro `GetFblMode` may be used to determine if the Operating Software has started the FBL.

When the FBL has not been started by the Operating Software, the response message CAN-ID in `CanInitTable` must be initialized by this function. If the FBL is configured for multiple-identities, then the physically-addressed request CAN-ID in `fblCanIdTable` must also be initialized. The correct IDs to use depend on the FBL configuration. If ECU SPS Type is configured to “TYPE\_B”, then the CAN-IDs must match the “permanently programmed” diagnostic message IDs used by the Operating Software. Otherwise, the identifiers must be calculated based upon the Diagnostics Node Address.

Furthermore, the correct identifiers to use depend on if the FBL is configured for multiple-identities. If your ECU supports MIM, then you must modify this function to perform any necessary Input/Output to determine the ECU’s identity, and select the appropriate CAN-IDs.

The FBL provides three macros that simplify the CAN-ID (and node-address) determination:

`DIAG_NODE_PHYSICAL_REQUEST_ID`, `DIAG_NODE_PHYSICAL_RESPONSE_ID`, and `DIAG_NODE_ADDRESS`. These macros will provide the correct values, based upon your database and configuration settings. If MIM is used, you must call the macro `ComSetCurrentECU` before using any of the `DIAG_NODE_xxx` macros. `ComSetCurrentECU` requires a parameter to select the node identity. The configuration tool, GENy, will define macros in `fbl_cfg.h` for each node the ECU may become. The macro names are in the form `FBL_ECU_HDLE_<node>`.

## AppIFblCanWakeUp

### Prototype

```
void AppIFblCanWakeUp( void )
```

### Parameter

-

-

Return code	
-	-
Functional Description	
<p>This function is used only if the configuration selects “Enable sleep mode”. The function is called from the ECU’s wake-up interrupt service routine (<code>FblCanWakeUpInterrupt</code>). The implementation should perform any necessary tasks (such as I/O port initialization, timer, watchdog, or phase-lock-loop synchronization) needed to restore the hardware to a normally-running state.</p> <p>If sleep-mode is disabled, this function is not called by the FBL. You may, however, implement your own wake-up interrupt-service-routine (referenced from the application-vector-table) that invokes this function.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> <li>&gt; Keep in mind that this function is called in the interrupt-context of your ECU. Special rules apply to such routines, for example: Any external variables altered by this function should be declared volatile to insure data consistency.</li> <li>&gt; See also <code>ApplFblSleepModeAllowed()</code> and <code>ApplFblEnterStopMode()</code></li> </ul>	

### ApplFblCheckProgConditions

Prototype	
<code>tFblResult ApplFblCheckProgConditions( void )</code>	
Parameter	
-	-
Return code	
<code>kFblOk</code>	Indicates that conditions are appropriate for (re)programming the ECU.
<code>kFblFailed</code>	Indicates that conditions are not correct for programming the ECU.
Functional Description	
<p>The implementation of this function should determine whether or not conditions are correct to enter the programming mode.</p> <p>The function is called when the FBL is started by the Operating Software, and may be called when a Programming-Mode request (service \$A5, sub-function \$01 or \$02) is received.</p>	
Particularities and Limitations	
None	

### ApplFblEnterStopMode

Prototype	
<code>void ApplFblEnterStopMode( void )</code>	
Parameter	
-	-
Return code	
-	-



### Functional Description

This function is called by the FBL when it is time to enter the “low power” mode of the ECU. This occurs if there has been no activity on the CAN bus for 60 seconds. In most cases, the function should be modified to enable a wake-up interrupt, and then enter a sleep or stop mode.

Care should be taken when enabling interrupts, since an interrupt may be pending when interrupts are enabled. The hardware must be in a valid state before the halt or stop instruction is executed.

The code following the halt instruction should disable interrupts before returning control to the FBL.

### Particularities and Limitations

> See also `ApplFblSleepModeAllowed()` and `ApplFblCanWakeUp()`

## ApplFblInit

### Prototype

```
void ApplFblInit( void )
```

### Parameter

-	-
---	---

### Return code

-	-
---	---

### Functional Description

This function is called from `main()` immediately after reset (It is also called when the FBL is started from the Operating Software, since the FBL is started via a reset).

The function may be used to perform any hardware and I/O initialization. Also, any global variables used exclusively by the callback components should be initialized at this point.

### Particularities and Limitations

- > For additional information, see the Flash Bootloader User Manual
- > You may choose to start the ECU's watchdog timer in this routine. However, the FBL will not start the timer used to call the watchdog-trigger routine (`ApplFblWDTrigger()`) until much later in the FBL's startup. See also Section 5.4 and `ApplFblWDInit()`.

## ApplFblInitErrStatus

### Prototype

```
void ApplFblInitErrStatus( void )
```

### Parameter

-	-
---	---

### Return 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. See also section 9.3.

## Particularities and Limitations

The FBL error state is available only when the configuration selects Project State “Integration”.

The variables available for error-state storage are:

```
errStatFlashDrvVersion[]
errStatFlashDrvErrorCode
errStatErrorCode
errStatFblStates
errStatLastServiceId
errStatTpError
errStatFileName
errStatLineNumber
errStatDescriptor
errStatHaveDriver
errStatAddress
```

The following macros/functions are used to set the error-state variables:

```
FblErrStatSetSId( id )
FblErrStatSetState( state )
FblErrStatSetFlashDrvError( error )
FblErrStatSetError( error )
FblErrDebugStatus( error )
FblErrDebugDriver( addr, drvError )
```

The GM FBL does not at this time save or use information in `errStatFlashDrvVersion`, or `errStatDescriptor`. These are reserved for future use.

The variables `errStatFlashDrvErrorCode` and `errStatFblStates` are updated as needed by the FBL, but cannot be retrieved over the CAN bus. They are intended to be accessed only during FBL development.

## ApplFblRamIntegrityCheck

### Prototype

```
void ApplFblRamIntegrityCheck( void )
```

### Parameter

-	-
---	---

### Return code

-	-
---	---

### Functional Description

This routine tests that there are no errors in Random Access Memory (RAM). By default, the function is called from `ApplFblStartup()`. This routine should set or clear the flag used by the Report-Programmed-State (service \$A2) callback (`ApplFblReportProgrammedState()`) to indicate that a RAM fault has occurred.

### Particularities and Limitations

- > This function is used only if the configuration selects “Enable RAM Integrity Check”.
- > The default implementation verifies that every bit of every byte in RAM may be set and cleared. You are required to obtain approval from GM to use this algorithm. See “RAM Integrity Check”.
- > The default implementation defines a table named `kRamTable`. You must initialize this structure with the starting and ending address of each address region before compiling the FBL. The result of the RAM test is stored in the `kProgrammedStateRamError` bit contained in the global variable `fblProgrammedState`.

### ApplFblReset

#### Prototype

```
void ApplFblReset( void )
```

#### Parameter

-	-
---	---

#### Return code

-	-
---	---

#### 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. For example, when a Return-To-Normal-Mode (service \$20) request is received when the FBL is in programming-mode.

#### 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.

### ApplFblResetVfp

#### Prototype

```
void ApplFblResetVfp( void )
```

#### Parameter

-	-
---	---

#### Return code

-	-
---	---

#### 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.

The function is called during initialization of the FBL, and when a Return-To-Normal-Mode (service \$20) request is received.

### Particularities and Limitations

> See also `ApplFblSetVfp()`.

### ApplFblRomIntegrityCheck

#### Prototype

```
void ApplFblRomIntegrityCheck( void )
```

#### Parameter

-	-
---	---

#### Return code

-	-
---	---

#### Functional Description

This routine calculates the checksum of the address-regions occupied by the bootloader, and compares the result to the Checksum (CS) field of the FBL's File-Header. By default, the function is called from `ApplFblStartup()`. This routine should set or clear the flag used by the Report-Programmed-State (service \$A2) callback (`ApplFblReportProgrammedState()`) to indicate that a ROM fault has occurred.

#### Particularities and Limitations

- > This function is used only if the configuration selects "Enable ROM Integrity Check".
- > The default implementation stores the result of the ROM test in the `kProgrammedStateRomError` bit contained in the global variable `fblProgrammedState`.

### ApplFblSetVfp

#### Prototype

```
void ApplFblSetVfp( void )
```

#### Parameter

-	-
---	---

#### Return code

-	-
---	---

#### 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.

The function is called when the FBL is started by the Operating Software, and when a Programming-Mode (service \$A5) requesting Enable-Programming-Mode (\$03) is received.

#### Particularities and Limitations

> See also `ApplFblResetVfp()`.

### ApplFblSleepModeAllowed

Prototype	
<code>tFblResult <b>ApplFblSleepModeAllowed</b>( void )</code>	
Parameter	
-	-
Return code	
<code>kFblOk</code>	Indicates that conditions are correct to go to sleep.
<code>kFblFailed</code>	Indicates that FBL should not go to sleep.
Functional Description	
<p>This function should determine if it is OK for the ECU to go to sleep. In most ECUs, an interrupt vector is required to service the wake-up event. A vector pointing to the appropriate interrupt-service-routine (ISR) in the FBL must be defined in the Application Vector Table. Since the Application Vector Table can be erased and replaced by the Operating Software, the default implementation calls an internal function, <code>FblCheckBootVectTableIsValid()</code>, to make sure that the table in flash is the “dummy” table compiled into the FBL. This ensures that the vector points to the service-routine in the FBL instead of a service-routine in the Operating Software. If this table is not present, then the ECU is not allowed to go to sleep.</p> <p>However, some ECUs have alternate methods of waking up that do not require execution of an ISR. In this case, the presence of the “dummy” Application Vector Table is irrelevant. If your ECU falls into this category, you should modify this function to return <code>kFblOk</code> if it is OK to enter sleep mode, or <code>kFblFailed</code> if not.</p>	
Particularities and Limitations	
<p>&gt; See also <code>ApplFblEnterStopMode()</code> and <code>ApplFblCanWakeUp()</code>.</p>	

## ApplFblStartup

Prototype	
<code>void <b>ApplFblStartup</b>( vuInt8 initposition )</code>	
Parameter	
<code>initposition</code>	<p>Indicates that the function is being called before or after hardware and state initialization. Possible values:</p> <p><code>kStartupPreInit</code> – Indicates function call before initialization.</p> <p><code>kStartupPostInit</code> – Indicates function call after initialization.</p>
Return code	
-	-
Functional Description	
<p>This function is called twice during the bootloader startup; See section 3.3.1. The pre-init call occurs after the FBL has determined that the FBL is being started by the Operating Software, or that the Operating Software is not present. The post-init call occurs immediately before the main loop (<code>FblRepeat()</code>) is started. You may perform any required hardware and software initialization before and after the CAN control and timer initialization is completed by <code>main()</code>.</p> <p>The default implementation will run the RAM and ROM integrity checks in the post-initialization call, if they are enabled by the FBL configuration.</p>	
Particularities and Limitations	
<p>&gt; See also the Flash Bootloader User Manual.</p>	

## ApplFblTask

Prototype	
void <b>ApplFblTask</b> ( void )	
Parameter	
-	-
Return code	
-	-
Functional Description	
<p>The purpose of this function is to perform any periodic background tasks.</p> <p>The FBL invokes this function on a regular periodic basis. The actual timing is dependent on both hardware and configuration. The hardware timing (which should not be changed) is indicated by the value of the macro <code>FBL_REPEAT_CALL_CYCLE</code>, defined in milliseconds (usually 1ms, see also your H/W specific documentation). The period at which <code>ApplFblTask</code> is called is a multiple of the repeat-call-cycle time, based on the value of <code>TpCallCycle</code>. The default definition of this parameter, shown below, is found in <code>ftp_cfg.h</code>, and may be edited (using GENy).</p> <pre>#define TpCallCycle      (1/FBL_REPEAT_CALL_CYCLE)</pre> <p>The value represents the period in milliseconds at which the Transport Protocol task and user task functions are called. The value must be an integer multiple of <code>FBL_REPEAT_CALL_CYCLE</code>.</p> <p>Only a limited number of tasks are performed during the start delay period if <code>FBL_ENABLE_STAY_IN_BOOT</code> is defined. During the delay period, the FBL does not invoke <code>ApplFblTask</code>.</p> <p>Since the timing of all events in the FBL are determined by the ability of the ECU to complete the main loop in less than the period defined by <code>FBL_REPEAT_CALL_CYCLE</code>, all code added to <code>ApplFblTask</code> must be kept as short as possible. After adding code to this function, you should verify that the time required to execute the main task loop (found in <code>FblRepeat</code>), is always less than the call cycle period.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> <li>&gt; This function is called only if the configuration selects “Enable ApplTask”.</li> <li>&gt; This function is only called while the FBL is idling. Calling intervals exceeding the <code>TpCallCycle</code> can occur while the FBL handles diagnostic service requests. This normally occurs while erasing and writing non-volatile memory.</li> </ul>	

### ApplFblTpErrorInd

Prototype
void <b>ApplFblTpErrorInd</b> ( vuint8 tpErrorCode )

Parameter	
tpErrorCode	<p>Indicates the error that was detected by the TP Layer. Possible values are (see also fbl_tp.h):</p> <p><b>kTpErrRxNotIdle</b> A single-frame or first-frame message was received while processing a previous request (i.e. the receive buffer is locked).</p> <p><b>kTpErrRxSFDL</b> A single-frame message was received containing an illegal value in the Data-Length (DL) field of the Protocol-Control-Information (PCI) byte.</p> <p><b>kTpErrRxCFNotExpected</b> A consecutive-frame message was received unexpectedly. CF messages must be preceded by a first-frame message.</p> <p><b>kTpErrRxWrongSN</b> A consecutive-frame message was received containing an illegal or unexpected value in the Sequence-Number (SN) field of the Protocol-Control-Information (PCI) byte.</p> <p><b>kTpErrRxTimeout</b> Too much time elapsed while waiting for a consecutive-frame message to arrive.</p>
Return code	
-	-
Functional Description	
<p>This function is called from the Transport-Protocol layer to indicate that an error has occurred. The cause of the error is supplied in the tpErrorCode argument.</p> <p>The implementation may choose to ignore the error; record the error, or respond to the error.</p> <p>The default implementation saves the error code in the global variable <code>errStatTpError</code>, and then sends a negative response indicating that a general programming error has occurred.</p>	
Particularities and Limitations	
<p>&gt; The TP Layer actually calls <code>TpErrorIndication()</code>. This is a function-like macro defined in <code>ftp_cfg.h</code> that redirects the call to <code>ApplFblTpErrorInd()</code>. The default implementation is to ignore the error.</p> <p>You must update the macro definition in order to call <code>ApplFblTpErrorInd()</code> (which by default will record the error and send a negative-response indicating a General-Programming-Failure).</p>	

**ApplFblVerifyChecksum**

**ApplTrcvrHighSpeedMode**

Prototype	
void <b>ApplTrcvrHighSpeedMode</b> ( void )	
Parameter	
-	-

Return code	
-	-
Functional Description	
<p>This function is used to configure your CAN bus transceiver to operate in high-speed mode. You must implement this routine to perform the necessary Input/Output operations to control the transceiver state.</p> <p>The function is called when the FBL is started by the Operating Software, and when a Programming-Mode (service \$A2) request to enable-programming-mode (subfunction \$03) is received (if preceded with a Programming-Mode, request-programming-mode-high-speed [subfunction \$02] request).</p>	
Particularities and Limitations	
<p>&gt; This function is used only if the configuration selects "Enable High-Speed". This is required if your ECU is communicating on the single-wire CAN "Body Bus".</p>	

### ApplTrcvrNormalMode

Prototype	
void <b>ApplTrcvrNormalMode</b> ( void )	
Parameter	
-	-
Return code	
-	-
Functional Description	
<p>This function is used to configure your CAN bus transceiver for normal CAN communications. You must implement this routine to perform the necessary Input/Output operations to control the transceiver state.</p> <p>This function is called when the FBL is started (either power-on or via Operating Software), when waking-up (if sleep-mode has been enabled), and when a Return-To-Normal-Mode (service \$20) request is received (if high-speed programming had been requested).</p>	
Particularities and Limitations	
None	

### ApplTrcvrSleepMode

Prototype	
void <b>ApplTrcvrSleepMode</b> ( void )	
Parameter	
-	-
Return code	
-	-
Functional Description	
<p>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.</p> <p>This function is called from the main loop after the FBL has determined that it is ok to go to sleep (see <code>ApplFblSleepModeAllowed()</code>). Upon waking up, the FBL will call <code>ApplTrcvrNormalMode()</code> to allow normal communications.</p>	



### Particularities and Limitations

> None

## AppIFblStartApplication

### Prototype

```
void AppIFblStartApplication ( void )
```

### Parameter

-	-
---	---

### Return code

-	-
---	---

### Functional Description

The function is called to start application. It contains an implementation we used for Demo purpose. You may adapt this function if your application needs to be started differently.

### Particularities and Limitations

> None

## AppIFblFatalError

### Prototype

```
void AppIFblFatalError( FBL_DECL_ASSERT_EXTENDED_INFO(vuint8 errorCode) )
```

### Parameter

-	-
---	---

### Return code

-	-
---	---

### Functional Description

The function is only available if Project state is "Integration" it is called if an assertion placed in the code is found to be invalid (assertXX-maros found in code). Define an action that makes the wrong condition visible to you (e.g. Led blinking). Per default Fbl enters a while(1) loop.

### Particularities and Limitations

> None

## AppIFblCheckDataFormatIdentifier

### Prototype

```
tFblResult AppIFblCheckDataFormatIdentifier(vuint8 formatId)
```

### Parameter

formatId	Data format identifier from the requestDownload service.
----------	--

Return code	
tFblResult	-
Functional Description	
<p>This function is called to check the data format identifier value.</p> <p>Data Processing interface function. This function is only required for certain configurations that have to be specifically ordered:</p> <ul style="list-style-type: none"> <li>- Pipelined Programming</li> <li>- Compression</li> </ul> <p>Encryption/decryption</p>	
Particularities and Limitations	
> None	

### AppIFblInitDataProcessing

Prototype	
tFblResult <b>AppIFblInitDataProcessing</b> ( tProcParam *procParam )	
Parameter	
tProcParam *procParam	-
Return code	
tFblResult	-
Functional Description	
<p>This function is called to initialize the application specific data processing function.</p> <p>Data Processing interface function. This function is only required for certain configurations that have to be specifically ordered:</p> <ul style="list-style-type: none"> <li>- Pipelined Programming</li> <li>- Compression</li> <li>- Encryption/decryption</li> </ul>	
Particularities and Limitations	
> None	

### AppIFblDataProcessing

Prototype	
tFblResult <b>AppIFblDataProcessing</b> ( tProcParam *procParam )	
Parameter	
tProcParam *procParam	-
Return code	
tFblResult	-

### Functional Description

Data processing function. This function is only required for certain configurations that have to be specifically ordered:

- Pipelined Programming
- Compression
- Encryption/decryption

### Particularities and Limitations

> None

### ApplFblDeinitDataProcessing

#### Prototype

**tFblResult ApplFblDeinitDataProcessing ( tProcParam \*procParam )**

#### Parameter

tProcParam *procParam	-
-----------------------	---

#### Return code

tFblResult	-
------------	---

### Functional Description

Data Processing deinitialization. This function is only required for certain configurations that have to be specifically ordered:

- Pipelined Programming
- Compression
- Encryption/decryption

### Particularities and Limitations

> None

## 5.2 Diagnostic Service Callbacks

The file fbl\_apdi.c contains all the routines used to handle diagnostic service requests. Each diagnostic service is described in detail in reference [1]. The complete list of routines in this component is shown below:

ApplFblEnablePrgMode	ApplFblInitiateDiagnosticOperation
ApplFblReadDataByIdentifier	ApplFblReadMemoryByAddress
ApplFblReportProgrammedState	ApplFblRequestDownload
ApplFblRequestPrgMode	ApplFblSecurityAccess
ApplFblTransferData	ApplFblRdbidProgrammedStateInd
ApplFblWriteDataByIdentifier	

Table 5-3 Diagnostic Callback Functions

\* These routines are also described in reference [4].

### ApplFblEnablePrgMode

#### Prototype

```
void ApplFblEnablePrgMode( void )
```

#### Parameter

-	-
---	---

#### Return code

-	-
---	---

#### Functional Description

This function is called to notify you that a Programming-Mode (service \$A5) request for enable-programming-mode (subfunction \$03) has been received. The function is not called when the FBL is started by the Operating Software.

This is for notification purposes only. Normally this function performs no action.

#### Particularities and Limitations

> See also ApplFblRequestPrgMode().

### ApplFblInitiateDiagnosticOperation

#### Prototype

```
void ApplFblInitiateDiagnosticOperation( vuint8 subfunction )
```

#### Parameter

subfunction	Value of the sub-function argument in the service request.
-------------	--

#### Return code

-	-
---	---

#### Functional Description

This function is called when the FBL receives an Initiate-Diagnostic-Operation (service \$10) request.

The function should validate the sub-function argument (normally, only wake-up-links (\$04) is required), and take the appropriate action. If the sub-function is invalid, the function should invoke the macro `DiagNRCSubFunctionNotSupportedInvalidFormat()`, and then return.

“Service \$10 – InitiateDiagnosticOperation” (Section 9.3.2.4.1 in reference [1]), requires this function only for a Gateway ECU.

### Particularities and Limitations

- > This function is called only if the configuration selects “Enable Initiate-Diagnostic-Operation”. When not enabled, the FBL will respond with a negative response indicating service-not-supported.
- If you add support for “Disable-All-DTCs” or “Enable-DTC-During-Dev-Cntrl” sub-functions, be aware that you must start the Tester-Present if it is not running. For example:

```
if (TimeoutTesterValue() == 0)
{
    ResetTesterTimeout();
}
```

### ApplFblReadDataByIdentifier

#### Prototype

```
void ApplFblReadDataByIdentifier (
    vuInt8 *pbDiagData,
    tTpDataType diagReqDataLen
)
```

#### Parameter

pbDiagData	Pointer to buffer containing the diagnostic request.
diagReqDataLen	The number of bytes in the diagnostic request. This should be a constant, kDiagRqlReadDataByIdentifier.

#### Return code

-	-
---	---

#### Functional Description

This function is called to handle Read-Data-By-Identifier (service \$1A) requests.

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.

The DID may be accessed at `pbDiagData[kDiagFmtDid]`. Symbolic names for GM standard identifiers may be found in `fbl_diag.h` with names like `kDiagDid<identifier>`. If the DID is not supported, the function should invoke the macro `DiagNRCRequestOutOfRange()`. If the DID is supported but not available, the function should invoke the macro `DiagNRCConditionsNotCorrectOrRequestSeqErr()`.

The response should be placed in the supplied buffer, starting at `pbDiagData[kDiagFmtDid+1]`.

### Particularities and Limitations

- > The FBL reserves the following DID values:  
 \$79 – Used to ping the ECU to stay in bootloader. See section 9.4.  
 \$7F – Used to retrieve extended error information. See section 9.3.
- > The default implementation supports the ECU-Diagnostic-Address (\$B0), Boot-Software-Part-Number (\$C0), Boot-Software-Part-Number-Alpha-Code (\$D0), the Software-Module-Identifier (\$C1 - \$CA), and the Software-Module-Identifier-Alpha-Code (\$D1-DA) DIDs.

Note that the implementation for DIDs \$C1-\$CA and \$D1-\$DA assume that only one Operating S/W module is downloaded.

You must implement support for additional DIDs as required by your application. See also “Appendix C – Corporate Standard Data Identifiers (DID)” (Section 12.3 in reference [1]).

### ApplFblReadMemoryByAddress

#### Prototype

```
void ApplFblReadMemoryByAddress (
    FBL_ADDR_TYPE address,
    FBL_MEMSIZE_TYPE len
)
```

#### Parameter

address	Starting address (logical) of memory region to read.
Len	Number of bytes to read from memory region.

#### Return code

-	-
---	---

#### Functional Description

This function is called when a Read-Memory-By-Address (service \$23) request is received. The function should obtain the data from the specified memory and place it in the diagnostics message buffer (starting at `DiagBuffer[kDiagRslReadMemoryByAddress]`).

The default implementation does not secure the access to memory. You may use the macro `GetSecurityUnlock()` if you wish to use the security lock state to restrict the memory access.

#### Particularities and Limitations

- > This function is used only if the FBL configuration selects “Enable Read-Memory-By-Address”. If not selected, the FBL will send a negative response indicating service-not-supported.
- > This service is not required in order to download modules to the ECU, and is normally enabled only during the development phase of your application.

### ApplFblReportProgrammedState

#### Prototype

```
vuint8 ApplFblReportProgrammedState ( void )
```

#### Parameter

-	-
---	---

Return code	
kDiagProgStateFullyProgrammed	Indicates that the Operating Software and all Calibration modules are present.
kDiagProgStateNoSoftwareOrCal	Indicates that both Operating Software and Calibration modules are not present.
kDiagProgStateNoCalibration	Indicates that the Operating Software is present, but that one or more Calibration modules are not present.
kDiagProgStateDefOrNoStartCal	Indicates that the Operating Software and all Calibration modules are present, but some calibration files are only dummy files, that still need to be replaced with real calibration files.
kDiagProgStateGeneralMemFault	Indicates that an unspecified memory failure has been detected.
kDiagProgStateRamFault	Indicates that the RAM Integrity Check has failed.
kDiagProgStateNvramFault	Indicates that a non-volatile RAM fault has been detected.
kDiagProgStateBootFault	Indicates that the ROM Integrity Check has failed.
kDiagProgStateFlashFault	Indicates that a flash memory fault has been detected.
kDiagProgStateEepromFault	Indicates that an EEPROM memory fault has been detected.
Functional Description	
This function is called when a Report-Programmed-State (service \$A2) request has been received. The function must evaluate the fault-state and module presence-patterns to determine the appropriate return value.	
Particularities and Limitations	
<ul style="list-style-type: none"> <li>&gt; The default implementation first checks the global variable <code>fblProgrammedState</code> to see if an integrity check failed. In order to allow the FBL to eventually report the module presence state, each memory fault state flag is cleared after it has been reported.</li> <li>&gt; The default implementation assumes that only one Operating Software module (MID == \$0001) is downloaded.</li> <li>&gt; The return values <code>kDiagProgStateGeneralMemFault</code>, <code>kDiagProgStateNvramFault</code>, <code>kDiagProgStateFlashFault</code>, <code>kDiagProgStateDefOrNoStartCal</code> and <code>kDiagProgStateEepromFault</code> are not presently used by the FBL. You may use these as required.</li> </ul>	

## ApplFblRequestDownload

Prototype	
<code>tFblResult ApplFblRequestDownload( void )</code>	
Parameter	
-	-
Return code	
kFblOk	Return this if conditions are correct to allow a module download to begin.
kFblFailed	Return this if a download is not allowed.

### Functional Description

The purpose of this routine is to determine if conditions are correct to permit a download to begin. If conditions are not correct, the routine should invoke one of the error-indication macros `DiagNRC<condition>` (see `fbl_diag.h`), and then return `kFblFailed`.

The function is called when a Request-Download (service \$34) request is received, and when the FBL is started by the Operating Software.

### Particularities and Limitations

None

## ApplFblRequestPrgMode

### Prototype

```
tFblResult ApplFblRequestPrgMode( vuint8 requestMode )
```

### Parameter

requestMode	Indicates if normal-speed versus high-speed programming is being requested.  If normal-speed, the value will equal <code>kDiagSubRequestProgMode</code> . If high-speed, the value will equal <code>kDiagSubRequestProgModeHSpeed</code> .  These values are based on the sub-function argument contained in the programming-mode service request.
-------------	--

### Return code

kFblOk	Return this if conditions are correct for enabling programming-mode
kFblFailed	Return this if conditions are not correct for enabling programming-mode

### Functional Description

The purpose of this function is to determine that conditions are correct for entering programming-mode.

This routine is called when a Programming-Mode (service \$A5) request is received containing a request-programming-mode (\$01) or request-programming-mode-high-speed (\$02) sub-function.

### Particularities and Limitations

- > This routine is not used when the FBL is started by the Operating Software.
- > The default implementation invokes `ApplFblCheckProgConditions()`.
- > There is no need to invoke an error-indication macro in the event of an issue. The FBL will invoke `DiagNRCConditionsNotCorrectOrRequestSeqErr()` if `kFblFailed` is returned.

## ApplFblSecurityAccess

### Prototype

```
void ApplFblSecurityAccess( vuint8 subfunction )
```

### Parameter

subfunction	Value of the sub-function argument contained in the Security-Access request.
-------------	--



Return code	
-	-
Functional Description	
<p>This function is called for all Security-Access (service \$27) requests other than the request-seed (\$01) and send-key (\$02) subfunctions.</p> <p>The function should verify that the sub-function is valid, and take the appropriate action. If the sub-function is not valid, the routine should invoke the macro <code>DiagNRCSubFunctionNotSupportedInvalidFormat()</code>. See "Negative Response (\$7F) Service Definition" and "SecurityAccess (\$27) Service" (Sections 9 and 10.8 in reference [1]) for other negative responses that may be used.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> <li>&gt; This function is used only if the configuration selects "Enable Security-Access". If security access is not enabled, the FBL will send a negative response indicating service-not-supported.</li> <li>&gt; See also <code>ApplFblSecuritySeed()</code> and <code>ApplFblSecurityKey()</code>.</li> </ul>	

### ApplFblSecurityKey

Prototype	
<code>vuint8 ApplFblSecurityKey( void )</code>	
Parameter	
-	-
Return code	
<code>kFblOk</code>	Return this if the security-key obtained from the service request matches the expected security-key.
<code>kFblFailed</code>	Return this if the security-key obtained from the service request does not match the expected security-key.
Functional Description	
<p>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 <code>ApplFblSecuritySeed()</code>.</p> <p>This function is called when a Security-Access (service \$27) request containing the send-key (\$02) subfunction is received.</p> <p>The security-key is retrieved directly from the diagnostics message buffer.  <code>DiagBuffer[kDiagFmtSeedKeyStart]</code> will retrieve the most-significant byte,  <code>DiagBuffer[kDiagFmtSeedKeyStart+1]</code> will retrieve the least-significant byte.</p>	

### Particularities and Limitations

- > This function is used only if the configuration selects “Enable Security-Access”. If security access is not enabled, the FBL will send a negative response indicating service-not-supported.
- > The default implementation uses the macros `KEY_SECURITY_1` and `KEY_SECURITY_2` to define the most-significant and least-significant (respectively) bytes of the expected security-key. You may modify or replace these as appropriate for your security implementation.
- > There is no need to invoke an error-indication macro in your implementation. The FBL will send a negative response indicating either invalid-key or exceeded-number-of-attempts, as appropriate, if `kFblFailed` is returned.
- > See also `ApplFblSecuritySeed()` and `ApplFblSecurityAccess()`.

### ApplFblSecuritySeed

#### Prototype

```
vuint8 ApplFblSecuritySeed( void )
```

#### Parameter

-	-
---	---

#### Return code

<code>kFblOk</code>	Return this value if the FBL should allow a Security-Access, send-key request.
<code>kFblFailed</code>	Return this value if conditions are not correct for returning a security-seed.

#### 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 (\$01) sub-function.

The security-seed must be placed directly in the diagnostics message buffer. The most-significant byte must be stored in `DiagBuffer[kDiagFmtSeedKeyStart]`, and the least-significant byte must be stored in `DiagBuffer[kDiagFmtSeedKeyStart+1]`. The security-seed must be in the range \$0001 - \$FFFE (inclusive).

### Particularities and Limitations

- > This function is used only if the configuration selects “Enable Security-Access”. If security access is not enabled, the FBL will send a negative response indicating service-not-supported.
- > There is no need to invoke an error-indication macro in your implementation. The FBL will send a negative response indicating conditions-not-correct-or-request-sequence-error if `kFblFailed` is returned.
- > The default implementation uses the macros `SEED_SECURITY_1` and `SEED_SECURITY_2` to define most-significant and least-significant (respectively) bytes of the security-seed. You may modify or replace these as appropriate for your security implementation.
- > See also `ApplFblSecurityKey()` and `ApplFblSecurityAccess()`.

### ApplFblTransferData

#### Prototype

```
tFblResult ApplFblTransferData( void )
```

Parameter	
-	-
Return code	
kFbIOk	Return this value if the FBL should proceed with writing transferred data to non-volatile memory.
kFbIFailed	Return this if conditions are not correct for continuing with the module download.
Functional Description	
<p>The purpose of this function is to verify that conditions are correct to write data to non-volatile memory. The function is called when a Transfer-Data (service \$36) request is received.</p> <p>If conditions are not correct, the function should call the appropriate error-indication macro, such as <code>DiagNRCVoltageOutOfRange()</code>. See <code>fbl_diag.h</code>.</p>	
Particularities and Limitations	
None	

### ApplFblWriteDataByIdentifier

Prototype	
<pre> vuint8 <b>ApplFblWriteDataByIdentifier</b> (                                 vuint8 *pbDiagData,                                 tTpDataType diagReqDataLen                                 ) </pre>	
Parameter	
pbDiagData	Pointer to the start of the diagnostics message buffer.
diagReqDataLen	The number of bytes in the service request. The number of bytes to be written will be equal to <code>(diagReqDataLen - kDiagFmtDid)</code> .
Return code	
kFbIOk	Return this value if the data was successfully saved.
kFbIFailed	Return this value if the data could not be written.
Functional Description	
<p>The purpose of this function is to respond to Write-Data-By-Identifier (service \$3B) requests.</p> <p>The function must obtain the Data-Identifier (DID) from the diagnostics message buffer, and determine that that the DID is supported. If the DID is not supported, the routine must invoke the macro <code>DiagNRCRequestOutOfRange()</code>.</p> <p>The Data-Identifier may be retrieved at <code>pbDiagData[kDiagFmtDid]</code>. The data to be written begins at <code>pbDiagData[kDiagFmtDid+1]</code>.</p>	

### Particularities and Limitations

- > This function is used only if your configuration selects “Enable Write-Data-By-Identifier”. If not enabled, the FBL will send a negative response indicating service-not-supported.
- > The default implementation does not support any Data-Identifiers (The implementation always indicates that the DID is out-of-range). You will need to implement support for each DID required by your application. Symbolic names for GM standard identifiers may be found in `fbl_diag.h` with names like `kDiagDid<identifier>`.
- > If using a device-driver API to write the data, you must ensure that the device-driver has been initialized. To determine if the driver has been initialized, you may use the macro `GetMemDriverReady()`. If multiple-device support is enabled, you must pass the device identifier to the macro. The device identifier may be obtained using the following code:

```
vsint16 memSegment;
vuint8 deviceIdentifier;
memSegment = FblMemSegmentNrGet(address);
if (memSegment >= 0) { deviceIdentifier =
FlashBlock[memSegment].device; }
```

- > The FBL implementation presently ignores the value returned from this function. To indicate an error, you must invoke one of the error-indication macros `DiagNRC<condition>`. See `fbl_diag.h`.

### ApplFblRdbidProgrammedStateInd

#### Prototype

```
tTpDataType ApplFblRdbidProgrammedStateInd ( void )
```

#### Parameter

<code>vuint8 *</code> <code>pbDiagData</code>	Pointer to data buffer for response data
--	--

#### Return code

<code>tTpDataType</code>	Length of response data
--------------------------	-------------------------

#### Functional Description

This function is called to fill the response data upon receiving `readDataByIdentifier` (0x1A) with data identifier `ProgrammedStateIndicator` (0xF0). The function checks the PSI state of each available partition and places the information in the response buffer.

#### Particularities and Limitations

None

### 5.3 Module Validation Callbacks

ApplFblExtProgRequest	ApplFblFillGaps	ApplFblChkModulePresence
ApplFblInvalidateBlock	ApplFblChkOpSwProgrammedState	ApplFblChkPSIState
ApplFblValidateBlock	ApplFblGetPresencePatternBaseAddress	ApplFblGetBaseModulePPRegion
ApplFblIsValidApp *	ApplFblSetModulePresence	ApplFblGetModuleHeaderAddress
ApplFblGetProgrammedState	ApplFblClrModulePresence	ApplFblUpdateChecksum
ApplFblINVMReadKeyNBID	ApplFblINVMReadECUID	ApplFblFinalizeChecksum
ApplFblINVMWriteKeyNBID	ApplFblINVMReadAppNBID	ApplFblINVMWriteAppNBID
ApplFblINVMReadSBATicket		

Table 5-4 Module Validation Callbacks

\* These routines are also described in reference [4].

#### ApplFblExtProgRequest

Prototype	
tFblProgStatus <b>ApplFblExtProgRequest</b> ( void )	
Parameter	
-	-
Return code	
kNoProgRequest	Return this value if the FBL has not been started by the Operating Software
kProgRequest	Return this value if the FBL has been started by the Operating Software
Functional Description	
This function is called during the startup of the ECU to determine if the FBL has been started by the Operating Software.	
Particularities and Limitations	
<ul style="list-style-type: none"> <li>&gt; The default implementation uses a macro called <code>FblChkFblStartMagicFlag()</code> to determine if the bootloader was started by the Operating Software. The macro uses a global variable named <code>fblStartMagicFlag[]</code>. This array must be located in a region of memory that is not destroyed when a reset occurs. If located in RAM, the region must not be zeroed by the ECU startup-code.</li> <li>&gt; The variable used to indicate startup by the Operating Software (<code>fblStartMagicFlag[]</code>) must be cleared before exiting this routine. The macro <code>FblClrFblStartMagicFlag()</code> may be used for this purpose.</li> </ul>	

## AppIFblFillGaps

Prototype	
<code>tFblResult <b>AppIFblFillGaps</b>( void)</code>	
Parameter	
-	-
Return code	
kFblOk	Return this if regions of currently downloaded module are successfully filled.
kFblFailed	Return this if regions of currently downloaded module are not successfully filled.
Functional Description	
<p>If the gap fill feature is enabled in GENy then this function calls the build in function FblHdrFillGaps to fill unused memory with the configured fill pattern.</p> <p>If the gap fill feature is disabled in GENy then the user may implement a custom gap fill algorithm in this function. Otherwise, the user can leave the function as is and no gap filling will be performed by the bootloader.</p> <p>See also 9.9.</p>	
Particularities and Limitations	
<p>&gt; The starting address of each address-region must be aligned to a write-segment boundary.</p>	

## AppIFblInvalidateBlock

Prototype	
<code>tFblResult <b>AppIFblInvalidateBlock</b>( tBlockDescriptor blockDescriptor )</code>	
Parameter	
blockDescriptor	Module-Id (MID) from the File-Header of the module being downloaded.
Return code	
kFblOk	Return this value if the module was successfully invalidated.
kFblFailed	Return this value if the module validation could not be changed.
Functional Description	
<p>The purpose of this function is to alter the flag(s) used by <code>AppIFblIsValidApp()</code> to indicate that the module is not present. The flag(s) used to validate a module is often referred to as a Presence-Pattern.</p>	

### Particularities and Limitations

The implementation provided usually does not have to be changed. If you want to change it please note the following things:

- > See also `ApplFblIsValidApp()` and `ApplFblValidateBlock()`.
- > If a failure occurs while invalidating the module, you must call one of the error-indication macros defined in `fbl_diag.h`. These have the form `DiagNRC<condition>`. For example, `DiagNRCGeneralProgError()`.
- > “Programmed State Indicator (PSI)” (Section 12.5.10 in reference [2]) requires that the the PSI be located at the end of the last sector used by the Operating Software and last calibration module. Per default Calibration modules PSI locations are automatically determined to the last `FlashBlock[]` of the calibration partition covered, whereas application/reserved modules pattern locations are to be configured (check `ApplFblGetPresencePatternBaseAddress()` ).
- > If using the erase method to invalidate a module, **you must insure that the presence-pattern is erased before any other data in the block.**

### ApplFblIsValidApp

#### Prototype

```
tApplStatus ApplFblIsValidApp( void )
```

#### Parameter

-	-
---	---

#### Return code

<b>kApplValid</b>	Return this to indicate that the Operating Software and all required supporting (e.g. Calibration) modules have been downloaded and are ready to run.
<b>kApplInvalid</b>	Return this if the Operating Software is not ready to run.

#### Functional Description

This routine is called to determine if the FBL should start the Operating Software. It is called during the startup of the ECU, after verifying that the FBL is not being started by the Operating Software (see `ApplFblExtProgRequest()`).

The routine should check the presence-pattern flag(s) managed by `ApplFblValidateBlock()` and `ApplFblInvalidateBlock()`. If all required modules are present, then the routine should indicate that the application is valid.

#### Particularities and Limitations

- > See also `ApplFblValidateBlock()`, `ApplFblInvalidateBlock()`, and Section 5.9.

### ApplFblValidateBlock

#### Prototype

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

#### Parameter

<b>blockDescriptor</b>	Module-Id (MID) from the File-Header of the module being downloaded.
------------------------	--

Return code	
kFblOk	Return this to indicate that the module presence-pattern has been successfully written.
kFblFailed	Return this to indicate that the module presence-pattern cannot be saved to non-volatile memory.
Functional Description	
The purpose of this function is to update the flag(s) used by <code>ApplFblIsValidApp()</code> to indicate that a module has been downloaded. The function is called at the end of a download, after all data has been written and validation (checksum) tests completed.	
Particularities and Limitations	
> See also <code>ApplFblIsValidApp()</code> , <code>ApplFblInvalidateBlock()</code> , and Section 5.9.	

### ApplFblGetProgrammedState

Prototype	
vuint8 <b>ApplFblGetProgrammedState</b> (void)	
Parameter	
-	-
Return code	
kDiagProgStateFullyProgrammed	Appl and all Cals are present
kDiagProgStateNoSoftwareOrCal	Appl not present
kDiagProgStateNoCalibration	Appl present, some Cal is missing
kDiagProgStateDefOrNoStartCal	Appl present, all Cals are present (special Cals) (Will never be reported currently, e.g. change <code>ApplFblCalsPresent()</code> implementation to detect default/no start calls to allow this return value).
Functional Description	
Checks if the application and/or calibration presence pattern of Primary Operating software all set	
Particularities and Limitations	
-	

### ApplFblChkOpSwProgrammedState

Prototype	
static vuint8 <b>ApplFblChkOpSwProgrammedState</b> (vuint8 opSwID)	
Parameter	
opSwID	Id of operation software to get programmed state for.
Return code	
kDiagProgStateFullyProgrammed	Appl and all Cals are present
kDiagProgStateNoSoftwareOrCal	Appl not present
kDiagProgStateNoCalibration	Appl present, some Cal is missing



### Functional Description

Check on given opSwID the Programmed State of OpSW and its calibration files

### Particularities and Limitations

-

## ApplFblGetModuleHeaderAddress

### Prototype

```
tFblAddress ApplFblGetModuleHeaderAddress( vuint8 blockNr )
```

### Parameter

blockNr	Describes Table entry number of Logical block. Be careful: this is not the "Block Index" information in GENy. It is simply the number of the entry: e.g.: 0 – First entry (usually appl with Block Index/MID of 0x01) 1 – Second entry (e.g. appl2 with MID of 0x15/21)
---------	---

### Return code

tFblAddress	Address of module header information structure
-------------	--

### Functional Description

This function has to return the address of a module header.

Only Application and Reserved Modules have to be configured in default configuration. These are the modules that have to be configured in Logical Block Table. You may change calibration pattern addresses from automatically determining to static configuration by replacing the call to FblHdrGetCalibrationPPRegion() by returning mid specific addresses.

### Particularities and Limitations

The module header may be placed at e.g.

- > the beginning of the logical block (offset = +0x00 ) or
- > with an offset to the beginning of the block (offset = +0xXX)
- > with an offset to the end of the block (offset = -0xXX)

Add check for blockNr that do not have header information located at the start of the logical block table region.

## ApplFblGetBaseModulePPRegion

### Prototype

```
void ApplFblGetBaseModulePPRegion(vuint8 mid, IO_PositionType *pPresPtnAddr, IO_SizeType *pPresPtnLen )
```

### Parameter

mid	Id of base module.
*pPresPtnAddr	Presence Pattern address for the given module
*pPresPtnLen	Presence Pattern Area length for the given module (for mask and pattern)

Return code	
None	
Functional Description	
This function has to return address and length of the covered presence pattern region (mask and pattern).	
Only Application and Reserved Modules, being base modules have to be configured. Note that these are the modules that have to be configured in Logical Block Table.	
Particularities and Limitations	
This callback is not required to be touched for standard single application configuration (APPL1_HDR_OFFSET_TO_PP_END is configured in GENy). Any further application or reserved module (= base module) however requires that the callback is checked. Recommendation is to use macros like BLOCKNR1_HDR_OFFSET_TO_PP_END, BLOCKNR2_HDR_OFFSET_TO_PP_END configured to GENy preconfig file and add checks for the appropriate blockNr.	

### AppIFblGetPresencePatternBaseAddress

Prototype	
static tFblAddress AppIFblGetPresencePatternBaseAddress ( vuInt8 blockNr, IO_PositionType *pPresPtnAddr, IO_SizeType *pPresPtnLen)	
Parameter	
blockNr	this is either the module ID of a downloaded module or the virtual blockNbr (starting above MAX_MODULE_ID) of a replacement key Descriptor to be stored inside update key section configured to FBL_UPDATEKEY_SEC_START_ADDR
pPresPtnAddr	Pointer to RAM location to place the address to the begin of presence pattern region
pPresPtnLen	Pointer to the RAM location where the length of the presence pattern shall be stored to.
Return code	
memSegment	memSegment of the presence pattern location or kFblDiagMemSegmNotFound in case of an error
Functional Description	
Returns the base address of the presence pattern and mask and the length of both fields.	
Particularities and Limitations	
<ul style="list-style-type: none"> <li>&gt; Currently addresses are automatically calculated by FblHdrGetPPAddrFromCoveredSegments for calibration files. The address for application 1 is configured in GENy and used inside AppIFblGetBaseModulePPRegion(). The callback will have to be directly edited for further applications or reserved module addresses. The address for a calibration module is calculated to be the last Flashblock Table entry touched by the module – 2*Write Segment size. This way pattern and mask will be the 2 last write segments inside the last block touched.</li> </ul>	

### AppIFblSetModulePresence

Prototype	
static tFblResult <b>App1Fb1SetModulePresence</b> (tBlockDescriptor *blockDescriptor);	
Parameter	
blockDescriptor	Pointer to the logical block descriptor
Return code	
kFbIOk	Presence pattern successfully set
kFbIFailed	Error writing presence pattern
Functional Description	
Writes the presence pattern into the flash memory. The location of the presence pattern will be taken from the logical block descriptor.	
Particularities and Limitations	
The function is invoked so that blockDescriptor->blockNr will contain the Module ID of the downloaded module.	

#### App1Fb1ClrModulePresence

Prototype	
static tFblResult <b>App1Fb1ClrModulePresence</b> (tBlockDescriptor *blockDescriptor);	
Parameter	
blockDescriptor	Pointer to the logical block descriptor
Return code	
kFbIOk	Mask for invalidation successfully written
kFbIFailed	Error writing invalidation mask
Functional Description	
Sets the mask presence pattern in flash memory to invalidate the block. The location of the presence pattern will be taken from the logical block descriptor.	
Particularities and Limitations	
The function is invoked so that blockDescriptor->blockNr will contain the Module ID of the downloaded module.	

#### App1Fb1ChkModulePresence

Prototype	
static tFblResult <b>App1Fb1ChkModulePresence</b> (tBlockDescriptor *blockDescriptor);	
Parameter	
blockDescriptor	Pointer to the logical block descriptor
Return code	
kFbIOk	Presence pattern are set and Mask value are OK
kFbIFailed	Presence pattern not set or mask flag not correct.

### Functional Description

Checks if mask and value of the presence pattern are set for a valid module.

### Particularities and Limitations

Note that other than stated in the function header, the location of the presence pattern is calculated from gmheader regions, and not taken from Logical block table (text is standard Api and cannot be changed).

## AppIFblChkPSIState

### Prototype

```
tPartPresState AppIFblChkModulePresence (vuint8 partId);
```

### Parameter

partId	Partition ID of module to be checked
--------	--------------------------------------

### Return code

PSI_PART_PRESENT	Module programmed
PSI_PART_INVALID	Module is invalid
PSI_PART_REVOKED	Module is Revoked

### Functional Description

Checks the PSI of a single module.

### Particularities and Limitations

See [2] section 12.5.10.1 "PSI States"

## AppIFblGetBaseModulePPRegion

### Prototype

```
void AppIFblGetBaseModulePPRegion (vuint8 mid, IO_PositionType *pPresPtnAddr, IO_SizeType *pPresPtnLen);
```

### Parameter

mid	Module ID
*pPresPtnAddr	Pointer to presence pattern address
*pPresPtnLen	Pointer to presence pattern length

### Return code

-	-
---	---

### Functional Description

Get the presence pattern address and length of the base module for the given module ID>

### Particularities and Limitations

-

### AppIFblUpdateChecksum

Prototype	
<code>tFblResult AppIFblUpdateChecksum (V_MEMRAM1 vuint16 V_MEMRAM2 * const checksum, SecM_LengthType regLen, const V_MEMRAM1 vuint8 V_MEMRAM2 * const regStartAddr);</code>	
Parameter	
*checksum	Pointer to current checksum value
regLen	Length of buffer
regStartAddr	Pointer to buffer to be added to checksum
Return code	
kFblOk	Checksum properly updated
kFblFailed	Failure updating the checksum
Functional Description	
This function adds the values in buffer to the current checksum value.	
Particularities and Limitations	
-	

### AppIFblFinalizeChecksum

Prototype	
<code>tFblResult AppIFblFinalizeChecksum (V_MEMRAM1 vuint16 V_MEMRAM2 * const checksum);</code>	
Parameter	
*checksum	Pointer to checksum value
Return code	
kFblOk	Checksum finalized correctly
kFblFailed	Failure finalizing checksum
Functional Description	
This function performs an operation on the checksum after it has been calculated. By default the operation is a twos-complement.	
Particularities and Limitations	

### AppIFblINVMReadKeyNBID

Prototype	
tFbIResult <b>AppIFb1NVMReadKeyNBID</b> (V_MEMRAM1 tNBIDInfo V_MEMRAM2* const keyNBIDInfo);	
Parameter	
*keyNBIDInfo	Pointer to key NBID info struct
Return code	
kFbIOk	Key NBID read successfully
kFbIFailed	Failure to read key NBID
Functional Description	
This function is used to read the stored key NBID from NVM.	
Particularities and Limitations	
-	

#### AppIFb1NVMWriteKeyNBID

Prototype	
tFbIResult <b>AppIFb1NVMWriteKeyNBID</b> (V_MEMRAM1 tNBIDInfo V_MEMRAM2* const keyNBIDInfo);	
Parameter	
*keyNBIDInfo	Pointer to key NBID info struct
Return code	
kFbIOk	Key NBID written successfully
kFbIFailed	Failure to write key NBID
Functional Description	
This function is used to write the key NBID that was received to NVM.	
Particularities and Limitations	
-	

#### AppIFb1NVMReadAppNBID

Prototype	
tFbIResult <b>AppIFb1NVMReadAppNBID</b> ( V_MEMRAM1 tNBIDInfo V_MEMRAM2 * const appNBIDInfo );	
Parameter	
*appNBIDInfo	Pointer to app NBID info struct
Return code	
kFbIOk	App NBID read successfully

kFbIFailed	Failure to read app NBID
<b>Functional Description</b>	
This function is used to read the stored app NBID from NVM.	
<b>Particularities and Limitations</b>	
-	

### AppIFbChkModulePresence

<b>Prototype</b>	
tFbIResult <b>AppIFbINVMWriteAppNBID</b> (V_MEMRAM1 tNBIDInfo V_MEMRAM2* const appNBIDInfo);	
<b>Parameter</b>	
*appNBIDInfo	Pointer to app NBID info struct
<b>Return code</b>	
kFbIOk	App NBID written successfully
kFbIFailed	Failure to write app NBID
<b>Functional Description</b>	
This function is used to write the app NBID that was received to NVM.	
<b>Particularities and Limitations</b>	
-	

### AppIFbINVMReadECUID

<b>Prototype</b>	
tFbIResult <b>AppIFbINVMReadECUID</b> (V_MEMRAM1 uint8 V_MEMRAM2 * const buffer);	
<b>Parameter</b>	
*buffer	Buffer to store ECUID
<b>Return code</b>	
kFbIOk	ECU ID read successfully
kFbIFailed	Failure to read ECU ID
<b>Functional Description</b>	
This function is used to read the ECU ID from NVM.	
<b>Particularities and Limitations</b>	
-	

### AppIFbINVMReadSBATicket

Prototype	
<pre>tFblResult ApplFblNVMReadSBATicket (V_MEMRAM1 uint8 V_MEMRAM2 * const buffer);</pre>	
Parameter	
*buffer	Buffer to store SBA ticket
Return code	
kFblOk	SBA ticket read successfully
kFblFailed	Failure to read SBA ticket
Functional Description	
This function is used to read the SBA ticket from NVM.	
Particularities and Limitations	
-	

## 5.4 Watchdog Callbacks

The watchdog is any hardware device (in most cases, built into the hardware of your ECU) that consists of a timer and a mechanism to reset the ECU when the timer expires. This allows the ECU to recover from situations that prevent the software from functioning properly (such as infinite loops). To prevent a reset, the software must periodically reset the watchdog-timer so that it does not expire.

The FBL configuration tool, GENy, provides a switch to enable or disable watchdog handling. When enabled, you must define how often the timer will be reset (also using GENy). The refresh-period you specify must be less-than the time-out period of the watchdog-timer.

GENy will define the constant `FBL_WATCHDOG_TIME` to represent the refresh-period in terms of how often tasks are performed in `FblRepeat()`. All operations within the bootloader must call either `FblLookForWatchdog()` or `FblRealTimeSupport()` (calls `FblLookForWatchdog()`, additionally may start triggering Response Pending) at least once per millisecond in order to maintain the watchdog. At the appropriate time, `FblLookForWatchdog()` will call the function `ApplFblWDTrigger()` (which you must implement) to reset the watchdog-timer.



### 5.4.1 Start of Watchdog

The watchdog timing is critical for three events: power-on reset of the bootloader, transfer of control from the bootloader to the application, and transfer of control from the application to the bootloader.

When a power-on reset occurs, instructions in the FBL will be performed first. If your ECU starts the watchdog-timer from power-on, you will have to initialize (or halt) the watchdog handling as soon as possible;

Normally, you should start the watchdog-timer in `ApplFblWDInit()`. Normally this function is called after the FBL has decided to stay in the bootloader, so there is no issue of the watchdog-timer expiring while starting the Operating Software (see section 3.3.1). However, you will also need to start the watchdog-timer in your Operating Software's startup-code. If you choose to start the watchdog-timer before application start add the configuration switch `FBL_ENABLE_PRE_WDINIT`, which will call `ApplFblWDInit()` early. You should also configure `FBL_ENABLE_PRE_TIMERINIT` in this case, to guarantee the watchdog triggering starts; this enables the triggering in `FblLookForWatchdog`. To start watchdog early may also be required if `ApplFblInit()` contains long lasting initialization (Eeprom-Manager).

Start watchdog early (before hw-init, before start of operating software)

Define

`FBL_ENABLE_PRE_WDINIT`,  
`FBL_ENABLE_PRE_TIMERINIT`

e.g. via GENy user preconfiguration file.

Start watchdog late (after hw-init, only when default executing boot)

### 5.4.2 Synchronize Watchdog with application

When the bootloader transfers control to the Operating Software, the watchdog-timer must have the maximum amount of time available. This is an issue if the watchdog is started by the hardware at power-on, or if you initialize the watchdog early. You must configure the watchdog hardware so that the watchdog time-out period is large enough to start the Operating Software. Before starting the Operating Software, the FBL will call `ApplFblWDLONG()`. You should implement `ApplFblWDLONG()` to reset the watchdog-timer and then return. In addition, if you enable the Stay-In-Boot feature, you must reset the watchdog-timer in `ApplFblWDLONG()` even if you start it late (alternatively disable it for this development feature).

### 5.4.3 Window Watchdogs

Some ECUs implement a *Windowed-Watchdog*. In this case, the watchdog-timer cannot be reset at any arbitrary time; it must be reset within a small time period (the *trigger-window*) immediately before the watchdog-timer expires.

When using a windowed-watchdog, you must set the refresh-period in GENy to occur within the trigger-window. When possible, set the refresh period in the middle of the trigger-window; you should avoid setting the period to refresh immediately after the

window opens, or immediately before watchdog-timer expires. In this configuration, it is especially important that the watchdog-timer and the refresh-timer are started at the same time.

In addition, you must take care that your implementation of `ApplFblWDLong()` waits until the trigger-window opens. This may be achieved by calling `FblLookForWatchdog()` continuously until it returns `FBL_WD_TRIGGERED`. Finally, you must devise a means of synchronizing the FBL's watchdog refresh-period with your Operating Software's refresh-period. One solution is to define and start an "elapsed-time" timer in `ApplFblWDLong()` that can be read by your Operating Software to determine when the trigger-window opens.

#### 5.4.4 Watchdog triggering during Memory operations

The bootloader faces a unique problem when erase and write operations are performed. In most cases, the ECU is unable to read instructions from the device while waiting for the erase/write to complete. Hence, the code to reset the watchdog-timer, which resides on the same device, cannot be executed. To resolve this issue, the bootloader makes a copy of `FblLookForWatchdog()` and `ApplFblWDTrigger()` in RAM. The device-driver calls the copy in RAM to maintain the system timers and refresh the watchdog-timer. To facilitate running from RAM, your implementation of `ApplFblWDTrigger()` should not contain any conditional operations, loops, or function-calls, especially if your compiler does not generate position-independent code.

Most ECUs initiate a device operation by setting a "command" register and then repetitively check a "status" register to indicate that the operation is complete. The device-driver will call the copy of `FblLookForWatchdog()` (in RAM) while waiting. However, in some systems the device operations are performed by calling a library function provided by the device manufacturer. In this case, the system timers cannot be maintained while a device operation is performed. You must configure the watchdog-timer period so that it is greater-than the worst-case time need to perform an erase or write operation. Additional callbacks may be available so that the elapsed time spent performing device operations can be accounted for. A detailed description of these functions will be provided in the Hardware Technical-Reference included with your delivery.

<code>ApplFblWDInit *</code>	<code>ApplFblWDLong *</code>	<code>ApplFblWDTrigger *</code>
------------------------------	------------------------------	---------------------------------

Table 5-5 Watchdog Callbacks

\* These routines are also described in reference [4].

#### ApplFblWdInit

Prototype	
<code>void ApplFblWDInit( void )</code>	
Parameter	
-	-
Return code	
-	-

## Functional Description

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

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

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

## Particularities and Limitations

> In addition to initializing the hardware responsible for the watchdog, the global variable `WDTimer` must be initialized to the number of “ticks” that define the interval between calls to the trigger function. The interval is determined from your configuration, and is defined by the macro `FBL_WATCHDOG_TIME`.

> Use of the watchdog is managed by the configuration switch “Watchdog Enable” (see also section 4.4). You should encapsulate your implementation within conditional-compilation directives to enable or disable the WD as appropriate. For example:

```
#if defined( FBL_WATCHDOG_ON )
    /* Enable Watchdog */
#else
    /* Disable Watchdog */
#endif
```

## ApplFblWDLONG

### Prototype

```
void ApplFblWDLONG( void )
```

### Parameter

-	-
---	---

### Return code

-	-
---	---

### Functional Description

The purpose of this function is to synchronize the start of the Operating Software with the watchdog. The call gives you the opportunity to ensure that the watchdog will not interrupt the Operating Software’s startup. The implementation should either wait for the next watchdog trigger-event (to maximize the period to the timeout), or disable the watchdog (Note: You should not disable the WD if your implementation of `ApplFblReset()` requires the WD to reset the ECU).

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

### Particularities and Limitations

- > The function is called at the end of a programming session and when the FBL jumps to the Operating Software directly after power-on. Care should be taken if the watchdog (WD) is initialized in `ApplFblWDInit()`, since this function could be called before the WD is started. See also Section 3.3.1.
- > Use of the watchdog is managed by the configuration switch “Watchdog Enable”. You should encapsulate your implementation within conditional-compilation directives to avoid waiting for a watchdog event that never happens. For example:

```
#if defined( FBL_WATCHDOG_ON )
    /* wait for watchdog event */
    /* !! Make sure WD is running before entering this loop (See
above)!! */
    /* !! This example does not contain the necessary check!! */
    while (FblLookForWatchdog() != FBL_WD_TRIGGERED)
        ;
#endif
```

### ApplFblWDTrigger

#### Prototype

```
void ApplFblWDTrigger( void )
```

#### Parameter

-	-
---	---

#### Return code

-	-
---	---

#### Functional Description

The purpose of this function is to reset the watchdog logic to prevent it's timer from resetting the ECU. The function will be called periodically (based on the “Watchdog time (ms)” entry in your configuration).

The implementation should not call any other function, nor reference any data outside of RAM!

Many non-volatile devices do not allow the ECU to fetch instructions (or any other data) from them while the device is being erased or programmed. Since the FBL often resides in the same device as the Operating Software and other downloaded modules, the routines to erase and write to the device are usually executed from RAM. The FBL also copies this function to RAM, so that it may be invoked from the device-driver while waiting for erase & write operations to complete. During this time any calls to routines resident in the device being programmed will lead to failure.

### Particularities and Limitations

- > Not all ECUs allow code to be executed from RAM. Please refer to your hardware-specific FBL documentation for implementation details.
- > Since this function is usually copied from the FBL to RAM, the code must be compiled as position-independent (relocatable). If your compiler does not support this feature, then the implementation in this function must not contain any conditional expressions (any code that results in non-sequential execution, such as calls, `if`, `do`, `while`, and `for` statements). See also Section 5.8.
- > This function can be called even if your configuration does not enable watchdog handling (for example, the function could be called when waking from sleep-mode).
- > You may use the conditional-compilation switch `FBL_WATCHDOG_ON` within your implementation to selectively compile the code.

## 5.5 Callback configuration summary

The Gm bootloader offers a lot of callbacks that are open for user modifications.

Some are required to be filled by you; some allow you to modify the behavior of the bootloader to your specific requirements. Many callbacks are only required for special use cases, or in case of hw-specific requirements. You also may reduce code size in changing them; e.g. by modifying generic implementations to a more specific use case. If you do not run into problems it is generally recommended to use our default implementations.

Several callbacks however need to be touched by you to complete the requirements demanded by the references [1] and [2]. These are listed in the following sub-chapter

### 5.5.1 Required callback configuration

These callbacks need to be checked or touched by you:

- > All hw-related callbacks required for hw-initialisation. These are
  - ▶ `ApplFblCanBusOff` (define Busoff behavior)
  - ▶ `ApplFblInit` (hw initialization)
  - ▶ `ApplFblReset` (hard hw-reset)
  - ▶ `ApplFblSetVfp` / `ApplFblResetVfp` (if programming voltage is required; rather rare)
  - ▶ `ApplTrcvrHighSpeedMode` / `ApplTrcvrSleepMode` / `ApplTrcvrNormalMode`
  - ▶ `ApplFblEnterStopMode` / `ApplFblSleepModeAllowed` (for low power mode)
  - ▶ `ApplFblRamIntegrityCheck` (if more than one check address region)
- > These Diagnostic callbacks:
  - ▶ `ApplFblReadDataByIdentifier` (check if DIDs are complete, add missing, test read them)
  - ▶ `ApplFblSecurityKey` / `ApplFblSecuritySeed` / `ApplFblCheckMecAndVFlag` (change required keypairs, add reading of flags if required )

- > These Module Validation Callbacks:
  - ▶ ApplFblGetBaseModulePPRegion, ApplFblGetModuleHeaderAddress  
(in case of multi-processor /reserved module configuration)
- > All watchdog related callbacks

## 5.6 Application Vector Table

In order to start the bootloader, the reset-vector in the ECU's interrupt vector must **always** point to the startup code of the FBL. To insure that the bootloader is always able to start up, the interrupt vector table should never be erased or reprogrammed. However, if the interrupt vector table cannot be programmed, how can the ECU invoke interrupt service routines in the Operating Software?

The solution to create a second structure, the Application Vector Table, that contains the instructions needed to jump to each interrupt service routine (ISR). This structure is linked with the Operating Software, and is programmed when the Operating Software is programmed. The ECUs vector table, programmed when the bootloader is programmed, will contain a fixed address for each interrupt that points to a corresponding entry in the Application Vector Table.



The location of the Application Vector Table cannot be changed once the FBL is programmed into your ECU. All subsequent downloads of the Operating System must use the same address for the Application Vector Table.

Your bootloader delivery contains two files, `applvect.c` and `fbl_applvect.c`. Each contains a definition of the application vector table. The file `fbl_applvect.c` should be linked to your flash bootloader, while `applvect.c` should replace the interrupt vector table in your Operating Software (see Section 6).

When running, the FBL does not use any interrupts. However, on many ECUs an interrupt is required to recover from sleep-mode (this depends on your implementation of `ApplFblEnterStopMode()`). If you require an interrupt to wake-up, but are not using the CAN-controller to manage this ("Enable sleep mode" is disabled in your configuration), you should edit the Application Vector Table in `fbl_applvect.c`, and modify the vector used by your wake-up interrupt to point to your ISR handler. If you are using the CAN-controller to wake-up, and have more than one CAN-cell, you may need to edit `fbl_applvect.c` to call `FblCanWakeUpInterrupt()` from the appropriate CAN vector.

For additional information on sleep-mode, please see the function descriptions for `ApplFblIsSleepModeAllowed()`, `ApplFblEnterStopMode()`, and `ApplFblCanWakeUp()`.

## 5.7 Transport-Layer Configuration

The Transport-Layer configuration switches are generated from GENy to the file `ftp_cfg.h`. There are no OEM specific features in this file. The configuration of the Transport layer usually should not be touched.. Contact us if you require changes.

## 5.8 [#hw\_wd] – Compiling the Watchdog components

Since the operations that erase and write the flash hardware may take some time, it is necessary for the flash driver to invoke the watchdog handling functions. These functions, initially in flash, may not be accessible while the flash is being erased or written. To resolve this issue, the FBL copies the watchdog functions, `FblLookForWatchdog` and `ApplFblWDTrigger` from flash to RAM.

To facilitate this, the source-modules containing this functions, `fbl_wd.c` and `fbl_apwd.c`, must be compiled as **position-independent** code when not copied via a linker mechanism. This directs the compiler and linker to create the code so that all branch and call instructions are relative to the current program-counter value. This allows the code to be executed from anywhere in the memory-map of the ECU.

For some compilers the code must be linked to execute from RAM, and be located in ROM. The mechanism to implement this depends highly on the compiler you are using. You should refer to the hardware-specific manual of your delivery for instructions on how to compile the watchdog components.

## 5.9 [#oem\_valfunc] – Flashing After A Reset

For more general information about this see the `UserManual_FlashBootloader` in the chapter **Your Application Initiates the Flashing Process**.

The GM FBL uses different names for the callback functions used to manipulate the validation area as those indicated in the general user manual.

Reference [4] defines the function `ApplFblValidateApp()` to write the signature that indicates that a module is present. The GM FBL uses the function `ApplFblValidateBlock()` instead of `ApplFblValidateApp()`.

Reference [4] defines the function `ApplFblInvalidateApp()` to remove the signature that indicates that a module is present. The GM FBL uses the function `ApplFblInvalidateBlock()` instead of `ApplFblInvalidateApp()`.

## 5.10 [#oem\_valid] – Proposals for Handling The Validation Area

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

The Vector FBL uses three functions to control the validation area:

`ApplFblIsValidApp()`, `ApplFblInvalidateBlock()`, and `ApplFblValidateBlock()`.

`ApplFblIsValidApp()` is called when the FBL is started up to determine if the Operating Software and all other required modules have been downloaded.

`ApplFblInvalidateBlock()` is called at the start of download so that you may update the mask and pattern used by `ApplFblIsValidApp()` to indicate that a module is not present.



`ApplFblValidateBlock()` is called when a download completes, so that you may update the mask and pattern used by `ApplFblIsValidApp()` to indicate that a module is present.

The implementation of the functions provided with the bootloader conforms to the requirements defined in “Programmed State Indicator (PSI)” (Section 12.5.10), which states::

- The PSI for the application SW shall be located in the last two bytes of application SW partition.
- For each calibration partition, there shall be one PSI assigned to that partition. The PSI shall be located in the last two bytes for that partition.
- The number of PSIs is equal to the number of partitions and not the number of programmable modules.

Conformance to the first two requirements imply separate validation areas are used for the Operating Software and Calibration data modules, and that the location of the validation areas cannot be set to a fixed address. The implementation of the three aforementioned API functions uses several local functions to write the presence patterns. The function `ApplFblGetPresencePatternBaseAddress()` determines the location of the presence pattern. In case of calibration files, the routine calls `FblHdrGetCalibrationPPRegion()` to determine the largest address used by the module, and map the address to an entry in the Flash-Block table to determine the address of the “last 2 bytes”. `ApplFblGetBaseModulePPRegion()` shall provide the addresses for Operational software and Reserved modules (GENy configured Operational Software 0x01 is preconfigured inside this callback, further modules have to be added manually). For more information about the Flash-Block table, see section 4.5.2.

At startup, `ApplFblIsValidApp()` checks both the mask and presence-pattern. Normally, the mask is in the erased state (nothing is written to the mask during download and programming). If the validation mask contains any non-erased value, the module will be considered invalid.

At the start of a download, `ApplFblInvalidateBlock()` will invert the erased state of the validation mask instead of erasing the entire sector containing the presence-pattern and Application/Calibration data.

Mask and pattern are located in the same block and have to be erased together. You must refer to the documentation supplied by Vector for your device-driver(s) to verify that the erase function is implemented (some drivers, notably the EEPROM drivers, do not implement the erase function – this is integrated into the write routines). In this case, you must modify `ApplFblValidateBlock()` to write the “erase” character to the validation mask.

You may choose to store the presence patterns separately (for example, in EEPROM). In this case, you must explicitly erase (or write the erase character, depending on your device-driver) the presence pattern in `ApplFblValidateBlock()`.

The bottom-line is that the last three bytes (and possibly more, at least two-segments worth of memory, depending on the segment-size of your non-volatile device) of the last



block used by each module (application and calibration) cannot be used to store downloaded data. The Fbl will send an NRC 85, Debug status kDiagErrPPOverlapping, if it is tried to write to this location.

### 5.11 [#oem\_start] Startup

The FBL is always started upon a power-on reset, and should be started via a reset (depending on the implementation of `ApplFblReset()`) when started by the Operating Software. When the programming session completes, the FBL will invoke `ApplFblReset()`, per the requirements of “ReturnToNormalMode (\$20) Service” (Section 8.5 in reference [1]).

Upon startup, the FBL will execute the compiler-specific startup code (which may be customized), and then control will be transferred to the main module.

The compiler-specific startup code is generally responsible for initializing the run-time environment, such as setting RAM to zero, copying the initial value of variables to RAM, and initializing the stack. In some hardware designs, it is necessary to modify the startup source module (for example some systems use an assembly-language file called `crt.s`) provided by your compiler vendor in order to initialize special registers, such as the watchdog, memory configuration, and system timing (PLL). In some instances, the special registers and/or instructions used in the startup code may only be executed immediately after reset.

When configuring the FBL, keep in mind that the ECU's reset vector will always point to the startup compiled as part of the bootloader. If your Operating Software uses non-default startup code, it may not be able to modify the special configuration registers or execute certain instructions. Attempting to do so may reset your ECU.

If your Operating Software contains a customized startup module, you should move/copy your initialization from the Operating Software to the bootloader.

The main module will perform some minimal hardware initialization, and then perform a check to determine if the bootloader is being started by the Operating Software. If not, the FBL will determine if the Operating Software is ready to run. If so, then the FBL will transfer control to the Operating Software via the reset jump vector in the Application Vector Table.

If the FBL is being started by the Operating Software, or if the Operating Software is not ready, the FBL will continue with hardware initialization (for example, start the CAN controller and hardware timers), and then proceed to the main-loop, where it will wait for CAN messages to arrive.

Initialization of the CAN controller depends highly on the FBL configuration, and on whether or not the FBL is started by the Operating Software. When started by the Operating Software, the FBL obtains the CAN controller settings (such as baud-rate and CAN-Identifiers) from a structure called the CAN Initialization Table. For details on starting the FBL from the Operating Software, please refer to Chapter 6.

## 5.12 [#oem\_ref] – Label Reference File

The only structures shared between the Operating Software and the FBL are the File-Header and Application Vector Table. Application header address is configured in GENy.

The address of the Application Vector Table must be known to the FBL when it is compiled. The link address of the structure **MUST** be the same in both the bootloader and Operating Software. Additional details regarding the Application Vector Table may be found in section 5.5.

## 6 Adapting the Operating Software

Typically, the Operating Software is initially developed as a stand-alone program, independent of the bootloader environment. To download the Operating S/W with the FBL, several changes need to be made.

For the following, many references are made to your 'Makefile' and 'Linker Directives'. You should interpret these as references to the tools in your development environment that control which files are compiled into the Operating System, how they are compiled, and where in memory the compiled objects are located. In many instances, the Integrated Development Environment (IDE) supplied by your compiler vendor is used to manage how the software is built.

There are four files that are shared between the Operating Software and the Flash Bootloader: `fbl_def.h`, `v_def.h`, `v_cfg.h`, and `fbl_cfg.h`. You should add the paths to folders containing these files to the file-search (include) path list in your Makefile. You must recompile both the FBL and your Operating Software when you reconfigure your bootloader since `fbl_cfg.h` is a generated (using GENy) file.

### STEP 1: **ADAPT START-UP CODE**

When the ECU is powered-up, register initialization, memory configuration, etc, will be handled by the startup code of the bootloader. If you have modified the startup code provided by the compiler-vendor to perform any hardware-specific I/O or register initialization, the modifications should be copied to the startup code of the FBL. Keep in mind that many ECUs contain registers that may be written only once.

### STEP 2: **REPLACE THE INTERRUPT VECTOR TABLE**

Remove the Interrupt Vector Table in use by the Operating Software. Add the Interrupt Service Routines for the Operating Software to the application vector table, found in `applvect.c`. Add `applvect.c` to the Makefile/Link-list for the Operating Software (set the start address to the same location as that used by the FBL).

### STEP 3: **ADD THE FILE-Container**

For this release, no scripts are available to add/modify the file containers. Example container files have been manually created and included with the delivery. Use these dummy files as long as possible while setting up your FBL. A tool to create the appropriate containers will be provided in later releases. Please contact us to ask for the development status. The tool can be provided in between releases as soon as it is available.

### STEP 4: **[#oem\_trans] – ADD DIAGNOSTICS TO INVOKE BOOTLOADER**

Refer to "Read Identification Information Process" (Section 7.4.1 in reference [1]). The Operating S/W must support all diagnostic services described in the download process up to and including "Step 23: RequestDownload". At a minimum, the application must support the following services:

ReadDataByIdentifier (\$1A), Identifiers \$B0 and \$C1; ReturnToNormalOperation (\$20); DisableNormalCommunication (\$28); RequestDownload (\$34); TesterPresent (\$3E); ReportProgrammedState (\$A2); ProgrammingMode (\$A5), sub-functions \$01, \$02, and \$03.

In addition, the application may be required to support the following services:

InitiateDiagnosticOperation (\$10); SecurityAccess (\$27), sub-functions \$01 and \$02.

When a request to download (service \$34) is received, the Operating S/W should initialize the structure defined by `tCanInitTable` and invoke the bootloader. The following table describes the fields found in the structure:

Name	Description
<code>TpRxIdHigh</code>	Most significant 8 bits of USDT Request Message ID. Note that the field is formatted to match the ID register in the CAN controller (not usually as an integer). To convert the ID to the proper format, you should initialize this field with the expression: $= (\text{canuint8}) (\text{MK\_ID}(\text{USDT\_Req\_to\_xxx}) \gg 8);$ Where <code>USDT_Req_to_xxx</code> is the CAN ID of the diagnostics request message to your ECU expressed as an integer. Refer your message database for the CAN ID value.
<code>TpRxIdLow</code>	Least significant 8 bits of USDT Request Message ID. Note that the field is formatted to match the ID register in the CAN controller (not usually as an integer). To convert the ID to the proper format, you should initialize this field with the expression: $= (\text{canuint8}) (\text{MK\_ID}(\text{USDT\_Req\_to\_xxx}) \& 0xFF);$ Where <code>USDT_Req_to_xxx</code> is the CAN ID of the diagnostics request message to your ECU. Refer your message database for the CAN ID value.
<code>TpTxIdHigh</code>	Most significant 8 bits of USDT Response Message ID. Note that the field is formatted to match the ID register in the CAN controller (not usually as an integer). To convert the ID to the proper format, you should initialize this field with the expression: $= (\text{canuint8}) (\text{MK\_ID}(\text{USDT\_Resp\_from\_xxx}) \gg 8);$ Where <code>USDT_Resp_from_xxx</code> is the CAN ID of the diagnostics response message from your ECU. Refer your message database for the CAN ID value.

TpTxIdLow	<p>Least significant 8 bits of USDT Response Message ID. Note that the field is formatted to match the ID register in the CAN controller (not usually as an integer). To convert the ID to the proper format, you should initialize this field with the expression:</p> <pre>= (canuint8) (MK_ID(USDT_Resp_from_xxx) &amp; 0xFF);</pre> <p>Where USDT_Resp_from_xxx is the CAN ID of the diagnostics response message from your ECU. Refer your message database for the CAN ID value.</p>
ProgrammedState	<p>Represents the initial return value used for service \$A2 (ReportProgrammedState). See Section 10.16.3 in reference [1]. In general, this is initialized to 0x00 to indicate that the ECU is fully programmed.</p>
ErrorCode	<p>Normally, the field is initialized as zero. The value is no longer used by the GM FBL.</p>
RequestProgrammingMode	<p>Identifies the Programming Mode request state. The FBL uses this value to initialize its data structures to the same state. Normally the FBL is started upon receipt of a \$34 service request. In this case, the value should be either</p> <pre>REQUEST_DOWNLOAD_MODE_LOWSPEED REQUEST_DOWNLOAD_MODE_HIGHSPEED</pre> <p>The value should be selected based on the programming mode (high vs. low speed) requested in the \$A5 service.</p>
RequestDownloadFormatID	<p>This field should be set to the value of the Data-Format-Identifier parameter in the service \$34 (RequestDownload) request. See also Section 10.12.2 in reference [1].</p>
requestDownloadMemorySize	<p>This field should be set to the value of the Uncompressed-Memory-Size parameter in the service \$34 (RequestDownload) request. See also Section 10.12.2 in reference [1].</p>
CAN-H/W Specific bus-timing initialization fields	<p>In addition to the above fields, the structure may contain one or more hardware-specific fields to initialize the bus-timing registers in the CAN controller. The value will also depend on whether high-speed or low-speed was selected. In some implementations, these fields may be ignored. Examples of how to initialize the structure may be found in the demo application provided with the FBL.</p>

Table 6-1 Parameters passed to FBL by Operating Software

An example of the table initialization and FBL call may be found inside the DemoAppl project we deliver (usually fbl\_jmpToFbl.c).

Once the table has been initialized, the FBL may be started by invoking the macro `CallFblStart()`. The Operating S/W should not send a positive-response to the download request (the response will be sent by the FBL). However, depending on the amount of time needed to start the FBL, you may need to send a Request-Correctly-Received; Response-Pending (RCR-RP) message.

**Caution**

In most cases, the FBL will ultimately be started by forcing the ECU to reset (depends on your implementation of `ApplFblReset()`).

If your ECU is not started via a reset, you must be careful that all register (such memory-mapping or timing) settings are consistent with the power-on settings required to run the FBL.

In addition, if a reset is not employed, you should be careful to avoid starting the FBL from an interrupt service routine (such as a transmit confirmation callback for the RCR-RP message).

If you are using the CANdesc component (with CANdela Studio), you should refer to the Vector Application Note: Calling the GM-Bootloader from CANdesc-Applications (AN-ISC-2-10100\_CANfblGM\_Call\_from\_CANdesc). This contains directions on how to invoke the FBL from the Operating Software.

**STEP 5: RESERVE SPACE FOR FBL AND PRESENCE-PATTERN(S)**

The Makefile/Linker directives should be modified to reserve space in flash for the bootloader and presence patterns. These areas must not be used by the Operating Software.

If a map file is available after re-compiling the Operating Software, you may want to verify that the addresses of the code and constant-data sections are within the bounds set in `fbl_apfb.c::FlashBlock`. The download will not succeed unless all sections mapped to flash memory fit within the regions defined in the flash block table. Adjust the Makefile/Linker directives of the Operating Software as needed.

**STEP 6: BUILD CALIBRATION DATA MODULES (optional)**

This step is only necessary if the FBL was configured to require modules in addition to the Operating Software. There are several ways of creating the data modules – it is up to you to decide upon the most appropriate action. In general, edit your Makefile to build new targets based on the files that define the additional modules. Headers for calibration files are generated together with the application header script.

## 7 Device Driver

### 7.1 General Information

The device-driver is responsible for erasing and writing data to non-volatile memory. This program is stored in the ECU flash memory and copied to RAM before downloading any other module. Most deliveries will include one device-driver capable of programming your ECU's internal flash-memory. The FBL will transfer the driver to a RAM array called `flashCode[]`. The size of the array, defined by the configuration tool, GENy, must be large enough to contain the driver.

Each device-driver is compiled as an independent program, although it does not contain ECU-startup code, nor a “main” entry point. Access to the routines should be performed by a set of high-level API functions defined by the FBL Memory Input/Output component (`fbl_mio`). See also section 7.2. The low-level API of the device-driver module is based on the HIS flash programming standard (see reference [9]).

The device-driver is compiled as either relocatable or non-relocatable code, depending on the hardware and development environment. Relocatable code is more flexible in that the code can be executed from anywhere in the memory space of the ECU; non-relocatable code has to be compiled to run from a specific RAM address: the starting address of `flashCode[]`.

If your driver supports relocatable code, the macro `FLASHCODE_RELOCATABLE` will be defined in `flashdrv.h`. In this case, the “starting” address of the driver link-file may be set to zero, since the FBL will relocate the downloaded code to `flashCode[]`.

If your driver does not support relocatable code, the “starting” address of the driver's link-file must correspond to the starting address of the `flashCode[]` array in the FBL. The FBL will reject the driver if the addresses do not match. In most cases, you should examine the address assigned to the link-section named “FLASHDRV” in the FBL, and use the same address in the link-section named “SIGNATURE” in the driver. Note that the section names will vary depending on the abilities of your development environment.

A script is provided to convert flash driver hex files into a C-array that can be compiled and linked with the bootloader.

### 7.2 High-Level Device-Driver Functions

All accesses to non-volatile memory should be performed via the bootloader's Memory-I/O component. The API redirects your request to the appropriate device-driver interface. The component is capable of reading from any device at any time, even if the device-driver has not been downloaded. However, the erase and write functions should not be called until the device-driver has been downloaded (See `GetMemDriverReady()`).

When the FBL has been configured to support multiple devices, some API functions require that you define and set a variable named `memSegment`. The variable should be set to the index of the flash-block table record that corresponds to the address of memory



that you are accessing. The routine `FblMemSegmentNrGet()` may be used to obtain the correct index.

### GetMemDriverReady

Prototype	
Single Device	
<code>vuint16 GetMemDriverReady( void )</code>	
Multiple Devices	
<code>vuint16 GetMemDriverReady( vuint8 device )</code>	
Parameter	
device	<p>Handle of device-descriptor table (see <code>fbl_apfb.c::memDrvLst[]</code> – the handle corresponds to the index into this array).</p> <p>The handle may be obtained from the flash-block table:  <code>FlashBlock[memSegment].device</code>, where <code>memSegment = FblMemSegmentNrGet(address)</code>.</p>
Return code	
-	<p>Zero is returned if the device-driver is not ready.</p> <p>Any non-zero value indicates that the device driver has been downloaded and initialized.</p>
Functional Description	
<p>This function is used to determine if the device-driver is to accept requests. You should verify that the driver is ready before calling any of <code>MemDriver_ReraseSync()</code>, <code>MemDriver_RwriteSync()</code>, or <code>MemDriver_VerifySync()</code>.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> <li>&gt; The function <code>MemDriver_InitSync()</code> must be called before using this function.</li> <li>&gt; The definition of this function is different when multiple device-drivers may be downloaded. The conditional-compilation switch <code>FBL_ENABLE_MULTIPLE_MEM_DEVICES</code> should be used to select between single-device and multiple-device configurations.</li> </ul>	

### MemDriver\_InitSync

Prototype	
<code>IO_ErrorType MemDriver_InitSync( void *address )</code>	
Parameter	
address	Usage depends on the device-driver (in most cases, the parameter is not used, so NULL should be passed). See also your Hardware Technical-Reference documentation.



**Return code**

IO_E_OK	Indicates that the device-driver(s) were successfully initialized. Any other value indicates that one or more drivers were not initialized. This is not necessarily an error condition. For example, when configured for multiple devices, the return value will indicate that only the device supported by the most recently downloaded driver is initialized.
---------	--

**Functional Description**

MemDriver\_InitSync() is used to initialize all available device-drivers.

The bootloader will call this function after downloading a device-driver.

**Particularities and Limitations**

- An initialization routine must be called before invoking the driver-interface functions GetMemDriverReady(), MemDriver\_RwriteSync(), MemDriverDevice\_RwriteSync(), MemDriver\_VerifySync(), or MemDriverDevice\_VerifySync(). **Note that MemDriver\_RreadSync() and MemDriverDevice\_RreadSync() may be called at anytime without initialization.**
- > This function may take a considerable amount of time to complete. See also the limitations for MemDriver\_ReraseSync().

**MemDriver\_ReraseSync****Prototype**

```
IO_ErrorType MemDriver_ReraseSync (
    IO_SizeType eraseLength,
    IO_PositionType eraseAddress
)
```

**Parameter**

eraseLength	Number of bytes to be erased. The eraseAddress + eraseLength should be aligned to the end of an erase-sector.
eraseAddress	Starting address of the region to be erased. The address must be aligned to the start of a sector-boundary.

**Return code**

IO_E_OK	Indicates that the region was successfully erased. Any other value represents a device-specific error code.
---------	--

**Functional Description**

MemDriver\_ReraseSync() is used to erase one or more sectors of non-volatile memory.

### Particularities and Limitations

- The function `MemDriver_InitSync()` must be called before using these functions.
- > Before calling these functions, you should call `GetMemDriverReady()` to determine that the device-driver has been downloaded and initialized.
- > When multiple-devices are supported by the FBL (`FBL_ENABLE_MULTIPLE_MEM_DEVICES`), you must set a local variable named `memSegment` before calling `MemDriver_ReraseSync()`. For example:  

```
memSegment = FblMemSegmentNrGet(eraseAddress);
```
- > These functions may take a considerable amount of time to complete. Maintenance of the watchdog-timer and P2Timer is handled inside delivered drivers. For you own drivers added please poll for `FblRealTimeSupport()` in a cycle <1ms.

### MemDriver\_RreadSync

#### Prototype

```
IO_ErrorType MemDriver_RreadSync (
    unsigned char* readBuffer,
    IO_SizeType readLength,
    IO_PositionType readAddress
)
```

#### Parameter

<code>readBuffer</code>	Pointer to RAM buffer where the retrieved data will be copied to.
<code>readLength</code>	Number of bytes to obtain.
<code>readAddress</code>	Starting (logical) address of memory to read.

#### Return code

<code>IO_E_OK</code>	Indicates that the region was successfully obtained. Any other value represents a device-specific error code.
----------------------	--

#### Functional Description

Use these functions to read data from non-volatile memory into RAM.

#### Particularities and Limitations

- > When multiple-devices are supported by the FBL (`FBL_ENABLE_MULTIPLE_MEM_DEVICES`), you must set a local variable named `memSegment` before calling `MemDriver_RreadSync()`. For example:  

```
memSegment = FblMemSegmentNrGet(readAddress);
```
- > These functions may take a considerable amount of time to complete. See also the limitations for `MemDriver_ReraseSync()`.
- > These functions may be called at any time without regard to whether or not the device-driver has been downloaded or initialized.

### MemDriver\_RwriteSync

**Prototype**

```
IO_ErrorType MemDriver_RwriteSync (
    unsigned char* writeBuffer,
    IO_SizeType writeLength,
    IO_PositionType writeAddress
)
```

**Parameter**

writeBuffer	Pointer to the source of data that is to be written to non-volatile memory. Many device-drivers require that the pointer is aligned to a specific (e.g. longword) boundary.
writeLength	Number of bytes to be written. The value must be a multiple of the devices write-segment size (See <code>MemDriver_SegmentSize()</code> ).
writeAddress	Starting address (logical) where the data will be written to. The address must be aligned to a segment boundary (i.e. The value should be a multiple of the write-segment size).

**Return code**

IO_E_OK	Indicates that the region was successfully written. Any other value represents a device-specific error code.
---------	---

**Functional Description**

Use these functions to write data to non-volatile memory.

**Particularities and Limitations**

- > The function `MemDriver_InitSync()` must be called before using these functions.
- > Before calling these functions, you should call `GetMemDriverReady()` to determine that the device-driver has been downloaded and initialized.
- > When multiple-devices are supported by the FBL (`FBL_ENABLE_MULTIPLE_MEM_DEVICES`), you must set a local variable named `memSegment` before calling `MemDriver_RwriteSync()`. For example:  

```
memSegment = FblMemSegmentNrGet(writeAddress);
```
- > These functions may take a considerable amount of time to complete. See also the limitations for `MemDriver_ReraseSync()`.

**MemDriver\_VerifySync****Prototype**

```
IO_ErrorType MemDriver_VerifySync( void *address )
```

**Parameter**

address	Starting address of the sector to be verified. NULL may be passed to verify all sectors that have been modified (erased or written) since the initialization of the driver, or since the last call to this function.  Note that on some systems, a (void *) cannot reach all addresses on the system. In this case, NULL must be passed as the argument. See also your Hardware Technical Reference.
---------	--

**Return code**

IO_E_OK	Indicates that the region was successfully verified. Any other value represents a device-specific error code.
---------	--

**Functional Description**

Use these function to verify the data written to non-volatile memory. This is required by some devices to ensure data-retention time.

**Particularities and Limitations**

- > The function `MemDriver_InitSync()` must be called before using these functions.
- > In most cases, the verify operation should be called only once after all erase and write operations have been completed.
- > These functions are not required by all device-drivers. See also your Hardware Technical Reference documentation.
- > Before calling these functions, you should call `GetMemDriverReady()` to determine that the device-driver has been downloaded and initialized.
- > When multiple-devices are supported by the FBL (`FBL_ENABLE_MULTIPLE_MEM_DEVICES`), you must set a local variable named `memSegment` before calling `MemDriver_VerifySync()`. For example:  

```
memSegment = FblMemSegmentNrGet(writeAddress);
```
- > These functions may take a considerable amount of time to complete. See also the limitations for `MemDriver_ReraseSync()`.

## 8 Using the Flash Tool for GM

vFlash is a Download Tool provided by Vector to be used for all OEM flash processes supported. It is a PC (Microsoft Windows<sup>™</sup>) based application that can transfer your Operating Software and other data modules to your ECU via the Flash Bootloader. The vFlash tool has to be purchased separately. The delivery includes a template for the GM use case that needs to be installed to be able to use vFlash with this bootloader.

### 8.1 Preparing the File-Header

Every module downloaded via the bootloader is required to have a File-Container. The container identifies the type of data, where the data should be stored, and so on. The format and contents of the header varies depending on the type of module

We recommend using our provided scripts to generate the required header information. A description of how to create the file containers using HexView can be found in [10].

### 8.2 Configuring vFlash

vFlash has several tabs that must be configured for your system.

#### 8.2.1 vFlash Communication Tab

The vFlash communication allows you to configure the communication settings of vFlash.

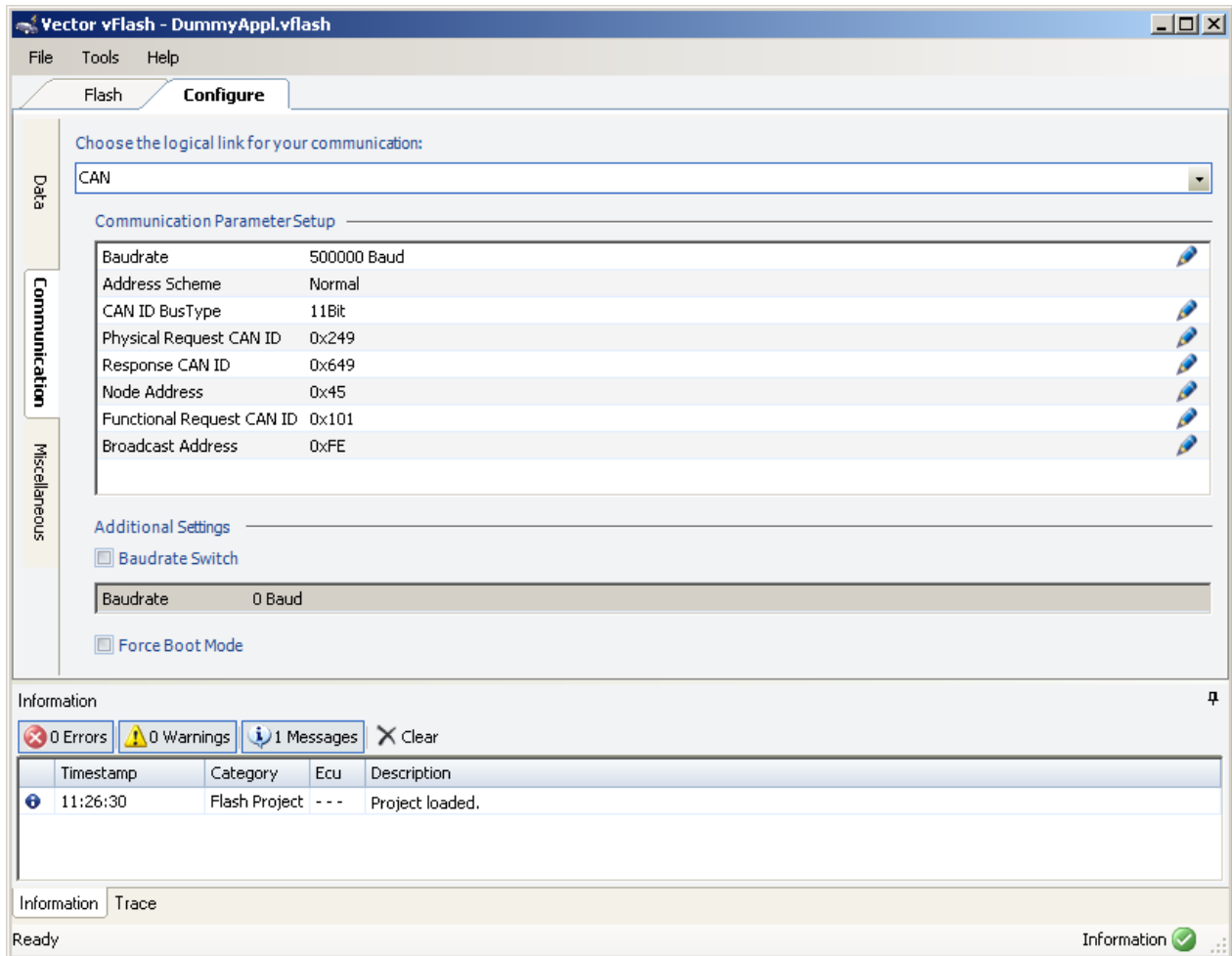


Figure 8-1 Example vFlash Communication Configuration Dialogue

Parameter	Configuration
Communication link	Only CAN is supported for this bootloader.
Baudrate	Baudrate of the bus that vFlash is connected to.
Address Scheme	Only Normal is supported for this bootloader.
CAN ID BusType	Length of CAN IDs used. Only 11Bit is supported for this bootloader.
Physical Request CAN ID	The physical receive ID of the ECU.
Response CAN ID	The physical response ID of the ECU.
Node Address	Node address of the ECU.
Function Request CAN ID	The functional (broadcast) ID of all ECUs on the bus.
Broadcast Address	The broadcast address used by a functional request. Normally 0xFE.
Baudrate Switch	When enabled, vFlash will request 'high-speed' mode during the download and switch its baudrate to the one specified in Additional Settings.

Table 8-1 Parameter description of the communication tab in vFlash.

## 8.2.2 vFlash Miscellaneous Tab

The vFlash miscellaneous tab allows you to configure additional options of vFlash.

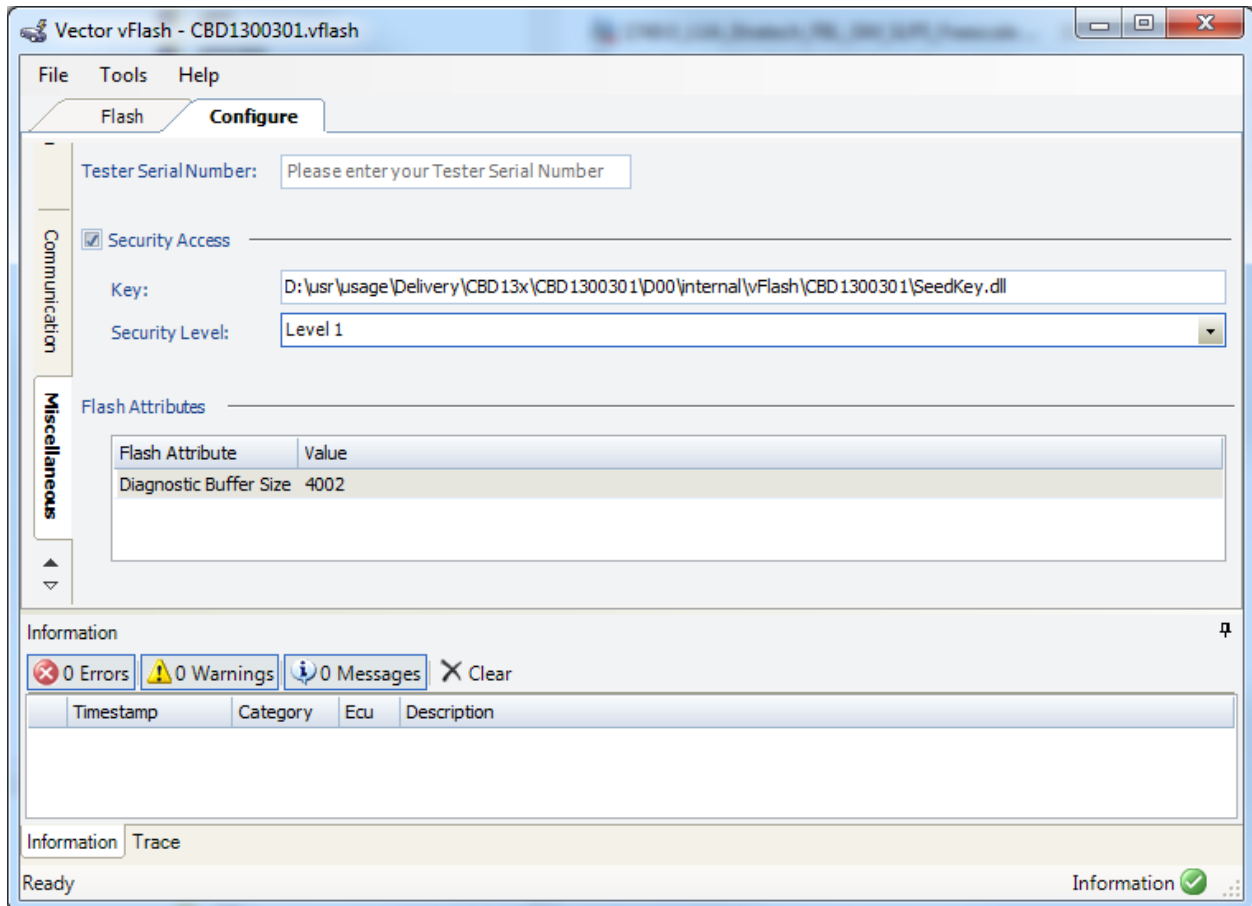


Figure 8-2 Example vFlash Miscellaneous Configuration dialog

Parameter	Configuration
Tester Serial Number	Not relevant for this bootloader.
Security Access	When enabled, vFlash will send the security access requests.
Key	DLL used to calculate the key from the seed received.
Security Level	Always Level 1 for this bootloader.
Diagnostic Buffer Size	Maximum size of transfer data requests that will be used by vFlash.

Table 8-2 Parameter description of the miscellaneous tab in vFlash

## 8.2.3 vFlash Data Tab

The vFlash data tab allows you to configure the modules to be downloaded to the ECU. The modules will be downloaded in the order specified.

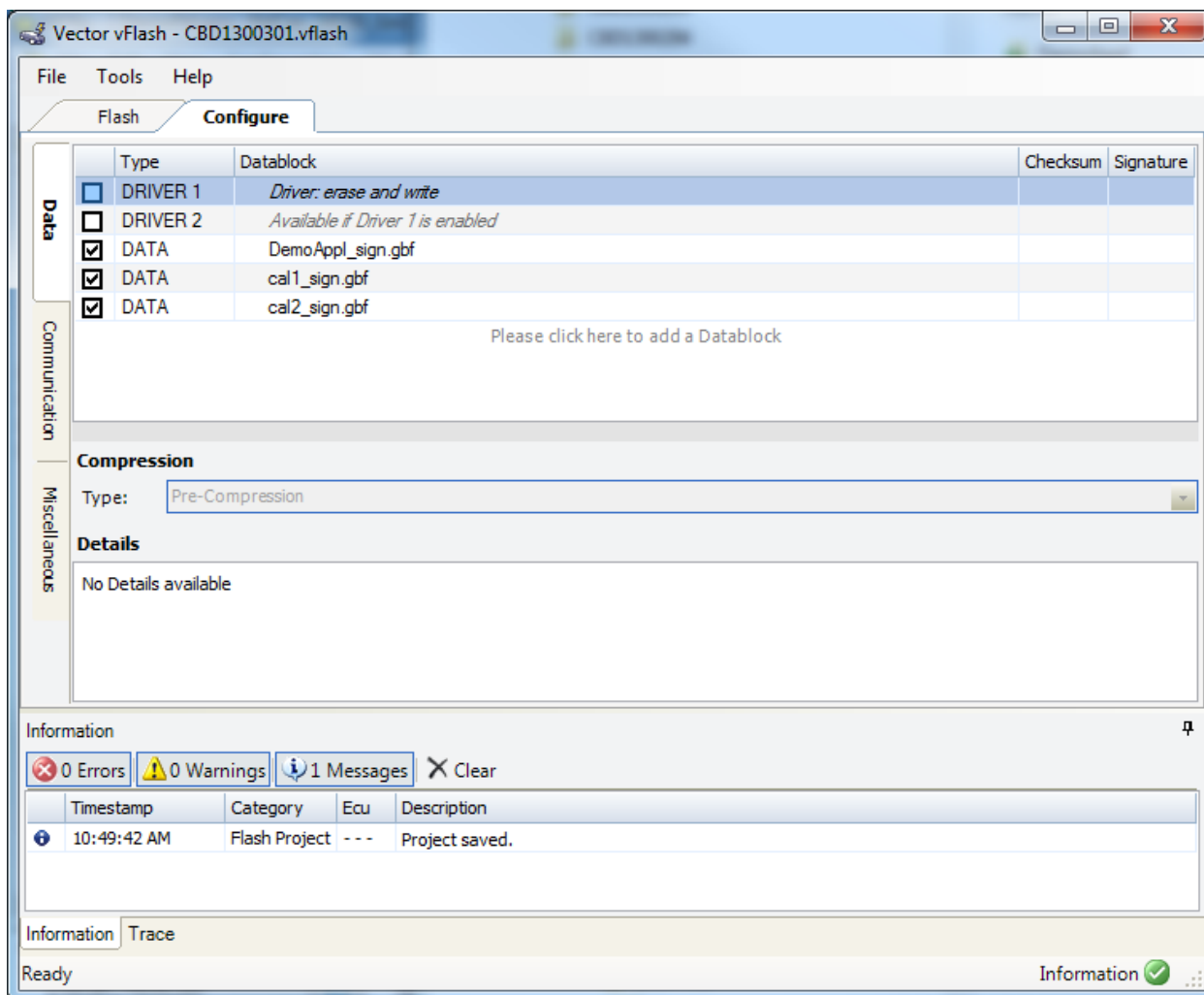


Figure 8-3 Example vFlash Data Configuration Dialogue.

**Note**

- Do not configure DRIVER1/DRIVER2. It is not required for the use case GM SLP5.

### 8.3 Starting the flash sequence with vFlash

To start the flash process, click the 'Flash' button in the upper right of the Flash tab in vFlash. The result of the flashing session will be displayed in the windows below.



## 9 Miscellaneous

### 9.1 [#oem\_multi] – Multiple-Identity-Modules

For more general information about this see the UserManual\_FlashBootloader in the chapter **Multiple ECU Support**.

Multiple Identity Modules (MIM) support is enabled when multiple nodes are selected in the configuration tool (GENy) (see Section 4.2). When enabled, you must add code to the FBL to determine the identity of the module, and select the appropriate CAN identifiers.

The FBL calls the function `ApplFblCanParamInit()` to obtain this information. You must implement this function to initialize the FBL's structures appropriately. Please refer to the description of `ApplFblCanParamInit()` for details.

### 9.2 Multiple Processor Support

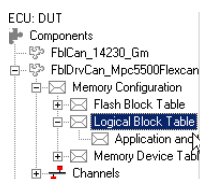


#### Caution

Multiple Processor support is not tested and thus cannot be configured by default if the delivery has not been, ordered, prepared, and tested explicitly for this use case. Contact Vector if this use case is required.

For multiple processor support, please configure the `FblCan_14230_Gm->GM Modules configuration->(Multiple application modules support)` option.

You will have to add entry(s) for your additional application + calibration areas. Please note that you will have to configure `Header Address` and `Presence Pattern Address` configuration describing application header start and validity info location. These items are returned inside the callbacks `ApplFblGetModuleHeaderAddress()` and `ApplFblGetBaseModulePPRegion()`.



Name	Block Index	Disposability	Start Address	End Address	Header Address	Presence Pattern Address	Verification RAM	Verification ROM	Description
Application and Calibration Area1	Application and Calibration Area1	0x1*	mandatory	0x10000	0x80fff	0x10000	0x80fff0	FblHdPipelineVerifyIntegrity*	FblHdVerifyIntegrity*

Figure 9-1 Multi-Processor logical block table configuration

#### Attributes:

- **Block Index:** Please configure Block Index attribute for the required module ID (OpSw1 0x01, OpSw2 0x21, OpSw3 0x31, OpSw4 0x41).
- **Other Attributes:** Compare Logical Block Table configuration.

Multi-processor solutions typically use a communication protocol to allow flashing of a slave ECU from a master ECU. The physical layer will typically be CAN/LIN/SPI. The required protocols are not part of the standard delivery. Please contact us to discuss possible solutions.

### 9.3 Request-Data-By-Identifier (Debug-Status)

Many diagnostics services provide very limited information regarding the cause of an error (for example, “General Programming Failure”). If desired, more detailed information may be obtained following a negative response. When enabled by your configuration (Project State “Integration”), the FBL will respond to a Read-Data-By-Identifier request with Data-Identifier (DID) \$7F. The response contains information regarding the cause of the most recent negative response. In any case the Fbl will answer with some extended information on GM specified DIDs F1 (PEC error code) and F2 (Boot Initialization status). These include already helpful information.

The Read-Data-By-Identifier request for DID 7F for debugging status returns the following information additionally (see also the description of `ApplFblInitErrStatus()`):

Byte	Description
0	PCI: Least-significant 4 bits indicates number of bytes in message frame.
1	Positive Response ID: \$5A
2	Data Identifier: \$7F
3	Service ID of last failed request ( <code>errStatLastServiceId</code> )
4	Failure Reason ( <code>errStatErrorCode</code> ) Possible values are defined in <code>fbl_diag.h</code> : See macros <code>kDiagErr&lt;fault&gt;</code> .
5-7	Optional: most-significant three bytes of error address ( <code>errStatAddress</code> ). Alternatively, if <code>errorStatErrorCode</code> indicate a Transport-Protocol error ( <code>kDiagErrTPFailed</code> ), a single byte value is supplied returning the TP error code ( <code>errStatTpError</code> ). In this case, the next field (file-name) starts at byte 6.
8-n	Zero-terminated string of ASCII characters identifying name of the file in which error was detected.
n-end	Line-number in file that detected error (number-of-bytes is ECU specific).

Table 9-1 Response for debug-status request

The information regarding a failure is stored via the functions `FblErrDebugStatus()` and `FblErrDebugDriver()`, as well as the macros `FblErrStatSetSid`, `FblErrStatSetState`, `FblErrStatSetFlashDrvError`, and `FblStatSetError`.

Please see `fbl_diag.h` for more information on the error status codes.

**Note**

You can retrieve additional error information reading the Programming Error Code (see section 12.6 of [2]).

## 9.4 [#oem\_time] – Stay-In-Boot mode

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

A common feature of Vector bootloaders is the ability to stay in the bootloader even if the Operating Software is ready to run (`ApplFblIsValidApp()` returns `kApplValid`). For production, GM requires you to disable this feature. However, it may be very helpful during development.

For example, you download an Operating Software module that contains an issue that causes the ECU to reset. The result is that you will be unable to replace the module, since the FBL will start the Operating Software, which then resets. This starts the FBL, which starts the Operating System again, over and over. In this situation, there is no means to start a download.

To escape this situation, the Stay-In-Boot feature inserts a small delay between the time the ECU is reset, and the time the FBL starts the Operating Software. If a “ping” message is received during this delay, the FBL will stay in its main loop instead of starting the Operating Software. Once the FBL is executing its main loop, a new module may be downloaded normally.

To enable this feature, you must configure the Stay-In-Boot feature in the `FblDrvCan_<Hw>` component.

When enabled, the FBL will respond to a Read-Data-By-Address (\$1A) service request containing the Data-Identifier (DID) of \$79. This message must be sent using the functionally-addressed CAN-Id (\$101). The message must be sent less-than 12 milliseconds after resetting the ECU. The default delay time may be set by uncommenting the `FBL_START_DELAY` defined above, and replacing ‘12’ with the desired delay time in milliseconds.

## 9.5 User-Callable Support Functions

The following documents commonly used functions in the FBL that may be invoked from callback functions.

### FblStart

#### Prototype

```
void FblStart( tCanInitTable *pCanInitTable )
void CallFblStart( tCanInitTable *pCanInitTable )
```

Parameter	
pCanInitTable	Pointer to CAN Initialization table in Operating S/W memory. NULL may be passed if Operating S/W has directly initialized the FBL initialization table.
Return code	
-	-
Functional Description	
<p>FblStart() is normally called indirectly to start the bootloader. The Operating Software will usually invoke CallFblStart() upon receipt of the Request-Download (service \$34) request. The macro will lookup the address of FblStart() and transfer control to the FBL.</p> <p>FblStart() will copy the CAN Initialization Table from memory in the Operating Software to memory controlled by the FBL. Next, the function will set the array FblStartMagicFlags[] to indicate that the FBL is being started by the Operating Software. Finally, the function will call ApplFblReset() to reset the ECU and start the FBL. See also section 3.3.2.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> <li>&gt; The download-tool will expect a response to the Request-Download (service \$34) within P2<sub>CE</sub> milliseconds (typically 100ms). Since the Operating S/W does not send the response, attention should be paid to the amount of time needed to start the FBL. If the amount of time needed by the FBL to startup and send the response exceeds P2<sub>CE</sub>, the Operating S/W should send a Response-Pending (RCR-RP) message and wait for transmit-confirmation before invoking CallFblStart(). The RCR-RP response will direct the download-tool to wait an additional 5000ms for the final response.</li> <li>&gt; FblStart() does not maintain the watchdog timer. Care should be taken to ensure that the watchdog timer does not expire before the time to reset the ECU. To avoid this, the Operating Software should call FblStart() immediately after resetting the watchdog timer to make the maximum time available to the FBL.</li> </ul>	

## FblReadMem

Prototype	
tFblLength <b>FblReadProm</b> (tFblAddress address, vuInt8* buffer, tFblLength length)	
Parameter	
address	Logical address of memory to read.
Buffer	Pointer to RAM where the values from source memory will be saved at.
Length	Number of bytes to copy from source memory.
Return code	
Read length	If memory was successfully read.
Functional Description	
<p>The function copies the contents of memory starting at the specified logical address to the specified destination using the appropriate device-driver (via MemDriver_RreadSync()). Logical addresses are defined by the Flash-Block table found in fbl_apfb.c.</p> <p>If the source address does not correspond to an entry in the Flash-Block table, the routine will treat the address as a physical address; the data will be copied in a manner similar to a call to memcpy().</p> <p>While copying data, FblRealTimeSupport() or FblLookForWatchdog() may be called to support response-pending messages and watchdog handing.</p>	

### Particularities and Limitations

- > The address-range of physical memory should not overlap the address-range of logical memory (logical memory is defined by the Flash-Block table).

### GetDiagInProgress

#### Prototype

```
Boolean GetDiagInProgress( void )
```

#### Parameter

-	-
---	---

#### Return code

False	If the FBL is not currently processing a diagnostics request.
True	If the FBL is currently processing a diagnostics request.

#### Functional Description

This macro will indicate if a request is being processed.

#### Particularities and Limitations

None

### FblRealTimeSupport

#### Prototype

```
vuint8 FblRealTimeSupport( void )
```

#### Parameter

-	-
---	---

#### Return code

FBL_NO_TRIGGER	Indicates that the watchdog timer was not reset during this call.
FBL_WD_TRIGGERED	Indicates that the watchdog timer was reset during this call.

#### Functional Description

This function maintains operations that must occur in real-time:

1. The watchdog timer is updated as needed (See `FblLookForWatchdog()`).
2. The response-pending message is sent as needed (See `DiagExRCRResponsePending()`).

The function must be called at least once per millisecond.

#### Particularities and Limitations

- > This function should usually be called rather than `FblLookForWatchdog()`, as P2 timing handling is included if required.

### FblLookForWatchdog

#### Prototype

```
vuint8 V_API_NEAR FblLookForWatchdog( void )
```

Parameter	
-	-
Return code	
FBL_NO_TRIGGER	Indicates that the watchdog timer was not reset during this call.
FBL_WD_TRIGGERED	Indicates that the watchdog timer was reset during this call.
Functional Description	
<p>This function manages the hardware-specific and P2 (response-pending) timers. The function must be invoked at least once per millisecond.</p> <p>The function also checks the hardware timer to determine if it is time to reset the watchdog timer. If true, the function will invoke <code>ApplFblWDTrigger()</code> to reset the watchdog timer.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> <li>&gt; No action is taken unless the hardware timer is running (the timer is started by <code>FblTimerInit()</code>).</li> <li>&gt; No action is taken unless the watchdog handler has been initialized (via <code>FblInitWatchdog()</code>).</li> <li>&gt; Watchdog support is provided only if enabled in the FBL configuration (<code>fbl_cfg.h</code> contains <code>FBL_WATCHDOG_ON</code>).</li> <li>&gt; Note that to support watchdog handling while writing or erasing non-volatile memory, this function and <code>ApplFblWDTrigger()</code> are normally executed from RAM (most types of flash cannot be read while an erase or write operation is in progress). See also section 5.8.</li> </ul>	

### DiagExRCRResponsePending

Prototype	
<code>void DiagExRCRResponsePending( vuint8 forceSend )</code>	
Parameter	
forceSend	<p><code>kNotForceSendResponsePending</code> Response-Pending message will be sent only if it is time to do so (P2 timer is less-than 10 milliseconds).</p> <p><code>kForceSendResponsePending</code> Response-Pending message will be sent unconditionally.</p>
Return code	
-	-
Functional Description	
<p>The function will send a Request-Received-Correctly-Response-Pending message when the response-pending (P2) timer is near or at zero. Once the message has been queued for transmission, the P2 timer will be reset (5000 milliseconds if download is not in progress, 30000 milliseconds if a download is in progress).</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> <li>&gt; This function should be called only if the FBL is handling a diagnostic request. See also <code>FblRealTimeSupport()</code> and <code>GetDiagInProgress()</code>.</li> </ul>	

### FblMemSegmentNrGet

**Prototype**

```
vsint16 FblMemSegmentNrGet( FBL_ADDR_TYPE address )
```

**Parameter**

address	Logical address of memory to map to Flash-Block table
---------	---

**Return code**

-1	Indicates that the Flash-Block table does not contain an entry corresponding to the requested address.
0 – (kNrOfMemDrv-1)	Index of Flash-Block table entry corresponding to the requested address.

**Functional Description**

This function is used to search the Flash-Block table (`fbl_apfb.c::FlashBlock[]`) for an entry that corresponds to a specified address.

This may be used, for example, to find address-boundaries for erase operations, or to help determine where a module's presence-pattern may go.

**Particularities and Limitations**

> The Memory-I/O functions (`MemDriver_ReraseSync()`, `MemDriver_ReadSync()`, `MemDriver_RwriteSync()`, `MemDriver_VerifySync()`, and `MemDriver_SegmentSize()` require a variable named `memSegment` to contain the block-table index to be set before being called if configured to support multiple-modules (`fbl_cfg.h` contains `FBL_ENABLE_MULTIPLE_MODULES`). This function may be used to obtain the index.

**GetFbl<XXX>Version****Prototype**

```
vuint8 GetFblMainVersion( void )
vuint8 GetFblSubVersion( void )
vuint8 GetFblReleaseVersion( void )
```

**Parameter**

-	-
---	---

**Return code**

-	FBL version information (0x00 – 0xFF)
---	---------------------------------------

**Functional Description**

These macros return the version identifiers from the bootloader's File-Header. The version information is composed of three parts, a main-version, sub-version, and release-version.

The values are associated with the macros in `fbl_diag.h` named `FBLOEM_GM_VERSION` and `FBLOEM_GM_RELEASE_VERSION`

**Particularities and Limitations**

> These macros may be invoked from your Operating Software implementation (see `fbl_def.h`).

## GetFblDCID&lt;X&gt;

Prototype	
vuint8 <b>GetFblDCID0</b> ( void )	
vuint8 <b>GetFblDCID1</b> ( void )	
Parameter	
-	-
Return code	
-	Data-Compatibility-Identifier (0x00 – 0xFF)
Functional Description	
<p>These macros return the Data-Compatibility-Identifier values from the bootloader's File-Header (see references [1] and [2]).</p> <p>GetFblDCID0 () returns the most-significant byte while GetFblDCID1 () returns the least-significant byte.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> <li>&gt; The range of GetFblDCID0 () is restricted to 0x80 – 0xFF, while GetFblDCID1 () may return any value in the range of 0x00 – 0xFF.</li> <li>&gt; These macros may be invoked from your Operating Software implementation (see fbl_def.h).</li> </ul>	

## GetFblSWMI&lt;X&gt;

Prototype	
vuint8 <b>GetFblSWMI</b> ( vuint8 index )	
vuint8 <b>GetFblSWMI0</b> ( void )	
vuint8 <b>GetFblSWMI1</b> ( void )	
vuint8 <b>GetFblSWMI2</b> ( void )	
vuint8 <b>GetFblSWMI3</b> ( void )	
Parameter	
index	Index into the SWMI array (typically 0 – 3, but may be up to 15).
Return code	
-	Software-Module-Identifier (0x00 – 0xFF)
Functional Description	
<p>These macros return the Software-Module-Identifier values from the bootloader's File-Header (see references [1] and [2]).</p> <p>GetFblSWMI0 () returns the most-significant byte, while GetFblSWMI3 () returns the least-significant byte.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> <li>&gt; Reference [2] constrains the SWMI field contain a 4-byte integer representing the FBL Part-Number.</li> <li>&gt; These macros may be invoked from your Operating Software implementation (see fbl_def.h).</li> </ul>	

## GetFblDLS&lt;X&gt;



Prototype	
<pre> vuint8 <b>GetFblDLS0</b>( void ) vuint8 <b>GetFblDLS1</b>( void ) </pre>	
Parameter	
-	-
Return code	
-	Design-Level-Suffix, also known as Alpha-Code (0x00 – 0xFF)
Functional Description	
<p>These macros return the Design-Level-Suffix values from the bootloader's File-Header (see references [1] and [2]).</p> <p><code>GetFblDLS0()</code> returns the most-significant byte, while <code>GetFblDLS1()</code> returns the least-significant byte.</p>	
Particularities and Limitations	
<p>&gt; These macros may be invoked from your Operating Software implementation (see <code>fbl_def.h</code>).</p>	

#### GetFblEcuNameAddr

Prototype	
<pre> vuint8 * <b>GetFblEcuNameAddr</b> ( void ) </pre>	
Parameter	
-	-
Return code	
-	Address of the array containing the ECU Name
Functional Description	
Particularities and Limitations	
<p>&gt; These macros may be invoked from your Operating Software implementation (see <code>fbl_def.h</code>).</p>	

#### GetFblSubjNameAddr

Prototype	
<pre> vuint8 * <b>GetFblSubjNameAddr</b> ( void ) </pre>	
Parameter	
-	-
Return code	
-	Address of the array containing the ECU Subject Name
Functional Description	

### Particularities and Limitations

- > These macros may be invoked from your Operating Software implementation (see fbl\_def.h).

### GetFblEcuIdAddr

#### Prototype

```
vuint8 * GetFblEcuIdAddr ( void )
```

#### Parameter

-	-
---	---

#### Return code

-	Address of the array containing the ECU ID
---	--

#### Functional Description

#### Particularities and Limitations

- > These macros may be invoked from your Operating Software implementation (see fbl\_def.h).

### FblBytesTo<XXX>

#### Prototype

```
vuint16 FblBytesToShort( vuint8 hi, vuint8 lo )
vuint32 FblBytesToLong( vuint8 hiWrd_hiByt,
                        vuint8 hiWrd_loByt,
                        vuint8 loWrd_hiByt,
                        vuint8 loWrd_loByt
                        )
```

#### Parameter

-	-
---	---

#### Return code

-	Word or long-word representation of byte array in ECUs native format.
---	---

#### Functional Description

The macros convert a set of bytes into a 16-bit (`FblBytesToShort()`) or 32-bit (`FblBytesToLong()`) quantity. The resulting value is in the ECUs native representation (Big-Endian or Little-Endian).

#### Particularities and Limitations

- > These macros may be invoked from your Operating Software implementation (see fbl\_def.h).
- > The macros behavior is modified for 8 and 16-bit big-endian ECUs (check `v_cfg.h` for `C_CPUYPE_BIGENDIAN` and `C_CPUYPE_8BIT` or `C_CPUYPE_16BIT`). In this case, only the 'hi' or 'hiWrd\_hiByt' parameter is used, the other parameters are ignored. For this configuration, the high-byte parameter is assumed to be the first byte of a `vuint8` array that will be accessed to obtain the `vuint16` or `vuint32` result.

## GetFblMode

## Prototype

```
vuint8 GetFblMode( void )
```

## Parameter

-

-

## Return code

START_FROM_APPL	Indicates that the FBL was started by request of the Operating Software as a result of receiving a Request-Download (service \$34) request.
START_FROM_RESET	Indicates that the FBL has been started via a power-on reset.
APPL_CORRUPT	Reserved for future use.
STAY_IN_FLASHER	Set when application is valid and Stay-In-Boot feature is enabled. Indicates that FBL is waiting for "ping" message.
FBL_START_WITH_RESP	Reserved for future use.
FBL_START_WITH_PING	Indicates a Stay-in-Boot ping message has been received. The Diagnostics-Layer will set the security-access-delay-timer to zero and send a response.

## Functional Description

This macro returns information about the current bootloader state. The returned value is a bit-mask; multiple bits may be set. You should perform a bitwise-AND to determine if the bootloader is in a particular state. For example:

```
if ((GetFblMode() & START_FROM_RESET) != (vuint8)0)
{
    /* Starting from reset (not from Operating Software) */
}
```

## Particularities and Limitations

> None

## 9.6 Low Power Mode in the bootloader

There are 3 basic configurations for the FBL sleep mode. They are as follows:

- > Integrated sleep mode enabled
- > Integrated sleep mode enabled with wakeup interrupt
- > Integrated sleep mode handling disabled

Each of the configurations are detailed below

.

### 9.6.1 Integrated sleep mode enabled

Check the checkbox in GENy for “Sleep Mode”. The Ecu wakes up on CAN-message reception (*AppIFblCanWakeUp()* is called on reception).

Only configurable if CAN-cell supports low power mode/wakeup on CAN-message reception.

Adapt *AppIFblSleepModeAllowed()* to return *kFblOk* if conditions are correct to go to sleep. If a user-specific reason exists to not go to sleep, then return *kFblFailed*.

### 9.6.2 Integrated sleep mode enabled with wakeup interrupt

9.6.1 Integrated sleep mode enabled above applies, with the difference that a real interrupt has to be generated for the given controller to wake up on CAN-message reception.

The tag `FBL_ENABLE_WAKEUP_INT` must additionally be defined ( via GENy-preconfig file ). This configuration should only be used if a non-interrupt based configuration is not possible for a given controller.

This configuration requires modification to the vector tables and callbacks. Be sure the required measures to again wakeup the CAN-cell are handled. See the Note on “Using wake up Interrupt” comment below for further configuration aspects and potential problems that may come along.

### 9.6.3 Integrated Sleep mode handling disabled

Simply uncheck the checkbox in GENy for “Sleep Mode”.

This is also the configuration to use if a user-specific low power mode / wake-up mechanism other than CAN-message reception is to be used (e.g. when a transceivers that allows turning of the ECU power is used.)

You may adapt *AppIFblSleepModeAllowed()* to return *kFblFailed* if no low power mode shall be used at all .



### Using Wakup Interrupt

**Using a wakeup interrupt is not recommended as long as there are other possibilities for waking up again without interrupt.**

Interrupts should be globally enabled inside *ApplFblEnterStopMode()* if Can Wakeup interrupt is to be used. CAN Wakeup interrupt is enabled in *FblCanSleep()*. All other peripheral interrupts except the CAN Wakeup interrupt should stay masked ( disabled ).

A sleep mode implementation using a wakeup interrupt should only be used before any application is ever programmed if the wakeup interrupt is called from reprogrammable memory ( this usually can only be avoided if there is a configurable vector table base address register ). This is to avoid enabling interrupts while there potentially is a corrupted application vector table.

The function *ApplFblSleepModeAllowed()* should call the function *FblCheckBootVectTablesValid()* to verify that it returns true ( no application was ever programmed ).

There is a potential deadlock problem when enabling global interrupts if the wakeup interrupt is served before the controller itself goes to low power mode: If the controller goes to low power mode after the CAN cell woke up, the ECU may not be able to wake up any more without externally resetting it. It must be verified if this is applicable for any given controller.

## 9.7 Example / hints to prepare containers for Type A Ecus

If you send your Ecu to the field as a Type A Ecu you may prepare the files to be programmed to your ECU as follows

- > Add the presence pattern information to the end of the last block touched the same way the bootloader writes the information for each module. This depends on the controllers minimum write size. Check in a debugger for the correct location and extract the pattern + mask information to a hexfile (e.g. insert in hexview). Verify that the application is started when the container including the patterns is programmed.
- > Merge the bootloader and all modules with presence patterns to a single file, fill the gaps with the fill pattern configured in the bootloader to fulfill the “unused bytes” requirement, compare section 7.5 in reference [2].

The same configuration can be used to load bootloader and application together to a debugger in order to debug application during development without having to program the application via CAN – this may be desirable during development.

## 9.8 Security Requirements

The bootloader is required to support two security services in order to keep malicious or unauthorized content out of the ECU [2]. See [2] for detailed descriptions of the required security handling.

- > Authentication: To ensure the content is genuine as authorized and released by GM [2].
  - > This is established through the digital signature and signer info.
- > Integrity: To ensure the content cannot be modified (intentionally or unintentionally) without being detected [2].
  - > This is established through the message digest.

The bootloader is also required to support two additional security features.

- > Application Software – Not Before Identifier (App-NBID)
- > Security Key – Not Before Identifier (Key-NBID)



---

**Note**

Refer to [10] for information on creating plain and signed headers.

---

### 9.8.1 Digital Signature

The digital signature is included as a parameter of the signed header. This applies to both application and calibration modules. The digital signature is used to check that the content of the remaining parameters of the signed header are correct (e.g. authorizes the module with this signed header can be programmed to flash). The bootloader uses the public key stored in the signer info to validate the digital signature.

The digital signature needs to be updated each time a module is updated (e.g. application is modified and re-compiled). The private key must be known to update the digital signature.

During development, Vector provides an example public/private key combination that can be used to generate the digital signature. This process is described in [10].

During production, the private key is only known to GM and therefore only GM can generate the digital signature. In this instance, GM is provided with modules that contain only the plain header. GM then 'signs' these files by adding the signed header.

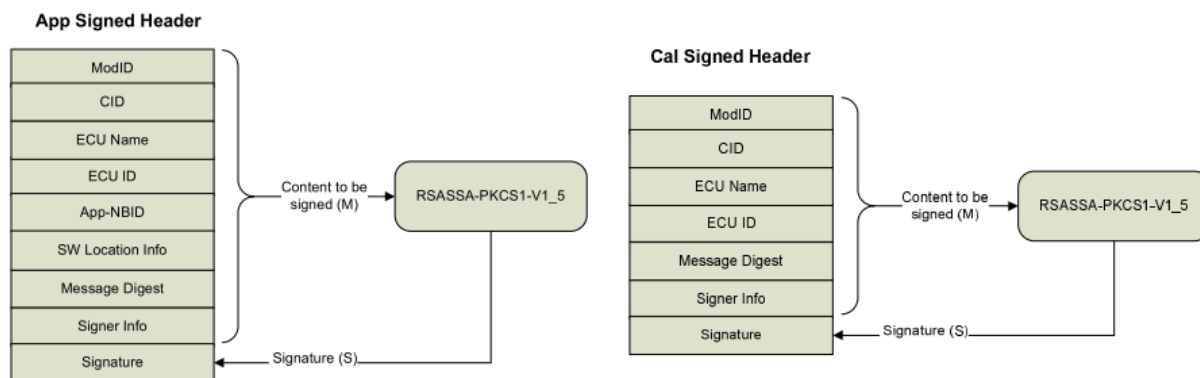


Figure 9-2 Application and Calibration signed header structures and signature calculation. "Signature" represents the digital signature.

### 9.8.2 Signature-Bypass Authorization (SBA) ticket

The SBA ticket allows a way to skip the digital signature and message digest checks. The SBA ticket is a module that consists only of the Signature-Bypass Authorization Header. This can be downloaded to the ECU and stored into the ECU NVM.

During bootloader initialization, the SBA ticket will be read from NVM via the user function `AppIFbINVMReadSBATicket`. If a valid SBA ticket is found, the bootloader will set the Signature Bypass Indicator flag with this state. Upon receiving download of an application or calibration module, the bootloader will skip the digital signature and message digest verification. Therefore, it is possible to download modules without having the correct digital signature or message digest.

The use of an SBA ticket can be useful while debugging an issue on a production ready ECU. In this case, an SBA ticket can be downloaded to the ECU to allow it to skip the digital signature and message digest checks. Now modifications can be made to the module (application or calibration) without the need for the module to be 'signed' (e.g. the module does not need to be sent to GM for signing each time an update is made).

The delivery contains an SBA ticket valid for the Demo configuration using Demo values for ECU ID, Subject name and ECU name. It is signed using the provided demo-key and can be used for testing during development.

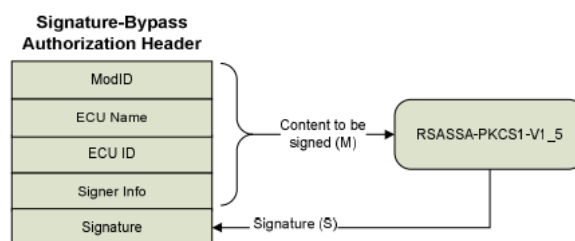


Figure 9-3 Signature-Bypass Authorization Header structure and signature calculation.

**Note**

An SBA ticket is only valid for a specific ECU ID and therefore can only unlock one unique ECU.

### 9.8.3 Message Digest

The message digest is included as a parameter of the signed header. This applies to both application and calibration modules. The message digest is used to check that the content of a module that is programmed to flash memory has not been altered (intentionally or unintentionally). The message digest is calculated using the hash algorithm SHA256.

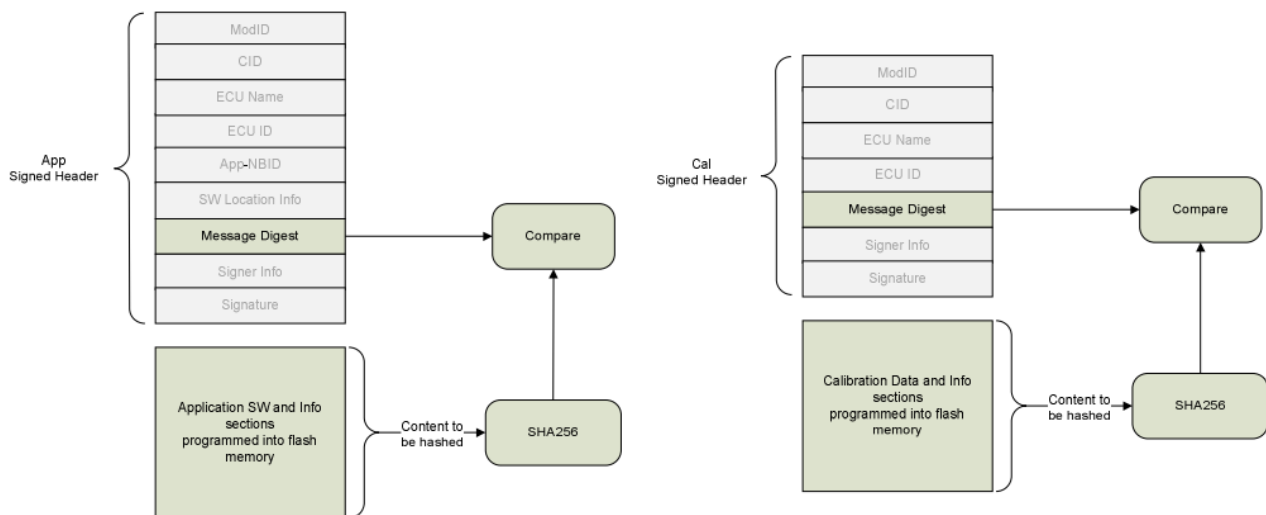


Figure 9-4 Application and Calibration Message Digest

#### 9.8.3.1 Pipelined Verification

The bootloader implements what is known as pipelined verification. Pipelined verification is when the bootloader actively calculates the hash on the programmed data while the bootloader is still downloading data. Each time a packet of data is transferred and programmed to the ECU the bootloader will calculate the hash of that packet while also receiving the next packet of data. This decreases the download time by significantly reducing the time to complete the hash calculation and message digest check at the end of programming.



### 9.8.3.2 Optional Integrity Word Check

In addition to the message digest check, the bootloader can also perform an integrity word check. The integrity word is stored inside the plain header and contains a checksum based on a wordsum with a 2s complement of the data that is programmed to flash. The bootloader will perform this check if the macro `FBL_ENABLE_VERIFY_INTEGRITY_WORD` is set. This is set by defining the macro in a user config file.



#### Caution

It is only recommended to use the integrity word check during development. This is because the integrity word check is not pipelined (like the message digest check). This means that the bootloader will need to re-read all of the data programmed to flash to perform the integrity word check. This eliminates the advantages of the pipelined verification used to calculate the message digest.

### 9.8.4 Signer Info

The signer info is included as a parameter of the signed header. This applies to both application and calibration modules. This is a certificate like structure used to validate the public key that is used as part of the digital signature. The signer info contains the root signature. The bootloader uses the public key stored in the bootloader ROM to validate the root signature (e.g. authorizes that this signer info can be used to validate the digital signature).

The signer info (including the root signature) does not need to be updated each time a module is updated. The signer info is only updated upon GM's discretion. The same signer info can be used as long as GM allows.

During development, Vector provides an example signer info block. The example signer info block contains the example public key that is used to validate the digital signature. It can be used for creating signed modules using again the `Demo_key`.

During production, the signer info will be provided by GM as part of the received signed containers.

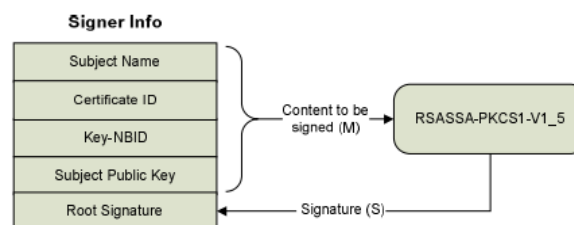


Figure 9-5 Signer Info Structure and signature calculation.

### 9.8.5 Application Software – Not Before Identifier (App-NBID)

The app-NBID is a security parameter that is primarily used to prevent roll back to previous application software version [2].

The app –NBID must be stored in NVM. The initial value is generally set to 0x0000. Upon downloading an application module, the app –NBID stored in the application header is compared to the app –NBID stored in NVM of the bootloader. If the app –NBID in the application header is less than the app –NBID stored in NVM then the module is rejected. If the app –NBID in the header is greater than the app –NBID stored in NVM then the NVM is updated with the new app –NBID.



---

**Note**

The bootloader uses the call-back functions `ApplFbINVMReadAppNBID` and `ApplFbINVMWriteAppNBID` to read and write the App-NBID. These functions are described in section 5.3 Module Validation Callbacks.

---

### 9.8.6 Security Key – Not Before Identifier (Key-NBID)

The Key-NBID is a security parameter that is primarily used to prevent use of a previous key [2].

The Key-NBID must be stored in NVM. The initial value is generally set to 0x0000. Upon downloading an application or calibration module, the key-NBID stored in the signer info of the header is compared to the key-NBID stored in NVM of the bootloader. If the key-NBID in the header is less than the key-NBID stored in NVM then the key (and also the module) is rejected. If the key-NBID in the header is greater than the key-NBID stored in NVM then the NVM is updated with the new key-NBID.



---

**Note**

The bootloader uses the call-back functions `ApplFbINVMReadKeyNBID` and `ApplFbINVMWriteKeyNBID` to read and write the key-NBID. These functions are described in section 5.3 Module Validation Callbacks.

---

## 9.9 Programming of Unused Flash Space / Gap Fill

If gap fill is enabled in GENy (see section 4.4.2) then the bootloader will automatically call a pre-built function (`FblHdrFillGaps`) to fill the unused memory space. This function is designed to cover all cases and is therefore bulky and slow. It is recommended that the user disable the gap fill function in GENy and implement a custom gap fill function in simple cases. This can be done using the function `ApplFbIFillGaps` (see section 5.3). For example, if there is only one fill region then this information can be embedded into a

module (i.e. start and end address of the fill region). ApplFblFillGaps can then be implemented to simply fill this one region.

## 10 Limitations

The GMLAN diagnostics specification (reference [1]) allows great flexibility in preparing and transferring data to your ECU. Reference [2] contains many clarifications that result in greater consistency between bootloader implementations.

To keep the FBL compliant to the specification, but otherwise minimize the amount of non-volatile memory used by the FBL, the number of variations to perform a download is reduced.

### 10.1 Implementation-specific limitations

- > The FBL usually obtains the destination address for transferred data from the File-Header of the module (for Calibration modules, the FBL looks in the File-Header of the Operating Software).

For this reason, the File-Header must be transferred in the first data block sent by the Download Tool. The first transfer-data request need not contain the entire header, but the FBL must be able to obtain the first PMA/NOB (address-region) definition.

### 10.2 GMW3110 V1.4 V.1.5 V1.6

GMW3110 V1.4 and GMW3110 V1.5 have been superseded by GMW3110 V1.6.

### 10.3 GMW3110 V 1.6

- > Section 9.13.2.2  
The FBL does not check the StartingAddress parameter. Starting address information is normally obtained from the File-Header of the Operating Software.
- > Section 9.3.4  
The FBL will not accept any module with a Module-ID of zero (\$00). This module ID is reserved for the SPS Utility File.

### 10.4 Global-A Secure Bootloader Specification

- > Compression/Decompression is not available with this release.

### 10.5 CG3532 ECU Security Requirements Ver. 1.0

- > CG3532 requires the ECU to dynamically check stack bounds. This is considered hardware and configuration specific and the delivered bootloader components do not meet this requirement. It is required for the user to consider this requirement..



## 11 Terminology

Acronym	Description
Address-Region	Block of consecutive data bytes. All addresses within the block contain data, there are no gaps.
API	Application Program Interface Defines the public methods within a software component that may be accessed by an application to perform certain programming tasks.
ASCII	American Standard Code for Information Interchange Defines the numeric representation of the English alphabet, plus several special and non-printable characters.
CAN	Controller Area Network
CPU	Central Processing Unit
CS	Check-Sum Refers to module checksum field of File-Header defined by GM. See “Software and Calibration File Header Requirements” (Section 9.3.3.1 in reference [1]).
CTS	Component Technical Specification Documentation developed by General Motors describing the detailed requirements of an ECU.
DCID	Data Compatibility Identifier This field is also referred to as the BCID (Bootloader Compatibility Identifier), and as the CCID (Calibration Compatibility Identifier). Refers to module compatibility field in File-Header defined by GM. See “Software and Calibration File Header Requirements” (Section 9.3.3.1 in reference [1]) and various sections in reference [2].
DLL	Data Link Layer Provides software interface to the CAN hardware (also known as the CAN driver).
DLS	Design Level Suffix Refers to module revision code in File-Header defined by GM. Also known as Alpha-Code. See also “Software and Calibration File Header Requirements” (Section 9.3.3.1 in reference [1]).
DPS	Development Programming System. PC based Download Tool created by General Motors.
ECU	Electronic Control Unit
EEPROM	Electrically Erasable Programmable Read-Only Memory EEPROM is a type of non-volatile memory. This type of memory is similar to flash memory, but may be erased and written one byte at a time.

FBL	Flash Bootloader. The FBL is a software application independent of the Operating Software. It is responsible for downloading and programming the Operating Software and/or related data modules into an ECU. The FBL is usually in protected memory, where it cannot be erased.
Flash-Driver	File containing the algorithms used to erase and write to flash memory. GM's specifications refer to the driver as the <i>Programming Routines</i> .
Download Tool	Any Hardware/Software system capable to connecting to the CAN bus for the purpose of reprogramming the Operating Software and/or Calibration or other data of an ECU. For example, Vector's <i>vFlash</i> and GM's <i>DPS</i> programs.
File-Header	Refers to file header defined by GM. All downloadable modules require a header. See "Software and Calibration File Header Requirements" (Section 9.3.3.1 in reference [1]).
GBF	Generic Binary File GBF files cannot be read using text editors. The provided hexview tool can do this instead. The GBF file type is compatible for use with GM's DPS tool (within an SPS archive file).
GM	General Motors
GMLAN	General Motors Local Area Network
HIS	Hersteller Initiative Software Refers to an effort made by a group of manufacturers to standardize aspects of software development. The FBL bases the low-level Memory I/O (e.g. the Flash Driver) APIs on this work. For additional information, see reference [9].
ISO	International Organization for Standardization
ISR	Interrupt Service Routine. Any software intended to be executed immediately upon receipt of an asynchronous event, thus interrupting whatever task the ECU was performing at the time of the event. In general, special restrictions apply to the actions an ISR may take.
KBPS	Kilo-Bits per Second Transfer rate of data across the CAN bus. Also known as the <i>Baud Rate</i> .
MEC	Manufacturers Enable Counter See also "The Vulnerability Flag And Manufacturers Enable Counter (MEC)" (Section 9.3.2.6 in reference [1]).
MID	Module-ID Refers to module identification field of File-Header defined by GM. See "Software and Calibration File Header Requirements" (Section 9.3.3.1 in reference [1]).
MIM	Multiple-Identity-Module. Some ECUs running common Operating Software may be used for multiple purposes within the vehicle. For example, the same ECU might be used in all doors. Such modules must configure themselves at startup to use unique diagnostic-identifiers and CAN ids.

Module	Any data file that may be transferred to an ECU via the Bootloader Software.
NOAM	<p>Number of Additional Modules</p> <p>Used in the File-Header of Operating Software modules. Defines the number of calibration modules associated with the Operating Software. See also “Software and Calibration File Header Requirements” (Section 9.3.3.1 in reference [1]).</p>
NOAR	<p>Number of Address Regions</p> <p>Used in File-Header defined by GM to define the number of regions in a module (See Address Region). See also “Software and Calibration File Header Requirements” (Section 9.3.3.1 in reference [1]).</p>
NOB	<p>Number of Bytes</p> <p>Refers to the number of bytes in an Address Region. Described in the File-Header defined by GM. See also “Software and Calibration File Header Requirements” (Section 9.3.3.1 in reference [1]).</p>
OEM	Original Equipment Manufacturer
Operating Software	This is the application program responsible for implementing the tasks that an ECU should perform. It is also known as the Application, Application Software, Operational Software, Operating System, and Opcode.
PCI	<p>Protocol-Control-Information</p> <p>Defines the type of message being exchanged by the Transport-Protocol layer. See also Reference [1], Section 4.5.1.1.2 “Protocol Control Information (PCI) formats for GMLAN”.</p>
PLL	<p>Phase Locked Loop</p> <p>Electronic circuit used to maintain accuracy in an ECU’s clock and timing hardware.</p>
PMA	<p>Product Memory Address</p> <p>Refers to the starting address of an Address Region. Described in the File-Header defined by GM. See also “Software and Calibration File Header Requirements” (Section 9.3.3.1 in reference [1]).</p>
PSI	<p>Programmed State Indicator</p> <p>Mechanism to determine if a logical partition is programmed with valid content.</p> <p>Interchangeable with presence pattern in this document.</p>
RAM	Random Access Memory
RCR-RP	A special type of negative response indicating Request Correctly Received, Response Pending.
ROM	Read Only Memory
Sector-Size	The number of bytes that are erased when an erase command is issued to a non-volatile device. This number must be a power of two (e.g. 64, 128, 256, etc). The sectors on a non-volatile device are not necessarily all the same size. Many devices allow multiple sectors to be erased with a single command.



Segment-Size	The number of bytes that must be written when a write command is issued to a non-volatile device. Most devices allow a multiple of the segment-size to be specified in write commands. The segment-size is expected to be the same for all sectors on a device. The FBL expects the value to be a power of two.
SIP	Software Integration Package Refers to all files contained in the delivery of Vector Software.
SPS	Service Programming System
SPS Type	Service Programming System Type. SPS refers to the architecture defined by GM used to program devices on a GMLAN serial data link. See GMW3110 Section 9.1. An ECU supporting SPS may be in one of three states: SPS_TYPE_A The ECU is fully programmed with both Operating Software and Calibration. SPS_TYPE_B The ECU is not fully programmed (may not be programmed at all), but the permanently-programmed diagnostic message CAN IDs are known to the bootloader. SPS_TYPE_C The ECU is not fully programmed and the permanently-programmed diagnostic message CAN IDs are not known. The FBL will communicate using the SPS-Prime Request and SPS-Prime Response IDs (these are based on the Diagnostic Node Address).
SWMI	Software Module Identification Refers to module part-number field in File-Header defined by GM. See “Software and Calibration File Header Requirements” (Section 9.3.3.1 in reference [1]).
TP	Transport Protocol
USDT	Unacknowledged Segmented Data Transfer. In GMLAN, this message type is used for transfer of Diagnostics data (and programming data).
UUDT	Unacknowledged Unsegmented Data Transfer

## 12 References

[1]	General Motors	GMW3110 v1.6 February, 2010
[2]	General Motors	Global-A Secure Bootloader Specification, V3.1, July 29, 2013 Ansaf Alrabady
[3]	General Motors	Service Programming System (SPS) Interpreter Programmers Reference Manual, Dec. 1, 2004
[4]	Vector-Informatik GmbH	Flash Bootloader User Manual, v2.7 September 01, 2006
[5]	Vector-Informatik GmbH	vFlash Manual
[6]	Vector-Informatik GmbH	+Calling the GM-Bootloader from CANdesc-Applications, v1.00 (App. Note: AN-ISC-2-1011_CANfblGM_Call_from_CANdesc) April 05, 2004
[7]	International Organization for Standardization	ISO 15765-2 Diagnostics on CAN / Part 2 Network Layer Services ISO 15765-3 Diagnostics on CAN / Part 3 Implementation of diagnostic services June 21, 2002
[8]	International Organization for Standardization	ISO 14230-3 Diagnostic systems – Keyword Protocol 2000 Version 1.5, October 1, 1997
[9]	Hersteller Initiative Software (HIS)	Functional Specification of a Flash Driver Version 1.3, June 6, 2002 <a href="http://www.automotive-his.de/">http://www.automotive-his.de/</a>
[10]	Vector-Informatik GmbH	Flash Bootloader OEM Technical Reference – CANfbl GM – Programmable Data File Creation
[11]	Vector-Informatik GmbH	TechnicalReference_NvWrapper.pdf
[12]	General Motors	CG3532 ECU Security Requirements Ver. 1.0

## 13 Contacts

<b>Vector Informatik GmbH</b> Tel: +49 711-80670-0 Email: FblSupport@vector-informatik.de  www.vector.com	<b>Vector CANTech, Inc.</b> Tel: (248) 449-9290 Email: fbisupport@vector-cantech.com  www.vector.com	<b>VecScan AB</b> Tel: +46 (0)31 764 76 00 Email: info@vecscan.com  www.vector.com
<b>Vector France SAS</b> Tel: +33 (0)1 42 31 40 00 Email: information@vector-france.fr  www.vector.com	<b>Vector Japan Co. Ltd.</b> Tel: +81 03(5769)6970 Email: info@vector-japan.co.jp  www.vector.com	

Visit our website for more information on

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

**[www.vector-informatik.com](http://www.vector-informatik.com)**