Flash Bootloader Hardware

TechnicalReference

CANfbl Renesas RH850

Version 1.00.01

Authors	Christian Bäuerle; Robert Schäffner
Status	Released

Document Information

History

Author	Date	Version	Remarks
Christian Bäuerle	2013-06-12	01.00.00	Creation
Robert Schäffner	2015-05-29	01.00.01	Removed template parts

Table 1-1 History of the Document

Reference Documents

No.	Source	Title	Document No.	Version
[1]	Renesas	RH850/F1L Group User’s manual		
[2]	Renesas	V850E3v5 Architecture Specifications		
[3]	Renesas	T01 - Self-Programming Library for Code Flash (RV40 Flash)	EALC-SD-1045-1E00b	

Table 1-2 References Documents

Contents

1 Introduction 6
1.1 Configurations covered by this Technical Reference 6
1.1.1 Microcontrollers 6
1.2 Compilers 6

2 Memory Model 7
2.1 [#hw_mem] - Design the Memory Layout 7
2.2 Memory Mapping 7
2.2.1 Memory mapping of application and FBL 7
2.2.2 Memory range of the FBL 7
2.2.3 Memory range of the application 8
2.2.4 Bootloader Sections 8
2.2.5 Compiler-/Linker specific memory mapping issues 9
2.3 [#hw_intvect] - The Interrupt Vector Tables 9
2.3.1 Bootloader Vector Table 10
2.3.2 Application Vector Table 10
2.3.3 Vector base address registers 11
2.3.4 Special measures for NMI 11

3 Hardware Layer 12
3.1 Flash Memory and Flash Driver 12
3.1.1 Special clock parameters and clock settings 12
3.1.2 Flash Code Buffer Size 12
3.1.3 ID Authentication 12
3.1.4 Rebuilding the flash driver binary 13
3.2 [#hw_size] - Flash Segment Size 14
3.3 CAN Driver 14
3.3.1 CAN Channels 14
3.3.1.1 Number of Supported Channels 15
3.3.1.2 Channel Settings for Pipelined Programming 16
3.3.2 Sleep Mode 16
3.4 Timer 16
3.5 Startup Code 17
3.6 Hardware Registers 17
3.7 Misc 18

4 Build 19
4.1 Code executed from RAM 19
4.1.1 Use of relocatable Code 19

4.1.2 Use of Code linked to RAM 19
4.1.3 Greenhills 20
4.2 Compiler Options 20
4.2.1 Inline Function Prologues 20
4.2.2 Optimizations 21
4.3 Linker Parameters 21

5 Glossary and Abbreviations 22
5.1 Abbreviations 22

6 Contact 23

Illustrations

Figure 1-1 Manuals and References for the Flash Bootloader 6
Figure 2-1 Memory Mapping 7
Figure 2-2 Flash Block Configuration using GENy 8
Figure 3-1 Clock parameter setting in GENy 12
Figure 3-2 Directory of flash driver project 13
Figure 3-3 CAN channel configuration in GENy 15
Figure 3-4 Maximum number of CAN channels 16
Figure 3-5 Component Hw_Rh850Cpu 18

Tables

Table 1-1 History of the Document 2
Table 1-2 References Documents 2
Table 2-1 Bootloader memory sections 9
Table 3-1 Timer Configuration Code 17

1 Introduction

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

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

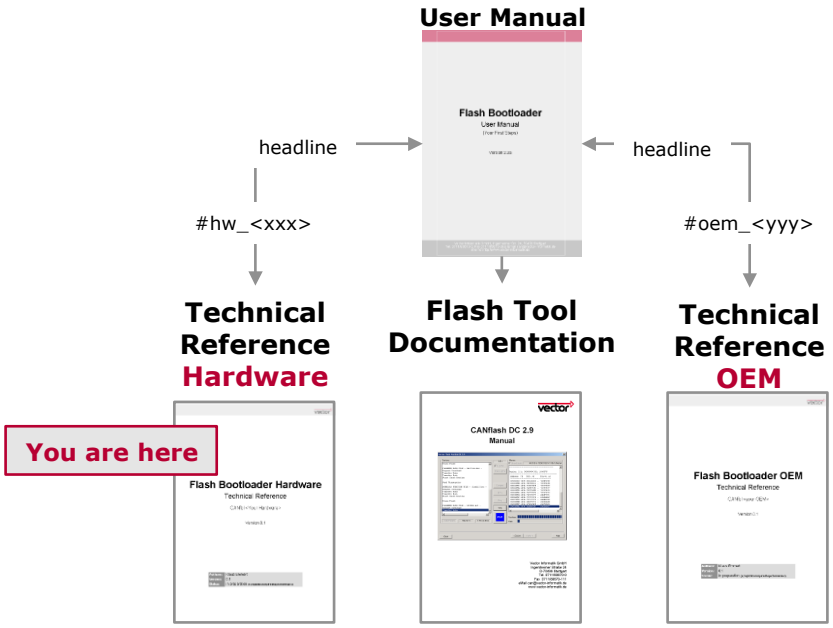

Figure 1-1 Manuals and References for the Flash Bootloader

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

1.1 Configurations covered by this Technical Reference

This Technical Reference covers the following controllers and compilers:

1.1.1 Microcontrollers

- Renesas RH850 F1L

1.2 Compilers

- GreenHills

2 Memory Model

2.1 [#hw_mem] - Design the Memory Layout

For more general information about this see the UserManual_FlashBootloader in the chapter **Design Memory Layout**.

2.2 Memory Mapping

2.2.1 Memory mapping of application and FBL

The FBL will reside in the memory space starting from the address 0, because the interrupt vector table is also located at this address and it must be part of the FBL. This is required so that the bootloader is started after PowerOn or reset.

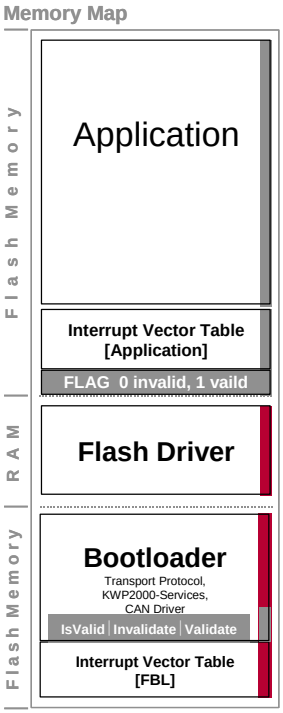

Figure 2-1 Memory Mapping

2.2.2 Memory range of the FBL

Depending on the Bootloader configuration and used compiler optimization, the Bootloader occupies one or more flash blocks. These flash blocks are used by the Bootloader exclusively and have to be configured to “Protected” in the flash block table to protect the Bootloader from being erased or overwritten.

The size of the FBL depends on the used car manufacturer specification and the configuration. Depending on these requirements, the FBL can exceed the size of 32Kbytes.

2.2.3 Memory range of the application

The application can start directly after the Bootloader and may occupy all memory regions which are not reserved for the Bootloader. The Flash sectors which can be used by the application are configured in the “FlashBlockTable” configuration dialog in the generation tool GENy, setting the device type to “Flash”.

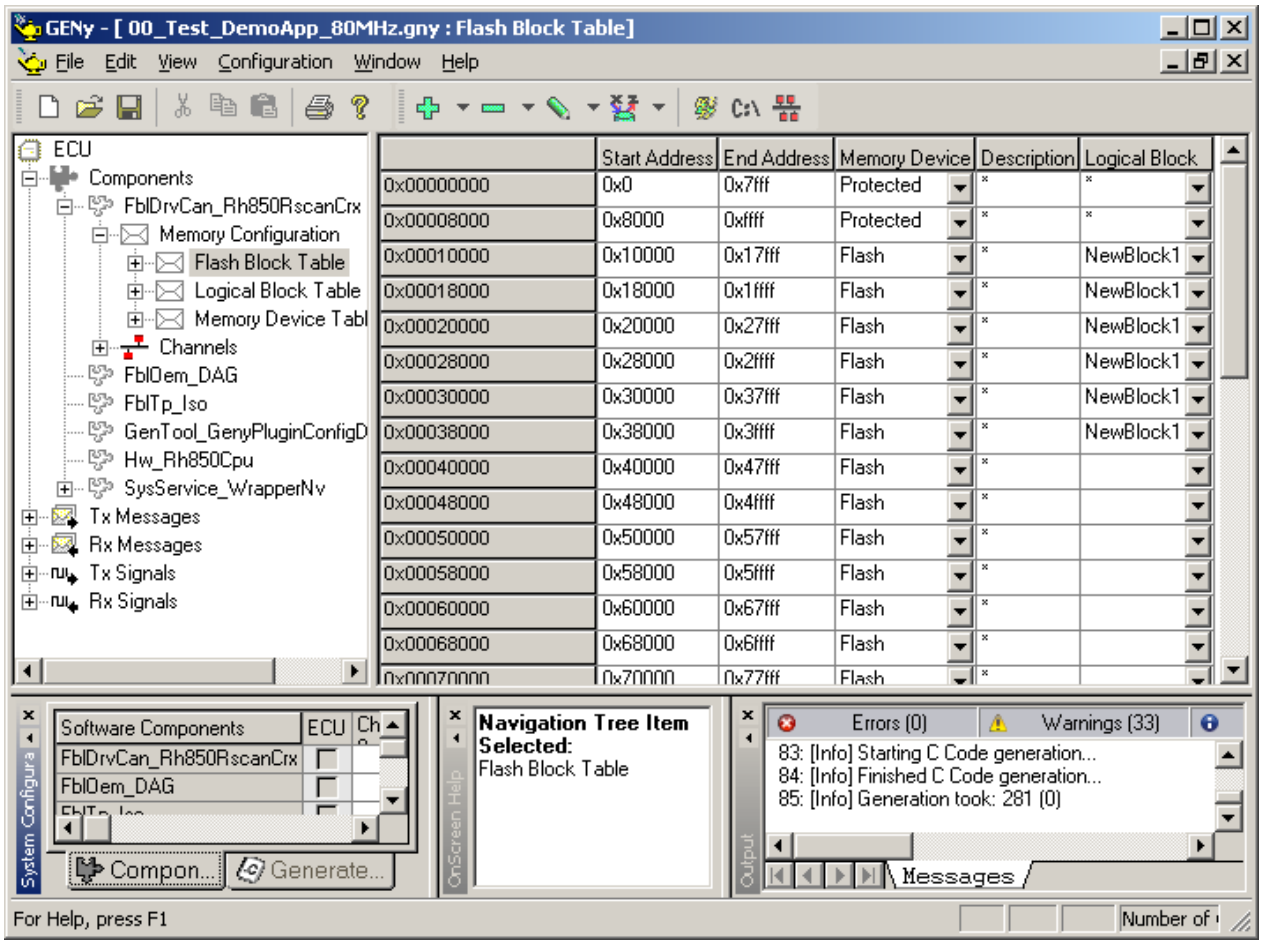

Figure 2-2 Flash Block Configuration using GENy

2.2.4 Bootloader Sections

There are a few RAM/ROM sections that must be defined to a fixed location. They are used to locate the data within a determined memory area. It may also be necessary to specify shared memory locations that are accessed within the application and the Bootloader (e.g. interrupt vector table of the application or the FBL header).

The following sections must be defined in the linker command file:

Section Name	Address
.intvect	The interrupt vector table of the bootloader at address 0
.FBLHEADER	This section defines the location of the Bootloader header structure “FblHeader”. This structure includes information like the Bootloader version, the start address of the Bootloader, etc. This address must be compliant to the address configured in the generation tool in the field “Header Address” in the FBL-options.
.APPLVECT	Start address of application interrupt vector table validation flag. The application interrupt vector table flag is used to determine if an application was downloaded and to get the start address of the application. Normally, the section APPLVECT should be placed right before the application interrupt vector table.
FLASHDRV	Allocates the memory for the flash driver. The size of the array is defined in the generation tool as the “Flash code buffer size (bytes)” parameter. The array flashCode[] allocates the memory. Please note that there can be additional flashCode[]-arrays if there is more than one flash driver variant present (e.g. if the OEM specification requires a downloadable driver and the Bootloader uses a compiled in driver for the EepM module.

Table 2-1 Bootloader memory sections

Note
Please note that depending on the used car manufacturer, some other sections must be also included.

Example
Linker command file examples can be found in document chapter Linker Parameters.

2.2.5 Compiler-/Linker specific memory mapping issues

2.3 [#hw_intvect] - The Interrupt Vector Tables

For more general information about this see the UserManual_FlashBootloader in the chapter The Interrupt Vector Tables and Fbl_vect.c / Applvect.c(.h) - The Interrupt Vector Tables.

2.3.1 Bootloader Vector Table

The interrupt vector table of the bootloader covers the exception vectors at the offsets from 0x00 – 0xF0. These are the vectors of RESET, SYSERR, HVTRAP, up to FEINT. All these vectors reference the startup code of the bootloader to make sure that these events are handled even if no valid application is available. If a special handling of one of these vectors is needed, please adapt the bootloader vector table fbl_vect.c to implement a specific handler call.

Besides these exceptions, no other vectors are implemented in the bootloader vector table. Since exception/interrupt control registers are available that permit the remapping of the vector table address, it is no longer necessary to use the redirection mechanism from bootloader vector table to application vector table with jump opcodes.

It is therefore mandatory that download applications initialize the vector base registers on startup and before interrupts are enabled.

2.3.2 Application Vector Table

The application vector table consists of the reset vector entry only. This entry is used by the bootloader to start the application if it is valid after Reset or PowerOn. It also provides a flag that tells the Bootloader that the application interrupt vector table is valid/present. This flag must be set to APPLVECT_FROM_APPL in the application.

Note
The fbl_applvect.c file doesn't contain the application vector table itself.

Caution
This table must be located at a fixed address to provide the Bootloader with the memory address of the application interrupt vector table validation flag. Please make sure to locate the application interrupt vector table validation flag at the same address in the Bootloader and in the application itself.

Example
An example implementation is provided with every Bootloader delivery:

- The Bootloader demonstration project includes a placeholder called fbl_applvect.c
- The demonstration application includes the file applvect.c. This file includes the address of the application's startup code which is used to start the application software by the Bootloader.

2.3.3 Vector base address registers

The RH850 has got three Exception/Interrupt control registers. RBASE is the Reset Base Register and contains the reset vector address. The exception vector base address is controlled by the EBASE register. The INTBP register provides the base address of the interrupt handler address table. For detailed information for these registers please see [3].

These registers permit to store the reset vector, the exception vectors and the interrupt vectors at three different locations. Since the bootloader only provides the reset- and the exception vectors, it is important to initialize RBASE and EBASE in the bootloader startup code.

2.3.4 Special measures for NMI

The NMI (FENMI) is handled by the Bootloader's startup code. Normally, the NMI handlers invoke the startup code of the Bootloader to restart the Bootloader after a NMI occurred. If NMI handling is required in your Bootloader, you have to implement the corresponding handlers.

Caution
While restarting the Bootloader can be an appropriate choice, the application software normally has to handle NMIs. Please ensure your application software includes the needed handlers and activates its own table by setting the corresponding base register.

3 Hardware Layer

The relevant hardware components of the Flash-Bootloader are the CAN-controller, the hardware-timer and flash memory. This chapter describes the relevant properties and configuration settings for these components.

3.1 Flash Memory and Flash Driver

The flash driver uses the Renesas Self Programming Library (FCL) for flash programming. The flash driver file flashdrv.c performs an interface adaption between the bootloader (HIS – interface) and the FCL.

3.1.1 Special clock parameters and clock settings

The FCL requires the CPU clock (CPUCLK) frequency as a parameter. The bootloader calls the flash driver with the value that is configured in the GENy FblDrvCan_-component Internal System Clock as a parameter. Please make sure that you enter the correct value in GENy as shown below.

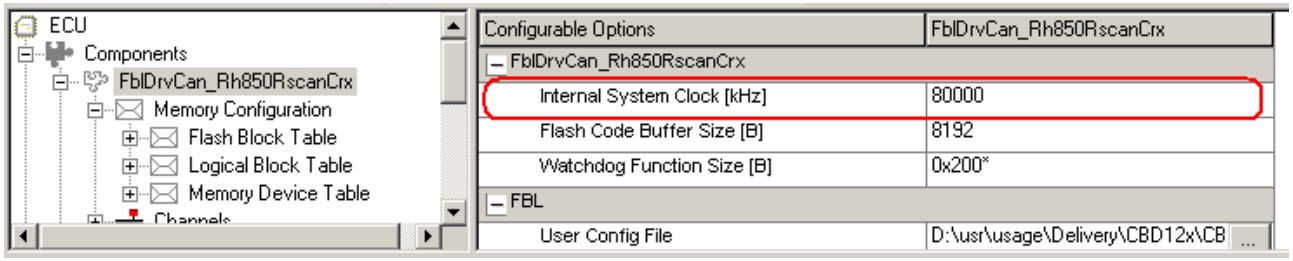

Figure 3-1 Clock parameter setting in GENy

3.1.2 Flash Code Buffer Size

The bootloader reserves a configurable amount of RAM to store the flash driver. Per default, 8KBytes are reserved for this buffer. The flash driver size is about 6KBytes. Depending on the used compiler, flash technology and the compiler options the size of the flash driver may be different. Please refer to the delivered flashdrv.hex file for the actual size of the flash driver. It is necessary to keep in mind that future updates of the flash driver can be of greater size so that it is recommended to have a sufficient buffer size in reserve.

3.1.3 ID Authentication

The RH850 requires an ID authentication when the flash library is initialized. The ID consists of four 32 bit unsigned integer values. The FBL uses the defines FLASH_AUTH_ID_0, FLASH_AUTH_ID_1, FLASH_AUTH_ID_2 and FLASH_AUTH_ID_3 to specify these four ID values. Per default, all ID- values are set to FFFFFFFFh. It is possible to redefine these ID codes to specific values by defining the codes in a user-configuration file for GENy:

Example
User configuration file fragment with authentication ID defines:

```
/*----------------------------------------------------------------------------*/
# define FLASH_AUTH_ID_0 0xFFFFFFFFul
# define FLASH_AUTH_ID_1 0xFFFFFFFFul
# define FLASH_AUTH_ID_2 0xFFFFFFFFul
# define FLASH_AUTH_ID_3 0xFFFFFFFFul
```

3.1.4 Rebuilding the flash driver binary

The flash driver is delivered with a build project based on the Vector MakeSupport. It can be found in the delivery in the Flash folder that is located in BSW.

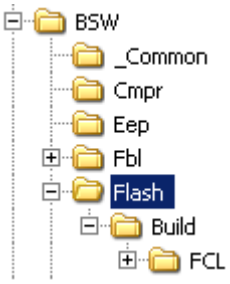

Figure 3-2 Directory of flash driver project

The subdirectory “Build” contains a build system to rebuild the flash driver. You have to adapt your compilers installation path in the file “Makefile”. Then the flash driver can be rebuilt by executing m.bat.

Example
Compiler path setting in Makefile:

```
#------------------------------------------------------------------------------
#------------------------ MUST be filled out -----------------------------------
# Define Compiler path
# E.g.: COMPILER_BASE = C:\Uti\HC08\HIWARE
#       COMPILER_BIN  = $(COMPILER_BASE)\prog
#       COMPILER_INC  = $(COMPILER_BASE)\lib\hc08c\include
#       COMPILER_LIB  = $(COMPILER_BASE)\lib\hc08c\lib
#------------------------------------------------------------------------------
COMPILER_BASE = D:\uti\PPC\WindRiver\5.9.0\diab\5.9.0.0
```

When rebuilding the flash driver it is important to consider compiler options, linker mapping and configuration settings of the FCL. Bootloader and flash driver have been tested with the settings of the delivery. Other settings have not been tested and should not be used.

The FCL permits several configurations. For further details see the FCL configuration header file fcl_cfg.h and [3].

It is mandatory that the execution mode of the FCL is configured to use the status check by user. This is done in file fcl_cfg.h by setting the define switch R_FCL_COMMAND_EXECUTION_MODE to R_FCL_HANDLER_CALL_USER. This permits watchdog servicing and handling of other dynamic tasks by the bootloader flash driver during flash operations. This setting is already configured in the bootloader delivery and must not be changed.

Some compilers support a small data area option (SDA). If this compiler option is used please make sure that the corresponding base pointers have got the same value in the bootloader and the flash driver.

The flash driver is not relocatable. The RAM buffer array for the flash driver in the bootloader needs to be located at the exactly same address as the address of the flash driver itself.

After the build process, the hex file of the flash driver is probably segmented and consists of more than on memory segment. Most bootloader implementations do not support the download of a segmented flash driver. Therefore, it is necessary to fill the gaps between the segments with fill code in order to obtain a single-region file. This can be done by a batch process using HexView.

3.2 [#hw_size] - Flash Segment Size

For more general information about this see the UserManual_FlashBootloader in the chapter **Flash Segment Size**

The flash segment size describes the minimum writable data size. The flash segment size of the RH850 F1L is 256 bytes. Depending on the OEM reprogramming requirements, it is necessary to align the download addresses to integer multiple of this size. This can be accomplished by an additional processing step with HexView after building the application.

3.3 CAN Driver

3.3.1 CAN Channels

The channel which is used for communication can be chosen in GENy. Depending on the derivative, different possibilities are provided.

The channel configuration dialog is the same for the bootloader CAN driver and the standard Vector CAN driver. Since the standard Vector CAN driver provides a greater flexibility than the bootloader CAN driver, there are several options in this dialog that are not relevant for FBL configuration.

The relevant settings for the FBL are:

- “BCFG” and “Bustiming Configuration” for setting up the baud rate
- “Physical Controller” for setting the CAN channel

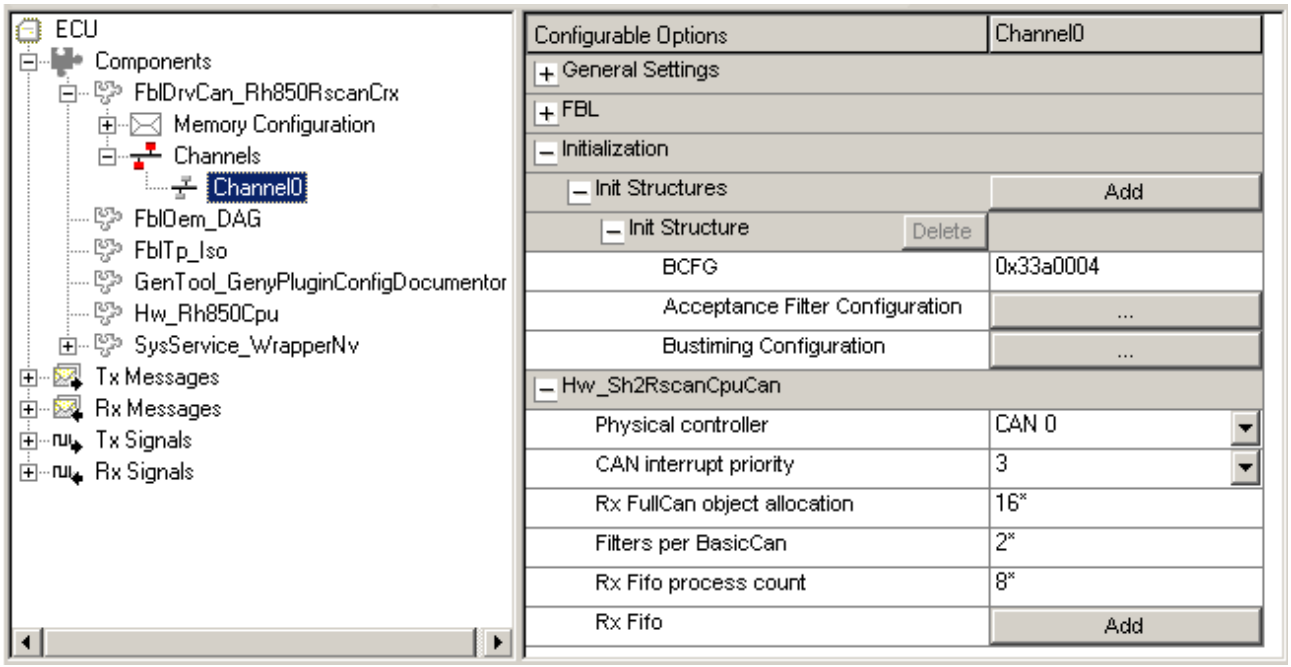

Figure 3-3 CAN channel configuration in GENy

The Acceptance Filter Configuration can be ignored because the FBL uses a fixed set of CAN message objects for download communication. CAN interrupt priority, FullCAN object allocation, BasicCan and RX fifo settings are also not relevant for the FBL.

3.3.1.1 Number of Supported Channels

The RSCAN cell supports a different number of physical CAN channels. In the field “Maximum number of CAN channels” of the component FblDrvCan_Rh850RscanCrx, the actual number of channels of the current derivative needs to be entered. When a physical CAN channel is chosen in the “Physical Controller” field, GENy checks if this selection complies with the configured maximum number of CAN channels in order to avoid incorrect configurations. In case of a mismatch, GENy generates an error message.

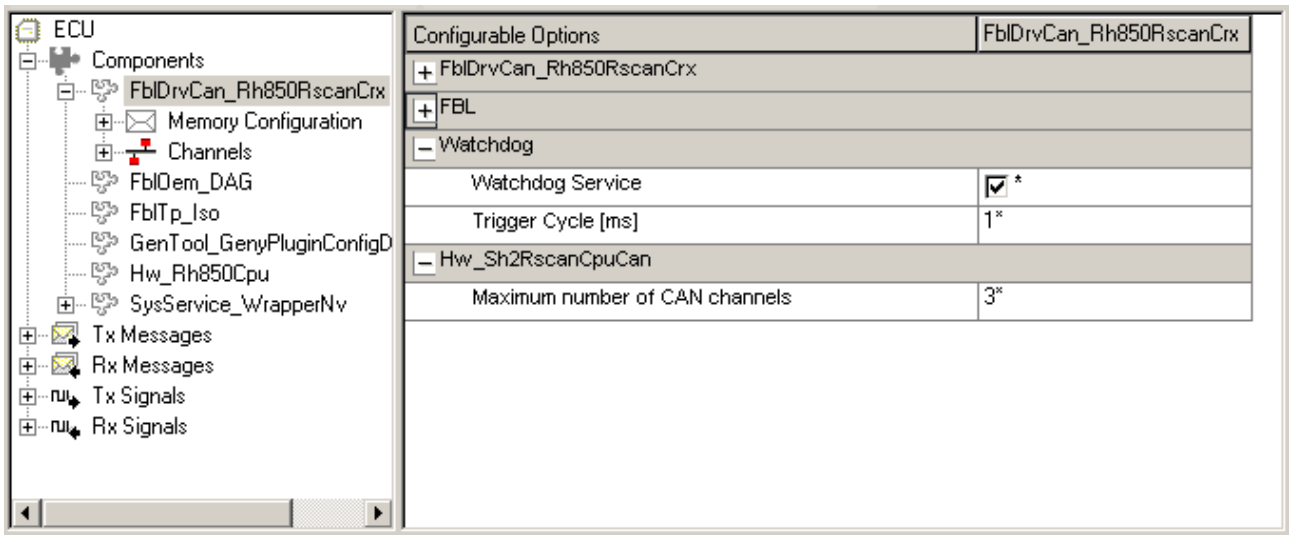

Figure 3-4 Maximum number of CAN channels

3.3.1.2 Channel Settings for Pipelined Programming

For pipelined programming, more than one (logical) channel can be configured, but only one physical channel is used. If more than one logical channel is mapped to the same physical channel in the “Physical controller” field, GENY generates an error message. Therefore, please set the Physical controller to a different value for the other logical channels. GENy generates the bootloader configuration for the physical channel according to the setting of logical channel 0.

3.3.2 Sleep Mode

The implementation of FblCanSleep() enters the reset mode of the bootloader CAN channel and deactivates the CAN clock.

Since the CAN cell does not provide a means to detect the sleep state, function FblCanIsAsleep() always returns 0 and cannot be used to detect a wakeup condition.

3.4 Timer

The hardware independent modules of the Bootloader require a 1ms time base to derive the necessary call cycles of cyclic functions from that time base.

The RH850 uses the Timer Array Unit D (TAUD). One channel is used to generate the 1ms time base. An additional channel is used as a free-running timer that can be used as a base for generating the security access seed value.

Like the other hardware specific configurations, the timer module is also configured with the configuration tool GENy. TAUD is supplied with the PCLK. Please specify the PCLK frequency of your system as the timer clock frequency.

Symbolic Constant	Description
FBL_TIMER_RELOAD_VALUE	Timer reload value for 1ms base cycle.

Table 3-1 Timer Configuration Code

3.5 Startup Code

There is a sample startup code provided with the Bootloader demonstration project. This startup code performs several configuration and preparation tasks before the Bootloader's main function can be started.

Caution
The startup code is provided as a part of the Bootloader demonstration project. Please use your own startup code or review the startup code carefully to ensure the startup code is correct and fits into your usecase.

The following tasks have to be handled by the startup code:

- Initialization of the global pointer (GP).
- Initialization of the text pointer (TP)
- Initialization of the stack pointer (SP)
- RAM initialisation.
- Copy routines linked to RAM from flash to RAM.

Code is copied into RAM using the __start() library function

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

Don't forget to change the TLB structure according to your own MMU configuration.

Please see the demonstration startup code for an example.

3.6 Hardware Registers

Access to hardware registers is provided by macros and type definitions in the fbl_sfr.h header file. There is only support of those hardware registers that are absolutely necessary for the Bootloader. These are the CAN controller registers, the ports for transceiver control, timer, watchdog and I/O port registers.

3.7 Misc

In GENy, the component Hw_Rh850Cpu is visible. Settings in this component are not relevant for the Flash-Bootloader. It is not necessary to make any changes here.

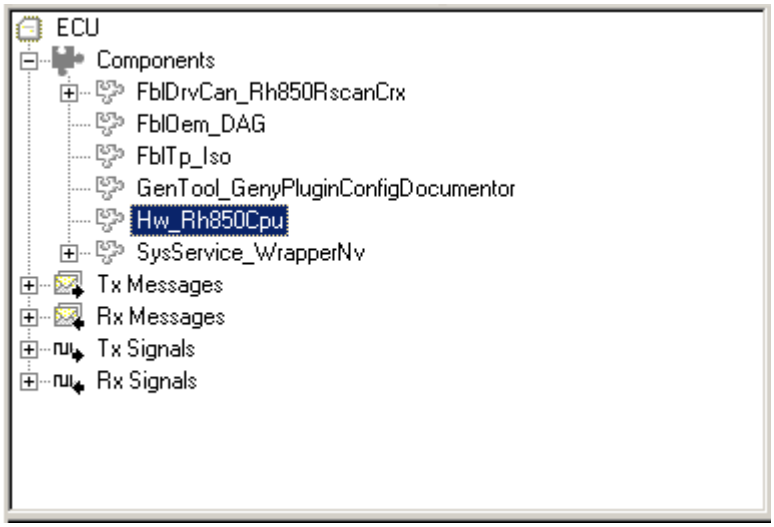

Figure 3-5 Component Hw_Rh850Cpu

4 Build

4.1 Code executed from RAM

The Bootloader has to execute some code from RAM during flash memory is erased or written. Depending on the amount of functionality which is needed during flashing, two approaches to execute code from RAM are used. Please note that one of these approaches is selected during the Bootloader package creation and that there is no option to configure this behavior.

4.1.1 Use of relocatable Code

If the Bootloader just needs to maintain watchdog handling functionality, the watchdog handling code can be linked to flash memory and be copied from flash memory to RAM during the Bootloader's startup. The two functions FblLookForWatchdog() and ApplFblWdTrigger() are copied to RAM in this configuration. No special precautions while rebuilding the Bootloader have to be taken.

This variant can is used if the function FblCopyWatchdog() is present in the fbl_wd module.

Caution

No function calls from ApplFblWdTrigger() are allowed if relocatable code is used for watchdog handling because function calls are normally not relocatable.

The functions copied to RAM are not debuggable because there are no symbols generated to the target addresses.

4.1.2 Use of Code linked to RAM

Some use cases require code to be linked to RAM because function calls have to be executed or more functionality has to be maintained during flashing, e.g. if pipelined programming is used or a complete communication stack has to be used to send response pending messages.

To locate this functionality in RAM, some special compiler features are used. Basically, all compilers use the same method:

- Functions which have to be linked to RAM are placed in special sections or complete modules are marked to be linked to RAM.
- The compiler links these modules to RAM address space.

- An image of the code and data linked to RAM is created. This image is linked into flash memory.
- Some address information to find the image in flash and target addresses in RAM is created.

Code linked to RAM using this method can be debugged normally because the debugging symbols are located at the execution addresses of the code.

Note
The addresses which can be supplied to normal jump instructions are not big enough to perform a jump from flash memory to RAM and back. Bootloader which have to execute such jumps either include a memmap.h file which inserts far jumps or have to be compiled with far jump compiler options.

The implementations of code in RAM differ for every compiler.

Example
Please see the Bootloader demonstration project's linker control file and startup code for examples.

4.1.3 Greenhills

- ROM images of RAM code are created using the linker's ROM() command.
- The linker automatically generates a table which is used to copy all images to all target addresses.
- The code is copied by the startup code supplied with the Bootloader demo using library routine.

4.2 Compiler Options

4.2.1 Inline Function Prologues

Greenhills compilers use independent functions to perform function prologue tasks. These function prologues cause errors while calling watchdog handling functions out of the flash driver. To avoid these problems, the function prologues have to be inlined on Greenhills compilers:

Example
Inline function prologues with Greenhills compilers:

```
-inline_prologue
```

4.2.2 Optimizations

The Greenhills compiler options –Ospace and –no_commons have been tested for the bootloader and should be applicable without problems.

4.3 Linker Parameters

There is a linker control file supplied with every Bootloader demonstration project. Please see this file for linker parameter and linker handling examples.

5 Glossary and Abbreviations

5.1 Abbreviations

Abbreviation	Description
CAN	Controller Area Network
DMA	Direct Memory Access
ECU	Electronic Control Unit
EepM	EEPROM Manager
EepMgr	EEPROM Manager
FBL	Flash Bootloader
NMI	Non-Maskable Interrupt
TAU	Timer Array Unit
SFR	Special Function Register

6 Contact

Visit our website for more information on

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

www.vector.com