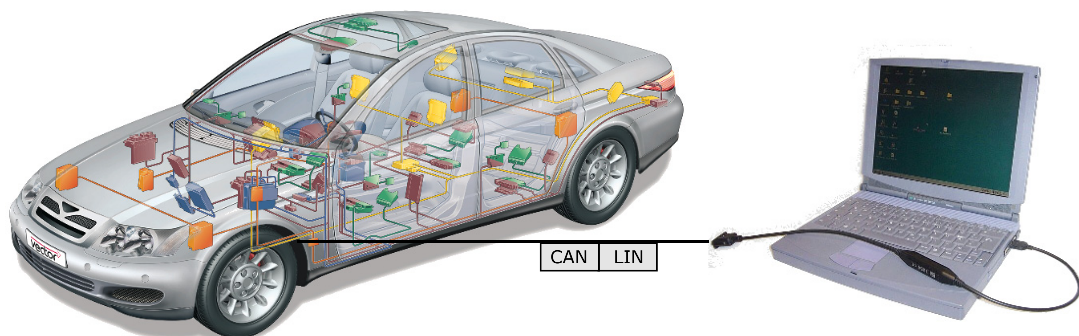
Flash Bootloader

User Manual

(Your First Steps)

Version 2.7

Vector Informatik GmbH, Ingerheimer Str. 24, 70499 Stuttgart
Tel. 0711/80670-0, Fax 0711/80670-399, Email can@vector-informatik.de
Internet http:\\www.vector-informatik.de



Authors:	Klaus Emmert
Version:	2.7
Status:	released (in preparation/completed/inspected/released)

©2006, Vector Informatik GmbH

History

Author	Date	Version	Remarks
Klaus Emmert	2004-07-09	2.2	Switch to new Layout Version 2.0 New symbols
Klaus Emmert	2004-02-09	2.3	Changes from Review Ra 2004-09-20, link labels
Klaus Emmert	2005-03-23	2.4	Chapter 3.8 and warning for startup-codes added
Klaus Emmert	2006-02-06	2.5	Change of file structure
Klaus Emmert	2006-08-18	2.6	File Structure illustration
Klaus Emmert	2006-09-01	2.7	WDtimer and some minor issues

Motivation For This Work

A short story about flashing

After a seemingly almost endlessly long development process, the software is finally finished and ready for the ECU, downloaded one last time, tested and the ECUs packaged for express delivery the next day.

Now it‘s 10:00 P.M.

Shortly before quitting time the next business day the telephone rings, and what is on the display makes your forehead break out in a sweat of alarm! Errors, nothing is working, says the message from your customer. You hastily start up another ECU in the lab and you have to also observe the same result just reported to you. After searching for a little while you realize that the error is in version management. You put the correct version together, recompile it, briefly test the result and send the hex code to your customer, who can now flash the new functioning software via CAN and diagnostics onto the ECU and as a result, can proceed with the planned tests without substantial delays.

WARNING

All application code in any of the Vector User Manuals is for training purposes only. They are slightly tested and designed to understand the basic idea of using a certain component or a set of components.

Contents

1 Welcome to the Flash Bootloader User Manual 7
1.1 Beginners with Flash Bootloader start here ? 7
1.2 For Advanced Users 7
1.3 Special topics 7
1.4 Documents this one refers to… 8

2 About This Document 9
2.1 How This Documentation Is Set-Up 9
2.2 Legend and Explanation of Symbols 10

3 Flashing – An Overall View 11
3.1 What Is Flashing? 11
3.2 What Happens During Flashing? 12
3.3 What Is The Flash Bootloader? 12
3.4 Bootloader 12
3.5 Flash Driver 13
3.6 Flash Tool 14
3.7 What The Flash Bootloader Does 15
3.8 What The Flash Bootloader NOT Does 15

4 Flashing – A More Detailed View 16
4.1 The Bootloader Is Always Started First 16
4.2 Flashing After A Reset 16
4.3 Your Application Initiates The Flashing Process 17
4.3.1 What might happen? 17
4.4 Handling of the Validation Concepts 18
4.4.1 Validation Area 18
4.4.2 Access to the Validation Area 18
4.4.3 ApplFblIsValidApp 18
4.4.4 ApplFblValidateApp 18
4.4.5 ApplFblInvalidateApp 18
4.5 Proposals For Handling The Validation Area 19
4.5.1 Proposal A 19
4.5.2 Proposal B 20
4.5.3 Proposal C 22
4.6 The Interrupt Vector Tables 23
4.7 Label Reference File 24

5 FLASHING IN 5 STEPS 25

©2006, Vector Informatik GmbH

5.1 STEP 1 Design The Memory Layout 26
5.2 STEP 2 Write A Test Application 27
5.3 STEP 3 Integrate The Bootloader 28
5.4 STEP 4 Adapt Your Test Application For The Tester 29
5.5 STEP 5 Download Your Test Application With The Tester 29

6 Details of Bootloader Integration Step (STEP 3) 30

6.1 Bootloader STEP 1 – Extract the files to a folder on your pc 31
6.2 Bootloader STEP 2 Adjust the marked files to fit your application 34
6.2.1 Make… Makefile and make.exe 34
6.2.2 fbl_cfg.h - The Configuration File For The Flash Bootloader 34
6.2.3 FBL_apxx.C 36
6.2.4 Fbl_vect.c / Applvect.c(.h) - The Interrupt Vector Tables 42
6.3 Bootloader STEP 3 Now compile the Flash Bootloader 44
6.4 Bootloader STEP 4 Transfer the Bootloader to the target hardware 44
6.5 Bootloader STEP 5 Use the Flash Tool to Test the Bootloader 44
6.6 Bootloader STEP 6 – Test the flashing after a reset 44
6.7 Bootloader Step 7 – Make your application ready for the transition to the Bootloader 45
6.8 Bootloader Step 8 – Start Bootloader from your application 45

7 Background Information 46

7.1 The Watchdog 46
7.1.1 Initializing The Watchdog 49
7.2 Multiple ECU Support 49
7.3 Validation Ok – Application Faulty 50
7.4 FlashSegmentSize 51
7.4.1 Why Does The Tool Have To Know This Block Length? 51
7.5 Frequently Asked Questions 52
7.5.1 Bootloader Crashes 52
7.5.2 Application Is Not Started 53
7.5.3 Bootloader Is Not Started 54
7.5.4 The Flash Tool's Error Codes 54

8 Index 2

Illustrations

Figure 1-1 Manuals and References for the Flash Bootloader 8
Figure 3-1 What Is Flashing 11
Figure 3-2 Bootloader, Flash Driver And Flash Tool Form The Flash Bootloader 12
Figure 3-3 Bootloader And Your Application Never Run Simultaneously.................... 13
Figure 3-4 Order And Way Of The Download Of The Software Components............. 14
Figure 4-1 Transition Between The Bootloader And Your Application And Vice Versa.. 16
Figure 4-2 Using A Flag Only In The Validation Area... 19
Figure 4-3 Separate Your Application Into Several Modules....................................... 20
Figure 4-4 Using A Validation Function For Validation Your Application..................... 22
Figure 4-5 Principle Of The Two Interrupt Vector Tables 23
Figure 5-1 Basic Memory Layout Of An Application with the Flash Bootloader 26
Figure 5-2 The Test Application In The ECU Memory Using Two Interrupt Vector Tables 27
Figure 5-3 Flashing via Flash Tool and OEM-specific Tester...................................... 28
Figure 6-1 Details of Bootloader Integration Step (Step 3).. 30
Figure 6-2 Function Calling Sequence During Flashing 37
Figure 6-3 Situation Directly After The Programming Of The Bootloader Together With The Dummy Application Vector Table 42
Figure 7-1 Memory Layout Of The Watchdog Trigger Functions 46
Figure 7-2 Functions For Manipulating The Watchdog 48
Figure 7-3 Modified Function Calling Sequence.. 50
Figure 7-4 Segmenting During Flashing.. 51

©2006, Vector Informatik GmbH

1 Welcome to the Flash Bootloader User Manual

1.1 Beginners with Flash Bootloader start here ?

You need some information about this document?

What is Flashing?

What is the Flash Bootloader?

Chapter 3.1
Chapter 3.3
Chapter 2

1.2 For Advanced Users

Start reading here.

5 Steps for Flash Bootloader integration.

Chapter 4
Chapter 5

1.3 Special topics

Why do I need 2 Interrupt Vector Tables?

How to define my application valid?

Validation Ok but application not valid…

How to handle my watchdog?

Chapter 4.6
Chapter 4.4
Chapter 7.3
Chapter 7.1

©2006, Vector Informatik GmbH

1.4 Documents this one refers to…

- FlashTool Documentation
- OEM-specific Documentation
- Hardware-specific Documentation

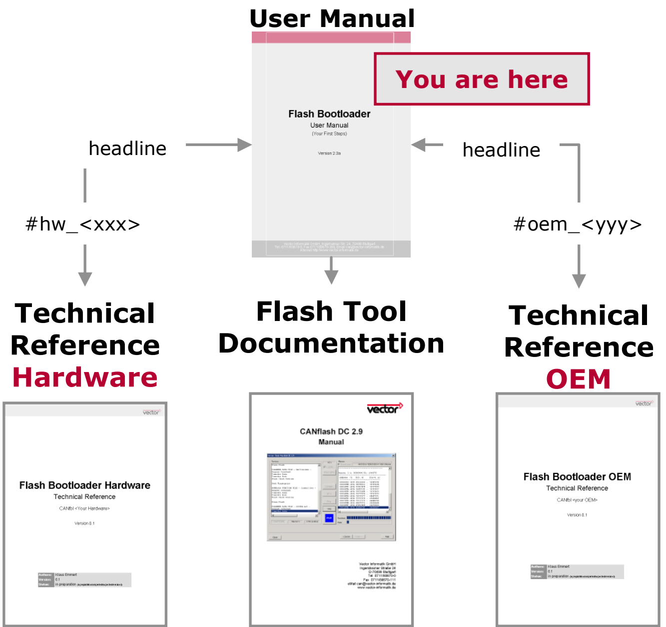


Figure 1-1 Manuals and References for the Flash Bootloader

The Flash Bootloader can be separated into a general part, that is equal to all Flash Bootloaders and parts that are dependent on the requirements of the OEM and the features of the hardware. All common topics are described within this user manual. Use the references when indicated to figure out the specifics for your Flash Bootloader.

For the OEM-specifics of your Flash Bootloader refer to the:

TechnicalReference_FBL_<OEM>.pdf. The ID for a reference to this document looks like: **[#oem_<xxx>].** This ID you will find in the corresponding headline there.

For the hardware-specifics of your Flash Bootloader refer to the :

TechnicalReference_FBL_<hardware>.pdf. The ID for a reference to this document looks like: **[#hw_<yyy>].** This ID you will find in the corresponding headline there.

©2006, Vector Informatik GmbH

2 About This Document

This document gives you an understanding of the Flash Bootloader. You will receive general information, a step-by-step tutorial to get the Flash Bootloader up and running, details regarding diagnostic services, directions to implement the watchdog, and instructions regarding the Flash Tool.

This is the general Flash Bootloader Document, independent of oem-specific settings or hardware. For more information about OEM solutions or hardware dependencies refer to those specific documents.

2.1 How This Documentation Is Set-Up

Chapter	Content
Chapter 1	The welcome page allows easy navigation throughout the document.
Chapter 2	Contains some formal information about this document, and explanation of legends and symbols.
Chapter 3	Provides a brief introduction to flashing and the different parts that make up the Flash Bootloader.
Chapter 4	Provides details about the flash process, states the validation concept, and the implementation of the two interrupt vector tables.
Chapter 5	Describes the five basic steps to integrate the Flash Bootloader, and download the application using the Vector Flash Tool.
Chapter 6	Describes how to download the application using the OEM-Specific flash tool.
Chapter 7	Lists some common problems encountered while integrating the bootloader and their proper solution.

©2006, Vector Informatik GmbH

2.2 Legend and Explanation of Symbols

You find these symbols at the right side of the document. They indicate special areas in the text. Here is a list of their meaning.

These areas to the right of the text contain brief items of information that will facilitate your search for specific topics.

Symbol	Meaning
	The building bricks mark examples.
Comments and explanation	You will find key words and information in short sentences in the margin. This will greatly simplify your search for topics.
	The footprints will lead you through the steps until you can use the described Flash Bootloader.
	There is something you should take care about.
	Useful and additional information is displayed in areas with this symbol.
	This file you are allowed to edit on demand.
	This file you must not edit at all.
	This indicates an area dealing with frequently asked questions (FAQ).

3 Flashing – An Overall View

3.1 What Is Flashing?

During the flashing process an application (or a part of it), which must be available in hex format (using the Flash Tool), is transferred into the ECU‘s memory. This transfer is done via a bus protocol like CAN, LIN, FlexRay, etc.

The Flash Bootloader downloads the application as a hex file to the ECU via CAN.

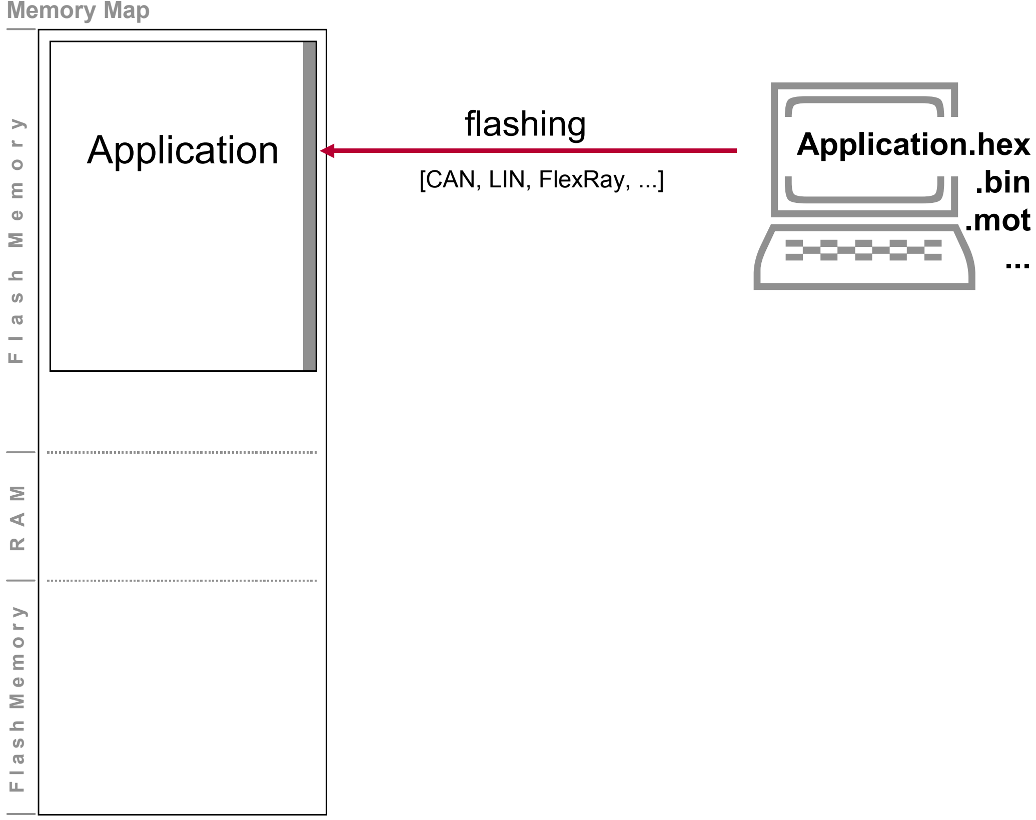


Figure 3-1 What Is Flashing

PC or Laptop, CAN Card and the Flash Tool – this is all you need for flashing.

To transfer your application to the target platform you simply need the Flash Bootloader, a PC or a laptop, a CAN card and the Flash Tool.

Using an emulator, a BDM tool, or the like to perform the flashing in an already built-in ECU would be expensive. In addition, the ECU’s hardware interface for the BDM or emulator is not always accessible.

The interface is based on a CAN bus as part of the Physical Layer, the Transport Protocol, and on KWP2000 for the specification of diagnostic services.

For flash data exchange between your new node and the tool you need two CAN messages, one for request and one for response.

©2006, Vector Informatik GmbH

3.2 What Happens During Flashing?

The Flash Driver (flash algorithms) is downloaded via the bus protocol. Afterwards the application is downloaded and written (flashed) into the Flash Memory (using the Flash Driver).

3.3 What Is The Flash Bootloader?

The Flash Bootloader is a combination of embedded software and PC tool, designed to do the flashing of the ECU software via a bus protocol (e.g. CAN).

Therefore the Flash Bootloader is divided up into 3 parts, the Bootloader, the Flash Driver and the Flash Tool.

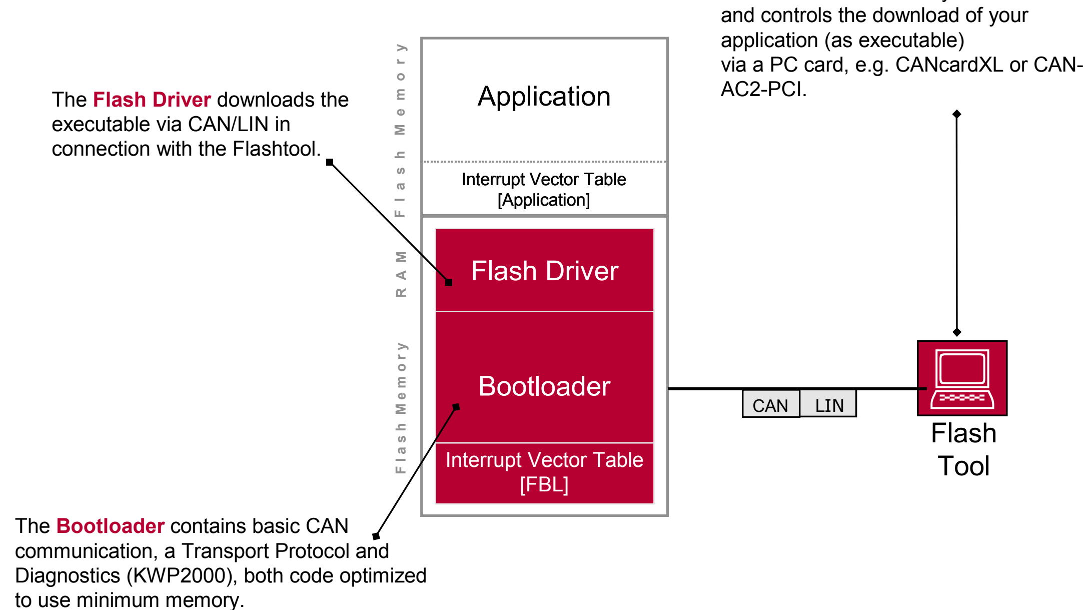
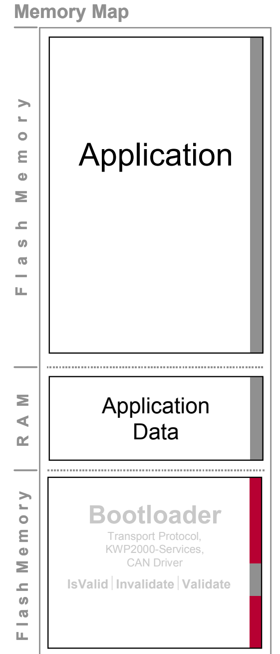


Figure 3-2 Bootloader, Flash Driver And Flash Tool Form The Flash Bootloader

3.4 Bootloader

The Bootloader is a stand-alone program. It is compiled, linked, and downloaded to the ECU separately from your application. The Bootloader and your application never run simultaneously.
The Bootloader uses Flash memory, in the protected area of the ECU. If your ECU does not support such a hardware protection, the Bootloader will protect itself from being overwritten.

Application is running



Flash Bootloader is running

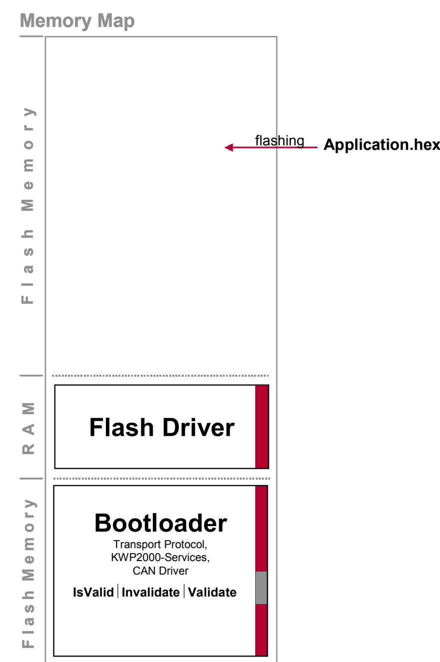


Figure 3-3 Bootloader And Your Application Never Run Simultaneously

As the Bootloader downloads data via CAN, it contains a CAN Driver and a Transport Protocol.

The Bootloader is transferred to the target platform via a Flash programmer or burner, which is hardware dependent (It is not downloaded over CAN).

3.5 Flash Driver

Reasons for Flash Driver to run in RAM:

Physical reasons

Protection against a faulty call deleting the flash

Save ROM memory

Easy to make updates.

The Flash Driver (actual flash algorithm) is the hardware dependent code for performing the flash functions.

In most cases, programming flash memory from flash is not possible. Therefore the Flash Driver is downloaded and executed into RAM to allow programming of the application.

The advantage of downloading the flash algorithm into RAM is that updates to the flash algorithms are possible without the need to reprogram the primary bootloader. The algorithm is cleared from RAM upon completion of the download to avoid accidental calls to the flash functions while in the application.

In special cases the flash algorithms are kept in flash memory and copied to RAM when needed. Of course the possibility of changing the flash algorithms is no longer available when this configuration is used. Moreover, there is a risk that the flash memory will be unintentionally erased from an accidental call to these functions. A remedy to correct this would be to encrypt the corresponding program code, such as e.g. an XOR or the like.

©2006, Vector Informatik GmbH

3.6 Flash Tool

The Flash Tool is a Windows™ based PC tool that controls the download of the application.

The Flash Tool reads the compiled and linked application data (Motorola-S or Intel-Hex format), triggers and controls the flash process, transfers the data via the bus protocol (e.g. CAN) and verifies this process via a checksum.

To do the download you additionally need a CAN card (CANcardXL, CAN AC2-PCI) to connect your hardware to the PC.

The following figure shows the Bootloader, the Flash Driver, the application, and how they get in the memory of the controller. As the Bootloader is the component to enable the download of the Flash Driver and the application, it cannot be transferred by itself. This job must be done with a suitable programmer or burner (development environment).

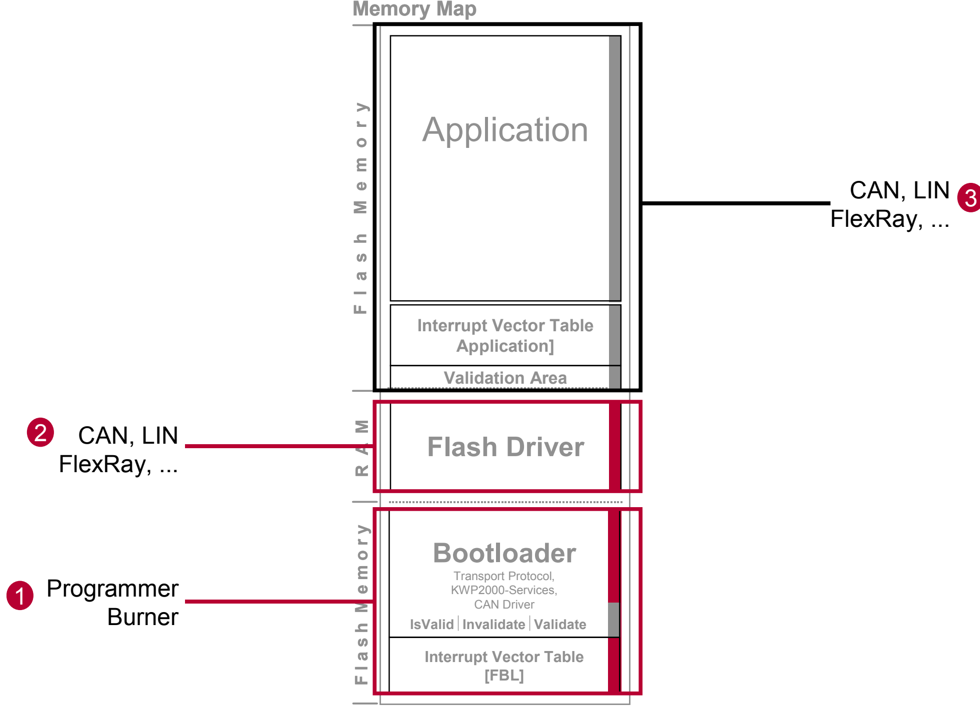
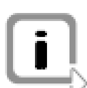


Figure 3-4 Order And Way Of The Download Of The Software Components

The numbers in the figure show the download order, Bootloader first, then the Flash Driver to be able to flash the application.



3.7 What The Flash Bootloader Does

- Initializes the CAN controller
- Sends diagnostic messages via CAN
- Receives diagnostic messages via CAN
- Erases and programs the flash memory

3.8 What The Flash Bootloader NOT Does

It is no “Ready-to-use” program:

- Adaptations of callback functions, startup / initialization is necessary
- Adaptation to runtime environment and specific hardware requirements are necessary

©2006, Vector Informatik GmbH

4 Flashing – A More Detailed View

4.1 The Bootloader Is Always Started First

As you now know, on the one hand the Bootloader resides in the protected area and is always resident on the ECU. On the other hand it is not guaranteed that a valid, executable application is also present in the ECU. This is the reason why the Bootloader is always executed first after a reset. The decision is then made to start the application or to stay in the Bootloader.

The Flash Bootloader is called after a reset or directly from your application.

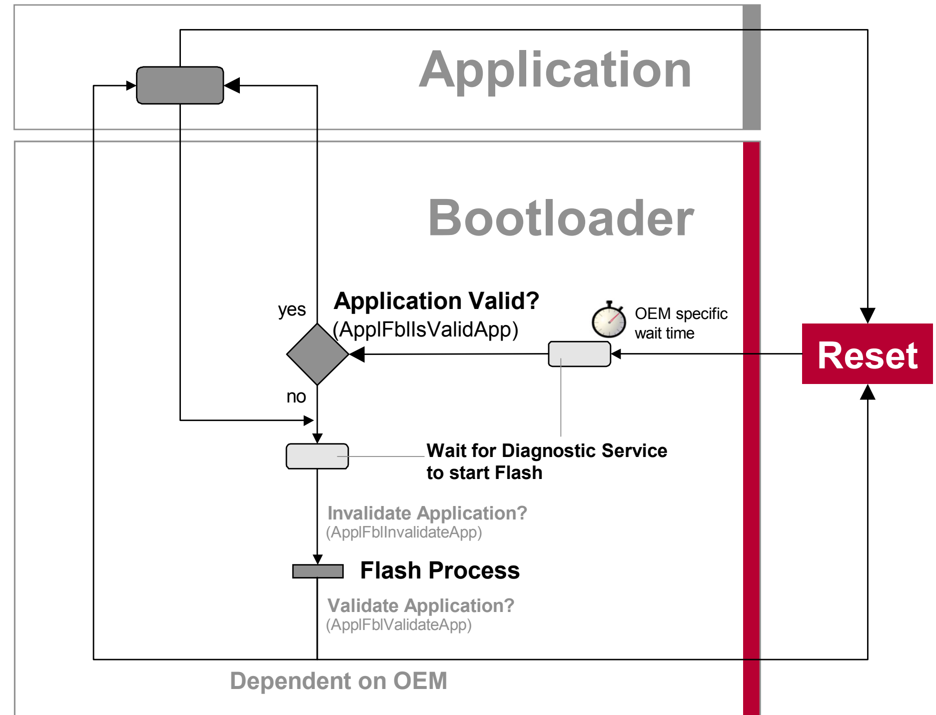


Figure 4-1 Transition Between The Bootloader And Your Application And Vice Versa

The Bootloader can be made to start in different ways, from your application and after a reset.

4.2 Flashing After A Reset

The Flash Bootloader is executed always first after a reset. At this point in time the bootloader has to determine if a valid application has been flashed. This test is done through the use of the **ApplFblIsValidApp** function.

You will find some example strategies of how an application can be set to valid or invalid in chapter 4.5.

The return value of the function **ApplFblIsValidApp** (see Figure 4-1) is used to decide whether the process should branch into the application or if the ECU should

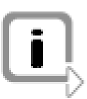
remain in the Flash Bootloader. The code in this function and the way you decide whether the application is valid or not, is up to you.

If the application area is blank or the code is faulty, the Bootloader waits for a Diagnostic CAN message to start the flash process, i.e. the Flash Driver is loaded into the ECU's RAM memory and started. Now the application is being downloaded, which is the actual flash process.

The application is connected to the Flash Bootloader via three functions **ApplFblIsValidApp**, **ApplFblValidateApp** and **ApplFblInvalidateApp.** These functions are to manipulate the Validation Area.

After a reset the **ApplFblIsValidApp** function decides whether to branch into the application.

The names of the functions ApplFblValidateApp and ApplFblInvalidateApp can vary slightly for some OEMs. Please refer to your OEM-specific Documentation for more detailed information **[#oem_valfunc]**.



In the following, these three functions are abbreviated sometimes only with IsValid, Validate and Invalidate. These abbreviations are found in most of the following figures.

4.3 Your Application Initiates The Flashing Process

The other way to start the flashing process is the one via your application and its diagnostics.

Before the Flash Bootloader manipulates data in the application area the function **ApplFblInvalidateApp** is called to set the Validation Area to invalid. Then the Flash Bootloader downloads the new application (the corresponding hex file) and executes a reset or starts the application directly (depending on the OEM **[#oem_start]**). Before starting the application, the function **ApplFblValidateApp** validates the application again.

Before the beginning of flashing, the application must be set to invalid. This prevents starting a partially flashed application if the flash process failed or was interrupted.

4.3.1 What might happen?

Imagine your validity check results in a valid application but it is actually faulty. Every reset leads in the application that is not working properly and the watchdog will provoke a reset again. Now the software is in an endless loop.

In this case it is not possible to flash the application again. See an optional selectable solution to this problem in the chapter 7.3 (Validation ok – Application Faulty).

4.4 Handling of the Validation Concepts

The concept for the flash mechanism is designed to allow maximum flexibility as well as a very easy implementation. The main thing you have to take care for is to call the application only if there is a valid version flashed. The manner you recognize a valid version is up to you and realized in the so-called Validation Area.

4.4.1 Validation Area

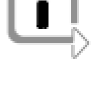
In this area you can store the indicators for a valid application. This can be a simple flag that indicates valid or even code for checking validity of the application. In the latter case the **ApplFblIsValidApp** should check this function before calling it.

4.4.2 Access to the Validation Area

The Bootloader provides 3 functions to access the validation area as mentioned just before. The functions reside in the Bootloader and the coding has to be done by you. The functions are:

4.4.3 ApplFblIsValidApp

This function checks the validity of the application. The return value decides about the further actions (see Figure 4-1).



> Since this function is called on every Reset, it is recommended to simply check a flag previously set. This speeds up the restart time.

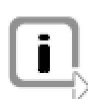
Use the following proposals to get a little comfortable with the possibilities of configuration arising with this concept.

4.4.4 ApplFblValidateApp

This function signs the application as valid by accessing the Validation Area directly. After the flash process the function **ApplFblValidateApp** is called to check the validity of the flashed application and to set the indicators (e.g. a Flag, a certain memory location etc.). The indicators are utilized on reset by the IsValid function.

4.4.5 ApplFblInvalidateApp

This function is called before erasing the flash memory. Herein you reset your indicators and mark the application as invalid in order to avoid the case where a possible reset or error while flashing occurs without invalidating the partially erased or programmed application.



> The implementation of the validation function is application-specific, however, once a solution is implemented it cannot be changed unless the Bootloader is re-programmed.

©2006, Vector Informatik GmbH

4.5 Proposals For Handling The Validation Area

The following three examples are examples of how you can deal with the validation concept offered by the Flash Bootloader. Before starting to develop your own strategy based on the concepts mentioned here, please refer to the OEM-specific Documentation [**#oem_valid**] in case your OEM wants you to do this in a predefined manner.

4.5.1 Proposal A

In this concept the validation area is just a flag. This is the simplest way to realize the validation concept:

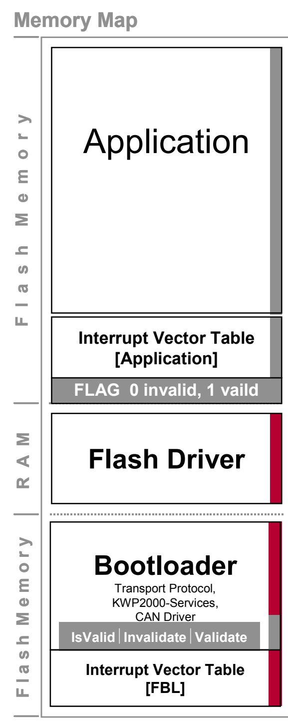


Figure 4-2 Using A Flag Only In The Validation Area

The code in the IsValid function has to know the location of the flag in the Validation Area. After a reset, the content of this flag shows the validity of the application. If you start flashing via CAN the function “Invalidate” clears the flag before erasing. After flashing, the function “Validate” sets the flag. Then after a reset, the “IsValid” function recognizes the flag to be set and returns a positive value. The application is valid and can be executed.

If you can guarantee that the flag (or byte) is the first byte to be erased and the last byte to be written while programming, you may leave the Invalidate and Validate functions empty.

Make sure that the valid indicator differs from the blank, non-existing flash contents.

4.5.2 Proposal B

Based upon this concept you can separate the application into separate sections. See the figure below for the memory mapping. The parts of the application are named as **Module n**. Every module needs a Validation Area of its own.

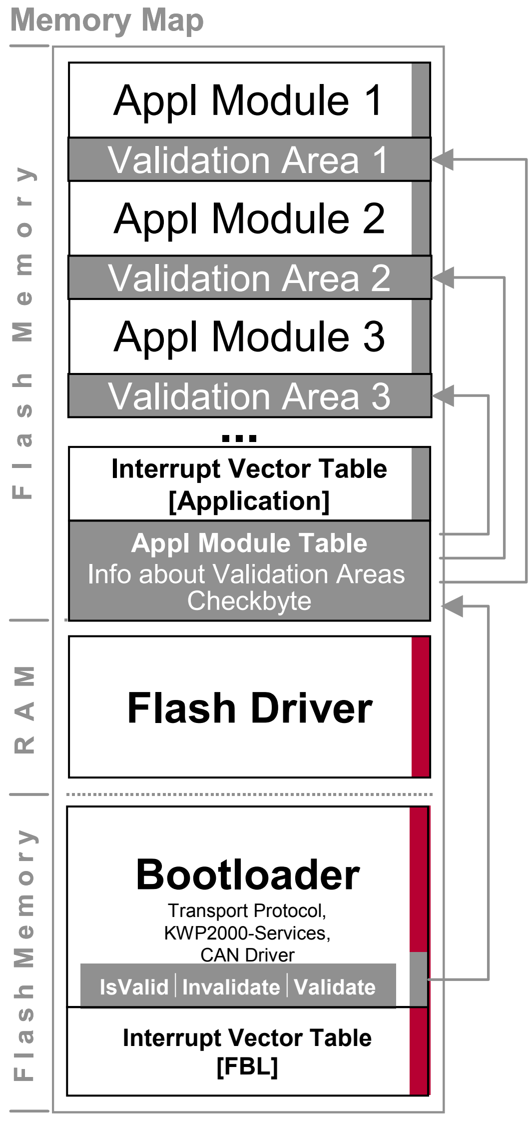
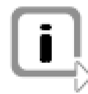


Figure 4-3 Separate Your Application Into Several Modules

To know the locations of the modules, their size and the location of their Validation Area you should add a Module Table. This table contains the latter information and enables the access to the single Validation Areas of the modules. The access is done via the known functions IsValid, Validate, and Invalidate.

Before you use the Module Table, make sure this information is in the memory. Use e.g. a check byte to ensure this.



The location of this check byte depends on the order of erasing and writing to the flash memory. It should be erased as the first byte in this module table area and written only if the complete module table area is valid.

4.5.2.1 The Module Table

The file FBL_MTAB contains the flash-erase sector table (more about the interrupt vector tables in the following). Whenever the sector selectable erase service is called, the Bootloader scans this table to get the address and length information of the sector that has to be erased.

The flash erase sector table is not required if the Bootloader configuration has been optimized for download of only one module.

The location of FBL_MTAB provides great flexibility. It is possible to change the memory size of the modules, without modifications of the Bootloader. To extend the number of modules you can add more entries to the table.

If it is not feasible or necessary to change the flash erase table, this table could also be compiled and linked into the protected area of the Bootloader. This has the advantage that this table cannot be erased.

©2006, Vector Informatik GmbH

4.5.3 Proposal C

The last example uses a validation function instead of just flags as shown in the two examples above.

This validation function resides in the application and is compiled and linked together with the application. Therefore you should take care before calling this function, that the code has been flashed correctly. For this demand you can use a check byte as shown in the Proposal B.

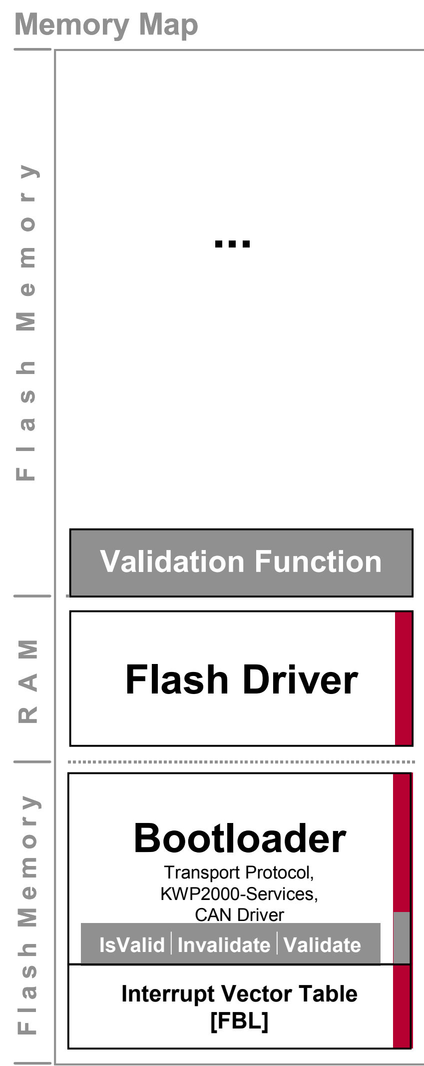


Figure 4-4 Using A Validation Function For Validation Your Application

©2006, Vector Informatik GmbH

4.6 The Interrupt Vector Tables

The application and Flash Bootloader each have their own interrupt vector table to use.

Interrupts are handled in a special way for applications with Flash Bootloader.

When an application uses the Flash Bootloader it must be guaranteed that the reset vector always points to the Flash Bootloader. After a reset the Flash Bootloader is started first.

Usually the ECU vector table is part of the protected area of flash memory, but the reset vectors for the application interrupt service function addresses have to be changeable.

The solution to this problem lies in using 2 interrupt vector tables as shown in the figure below.

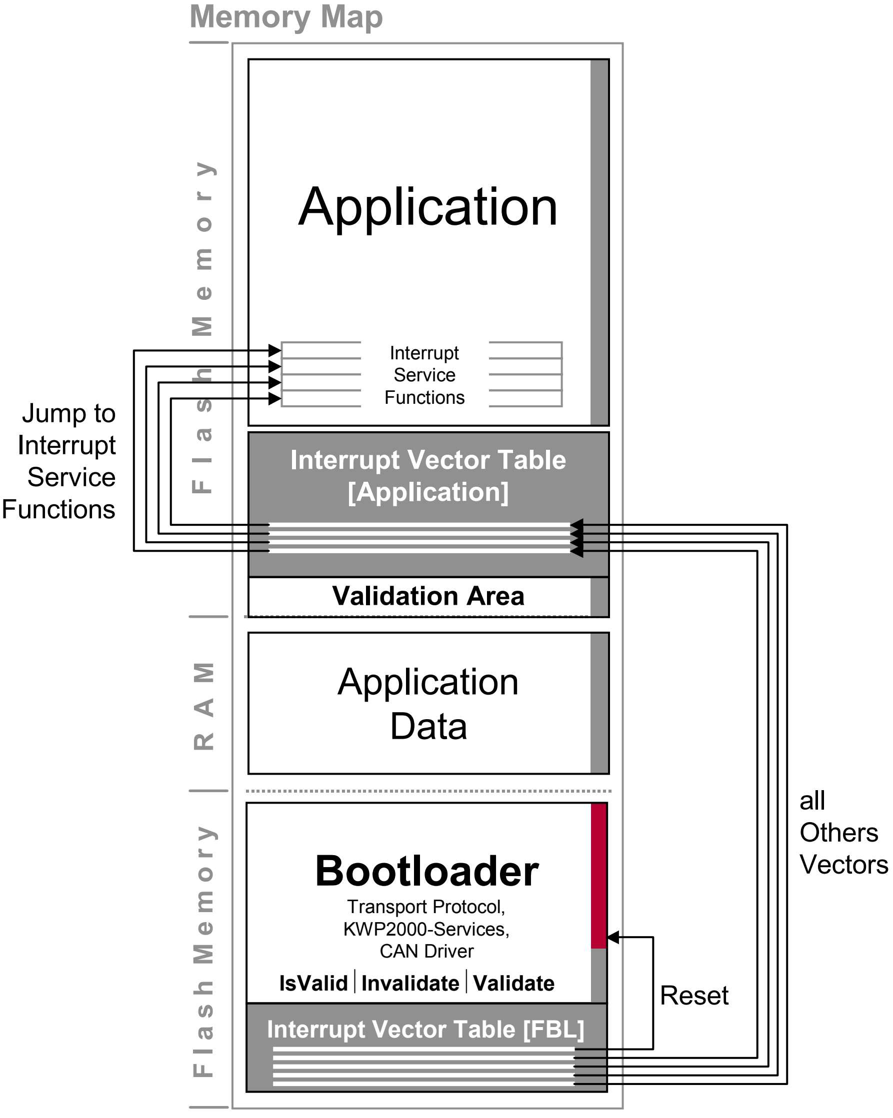


Figure 4-5 Principle Of The Two Interrupt Vector Tables

©2006, Vector Informatik GmbH

Every interrupt through Reset leads directly into the Flash Bootloader.

The interrupt vector table at the memory address provided by the hardware is used by the Flash Bootloader (Interrupt Vector Table [FBL]). Your application must use its own vector table (Interrupt Vector Table [application]).

Now if an interrupt occurs which is not a reset, then on the hardware side control branches to the memory location whose address is at the corresponding location in the Flash Bootloader's interrupt vector table (see Figure 4-5). Any interrupt but the reset points into the Interrupt Vector Table of your application. Within your application Vector Table control is branched now to the appropriate interrupt service function.

As a result, a slight overhead between detecting the interrupt and calling the interrupt service function is added to every interrupt service function. The delay is the duration of a jump instruction.

An exception to this is the reset interrupt. It always points to the Flash Bootloader.

There is no direct path from reset into the application.

The CAN Driver working in the Flash Bootloader needs no interrupts itself, since it is running in polling mode. It checks cyclically to see if CAN messages have been received. Reception is not indicated via an interrupt, as is usually the case. For this reason no additional interrupt service functions are needed in the vector table for the Flash Bootloader.

You will find further particulars about the modifications you have to deal with to adjust your interrupt vector table in Section 6.2.4.

Refer to your Hardware-specific Documentation [#hw_intvect] to get more detailed information.

4.7 Label Reference File

Some address information need to be shared between your application and the Bootloader. Some files such as the application vector table and the module table are referenced in both the application and the bootloader. These files are usually located in the application area, since they potentially change (module table, interrupt vector table).

The data contents of these files could be erased at any time because they reside in the non-protected area.

In special circumstances the Bootloader will eventually use the files. Therefore the application needs to compile and link these files to the **same** memory location. The module with these files should be downloaded as the first module to provide the data to the Bootloader whenever needed. This ensures a proper Bootloader execution.

Refer to OEM-specific documentation [#oem_ref].

5 FLASHING IN 5 STEPS

STEP 1 : DESIGN THE MEMORY LAYOUT

Figure out which component has to be placed at which memory location. Estimate the sizes.

STEP 2: WRITE A TEST APPLICATION

This can be any application or the application you later use for the ECU. It is very important, that this application is running correctly and you can recognize its running. This application is the evidence for the correct function of the Flash Bootloader.

STEP 3: INTEGRATE THE BOOTLOADER

In this step you have to integrate the Bootloader, download it via a programmer or burner and test it with the Flash Tool. This is the major step to be done!

Read the Introduction of this step or go directly to the 8 Bootloader Integration step.

STEP 4: ADAPT YOUR TEST APPLICATION FOR THE TESTER

To prepare the application hex file for being downloaded via the Tester, it has to be converted to an OEM-specific format. Do this in this step and refer to the description provided by your OEM.

STEP 5: DOWNLOAD YOUR TEST APPLICATION WITH THE TESTER

Now do the download the test application as before but now using the OEM-specific Tester.

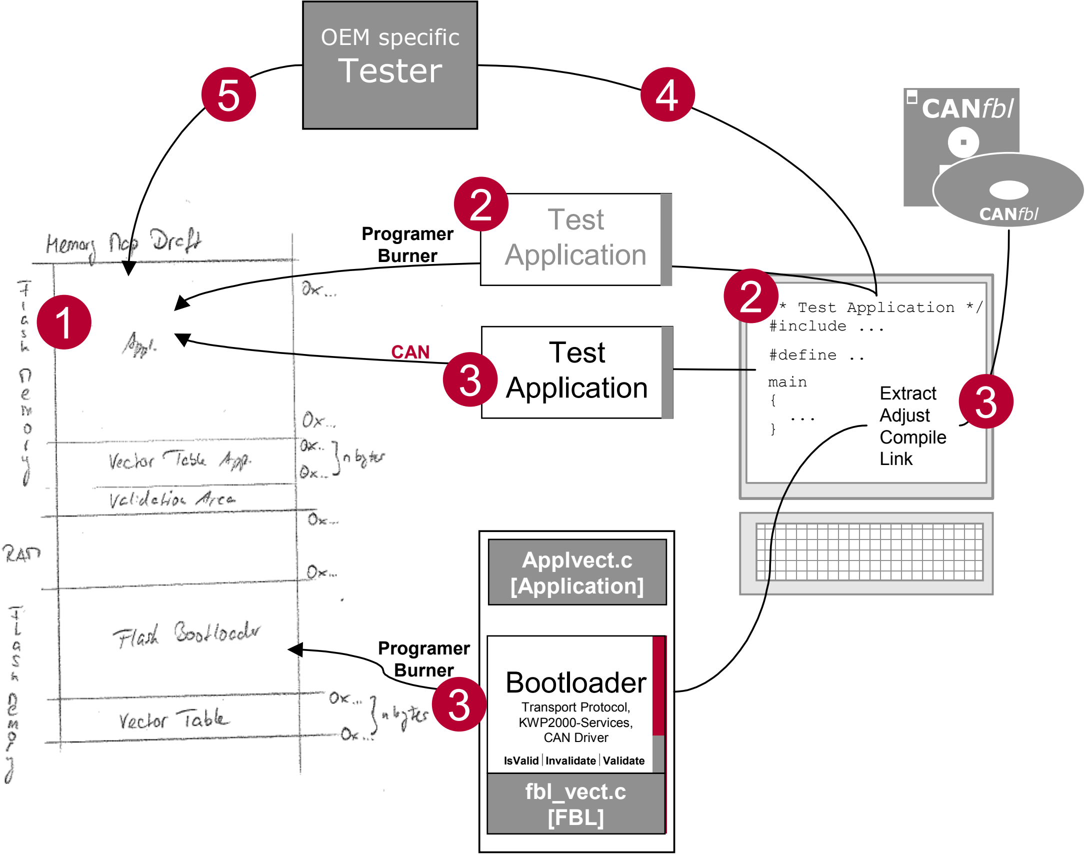


Figure 1: 5 Steps And You Are Flashing via Flash Tool and OEM-specific Tester

©2006, Vector Informatik GmbH

5.1 STEP 1 Design The Memory Layout

As flashing is a memory related activity you first need a basic estimation about the memory consumption. Based on the controller you use and the memory model start with a basic design for the memory layout.

Define where your application has its location, define the location of the application vector table and the Bootloader. Figure out where your controller has its "original" interrupt vector table.

Refer to the hardware-specific documentation for more information **[#hw_mem]**.

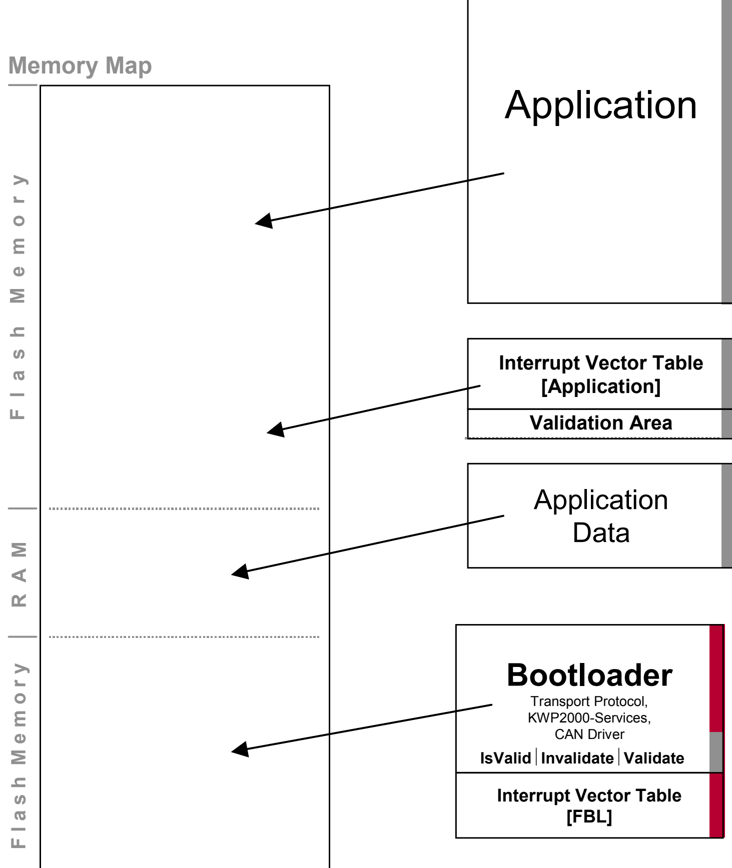


Figure 5-1 Basic Memory Layout Of An Application with the Flash Bootloader

Back to 5 Steps overview

5.2 STEP 2 Write A Test Application

The Flash Bootloader is designed to download your application via a bus protocol like CAN, LIN, etc. To test the correct function of the Flash Bootloader you first need a Test Application to verify the work of the Flash Bootloader.

In this step, write a Test Application.

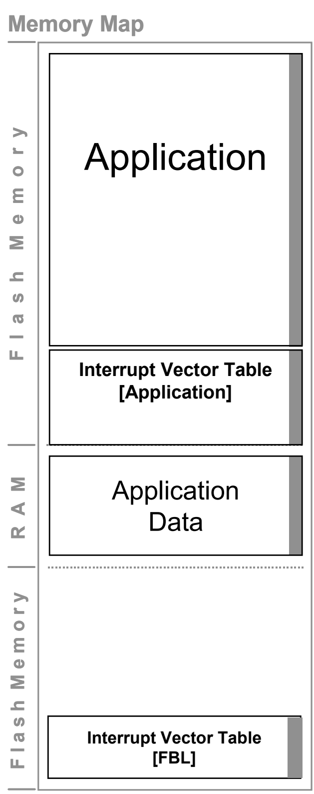


Figure 5-2 The Test Application In The ECU Memory Using Two Interrupt Vector Tables

It's recommended to write the Test Application with at least one interrupt service function, perhaps a timer to do some cyclic action. Make sure that the application is showing its correct behavior e.g. via LED blinking or even the transmission of a CAN message. This will be later on the indicator for the correct working Flash Bootloader.

Download and test this application via your development environment.

It's recommended to work with two interrupt vector tables from the beginning. Just map all interrupts from the "original" vector table to the interrupt vector table of your application.

For more information refer to the hardware-specific documentation **[#hw_tstappl]**.

Back to 5 Steps overview

5.3 STEP 3 Integrate The Bootloader

This is the major step for you to do. You have to unpack the delivered files, do some application specific adaptations, compile the Bootloader, and download it.

In order to test the Bootloader, you need to flash your test application that you created in the previous step. To do so you should configure the Flash Tool properly prior to testing.

Use the Flash Tool of the Flash Bootloader package to download your test application. Once you have managed to get your Bootloader to work with the Flash Tool then you can go on and use the OEM-specific Test Tool. Now the embedded side is working and you can concentrate on adapting the OEM-specific tester.

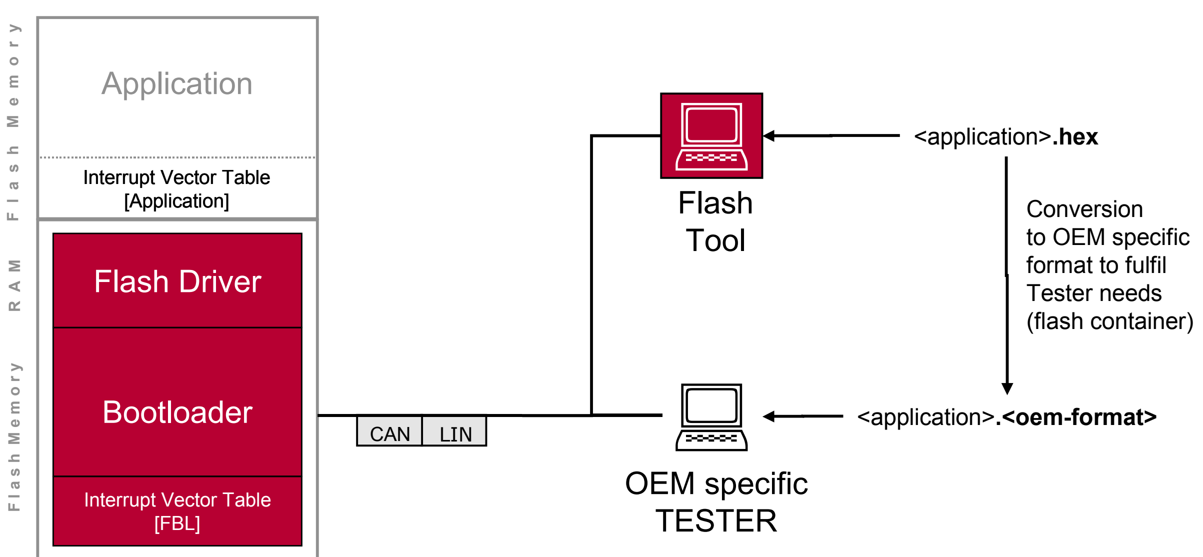


Figure 5-3 Flashing via Flash Tool and OEM-specific Tester

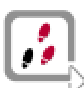
The application is compiled to a hex file to be used with the Flash Tool. The Flash Tool controls the download via the bus system (here CAN) and communicates with the Bootloader. To be able to flash with an OEM-specific tester, some adaptations have to be done to the application hex file.

For more information refer to documentation about this workflow provided by your OEM.

Integrate the Bootloader in 8 Steps more… (see Chapter 6)

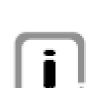
Follow the 8 Integration Steps for the Bootloader (see Chapter 6) before continuing with the next step.

Back to 5 Steps overview



5.4 STEP 4 Adapt Your Test Application For The Tester

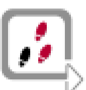
The next step after downloading the test application is to adapt your <application.hex> file to be downloaded via the OEM-specific tester (see explanation in the STEP 3).



To test these adaptations, no change of the Bootloader software is necessary.

Normally there are several steps necessary to convert the hex file into a file that can be read and used by the tester. Get more information about this workflow using the documentation provided by your OEM.

Back to 5 Steps overview



5.5 STEP 5 Download Your Test Application With The Tester

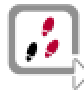
Now you can test the Bootloader using your OEM-specific Tester. You can test things such as starting a Flash programming triggered via reset, or initiated from the application, etc.

Is it still working?

Congratulations, you did it!

Back to 5 Steps overview.

Continue with the Background Information.



6 Details of Bootloader Integration Step (STEP 3)

Bootloader STEP 1: Extract the files to a folder on your PC
Bootloader STEP 2: Adjust the files to fit your application
Bootloader STEP 3: Now compile the Bootloader
Bootloader STEP 4: Transfer the Bootloader to the target hardware
Bootloader STEP 5: Use the Flash Tool to test the Bootloader
Bootloader STEP 6: Test the flashing after a reset
Bootloader STEP 7: Make your application ready for transition to Bootloader
Bootloader STEP 8: Start Bootloader from your application

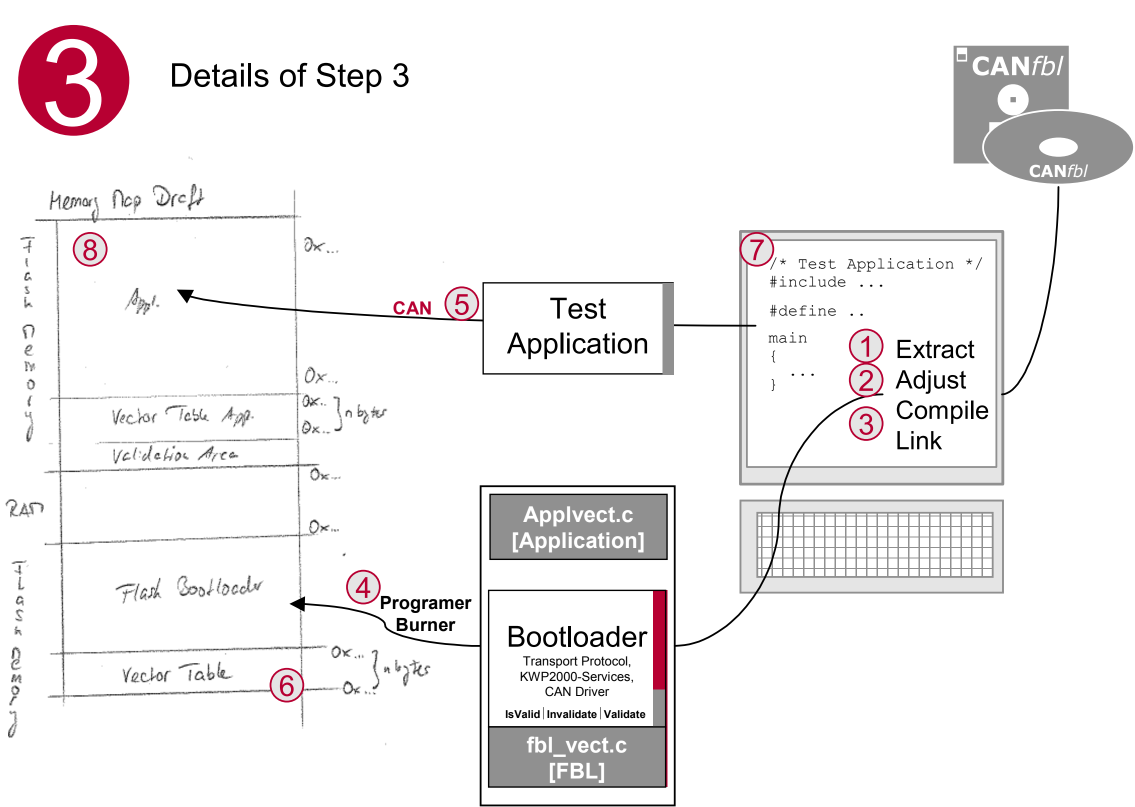
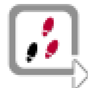


Figure 6-1 Details of Bootloader Integration Step (Step 3)



6.1 Bootloader STEP 1 – Extract the files to a folder on your pc

Extract the Flash Bootloader files just by starting the install shield. Per default the files are installed to the folder **C:\Programme\Vector\...**. You can also use the **Start\Programme\Vector…** to find your installation.

The files are installed with the following directory structure.

Name	Description
StandardECU	Root directory of the delivery, root name may differ.
_Common	Contains common files shared between the application and the bootloader. *[e.g. v_def.h, etc.]*
_Demo	Contains an example implementation.
DemoAppl	An application that is prepared to be downloaded to the micro via the Flash Bootloader and CANflash.
DemoFbl	The bootloader part of the demo. This directory contains an example of a full functional bootloader (e.g. including in-/validation, transition from application to bootloader and a comprehensive mapping of the bootloader sections).
_Doc	Contains documentation and test reports for the bootloader. *[e.g. UserManual_FlashBootloader.pdf, TechnicalReference_FBL_*.pdf, TestReport*.pdf, etc.]*
_FlashScript	CANdito flash scripts provided with the bootloader. This directory only exists if there are flashscripts available for the SLP.
_GenTool	Generation tool for the flash bootloader configuration files. In case of CANgen, the license file is located here. *[e.g. CanGen.exe, license.liz, GENyFramework_*.exe, etc.]*
Components	Directory for GENy component-DLLs. If GENy is used, the license can be found here. Otherwise, this directory does not exist. *[e.g. license.liz, Version.Info, preconfig*.pco, etc.]*
_MakeSupport	Make environment used by demo bootloader and application. This folder contains the global makefile. *[e.g. Global.Makefile.target.make.*, etc.]*
_Misc	Everything that doesn't fit into the other directories can be placed here (e. g. little tools, that don't require an installation procedure)
HexView	The best tool to generate, edit and process hex-files. Can build plenty of flash containers, too.
_Setup	Setup files for PC-installable software included in the delivery (e. g. CANflash) *[e.g. CANflashFord25.EXE, etc.]*
DrvEep	EEPROM-driver to be used with the bootloader *[e.g. EepDrv.c, EepIO.c, EepCfg.h, etc.]*
DrvFlash	Flash-EEPROM-driver to be used with the bootloader (aka secondary bootloader). This directory contains the source files of the

	driver and a ready-to-use function file container for the used SLP. *[e.g. flashdrv.c, flashdrv.h, FlashDrv_V850_f.hex, etc.]*
Fbl	Contains all bootloader files, except flash-driver and security module *[e.g. fbl_main.c, fbl_hw.c, fbl_diag.c, etc.]*
_Template	Contains files that are part of the flash bootloader but need to be adapted for integration purposes. The files contain a collection of callback-functions grouped in different files for different purposes. The functions mainly adapt to the specific needs of the bootloader to a particular project resp. adapts the hardware requirements of the Bootloader. *[e.g. _fbl_ap.c, _fbl_apdi.c, _fbl_apwd.c, _fbl_apfb.c, etc.]*
SecMod	Security module for the bootloader. *[e.g. secmod.c, _secmod.h, etc.]*

The files listed in the table are installed to the folder **Fbl**. The files with the _ (underscore) are installed to the folder **Fbl_Template, v_def.h** is in the **_Common** path and the flashdrv.h in the **DrvFlash** folder.

Since the Flash Bootloader is an independent application, it must be possible for you in most cases to take the directory structure, as unpack it.

File Name	Description	Status
Makfile of project file	Makefile or project file for the build process of the Bootloader.	
fbl_cfg.h	Global Bootloader configuration. This file is generated by the configuration tool.	
_fbl_apxx.c / _fbl_apxx.h	Hardware and system specific callback functions.	
_ftp_cfg.h	Transport layer configuration file. Very soon, this file is also generated by configuration tool.	
_fbl_inc.h	Include file for the Bootloader. Include additional header files if necessary.	
_applvect.c / applvect.h	Application vector table	
fbl_diag.c / fbl_diag.h	General Diagnostic Module that contains the diagnostic handling (KWP2000) and the basic bootloader functionalities.	
fbl_can.h	Definitions for CAN interface.	
fbl_def.h	Basic Bootloader definitions.	
fbl_hw.c / fbl_hw.h	Hardware-specific module for CAN and timer.	
fbl_main.c	Main module for Bootloader initialization and idle loop.	
fbl_tp.c / fbl_tp.h	Transport layer for the FBL.	

©2006, Vector Informatik GmbH

fbl_vect.c	Bootloader vector table.	
fbl_wd. / fbl_wd.h	Watchdog support.	
flashdrv.h	Interface header file for Flash Driver.	
v_def.h	Type definitions from Vector Informatik	

Caution
It is absolutely necessary that YOU adapt the startup-code for the Bootloader to your specific hardware platform!

Remind that there will be two startup-codes executed subsequently, first the startup-code of the Bootloader, then the startup-code of your application.

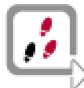
!!! be careful with registers that can be written only once after reset !!!

You can adapt these files according to your application (fbl_vect cannot be adapted by user). A detailed description of how that is to be handled can be found in Section 6.2. In your delivery all files that you have to adapt are marked with an underscore (_<file>) before the name and stored in the **_Template** folder. Create an own folder to store the adapted files without the underscore.

OEM specific

OEM specific – some more files, refer to your OEM specific reference [#oem_files].

Back to 8 Steps Bootloader integration overview



6.2 Bootloader STEP 2 Adjust the marked files to fit your application

Now you go through the files pointed to by the hand in the above diagram in detail and adapt them specifically to your application.

6.2.1 Make… Makefile and make.exe

To be able to compile the Flash Bootloader you just have to adapt the file makeconf by setting the compiler path to your demands, see an example

e.g. COMPILER_PATH = c:\uti\hc12\cx32

Then you can execute the make.exe to compile and link for the first time. Now you can start upon this basis and adapt the files for your demands.

6.2.2 fbl_cfg.h - The Configuration File For The Flash Bootloader

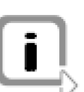
If you are using a Generation Tool with the bootloader then the Generation Tool creates this file. To modify the bootloader you would simply have to trigger the generation process again. If you are not using the generation tool with the bootloader then you can manually configure the bootloader by modifying defines such as clock frequency, CAN baudrate, etc…

See the list of the possible switches below to be changed followed by a brief description. To get more information refer to the comments in the file fbl_cfg.h.

`CAN_TP_RXID`	Receive ID for the transport protocol
`CAN_TP_TXID`	Send ID for the transport protocol
`FBL_ENABLE/DISABLE_DEBUG_STATUS`	Additional hints for debugging (if possible)
`FBL_ENABLE/DISABLE_SYSTEM_CHECK`	Checks if data buffer will be overwritten
`FBL_ENABLE/DISABLE_FLASHBLOCK_CHECK`	The pre-defined flashblocks in FBL_AP are checked and aligned during download
`FBL_ENABLE/DISABLE_APPL_TASK`	Enable a cyclic task for some timing adjustments (call cycle TpCallCycle typ. 1ms).
`FBL_MAX_NUMBER_OF_MODULES`	Set here the number of modules that shall be downloaded and programmed.
`FBL_ENABLE/DISABLE_SECTOR_ERASE_FCT`	Enable the usage of the callback function ApplFblSectorErase()
`FBL_ENABLE/DISABLE_FILECHECKSUM`	Enables an internal file checksum calculation. This checksum is generated during the download sequences

©2006, Vector Informatik GmbH

FBL_ENABLE/DISABLE_ENCRYPTION_MODE	If you enable this switch, further functions are called to decrypt coded data. (ApplFblEncryptInit, ApplFblEncryptData).
FBL_WATCHDOG_ON/OFF	Switch Watchdog on or off
FBL_WATCHDOG_TIME	Setting of the watchdog trigger cycle
FBL_PROCESSOR_40MHZ	Some clock settings may not be available on all CPUs. Please refer to fbl_hw.c, FblTimerInit
FBL_ENABLE/DISABLE_STAY_IN_BOOT	Setting FBL_DISABLE_STAY_IN_BOOT, it is not possible to force the bootloader not to start the application.
FBL_DIAG_BUFFER_LENGTH	This is the size of the diagnostic data buffer used for USDT reception/transmission (COMMON_BUFFER mode).
FBL_START	Start address of the FBL.
FLASH_SIZE	Specifies the number of bytes used for the Flashcode (aka flash driver). **Note**: Allocate as much memory as possible to be able to download a bigger flash driver in the future.
CAN_BTR01	Bus timing configuration for normal mode



Set up and initialize these functions according to your needs. When doing this, follow the descriptions for each and use Figure 6-2 for a complete overview.

6.2.3 FBL_apxx.C

The next thing you have to do is adapt the Flash Bootloader to your application and hardware. The following functions have to be adapted:

Function Name	File Name
ApplFblInit	fbl_ap
ApplFblStartup	fbl_ap
ApplTrcvrNormalMode	fbl_ap
ApplFblSetVfp	fbl_ap
ApplFblResetVfp	fbl_ap
ApplCanParamInit	fbl_ap
ApplFblFlashBlockNotFound	fbl_ap
ApplFblTask	fbl_ap
ApplFblIsValidApp	fbl_ap
ApplFblInvalidateApp *, ApplFblValidateApp *	fbl_ap
ApplFblSecuritySeed	fbl_ap
ApplFblSecurityKey	fbl_ap
ApplFblSectorErase	fbl_ap
ApplFblWDInit	fpl_apwd
ApplFblWDShort	fbl_apwd
AplFblWDTrigger	fbl_apwd
ApplFblWDLong	fbl_apwd

You will find these functions again in Figure 6-2.

©2006, Vector Informatik GmbH

Application

Bootloader

FblStart

ApplFblWDShort

yes

ApplFblIsValidApp

no

OEM specific
wait time

ApplFblInit
Optional: initialize WD
and set trigger time *

Reset

ApplFblStartup

ApplFblStartup

ApplFblWDInit
Optional: initialize WD
and set trigger time *

ApplFblCanParamInit

ApplTrcvrNormalMode
[End of Initialization Phase]

Wait for Diagnostic Service to start Flash

ApplFblWDTrigger

ApplFblSecuritySeed
[Queried via KWP2000]

ApplFblSecurityKey
[Queried via KWP2000]

Download of Flash Algorithms

ApplFblSetVfp

ApplFblInvalidateApp

Erase Flash Memory

ApplFblSectorErase (opt.)

ApplFblFlashBlockNotFound

Download Application Data

ApplFblValidateApp

ApplFblResetVfp

ApplFblWDLong

Dependent on OEM

Figure 6-2 Function Calling Sequence During Flashing

©2006, Vector Informatik GmbH

The Watchdog may only be initialized in one of the two optional functions. In most cases this is the function **ApplFblInit**. Further details on the background and possibilities of Watchdog handling can be found in Section 7.1.

The description of the functions below is done in same order as they are called beginning with the reset.

6.2.3.1 ApplFblInit

The ApplFblInit function

The function ApplFblInit is called after every reset. You can do your basic initializations, such as memory mapping, PLL setup, etc. Usually initialization of the Watchdog timer is dealt with here. (See 3.3 for more details).

6.2.3.2 ApplFblCanParamInit

The function ApplFblCanParamInit

Callback function for multi ECU support. See 7.2.

6.2.3.3 ApplFblIsValidApp

The function ApplFblIs ValidApp

Callback function to check the validity of the application. See 4.4.2.

6.2.3.4 ApplFblStartup

The function ApplFblStartup

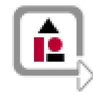
This function is called after **ApplFblInit** if no valid application was found or the Bootloader was started by the application (reprogramming request). Use this function to do initializations, which are needed by the application and the Bootloader. This is useful if you want to initialize hardware or software explicitly used for the Bootloader.

6.2.3.5 ApplFblWDInit

Be sure to only initialize the watchdog timer in one location

The function ApplFblWDInit is only called if control remains in the Flash Bootloader after a reset.

You can also initialize your Watchdog here if you have not done this already in the function **ApplFblInit** (see 6.2.3.1) (for more details about the watchdog see 3.1).



Example:

```
void ApplFblWDInit(void)
{
Your code for initializing your Watchdog can go here, or nothing, if you either
won't be using it or you want to initialize it somewhere else.
}
```

©2006, Vector Informatik GmbH

6.2.3.6 ApplTrcvrNormalMode

The function ApplTrcvNormalMode

The name of this function is self-explanatory. To send CAN messages it is first necessary to setup the transceiver. This can be done here.

Example:

```
void ApplTrcvrNormalMode(void)
{

  DDRCAN   = 0xff;  /* set the port direction */
  PORTCAN = 0x30;  /* set the port */

}
```

6.2.3.7 ApplFblSecuritySeed / Key

Refer to your OEM-specific Documentation [#oem_sec] to get the necessary information.

6.2.3.8 ApplFblSetVfp

The function ApplFblSetVfp

Whether you need to configure this function or not depends on your hardware setup (some flash memories require external programming voltage to erase and program). If you have to turn on the voltage for flash programming, you can do that in this function.

Normally you would set an I/O port to enable an external flash supply.

Example:

```
void ApplFblSetVfp(void)
{
for example:
  PORTA  &= ~0x01;
}
```

6.2.3.9 ApplFblInvalidateApp

The function ApplFblInvalidateApp

Callback function to invalidate an application. See 4.4.2.

6.2.3.10 ApplFblFlashBlockNotFound

The function ApplFblFlash BlockNotFoun

This function is called if a TransferData has been received but no address region was found in the defined FlashBlocks. It also allows you to support out of main memory downloads (for example: Additional Flash or EEPROM).

6.2.3.11 ApplFblValidateApp

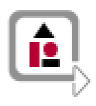
The function ApplFblValidateApp

Callback function to validate an application. See 4.4.2.

6.2.3.12 ApplFblResetVfp

The function ApplFblReset Vfp

The function ApplFblResetVfp is the counterpart of the function ApplFblSetVfp. This function allows you to turn off the programming voltage.



Example:

```
void ApplFblResetVfp(void)
{
For example:
  PORTA |= 0x01;
}
```

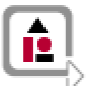
6.2.3.13 ApplFblWDLong

The function ApplFblWDLong

Adjusts watchdog timing for application if necessary.

This function is called immediately after flashing, before branching from the Flash Bootloader into your application. For watchdogs with changeable timer intervals, switching to the other interval can be done here.

> If you want to use a hardware reset for the transition, implement an infinite loop in this function. The Watchdog timer will expire and create the desired reset.



Example:

```
void ApplFblWDLong(void)
{
 Here is the place for your infinite loop or your changeover. If you do not need
this function, simply leave this blank.
}
```

Starting the flash process from the application, the following call back functions have to be adapted.

6.2.3.14 ApplFblWDShort

The function ApplFblWDShort

The function ApplFblWDShort is called during the transition from the application into the Flash Bootloader. With it you can again re-adjust the watchdog timer interval.

If you have a window watchdog, you can synchronize the watchdog here.

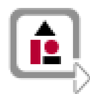
Example:

```
void ApplFblWDShort(void)
{
 Here is the place for your changeover. If you do not need this function, simply
leave this blank.
}
```

©2006, Vector Informatik GmbH

6.2.3.15 ApplFblWDTrigger

The function ApplFblWDTrigger



If you enable the watchdog timer, then you must refresh it in the function ApplFblWDTrigger. An example of how this is done follows.

Example:

```
void ApplFblWDTrigger(void)
{
/* You must operate your Watchdog here. The code could look like this: */
  PORTB   |=    cWatchdogPin;
  PORTB   &=   ~cWatchdogPin;
}
```

©2006, Vector Informatik GmbH

6.2.4 Fbl_vect.c / Applvect.c(.h) - The Interrupt Vector Tables

There are several files provided for the interrupt vector tables. Fbl_vect.c (.h) is the vector table for the Bootloader, Applvect.c (.h) for the application.

On some hardware platforms the vector table may also be implemented in assembly language. The files names will have corresponding extensions in this case.

For the integration of the Bootloader you have to compile and link the provided files fbl_vect **and** applvect.c / applvect.h to your Bootloader files. To adjust the location of the application vector table, just link the APPLVECT segment to the desired memory location. The location of the application vector table must be fixed and the same for the bootloader as well as the application.

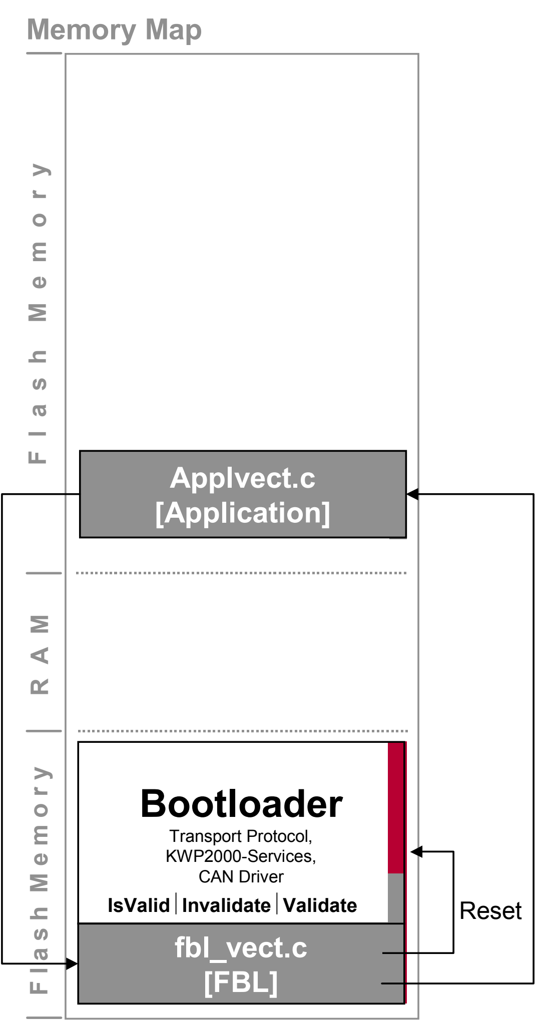


Figure 6-3 Situation Directly After The Programming Of The Bootloader Together With The Dummy Application Vector Table

The provided application vector table is just a dummy table to provide the Bootloader with the memory address of the application vector table. All vectors within the applvect.c point to the startup code of the Bootloader.

If you now flash your application for the first time, this dummy application interrupt vector table will be overwritten with your application vector table. Make sure that the memory location is exactly the same.

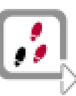
> It is recommended to use the applvect.c file from the delivered example application as basis for your application vector table and insert the name of your interrupt service functions at the appropriate locations in the file.

.

Refer To hardware-specific Documentation [#hw_intvect].

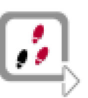
Back to 8 Steps Bootloader integration overview

6.3 Bootloader STEP 3 Now compile the Flash Bootloader



Now all the files and functions have been adapted. Call the file make.exe again to compile them. The result is a file XYZ.hex, that is, the Flash Bootloader in hex format.

6.4 Bootloader STEP 4 Transfer the Bootloader to the target hardware



In order to test your result, you now have to load the Bootloader hex file onto your target platform.

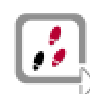
6.5 Bootloader STEP 5 Use the Flash Tool to Test the Bootloader

Open the Flash Tool (for installation and how to use it see the FlashTool Documentation). Go to **Options\Paths** and set the paths.

> Before you start the download, be sure that you have a CAN connection to your ECU.

Press the Start-Button to start the Flasher. The Flash Tool will now cyclically send the CAN Flash messages.

6.6 Bootloader STEP 6 – Test the flashing after a reset



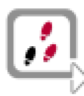
If you do a reset now, the ECU, triggered by the Flash Tool's cyclically sent CAN messages, should start the flash process. You can see the flash process in the window of the Flash Tool.

Did it work properly? If so, the main work is already behind you.

> **If not**: check the baud rate, the CAN connection and the hardware initializations.

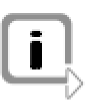
Back to 8 Steps Bootloader integration overview

6.7 Bootloader Step 7 – Make your application ready for the transition to the Bootloader



Another possible way to start the Flash Bootloader is from the application. A common way of starting the bootloader from the application is by receiving a certain diagnostic service or because by a special-purpose CAN message.

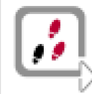
As you see in the Figure 6-2 the transition from the application to the Bootloader is done via the function FblStart. The parameter of this function is a hardware dependent structure that contains e.g. the baud rate, the bit timing, CAN-ID, etc.



Some OEMs realize the transition from the application to the Bootloader via a reset.

Refer to your OEM-specific Documentation [#oem_trans] to get the necessary information on the correct sequence of events to switch from your application to the Bootloader.

6.8 Bootloader Step 8 – Start Bootloader from your application



Compile your application with the jump to bootloader supported as described in the previous step. Then load it on your target hardware using the flash tool. Now flashing works in cooperation with the flash tool. The final two steps deal with the preparation of the application hex file for download via an OEM tester.

Ok? It is working?

If not, go back to the STEP 3, or continue with the STEP 4 if it works.

©2006, Vector Informatik GmbH

7 Background Information

7.1 The Watchdog

The Bootloader needs to trigger an on-chip or external watchdog while downloading the application.

Refreshing the watchdog is hardware/application specific and therefore must be implemented by the user. There are two watchdog functions:

- **FblLookForWatchdog:** Internal function to generate the time base for triggering the watchdog (must not be changed by the user)
- **ApplFblWDTrigger:** Hardware/application specific call-back function for refreshing the watchdog

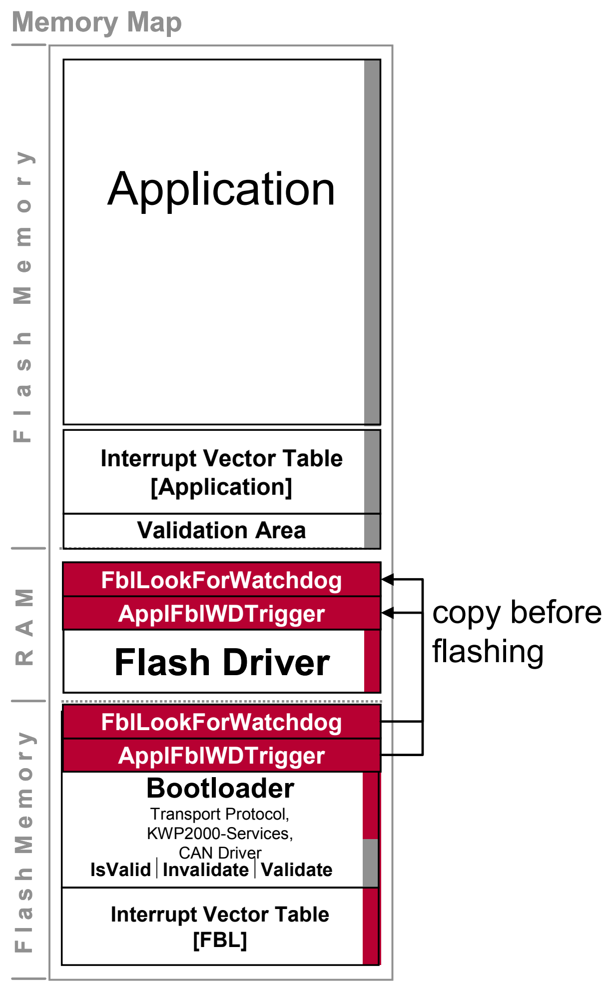


Figure 7-1 Memory Layout Of The Watchdog Trigger Functions

©2006, Vector Informatik GmbH

The watchdog trigger functions have to be relocatable!

Both functions are copied into RAM while programming the Flash, as it’s not possible on most MCUs to execute an application from Flash while reprogramming parts of the Flash. Copying the functions into RAM is either accomplished by a copy function of the Bootloader (**FblCopyWatchdog**) or by the startup function. In the first case, the watchdog trigger functions must be re-locatable (not really)!

You are already familiar with Figure 7-2. The main attention is now on those functions in which handling or manipulating the watchdog timer can take place.

Application

Bootloader

FblStart

ApplFblWDShort

yes

ApplFblIsValidApp

ApplFblInit

Reset

OEM specific wait time

no

Optional: initialize WD and set trigger time *

ApplFblStartup

ApplFblWDInit

Optional: initialize WD and set trigger time *

ApplFblCanParamInit

ApplTrcvrNormalMode
[End of Initialization Phase]

Wait for Diagnostic Service to start Flash

ApplFblWDTrigger

ApplFblSecuritySeed
[Queried via KWP2000]

ApplFblSecurityKey
[Queried via KWP2000]

Download of flash algorithms

ApplFblSetVfp

ApplFblInvalidateApp

Erase Flash Memory

ApplFblSectorErase (opt.)

ApplFblFlashBlockNotFound

Download Application Data

ApplFblValidateApp

ApplFblResetVfp

ApplFblWDLong

Dependent on OEM

Figure 7-2 Functions For Manipulating The Watchdog

©2006, Vector Informatik GmbH

7.1.1 Initializing The Watchdog

With some watchdogs double initialization can lead to a reset.

You can initialize the watchdog in one of two different places: either in the function **ApplFblInit** or in the function **ApplFblWDInit**. If you want to enable the watchdog within the Stay-In-Boot time where the bootloader waits 50ms (default) to receive the CAN message, then you have to do the initialization in the function **ApplFblInit**. In that case the function **ApplFblWDInit** MUST remain empty.

On the other hand it is also true, of course, that if you initialize in the function **ApplFblWDInit** you must not initialize in the function **ApplFblInit**.

Use WDTimer [in ms] to easily set your Watchdog operating times.

The user should initialize the watchdog since this is hardware dependent. To find your register settings to initialize the watchdog, please refer to the controller manual.

You can however use **WDTimer** for triggering. If you set **WDTimer** to **0**, then the Watchdog will timeout immediately. The unit for **WDTimer** is ms. The Watchdog is operated via the function ApplFblWDTrigger, which you have already adapted to your needs (see 6.2.3.5 and 6.2.3.15).

Example:

```
/* Initializing the WD, hardware-specific */
WDTimer = 250/*ms*/;  /* in this way the Watchdog is triggered after 250ms.
```

Changing the value of WDTimer will only influence the time of refreshing the watchdog for that particular cycle. The watchdog refresh rate will then be reinitialized to the value set in fbl_cfg.h as soon as the watchdog is serviced. (see 2.4.2).

If your watchdog is a window watchdog, or at least supports different monitoring windows and you want to use these as well, then you can use both of the functions **ApplFblWDLong** and **ApplFblWDShort** for toggling the monitoring times.

Refer to the hardware-specific documentation **[#oem_wd]** for more information about the watchdog.

7.2 Multiple ECU Support

The Flash Bootloader supports multiple ECUs (When similar ECUs are used multiple times in a vehicle, and their CAN identifiers are configurable).

This feature can be activated by setting the

#define ‘FBL_ENABLE_MULTIPLE_NODES’ in the file fbl_cfg.h or via the Generation Tool depending on your OEM.

Refer to your OEM-specific Documentation [#oem_multi] to get the necessary information.

©2006, Vector Informatik GmbH

If multiple ECUs are supported, the FBL will call the function **ApplFblCanParamInit()**, where the user has to decide which set of CAN identifiers are used. Some OEMs set additional information, e.g. like identification of the ECU.

7.3 Validation Ok – Application Faulty

As mentioned in the earlier chapter 4.3.1 the validity check may result in a positive response and sign the application valid, even if the application itself is faulty. In that case every reset will lead to an application that does not work properly (for example: The CAN communication of the application does not work anymore).

> The validation process does not recognize the problems in the application (e.g. a damaged byte in flash, etc.).

How can you flash an error free application in that case?

To do this it must be possible to react on a CAN message during the transition from a reset to the application. Using the switch FBL_ENABLE_STAY_IN_BOOT in the `fbl_cfg.h` file (see 6.2.1) you get the following modified function calling sequence.

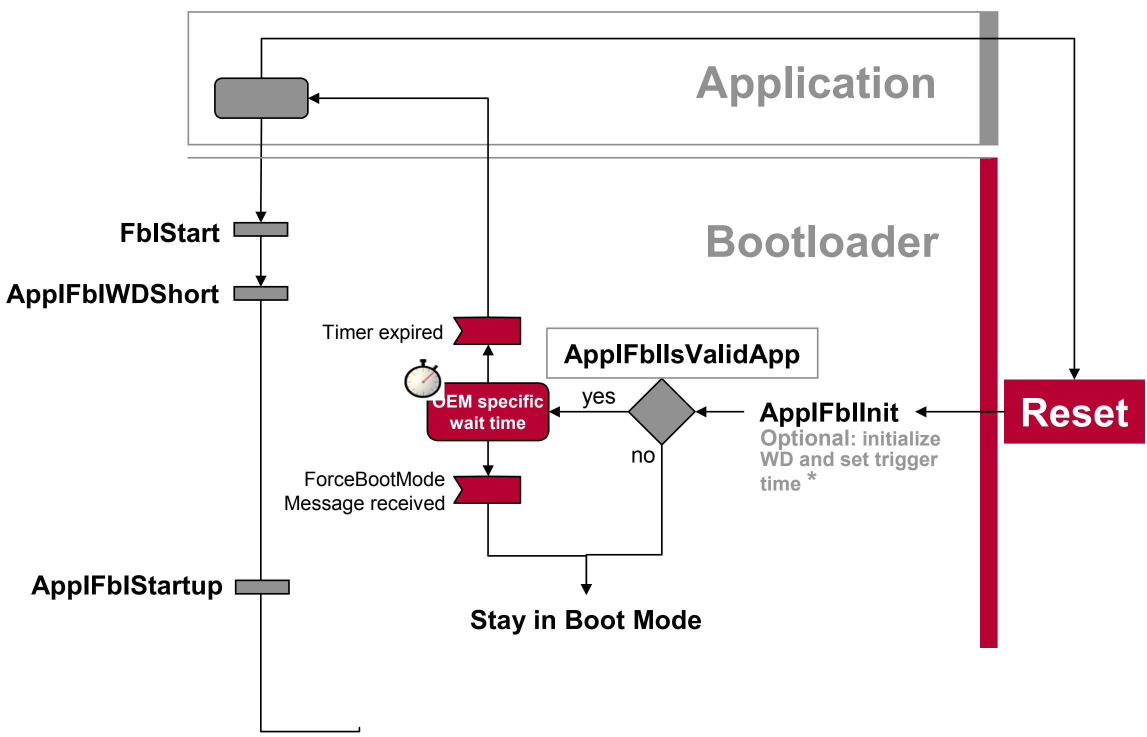


Figure 7-3 Modified Function Calling Sequence

After the application is checked to be valid a timer is started. The FBL waits a default time (refer to OEM-Specific Documentation **[#oem_time]**) to receive a CAN message (ForceBootMode Message) as a trigger to stay in the boot mode. Now a new flash process can be triggered.

©2006, Vector Informatik GmbH

If the timer expires with no receive message the application will be executed.

With this mechanism it is now possible to flash an application, which had been set valid erroneously.

The disadvantage is that the startup time will be longer (watchdog).

7.4 FlashSegmentSize

The Flash Segment Size is a configurable parameter in the Flash Tool.

When you are modifying the **FlashSegmentSize**, enter the smallest segment of flash memory that can be programmed by the Flash Bootloader (hardware dependent information **[#hw_size]**).

In other words, a block must start at the beginning of a flash segment, for Flash Bootloader to write to it. This specification depends on your hardware. The Flash Tool needs this information in order to optimize the write process. The tool must optimize exactly if area boundaries do not fall on flash block boundaries.

Enter the value for your Segment Size in the flash tool in the Flash File window. (See FlashTool Documentation)

FlashSegment-Size gives the minimal size of the data in bytes which the CANflasher can write to at once. Thus at least this much data must be written simultaneously. The size of FlashSegment-Size depends on your hardware.

7.4.1 Why Does The Tool Have To Know This Block Length?

In the following example let us take a Flash Bootloader that can write a minimum of 64 bytes; this corresponds to 40 in hexadecimal representation.

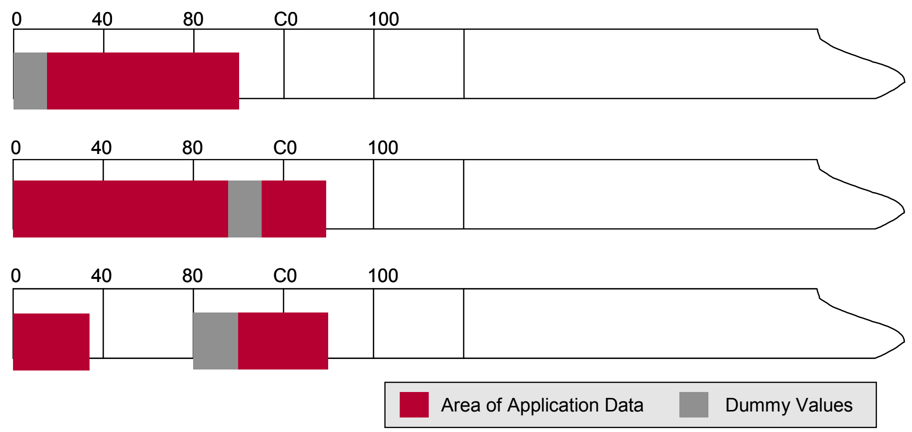


Figure 7-4 Segmenting During Flashing

In the first example the data area to be written to (red) is distributed over 3 consecutive segments and does not start precisely at the beginning of a segment. The Flash Bootloader then fills this gap with dummy values (gray) and in this way a segment consisting of 3x64 bytes can be written to memory at once. The gap at the end will be filled with the values that are already in the flash memory.

In the second example two data segments lay one after the other, but there is a gap between them. To prevent the Flash Bootloader from having to write to the segment from 0x80 to 0xC0 twice, it also fills up the gap here and the region up to the next segment boundary with dummy values, and now a segment with the length of 4X64 bytes can be written to at once. The gap at the end will be filled with the values that are already in the flash memory.

©2006, Vector Informatik GmbH

In the third example two data segments have to be written to here, too, but the gap between them is greater than one segment. In this case, in order to not have to write to a completely unused segment, the Flash Bootloader now divides up the task differently. For this reason the segment between 0x40 and 0x80 remains empty here.

7.5 Frequently Asked Questions

A list of frequently encountered problems is provided here to facilitate troubleshooting.

7.5.1 Bootloader Crashes

Q: The Bootloader simply crashes after reset

A: Check if the Bootloader accidentally started the application – check validity information by setting a breakpoint at **ApplFblIsValidApp**.

Also verify that the Bootloader and application locate the Application Vector Table to the same address.

Q: I can start the download but the software crashes when the Flash is erased

A: The Bootloader may crash for several reasons:

1. The Flash Driver was not correctly copied from ROM to RAM. Check if the byte-array (**flashCode**) for the Flash Driver is large enough to hold it.
1. Check your watchdog routine. Check if the byte-array for the watchdog function (**WDTriggerBuffer**) is large enough to hold the watchdog function.
2. Check that the watchdog trigger function is relocatable.
3. Check the **FlashBlock** structure in the “fbl_apfb.c” file. Make sure that the memory area occupied by the Bootloader is excluded from the **FlashBlock** structure.

Q: The Flash is erased, but the Bootloader crashes before the application is downloaded.

A: Check the **FlashBlock** structure in the “fbl_apfb.c” file. Make sure that the memory area occupied by the Bootloader is excluded from the **FlashBlock** structure.

Q: I can start the Bootloader software but after some time, the Bootloader crashes

A:

4. Check your watchdog routine. Check if the byte-array for the watchdog function (**WDTriggerBuffer**) is large enough to hold the watchdog function.
5. Check that the watchdog trigger function is relocatable.

Disable watchdog for testing purpose.

Q: The Bootloader crashes when programming a specific Logical Block

A: Check the **FlashBlock** structure in the “fbl_apfb.c” file. Make sure that the memory area occupied by the Bootloader is excluded from the **FlashBlock** structure. Also make sure to exclude non-Flash (RAM, Registers, EEPROM) areas from the **FlashBlock** structure.

Q: The Bootloader is cyclically restarted

A:

1. Check your watchdog routine. Check if the hardware watchdog is serviced correctly. Disable watchdog for testing purpose.
2. Maybe the application was started and didn’t trigger the watchdog. Check validity information.

Q: I can start the download but sometimes, the transfer is aborted with a timeout.

A:

1. Check the watchdog timeout and watchdog trigger
2. Check the setting of the FBL_DIAG_TIME_P3MAX and the diagnostic response timeout of your Tester/the Flash programming tool. The Bootloader must transmit cyclic “Response Pending” messages on the bus. You may check this with a CANoe/CANalyzer tool.

7.5.2 Application Is Not Started

Q: I can download the application, but after a reset the Bootloader is still active

A: There could be different reasons for this:

- Check the validity information. Set a breakpoint in **ApplFblIsValidApp** and verify it. Did you download all necessary parts of the application?

Q: When I download the application for the first time and restart the ECU, the application is running. If I reprogram the application, the Bootloader is active after reset.

©2006, Vector Informatik GmbH

A: Check if the reprogramming flag for the Bootloader is correctly reset after starting the Bootloader.

7.5.3 Bootloader Is Not Started

Q: I can download the application and the application is running, but I cannot reprogram the application

A: Check if the reprogramming flag for the Bootloader is set.

7.5.4 The Flash Tool's Error Codes

Q: How can I interpret the error codes of the Flash Tool?

A: Find the error code that occurred during flashing from the table in fbl_diag.h to be help narrow down your search for the cause.

8 Index

ApplFblInit ... 51

ApplFblInvalidateApp ... 19, 20, 41

ApplFblIsValidApp ... 18, 19, 20, 40

ApplFblValidateApp ... 19, 20, 42

ApplFblWDInit ... 51

Bootloader . 11, 12, 13, 14, 17, 18, 19, 20, 21, 25, 26, 33, 36, 38, 40, 42, 46, 47, 51, 53, 54

Call-Back Functions

- ApplFblStartup ... 40

Call-Back Functions

- AplFblWDTrigger ... 38
- ApplCanParamInit ... 38
- ApplFblFlashBlockNotFound ... 38
- ApplFblInit ... 38
- ApplFblInvalidateBlock ... 38
- ApplFblIsValidApp ... 38
- ApplFblResetVfp ... 38
- ApplFblSetVfp ... 38
- ApplFblStartup ... 38
- ApplFblTask ... 38
- ApplFblWDInit ... 38
- ApplFblWDLong ... 38
- ApplFblWDShort ... 38
- ApplTrcvrNormalMode ... 38

FBL_MTAB ... 23

Flash Tool ... 11

Flashcode ... 19

Flashing ... 13

FlashSegmentSize ... 53

Initializing ... 51

interrupt vector tables ... 25

Interrupt Vector Tables ... 25

Invalidate ... 19, 21, 22

IsValid ... 19, 21, 22

KWP2000 ... 13

Label Reference File ... 26

Module Table ... 23

Motivation ... 3

Proposal ... 21, 22, 24

RAM ... 15, 19

Reset ... 18, 26, 46

Segment Boundary ... 53

Step ... 46, 47

The Interrupt Vector Table ... 44

valid ... 19, 20, 21, 22

Validate ... 19, 21, 22

Validation Area ... 19, 20, 21, 22

validity ... 20, 21

vectortable.c ... 44

Watchdog ... 40, 42, 43, 51

- ApplFblWDTrigger ... 48
- FblCopyWatchdog ... 49
- FblLookForWatchdog ... 48

Watchdogs ... 40, 49

WDTimer ... 51

©2006, Vector Informatik GmbH