Documentation

HexFile Handling in Gen 9.3 projects

Project name: HexFile Handling in Gen 9.3 projects

Author: CC/ESM2 Heiko Eckert

Relevant for: **MTC 10.x, HawCC 2.x**

Subordinate tools (XFlash, SharCC_PMSe, DLM-Composer)
HSW component of ECU projects

Table of Contents:

1. Revision History

Document version	Date	Name	Comment
V0.1	13.12.12	CC/ESM2-Eckert	First draft version. Converted to Evo17 requirements
V0.2	18.02.13	CC/ESM2-Eckert	Added new hexfile structure requirements.
V0.3	04.03.13	CC/ESM2-Eckert	Added new hexfile structure requirements.
V0.4	15.03.13	CC/ESM2-Eckert	Added new hexfile structure requirements.
V0.6	21.06.13	CC/ESM2-Eckert	Added chapter about gaps in RawHexFile (chapter 3.5), reworked checksum handling (chapter 6.1, chapter 6.2)
V1.0	25.07.13	CC/ESM2-Eckert	Changed chapter6 (checksum handling) Changed chapter7 (DDF) Added explicite rules for structuring hexfiles.
V1.1	09.12.13	CC/ESM2-Eckert	Added chapter 3.6 about handling flash option bytes on Renesas devices.
V1.2	09.05.14	CC/ESM2-Eckert	Added chapter 5.1.2 (Complete hexinfo structure)
V1.3	18.07.14	CC/ESM2-Eckert	Adapted chapter 4.1
V1.4	08.09.14	CC/ESM2-Eckert	Adapted chapter 7, overworked complete document
V1.5	12.12.14	CC/ESM2-Eckert	Adapted chapter 5.1.x: changed coding format for Commit-ID, TimeStamp and CM-ID
V1.6	25.02.15	CC/ESM2-Eckert	Changed usage of dirty flags in chapter 5
V1.6.1	16.03.15	CC/ESM2-Eckert	Removed deprecated content of chapter 8
V1.7	04.05.15	CC/ESM2-Eckert	Adapted chapter 6 (CHECK block coding) to multi consistency handling.
V1.7.1	14.08.15	CC/ESM2-Eckert	Some minor changes to chapter 6
V1.7.2	15.10.15	CC/ESM2-Eckert	Adapted init values of hexinfo and CHECK structures
V1.8	19.06.17	CC/ESM2-Eckert	Added subblocktypes to HexBlockStructure, added CreateDummyBlock flag
V1.9	27.10.17	CC/ECC6-Wilhelm	Adapted chapter 10: HSW

V1.10	12.12.17	CC/ESM2-Eckert	Added chapter 9: Tools
V1.11	07.02.18	CC/ESM2-Eckert	Adapted chapter 4.3: HSM
V1.12	14.09.18	CC/ESM2-Eckert	Adapted chapter 5.2: added Emu devices
V1.12.1	27.09.18	CC/ESM2-Eckert	Added comment to chapter 4.3
V1.13	28.02.19	CC/ESM2-Eckert	Adapted chapter 5.2: added D6/D7 memory map
V1.14	19.03.19	CC/ECC6-Wilhelm	Added chapter 7 (Signature handling)
V1.14.1	04.07.19	CC/ESM3-Eckert	Additional extended BlockType: FOTA

2. Introduction

2.1. Scope / Purpose

The purpose of this specification is the definition of hexfile handling for Evo17 devices.

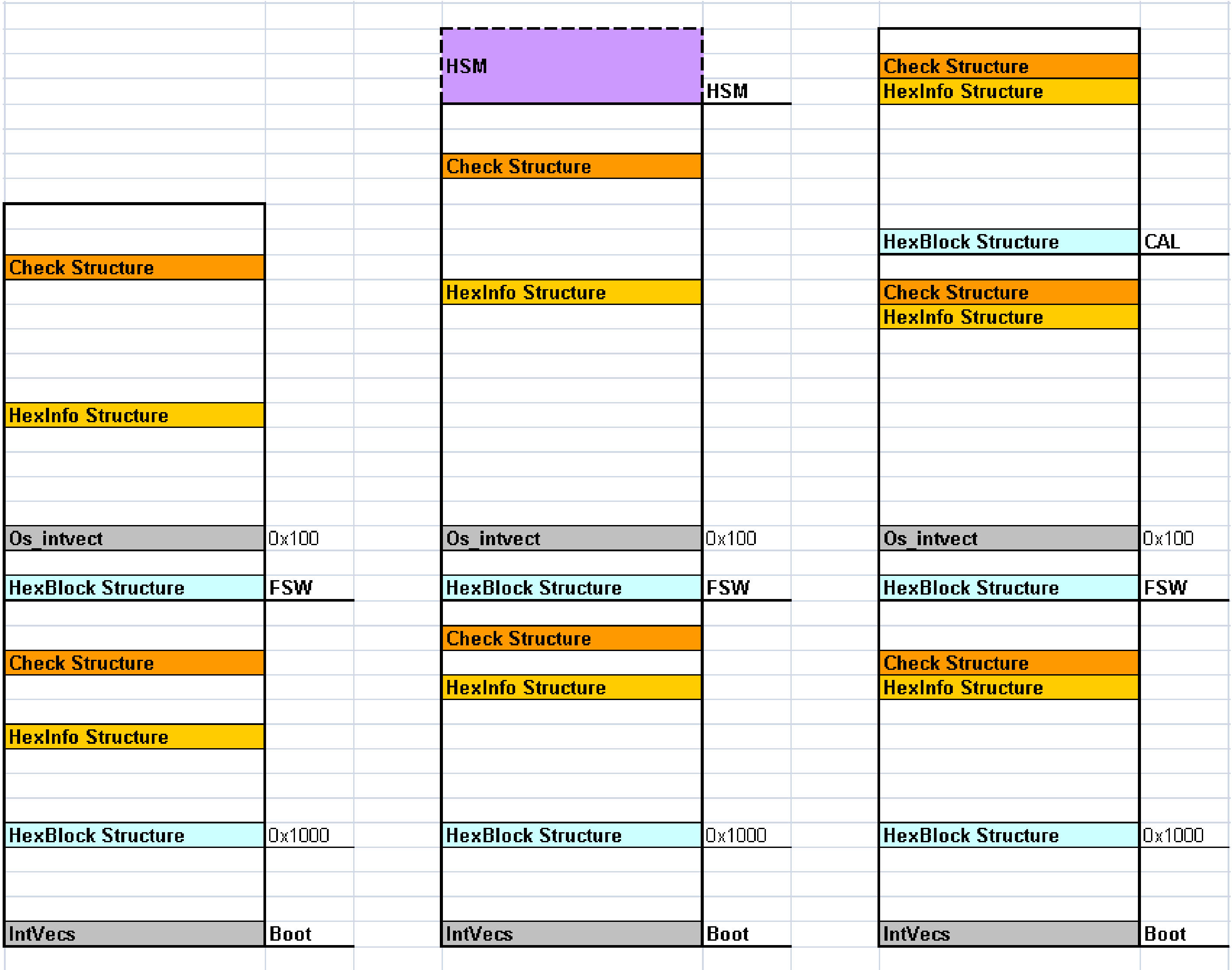
2.2. Terms / Abbreviations

Term / Abbreviation	Explanation
MTC	Make Tool Chain
Hexinfo	Structure containing project specific data like BBNumber, CommitID,…
Hexfile block	Defined part of a hexfile A hexfile block contains 0-n complete memory segments of current device
HawCC	Hexfile Alteration Wizard for CC. Tool to visualize hexfile data, fill in checksum and patch hexfile information
Multi-block-concept	Handling of more than one hexfile blocks within current hexfile
PMS/E	Parameter modifying system (CC application tool)
DDF	Device description file
LD file	Linker directive file (former LCF)
LCF	Linker command file (used in MTC9.x, now deprecated)
EOL	end-of the line (reprogramming feature)
CALPART	A calblock that needs not to start and end on HW section border, but is not (stand-alone) flashable
TPSW	Third party SW
HSM (ICU-M)	Hardware Security Module (Security core on cc-cube devices D3-D5)
PE	Processor element

3. Hexfile Layout

3.1. Base concept

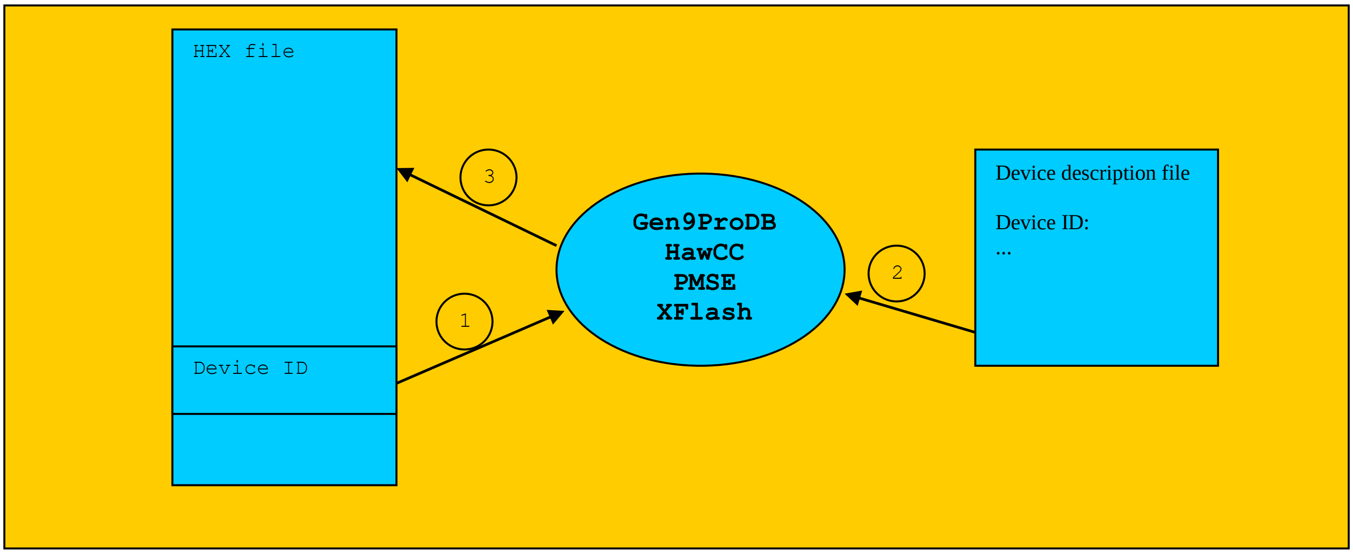
The current MTC 10.x versions support the handling of one RAW hexfile (generated by linker using LD file). This hexfile can be split to several block hexfiles by Gen9ProDB tool within MTC.



The Gen 9.3 devices have different memory layouts. To handle these different layouts when manipulating or interpreting hexfiles it is necessary to have a device description for each member. The complete device description is too big to program it directly into the hexfile thus it is just the device identifier which is programmed into the hexfile.

3.2. DeviceID

As the program, which interprets the hexfile does not know in advance which device the hexfile is created for, it is necessary to write this device ID at an address which can be determined by CC tools.



Any tool is now able to read the device identifier from the input hexfile and refer to the device description file, which provides all necessary information about the device.

Step1: The tool reads the device identifier from input hexfile.

Step2: The device description file provides the device specific data for the tool.

Step3: The tool is now able to manipulate/interpret the given hexfile.

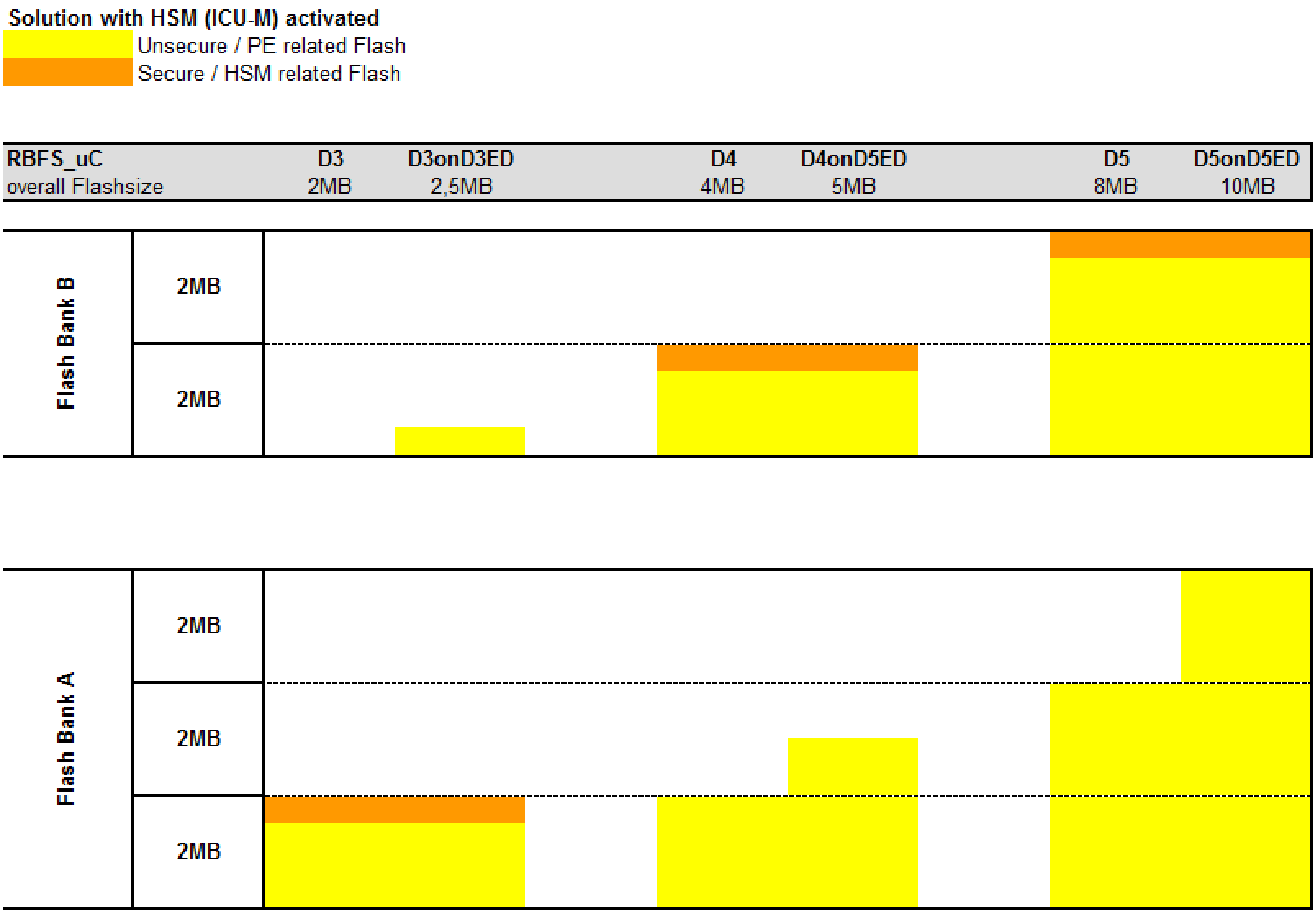
3.3. HSM area (ICU-M)

Exclusive Pflash area for HSM core, read/write protected for FSW (D3-D5)

Renesas: HSM area is always part of Pflash. Located at the end of Pflash (end address Pflash = end address HSM exclusive code flash).

Last 48 bytes before HSM area must be always protected from write as well.

The REL instruction fetch unit (IFU) preloads at most 48 bytes from the current program counter (PC) in advance if the code is executed from code flash. If it is executed from other memories (e.g. RAM), at most 32 bytes are read in advance. The preloads are performed by the instruction fetch pipeline, irrespective if the memory is available or not. In order to prevent exceptions due to access violations reported by guards, there must not be any instructions located in the last 48 (code flash) or 32 (other memories) bytes of a protected address range. For the code flash, the last 48 bytes of both the unsecure and secure (ICU-M) flash region must not be filled with instructions. This condition must be fulfilled for both banks of the code flash. Therefore the reserved area is introduced here.



3.4. HexBlock structure area

The hexblock structure will be located once per block. It contains DeviceID, BlockType, HexFileFlags, and HexID of all Blocks and all necessary pointers (NextBlockRef, HexInfoRef, CheckRef, SignatureRef, DMTRef, BlockInterfaceRef).

The HexBlock structure is located at the respective start address of the current block (except for the 1st hexblock).

In the 1st hexblock the HexBlock structure is located at the fix address 0x00001000 (because of the interrupt vectors, which reside at address 0x00000000 here).

Rule: A block which does not have some valid data at its blockstart address is invalid and will cause an error in MTC hexfile writer.

In hexfiles without bootblock the HexBlock structure in the first block is located at 0x00001000 as well.

HexBlockStructure Overall Size: 0x20 Byte						
Rel. Address	**Field name**	**Byte Size**	**Datatype**	**Example**	**Tool**	**Comment**
0x00	**Block structure ID**	1	**Byte**	0x01	HSW/PSW	Defines the internal structure of HexBlockStructure
0x01	**SupplierID**	3	**ASCII**	“STM”	HSW/PSW	STM | REL
0x04	**DeviceID**	4	**UInt32**	0x86E77777	HSW/PSW	Must correspond to content of DDF.ini
0x08	**BlockSize**	4	**UInt32**	0x00008000	HSW/PSW	Length of current block in bytes
0x0C	**BlockType**	1	**Byte**	0x01	HSW/PSW	0x01: BOOT 0x02: CODE 0x03: CAL 0x04: CALPART 0x05: HSM
0x0D	**BlockType (extended)**	1	**Byte**	0x01	HSW/PSW	0x00: NoEXT 0x01: BMGR 0x02: RBBLDR 0x03: RBBLDRNo 0x04: OEMBLDR 0x11: RBDiag 0x12: RBDiagNo 0x20: FOTA
0x0E	**CreateDummyBlock**	1	**Byte**	0x01	HSW/PSW	0x00: do not create dummy block 0x01: create dummy block
0x0F	**Reserved**	1	**-**	-	HSW/PSW	Must be 0x00
0x10	**CheckRef**	4	**UInt32**	0x00006300	HSW/PSW	Pointer to CHECK structure
0x14	**HexinfoRef**	4	**UInt32**	0x00006400	HSW/PSW	Pointer to hexinfo structure
0x18	**SignatureRef**	4	**UInt32**	0x00006500	HSW/PSW	Pointer to signature structure
0x1C	**BlockInterfaceRef**	4	**UInt32**	0x00006700	HSW/PSW	Pointer to blockinterface structure

Invalid pointers will have to be set to 0xFFFFFFFF by PSW/HSW.

3.4.1. Structure ID

The structure ID can be used to document changes in HexBlockStructure format. The value will be increased on any change made to HexBlockStructure.

Additional it will be used (together with SupplierID) to determine if a given file is a valid Gen 9.3 hexfile.

The only valid ID for Gen 9.3 is 0x01;

3.4.2. Block Size

The blocksize is a 32-bit value which determines the size of the current block. The size is coded in hexadecimal directly to hexfile. No need to multiply with a fix value (like in gen9).

3.4.3. BlockType

Valid blocktypes in Gen 9.3 hexfile handling are:

BOOT	0x01
CODE	0x02
CAL	0x03
CALPART	0x04
HSM	0x05

3.4.4. Extended BlockType

The purpose of a subblocktype is to distinguish between hex blocks of the same blocktype. In this regard, blocktype “BOOT” can be of subblocktype 0x00 to 0x04, while “CODE” can be of subblocktype 0x00, 0x11 and 0x12. All other blocktypes are always of subblocktype 0x00, which stands for “NoEXT” (No extension type used).

Valid subblocktypes in Gen9.3 hexfile handling are:

0x00: NoEXT	(no extension type used)
0x01: BMGR	(Identifies a code block being a Bootmanager)
0x02: RBBLDR	(Identifies a code block being a Robert Bosch Bootloader)
0x03: RBBLDRNo	(Identifies a code block being a Robert Bosch Bootloader with dummy code, containing no functionality)
0x04: OEMBLDR	(Identifies a code block being a Customer Bootloader)
0x11: RBDiag	(Identifies a code block being a Robert Bosch Diagnosis)
0x12: RBDiagNo	(Identifies a code block being a Robert Bosch Diagnosis with dummy code, containing no functionality)

3.4.5. CreateDummyBlock

If this flag is set to 0x01 am dummy block will be created for all blocks with below subblocktypes:

```
0x02: RBBLDR -> 0x03: RBBLDRNo
0x11: RBDiag -> 0x12: RBDiagNo
```

The subblocktype in dummy block has to be changed to the respective dummy subblocktype while creating dummy block.

The dummy blocks “RBBLDRNo” and “RBDiagNo” will be available as separate hexfiles. The reason for creating dummy blocks is to use them as substitutes for block “RBBLDR” and “RBDiag in a later manufacturing step.

3.4.6. Reference Pointers

<u>CheckRef:</u>

32-bit pointer to CHECK area, needed for checksum calculation and consistency checks

<u>HexinfoRef:</u>

32-bit pointer to Hexinfo area.

<u>SignatureRef:</u>

32-bit pointer to signature structure.

<u>BlockInterfaceRef:</u>

32-bit pointer to the location, where block interface structure is located.

3.5. Handling Gaps

3.5.1. Gaps within flash area (software gaps)

All gaps which reside in the Flash area of the RawHex must be filled by MTC. MTC generated hexfiles must not have any gaps.

The standard fill pattern will be patched by Gen9ProDB on every run of the MTC. For Gen 9.3 devices we will use the RIE Opcode 0x0040 (Reserved Instruction Exception).

3.5.2. Gaps outside flash area (hardware gaps)

Gaps outside flash area (so called “hardware gaps”) are not covered by any HW segment in DDF, nor are they covered by any hexblock defined in LCF.

Rule: A hardware gap within a hexblock (boot, code, cal) is not allowed and will cause an error in MTC HexFileWriter.

Hardware gaps must not be filled by MTC HexFileWriter because they do not contain any flashable content.

Bank B:

Each flash bank requires at least one separate hexblock

Bank A:

3.6. Security concepts

3.6.1. Flash Option Bytes

On Renesas devices a dedicated flash memory address space is provided to hold various configuration settings, called flash option bytes. Via these options, the start-up configurations for the controller can be set.

(The settings itself are device dependant and may not be available in all devices). The flash option bytes can be written by use of an external flash programmer and in self-programming mode.

On device start-up (during reset sequence) the option bytes will be copied from this dedicated flash area into internal peripheral module registers. After copying these register values will be used as initial start-up configuration for related modules like Clock Control, ICU-M, Emulation Device settings, …

Within CC-AS the Renesas flash option bytes will be allocated directly within the hexfile during normal build process.

MTC will read these security bytes and check if they are covered by the sectioning data read from DDF.ini. MTC will write out Flash option bytes without any change to target hexfile. Empty flash option byte areas in input hexfile will not cause an error and will NOT be filled by MTC.

The following output target hexfiles should contain flash option bytes:

- complete
- CODE block
- BOOT block

3.7. Multi-block hexfile handling

See document: Multi block hexfile handling.pdf

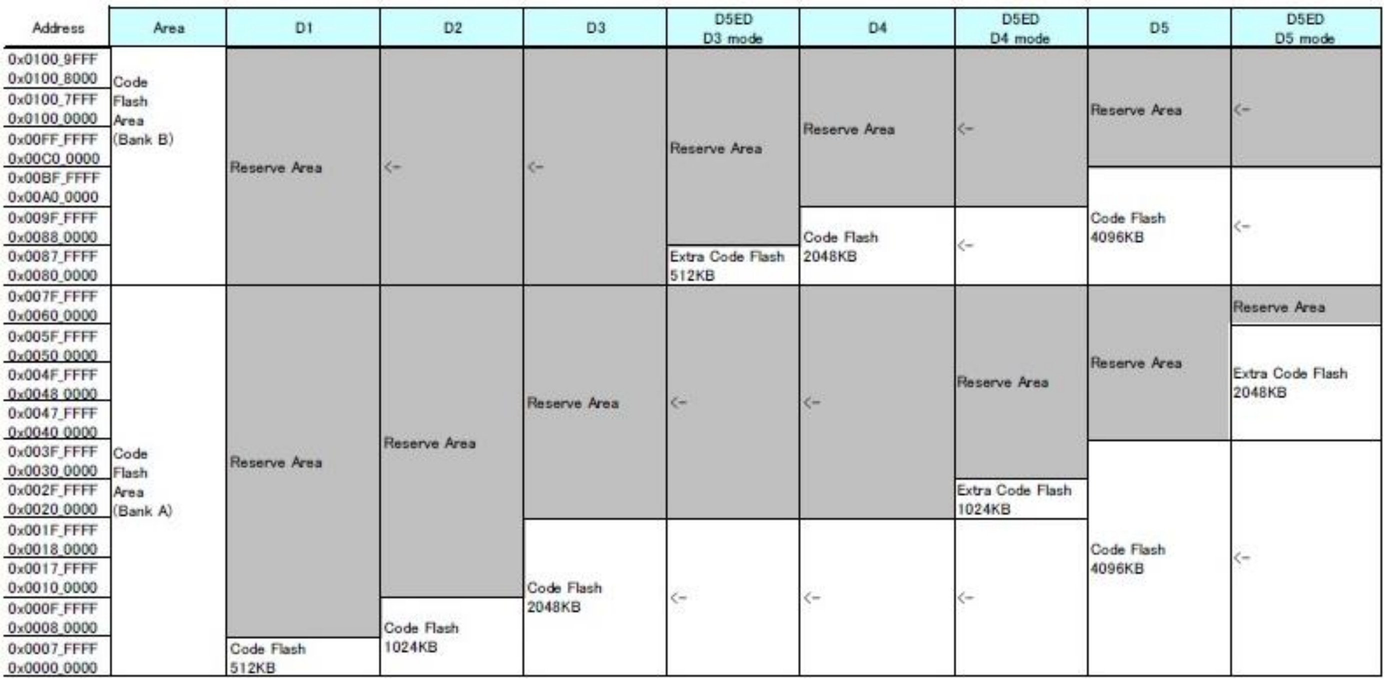
4. Supported devices

4.1. Memory layout

To enable (stand-alone) flashing of split code and data sections these sections must be allocated at flash section boundaries (see chapter 4.2). This will be checked by MTC on buildtime. It is the task of the ECU SW project to set the correct definitions inside the linker directive file (LDF). A side effect of this is, that for each hexfile block, up to one flash sector may be left unused.

Additionally MTC will check for invalid hexfile content on buildtime. If the MTC buildtarget in the current configuration equals to RAMBlocksOnly all content outside the **RamBorders** of the current device (see DDF.ini) will cause an error. In all other cases all content outside the **FlashBorders** of the current device (see DDF.ini) will cause an error.

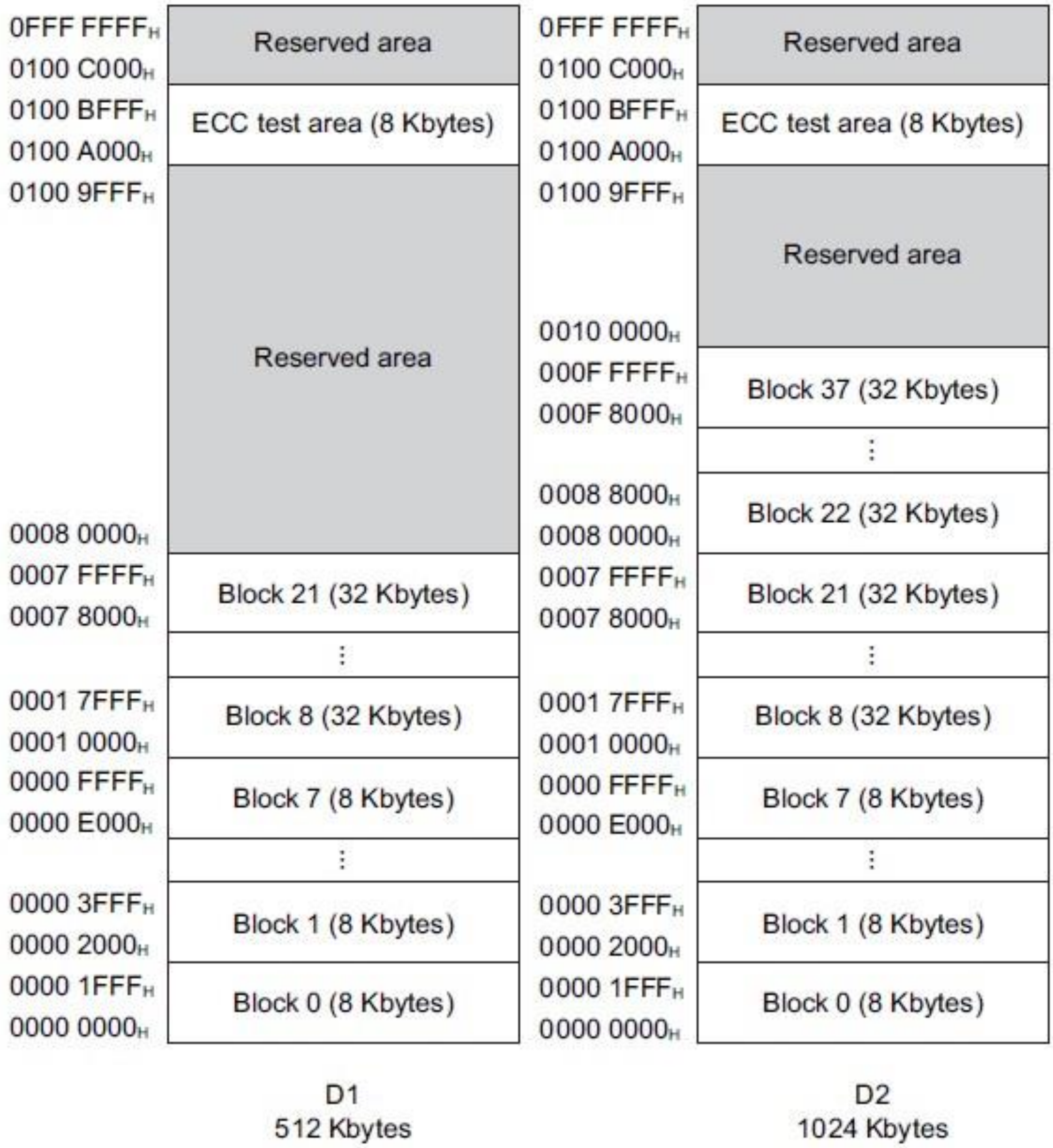
There are two banks of memory (BankA, BankB) which are used for code flash:



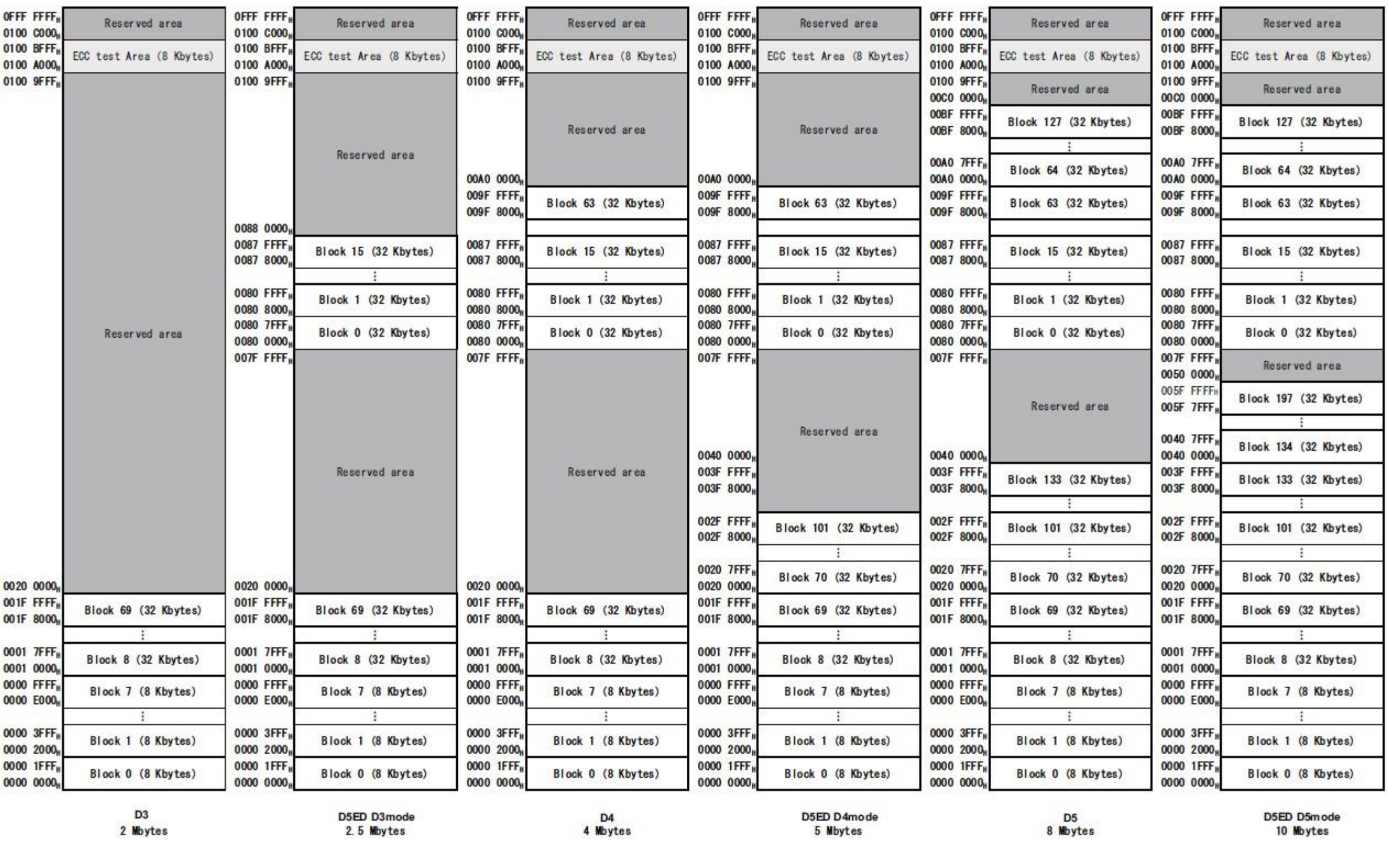
The flash boundaries will be checked by MTC. All hexcode which resides ouside the borders of the current device will cause an error.

4.2. Flash sectoring on supported Renesas devices

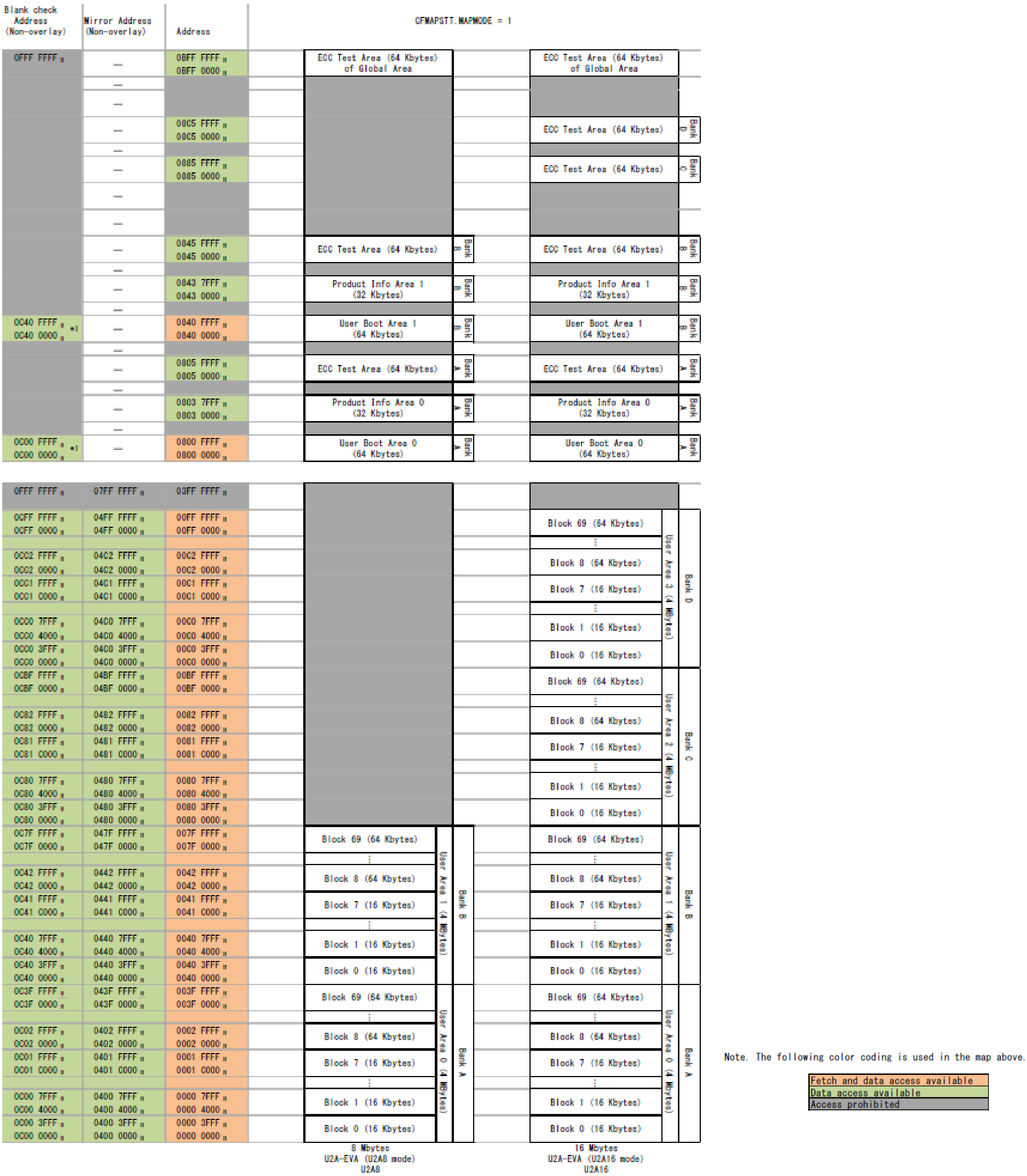
4.2.1. D1-D2



4.2.2. D3-D5


0FFF FFFFH
0100 C000H
0100 BFFFH
0100 A000H
0100 9FFFH
Reserved area
ECC test Area (8 Kbytes)
0088 0000H
0087 FFFFH
0087 8000H
Block 15 (32 Kbytes)
0080 FFFFH
0080 8000H
Block 1 (32 Kbytes)
0080 7FFFH
0080 0000H
Block 0 (32 Kbytes)
007F FFFFH
00A0 0000H
009F FFFFH
009F 8000H
Block 63 (32 Kbytes)
00C0 0000H
00BF FFFFH
00BF 8000H
Block 127 (32 Kbytes)
00A0 7FFFH
00A0 0000H
Block 64 (32 Kbytes)
0050 0000H
005F FFFFH
005F 7FFFH
Block 197 (32 Kbytes)
0040 7FFFH
0040 0000H
Block 134 (32 Kbytes)
003F FFFFH
003F 8000H
Block 133 (32 Kbytes)
002F FFFFH
002F 8000H
Block 101 (32 Kbytes)
0020 7FFFH
0020 0000H
Block 70 (32 Kbytes)
001F FFFFH
001F 8000H
Block 69 (32 Kbytes)
0001 7FFFH
0001 0000H
Block 8 (32 Kbytes)
0000 FFFFH
0000 E000H
Block 7 (8 Kbytes)
0000 3FFFH
0000 2000H
Block 1 (8 Kbytes)
0000 1FFFH
0000 0000H
Block 0 (8 Kbytes)
D3
2 Mbytes
D5ED D3mode
2.5 Mbytes
D4
4 Mbytes
D5ED D4mode
5 Mbytes
D5
8 Mbytes
D5ED D5mode
10 Mbytes

4.2.3. D6-D7



Blank check Address (Non-overlay)
Mirror Address (Non-overlay)
Address
CFMAPSTT.MAPMODE = 1
ECC Test Area (64 Kbytes) of Global Area
ECC Test Area (64 Kbytes)
Product Info Area 1 (32 Kbytes)
User Boot Area 1 (64 Kbytes)
Product Info Area 0 (32 Kbytes)
User Boot Area 0 (64 Kbytes)
Block 69 (64 Kbytes)
Block 8 (64 Kbytes)
Block 7 (16 Kbytes)
Block 1 (16 Kbytes)
Block 0 (16 Kbytes)
User Area 3 (4 MBytes)
User Area 2 (4 MBytes)
User Area 1 (4 MBytes)
User Area 0 (4 MBytes)
Bank A
Bank B
Bank C
Bank D
8 Mbytes
U2A-EVA (U2A8 mode)
U2A8
16 Mbytes
U2A-EVA (U2A16 mode)
U2A16
Note. The following color coding is used in the map above.
Fetch and data access available
Data access available
Access prohibited

5. Hexfile information

The HSW/PSW development has to provide a const structure inside the hexfile as placeholder for the MTC hexfile handling tools. The first byte of that block is an identifier describing the subsequent block structure enabling the tool to write the appropriate data into it. The identifier will be incremented if changes in the data structure are made.

A pointer to the base address of the SW hex info block will be added to the hexfile structure area at offset 0x10.

The hexfile information is written to a location inside every specified block of the current hexfile.

It is a requirement from CC/ECC SIKO-Team to include hexinfo structure into Flash checksum calculation!

5.1. Valid HexInfo structure IDs

5.1.1. Standard

Typically used for Bootblocks (e.g. BootManager, Bootloader), general purpose datablocks (e.g. TPSW).

Structure ID 0x01 is supported by MTC10.0.7.0 earliest. For older MTC usage please refer to Deprecated hexinfo structures.pdf

HexInfo Overall size: 0x24 Byte						
Rel. Address	**Field name**	**Size**	**C# Type**	**Example**	**Tool**	**Comment**
0x00	**Block structure ID**	1 Byte	byte	**0x01**	HSW/PSW	See chapter 5.2.1 to get all valid IDs
0x01	**Status MTC**	1 Byte	byte	0x00	Initialized to "0" by HSW/PSW Set by Gen9ProDB	0: Invalid 1: Valid See chapter 5.2.2.2
0x02	**Status SW**	1 Byte	byte	0x00	Initialized to "0" by HSW/PSW Set by Gen9ProDB	0: Invalid 1: Valid See chapter 5.2.2.1
0x03	**Status ET**	1 Byte	byte	0x01	Initialized to "1" by HSW/PSW Set by AppTools/HawCC	0: Invalid 1: Valid See chapter 5.2.2.3
0x04	**BB number**	5 Byte	string (ASCII)	"25409"	Gen9ProDB	nnnnn mit n=[0-9], see chapter 5.2.3
0x09	**MTC Configuration Number**	3 Byte	string (ASCII)	"002"	Gen9ProDB	Defines a valid MTC configuration, see chapter 5.2.4
0x0C	**CM-ID**	1 Byte	string (ASCII)	"0"	Gen9ProDB	See chapter 5.2.5
0x0D	**CommitID**	23 Byte	string (ASCII)	"_U62fCw3VEe SfCsb2qB8YLg"	Gen9ProDB	CM unique version ID (GIT Commit-ID, TCM Versionnumber), see chapter 5.2.6

5.1.2. FSW

Typically used in main FSW block of hexfile. Can be used only once per file.

Structure ID 0x11 is supported by MTC10.0.7.0 earliest. For older MTC usage please refer to Deprecated hexinfo structures.pdf

HexInfo Overall size: 0x38 Byte						
Rel. Address	**Field name**	**Size**	**C# Type**	**Example**	**Tool**	**Comment**
0x00	**Block structure ID**	1 Byte	byte	**0x11**	HSW/PSW	See chapter 5.2.1 to get all valid IDs
0x01	**Status MTC**	1 Byte	byte	0x00	Initialized to “0” by HSW/PSW Set by Gen9ProDB	0: Invalid 1: Valid See chapter 5.2.2.2
0x02	**Status SW**	1 Byte	byte	0x00	Initialized to “0” by HSW/PSW Set by Gen9ProDB	0: Invalid 1: Valid See chapter 5.2.2.1
0x03	**Status ET**	1 Byte	byte	0x01	Initialized to “1” by HSW/PSW Set by AppTools/HawCC	0: Invalid 1: Valid See chapter 5.2.2.3
0x04	**BB number**	5 Byte	string (ASCII)	“25409”	Gen9ProDB, **EPK**	nnnnn mit n=[0-9], see chapter 5.2.3
0x09	**MTC Configuration Number**	3 Byte	string (ASCII)	“002”	Gen9ProDB, **EPK**	Defines a valid MTC configuration, see chapter 5.2.4
0x0C	**CM-ID**	1 Byte	string (ASCII)	“0”	Gen9ProDB, **EPK**	See chapter 5.2.5
0x0D	**CommitID**	23 Byte	string (ASCII)	“_U62fCw3VEe SfCsb2qB8YLg”	Gen9ProDB, **EPK**	CM unique version ID (GIT Commit-ID, TCM Versionnumber), see chapter 5.2.6
0x24	**Date**	11 Byte	string (ASCII)	“_2007-05-05”	Gen9ProDB, **EPK**	see chapter 5.2.5 Date & Time
0x2F	**Time**	9 Byte	string (ASCII)	“_15:34:07”	Gen9ProDB, **EPK**	see chapter 5.2.5 Date & Time

All fields which are content of the EEPROM key (EPK) are marked in red.

5.1.3. CAL1 (no EOL)

Structure ID 0x21 is supported by MTC10.0.7.0 earliest. For older MTC usage please refer to Deprecated hexinfo structures.pdf

HexInfo Overall size: 0x48 Byte						
Rel. Address	**Field name**	**Size**	**C# Type**	**Example**	**Tool**	**Comment**
0x00	**Block structure ID**	1 Byte	byte	**0x21**	HSW/PSW	See chapter 5.2.1 to get all valid IDs
0x01	**Status MTC**	1 Byte	byte	0x00	Initialized to “0” by HSW/PSW Set by Gen9ProDB	0: Invalid 1: Valid See chapter 5.2.2.2
0x02	**Status SW**	1 Byte	byte	0x00	Initialized to “0” by HSW/PSW Set by Gen9ProDB	0: Invalid 1: Valid See chapter 5.2.2.1
0x03	**Status ET**	1 Byte	byte	0x01	Initialized to “1” by HSW/PSW Set by AppTools/HawCC	0: Invalid 1: Valid See chapter 5.2.2.3
0x04	**BB number**	5 Byte	string (ASCII)	“25409”	Gen9ProDB	nnnnn mit n=[0-9], see chapter 5.2.3
0x09	**MTC Configuration Number**	3 Byte	string (ASCII)	“002”	Gen9ProDB	Defines a valid MTC configuration, see chapter 5.2.4
0x0C	**CM-ID**	1 Byte	string (ASCII)	“0”	Gen9ProDB	See chapter 5.2.5
0x0D	**CommitID**	23 Byte	string (ASCII)	“_U62fCw3VEe SfCsb2qB8YLg”	Gen9ProDB	CM unique version ID (GIT Commit-ID, TCM Versionnumber), see chapter 5.2.6
0x24	**Application Tool**	8 Bytes	string (ASCII)	“PMS/E”	Application tool	Only valid tool is PMS/E, see chapter 5.3
0x2C	**Application version**	2 Byte	uint (UInt16)	0x0012	Application tool	see chapter 5.3
0x2E	**Application date**	10 Byte	string (ASCII)	“2000-11-06“	Application tool	"yyyy-mm-dd", see chapter 5.3
0x38	**Application time**	8 Byte	string (ASCII)	“17:33:05“	Application tool	"hh:mm:ss", see chapter 5.3
0x40	**Application engineer**	8 Byte	string (ASCII)	“ECK2SI “	Application tool	<NT user>, see chapter 5.3

5.1.4. CAL2 (EOL)

Structure ID 0x31 is supported by MTC10.0.7.0 earliest. For older MTC usage please refer to Deprecated hexinfo structures.pdf

HexInfo Overall size: 0x6E Byte						
Rel. Address	**Field name**	**Size**	**C# Type**	**Example**	**Tool**	**Comment**
0x00	**Block structure ID**	1 Byte	byte	**0x31**	HSW/PSW	See chapter 5.2.1 to get all valid IDs
0x01	**Status MTC**	1 Byte	byte	0x00	Initialized to “0” by HSW/PSW Set by Gen9ProDB	0: Invalid 1: Valid See chapter 5.2.2.2
0x02	**Status SW**	1 Byte	byte	0x00	Initialized to “0” by HSW/PSW Set by Gen9ProDB	0: Invalid 1: Valid See chapter 5.2.2.1
0x03	**Status ET**	1 Byte	byte	0x01	Initialized to “1” by HSW/PSW Set by AppTools/HawCC	0: Invalid 1: Valid See chapter 5.2.2.3
0x04	**BB number**	5 Byte	string (ASCII)	“25409”	Gen9ProDB	nnnnn mit n=[0-9], see chapter 5.2.3
0x09	**MTC Configuration Number**	3 Byte	string (ASCII)	“002”	Gen9ProDB	Defines a valid MTC configuration, see chapter 5.2.4
0x0C	**CM-ID**	1 Byte	string (ASCII)	“0”	Gen9ProDB	See chapter 5.2.5
0x0D	**CommitID**	23 Byte	string (ASCII)	“_U62fCw3VEeSfCsb2qB8YLg”	Gen9ProDB	CM unique version ID (GIT Commit-ID, TCM Versionnumber), see chapter 5.2.6
0x24	**Application Tool**	8 Bytes	string (ASCII)	“PMS/E”	Application tool	Only valid tool is PMS/E, see chapter 5.3
0x2C	**Application version**	2 Byte	uint (UInt16)	0x0012	Application tool	see chapter 5.3
0x2E	**Application date**	10 Byte	string (ASCII)	“2000-11-06“	Application tool	"yyyy-mm-dd", see chapter 5.3
0x38	**Application time**	8 Byte	string (ASCII)	“17:33:05“	Application tool	"hh:mm:ss", see chapter 5.3
0x40	**Application engineer**	8 Byte	string (ASCII)	“ECK2SI “	Application tool	<NT user>, see chapter 5.3
0x48	**Logistic Number (CAL-BB)**	5 Byte	string (ASCII)	“12345”	Gen9ProDB	nnnnn mit n=[0-9], see chapter 5.4
0x4D	**Default Variant**	1 Byte	byte (Byte)	0x02	Gen9ProDB	= 01-99, see chapter 5.4
0x4E	**Customer Data**	32 Byte	string (ASCII)	“Opel X4400”	Gen9ProDB	32byte freetext, see chapter 5.4

5.1.5. Complete

Typically used in CSW hexfiles without calblock. Can be used only once per file.

Structure ID 0x41 is supported by MTC10.0.7.0 earliest. For older MTC usage please refer to Deprecated hexinfo structures.pdf

HexInfo Overall size: 0x5C Byte						
Rel. Address	**Field name**	**Size**	**C# Type**	**Example**	**Tool**	**Comment**
0x00	**Block structure ID**	1 Byte	byte	**0x41**	HSW/PSW	See chapter 5.2.1 to get all valid IDs
0x01	**Status MTC**	1 Byte	byte	0x00	Initialized to “0” by HSW/PSW Set by Gen9ProDB	0: Invalid 1: Valid See chapter 5.2.2.2
0x02	**Status SW**	1 Byte	byte	0x00	Initialized to “0” by HSW/PSW Set by Gen9ProDB	0: Invalid 1: Valid See chapter 5.2.2.1
0x03	**Status ET**	1 Byte	byte	0x01	Initialized to “1” by HSW/PSW Set by AppTools/HawCC	0: Invalid 1: Valid See chapter 5.2.2.3
0x04	**BB number**	5 Byte	string (ASCII)	“25409”	Gen9ProDB, **EPK**	nnnnn mit n=[0-9], see chapter 5.2.3
0x09	**MTC Configuration Number**	3 Byte	string (ASCII)	“002”	Gen9ProDB, **EPK**	Defines a valid MTC configuration, see chapter 5.2.4
0x0C	**CM-ID**	1 Byte	string (ASCII)	“0”	Gen9ProDB, **EPK**	See chapter 5.2.5
0x0D	**CommitID**	23 Byte	string (ASCII)	“_U62fCw3VEe SfCsb2qB8YLg”	Gen9ProDB, **EPK**	CM unique version ID (GIT Commit-ID, TCM Versionnumber), see chapter 5.2.6
0x24	**Date**	11 Byte	string (ASCII)	“_2007-05-05”	Gen9ProDB, **EPK**	see chapter 5.2.5 Date & Time
0x2F	**Time**	9 Byte	string (ASCII)	“_15:34:07”	Gen9ProDB, **EPK**	see chapter 5.2.5 Date & Time
0x38	**Application Tool**	8 Bytes	string (ASCII)	“PMS/E”	Application tool	Only valid tool is PMS/E, see chapter 5.3
0x40	**Application version**	2 Byte	uint (UInt16)	0x0012	Application tool	see chapter 5.3
0x42	**Application date**	10 Byte	string (ASCII)	“2000-11-06“	Application tool	"yyyy-mm-dd", see chapter 5.3
0x4C	**Application time**	8 Byte	string (ASCII)	“17:33:05“	Application tool	"hh:mm:ss", see chapter 5.3
0x54	**Application engineer**	8 Byte	string (ASCII)	“ECK2SI “	Application tool	<NT user>, see chapter 5.3

All fields which are content of the EEPROM key (EPK) are marked in red.

5.1.6. Deprecated hexinfo structures

All deprecated hexinfo structures can be found in the following document: Deprecated hexinfo structures.pdf

5.2. Coding of HexInfo entries

5.2.1. Block structure ID

The block structure ID is used to determine the internal structure of the hexfile information.

Currently the only valid IDs for Gen 9.3 projects are:

STANDARD:	0x00/0x01
FSW:	0x10/0x11
CAL1:	0x20/0x21
CAL2:	0x30/0x31
COMPLETE:	0x41

In future SW releases there may be additional values. The block structure ID is increased on every change which is done to the internal structure of the hexfile information.

5.2.2. Status Flags

The status flags are necessary to determine if a given hexfile is valid for the plant. MTC will set the flags dependent of the current build settings. The third flag is set if external modifications have been done to the file.

The RomTrans tool must check all three flags within all blocks of a given file. Only if all checked flags are confirmed to be “1” the hexfile is ready for production. In any other case RomTrans must refuse to pass the file to the plant.

5.2.2.1. Status SW

This Boolean “Project SW state” value indicates if the current hexfile is valid for delivery to the plant:

0: invalid for the plant

1: valid for the plant

The hexfile is marked as invalid for the plant if one of the following conditions is true:

- local build (Jenkins server not used)
- RTC UUID (Snapshot TimeStamp) is not available

The “Project SW state” flag has to be checked by RomTransTool.

5.2.2.2. Status MTC

This Boolean “MTC build environment state” value indicates if the current hexfile is valid for delivery to the plant:

0: invalid for the plant

1: valid for the plant

The hexfile is marked as invalid for the plant if one of the following conditions is true:

- Status SW is invalid (respective flag is set to “0”)
- Release flag within respective MTC configuration not set
- Incremental build
- DevLatest MTC is used

For additional possible reasons see chapter “Dirty flag Handling” of MTCUserGuide:
C:\MTC10Tools\GenMake\V<currentversion>\manuals\MTC10UserGuide.pdf

To evaluate the reasons for a distinct build see the file _DirtyFlagReasons.txt in the respective out folder.

The “MTC build environment state” flag has to be checked by RomTransTool.

5.2.2.3. Status ET

This is the status set by external tools (e.g. PMS/E, HawCC, …):

0: invalid for the plant

1: valid for the plant

The hexfile is marked as invalid for the plant if one of the following conditions is true:

- The hexfile was locally modified by an external tool (e.g. PMS/E, HawCC)

The “External Tool state” flag has to be checked by RomTransTool.

The external Tool has to set the “Status ET” flag to invalid in every block which has been externally modified.

5.2.3. BBNumber

The BBNumber is a unique 5-Byte ASCII number which identifies the current project SW for the plant.

5.2.4. MTC Configuration

Every SW generated with MTC10 gets a configuration name which can be specified by user. This configuration name will be coded in the name of every MTC generated output file.

Example for a valid configuration name: EV7130xD5EDxAPB

Example for the naming of the corresponding MTC generated hexfile:
PRJ_Hexfile_BB90058_$\boxed{\textbf{EV7130xD5EDxAPB}}$.hex

Every configuration name refers to a unique configuration number which will be coded in hexinfo as an hexadecimal 3-byte ASCII string.

Example:

EV7130xD5EDxAPB	0x000	this leads to "000" in hexfile
EV7130xD5EDxEPSW	0x001	this leads to "001" in hexfile
EV7130xHSWSim	0xFFF	this leads to "FFF" in hexfile

The MTC configuration number will be added to EPK since MTC 9.3.

5.2.5. CM-ID

This ID specifies which Configuration Management System was used to archive the sources for the current ECU project.

0: unknown (e.g. local build)

1: TCM

2: GIT

3: RTC (ALM)

4: others

5.2.6. Commit ID

The COMMIT ID in GIT (= UUID in RTC) is a unique id (23 ASCII characters) and will replace gen9.1/gen9.2 TCM version number in gen9.3.

The COMMIT ID will change every time a new version of the project is checked into the repository.

If SCM-Tool will be replaced/changed this could be handled with different Block Structure IDs

5.2.7. Date & Time

Date & Time Information is patched by Gen9ProDB into the hexfile.

Coding of **Date & Time** if field „Status SW" == „1":

Date/Time of the project version of last GID commit.

Coding of **Date & Time** if field „Status SW" == „0":

Current system time.

The date/time stamp is typically content of the first CodeBlock in hexfile. It contributes to the EPK information.

5.3. ApplicationData

All application tools modifying a MTC generated hexfile must fill in the following fields of the hexfile information:

Application Tool:	currently the one and only tool is PMS/E
Application Date:	Date of modification
Application Time:	Time of modification
Application Engineer:	The UserID of the person, which did the modification
Application Version:	BuildNumber of the current used version of application tool

"Application Version" must be a unique number which identifies the version of the used application tool. It must be provided by the application tool itself.

The "Application Data" fields are typically content of all valid CALBlocks in hexfile.

5.4. EOL VariantData

Logistic data will be available in an additional hexfile information structure for variant calibration blocks only.

Logistic data will be read out of file ParameterAllocation_BB<xxxxx>.xml and stored to CalInfo by Gen9ProDB.

Logistic data must contain the following information:

- DEFAULT_VARIANT: This variant will be patched to complete hexfile generated by MTC.
- LOGISTIC_NUMBER: BB-Number of actual variant CAL block (or a project specific number if a separate Cal-BB is not required).
- CUSTOMER_DATA: special comment / customer specific number

For additional information on configuring the Hexinfo content via ParameterAllocation_BB<xxxxx>.xml for EOL projects see the following document: Multi block hexfile handling.pdf

The “EOL VariantData” is typically content of all variant CALBlocks within an EOL project SW.

5.5. EEPROM key (EPK)

The EPK consists of the following ASCII data:

- BBNumber
- MTCConfigNumber
- CM-ID
- CommitID
- Date/Time-Stamp

All entries used for calculating EPK are marked in red within the table in chapters 5.1.2/5.1.3. The EPK is typically content of the hexfile information marked as FSW.

6. CHECK structure area

Each hexblock requires a checksum structure allocated near to the end of its data area. The starting address of this structure must be 4 bytes aligned. No safety critical parts are allowed to be placed/allocated behind that structure.

The CheckRef pointer at address <HexBlockStructureStart>+0x00000010 points to the beginning of the CHECK structure within the current block.

The checksum is calculated for every HexBlock twice: Once for the first half of the area relevant for checksum calculation and once for the second half. The separator address for the begin of the area for the second checksum is calculated and patched into the hexfile by MTC. (Hint: Separator address must be 4k aligned)

This is mainly done to utilize within a MultiCore controller both cores. This means that both cores are able to calculate one checksum per HexBlock during uC startup, which reduces startup time.

Additionally there is a third checksum used for consistency check which only includes the data of a defined interface.

6.1. Valid CHECK structure IDs

6.1.1. Standard

The standard CHECK structure is used if a consistency check is not planned for current block.

Standard Overall Size: 0x10 Bytes						
Offset	**Field name**	**Size**	**C# Type**	**Example**	**Tool**	**Comment**
0x00	**Block structure ID**	1 Byte	Byte	0x02	HSW/PSW	0x01 - 0x0F
0x01	**AlgorithmID**	1 Byte	Byte	0x01	HSW/PSW	Valid checksum IDs are 0x00-0x10
0x02	**ChecksumSegmentation**	1 Byte	Byte	0x01	Initialized to 0xFF by HSW/PSW, Set by Gen9ProDB	Flag which indicates if Checksum2 is used or not
0x03	**Reserved**	1 Byte	-	-	Initialized to 0x00 by HSW/PSW, Set by Gen9ProDB	Used for alignment, must be 0x00
0x04	**SeparatorRef**	4 Byte	UInt32	0x00020000	Initialized to 0xFFFFFFFF by HSW/PSW, Set by Gen9ProDB	Separates the two different checksum areas from another (points to startadr. of second part)
0x08	**Checksum1**	4 Byte	UInt32	0x12345678	Initialized to 0xFFFFFFFF by HSW/PSW, Set by Gen9ProDB	Checksum for first part of current hexblock
0x0C	**Checksum2**	4 Byte	UInt32	0x12345678	Initialized to 0xFFFFFFFF by HSW/PSW, Set by Gen9ProDB	Checksum for second part of current hexblock

6.1.2. Extended

The extended CHECK structure is used if a consistency check to at least one other block is planned. The lines marked in green will be available n times whereas the lines marked in red will be available m times.

Extended Overall Size: 0x14 Bytes minimum						
Offset	**Field name**	**Size**	**C# Type**	**Example**	**Tool**	**Comment**
0x00	**Block structure ID**	1 Byte	Byte	0x12	HSW/PSW	0x12 - 0x1F
0x01	**AlgorithmID**	1 Byte	Byte	0x01	HSW/PSW	Valid checksum IDs are 0x00-0x10
0x02	**ChecksumSegmentation**	1 Byte	Byte	0x01	Initialized to 0xFF by HSW/PSW, Set by Gen9ProDB	Flag which indicates if Checksum2 is used or not
0x03	**Reserved**	1 Byte	-	-	Initialized to 0x00 by HSW/PSW, Set by Gen9ProDB	Used for alignment, must be 0x00
0x04	**SeparatorRef**	4 Byte	UInt32	0x00020000	Initialized to 0xFFFFFFFF by HSW/PSW, Set by Gen9ProDB	Separates the two different checksum areas from another (points to startadr. of second part)
0x08	**Checksum1**	4 Byte	UInt32	0x12345678	Initialized to 0xFFFFFFFF by HSW/PSW, Set by Gen9ProDB	Checksum for first part of current hexblock
0x0C	**Checksum2**	4 Byte	UInt32	0x12345678	Initialized to 0xFFFFFFFF by HSW/PSW, Set by Gen9ProDB	Checksum for second part of current hexblock
0x10	**Reserved**	2 Byte	-	-	HSW/PSW	Used for alignment, must be 0x00
0x12	**Interface count (provide)**	1 Byte	Byte	0x01	HSW/PSW	Number of interfaces provided by current block
0x13	**Interface count (consume)**	1 Byte	Byte	0x01	HSW/PSW	Number of interfaces consumed by current block
0x14	**InterfaceRef (provide)**	4 Byte	UInt32	0x00040000	HSW/PSW	Pointer to an interface structure within current block . This entry will be available n times.
0x18	**InterfaceRef (consume)**	4 Byte	UInt32	0x00050000	HSW/PSW	Pointer to an interface checksum within another block. This entry will be available m times.
0x1C	**InterfaceChecksum**	4 Byte	UInt32	0x12345678	Initialized to 0xFFFFFFFF by HSW/PSW, Set by Gen9ProDB	Interface checksum copied from another block

6.1.3. Interface

The interface CHECK structure is used to define an interface for consistency checking. Each InterfaceRef in EXTENDED structure points to a related INTERFACE structure.

All interface CHECK structures in a hexblock must be located behind the related EXTENDED structure in the same block because the calculated interface checksum must not be part of the standard ROM checksum.

Interface Overall Size: 0x14 Byte						
Offset	**Field name**	**Size**	**C# Type**	**Example**	**Tool**	**Comment**
0x00	**Block structure ID**	1 Byte	Byte	0x22	HSW/PSW	0x22 - 0x2F
0x01	**AlgorithmID**	1 Byte	Byte	0x00	HSW/PSW	Valid ID which defines the algorithm to claculate interface checksum
0x02	**InterfaceName**	6 Byte	String (ASCII)	“CAL1”	HSW/PSW	Name of current interface
0x08	**InterfaceStartAddress**	4 Byte	UInt32	0x0004000	HSW/PSW	Start address of the valid area to calculate interface checksum
0x0C	**InterfaceEndAddress**	4 Byte	UInt32	0x0005000	HSW/PSW	End address of the valid area to calculate interface checksum
0x10	**InterfaceChecksum**	4 Byte	UInt32	0x456789ab	Initialized to 0xFFFFFFFF by HSW/PSW, Set by Gen9ProDB	Interface checksum calculated by Gen9ProDB

6.2. Deprecated CHECK structures

All deprecated CHECK structures can be found in the following document:

Deprecated_CHECK_structures.pdf

6.3. Coding of CHECK Structure

6.3.1. BlockStructureID

The BlockStructureID tells details about the used version of the CHECK structure. All related tools will read this ID and treat the following data accordingly

6.3.2. AlgorithmID

The algorithm ID tells details about how to generate a valid checksum out of the available data.

Valid CHECK algorithm IDs	
StandardChecksum	0x01
ConsistencyChecksum	0x10
InterfaceChecksum1	0x20
InterfaceChecksum2	0x30

6.3.2.1. StandardChecksum

In Gen 9.3 a CRC32 algorithm will be used for default to calculate ROM checksum.

Valid checksum data will be all data except for the following areas:

- CHECK
- all data between CHECK and the end of current block.

Range Checksum1: <BlockStart> … <CheckSeparator-1>

Range Checksum2: <CheckSeparator> … <CheckRef-1>.

CHECK = CRC32(valid 32-bit words of address range)

The CheckRef address can be read at <HexBlockStructureStart>+0x00000010 of RawHexfile.

This ID is only valid in combination with STANDARD or EXTENDED BlockStructureID.

6.3.2.2. ConsistencyChecksum

For consistency checksum calculation the same algorithm is used than for the standard ROM checksum, but with a different range.

Address range = from <blockstart> to <CheckRef>, but without the complete hexinfo area

This ID is only valid in combination with INTERFACE BlockStructureID.

The range for checksum calculation can be limited additionally by InterfaceStartAddress and InterfaceEndAddress.

6.3.2.3. InterfaceChecksum

These algorithms will not use any data from RawHexFile for checksum calculation at all. We will use external metadata instead.

There are several possibilities how to calculate/get interface checksum.

These AlgorithmIDs are only valid in combination with INTERFACE BlockStructureID.

Code/Data Separation (InterfaceChecksum1):

The parameter data (address, modelname, datatype) collected by MTC will be used to calculate a CRC32 checksum. **This will only work with CalBlocks which contain Ascet parameter values only. This will not work in reduced projects using GenExSW.**

Code/Code Separation (InterfaceChecksum2):

This algorithm for consistency checksum calculation is currently not supported by MTC. If you are in an urgent need for this feature please contact “ISM-Box MTC”.

6.3.3. Checksum1/Checksum2

All checksums used in Gen 9.3 hexfile handling have CRC 32 type.

Gen9ProDB has to calculate a 32 bit checksum depending on the hexfile content and patch the result into the CHECK structures. These checksums will be verified by the ECU SW at run time.

The algorithm for the checksum calculation is defined by the identifier at offset 0x01 of the CHECK block. Typically it will have STANDARD type.

6.3.4. CheckSeparator

The CheckSeparator is the address at which Checksum1 range ends and Checksum2 range starts. It has no influence on calculating consistency checksum.

The CheckSeparator address is calculated by MTC using the following algorithm:

CheckSeparator = <BlockStart> + <CheckRef-BlockStart>/2 + Alignment

Alignment: If the result of this calculation is not a valid 32-bit address an alignment will be added.

6.3.5. CheckSegmentation Flag

If the splitting of the checksum area was done and 2 checksums are patched, MTC must set the ChecksumSegmentation flag to TRUE.

If the CheckStructure is located within the first 4kB area of a hexblock (e.g. partial CAL blocks) only Checksum1 will be used/patched and the ChecksumSegmentation flag will be set by MTC to FALSE.

In this case Checksum2 must be set to 0xFFFFFFFF by MTC and SeparatorRef will be set by MTC to StartAdr of CheckStruct.

6.3.6. InterfaceRef/InterfaceChecksum

There are two types of InterfaceRef pointer values.

InterfaceRef (provide) points to an interface structure within the current block. These interface structures will be filled with calculated checksum data during MTC run.

InterfaceRef (consume) points to an interface structure in another block. The checksum data available there will be copied to the InterfaceChecksum field of the current CHECK structure.

6.3.7. InterfaceName

For better determination of a distinct interface a CHECK structure with INTERFACE type can be marked with a unique name (6 Byte ASCII string)

.

6.3.8. Interface address borders

These values can only be used in CHECK structures with INTERFACE type. They are used to handle a user defined address range for checksum calculation. Interface address borders are only valid in combination with ConsistencyChecksum algorithmID and will override the borders of the default range.

If the default range should be used both values must be set to zero (default).

6.4. Consistency Check

The verification that two given hexblocks are consistent is mandatory for the correct behavior of the ECU SW.

The consistency information is created by the Gen9ProDB tool and patched during build into the different SW blocks. **The data structure containing the consistency check data must be located in an area, which is not regarded for checksum calculation**, thus we decided to add this to a separate interface structure in Gen 9.3.

The consistency check part of the CHECK structure contains references to the interface data, provided by other blocks within the same hexfile. It is the task of the ECU project to set up the number and the content of these reference pointers correctly. The Gen9ProDB tool will – after calculation of all interface checksums – copy the data described by the pointers to the appropriate data fields. Thus any consistency check between any hexfile blocks could be implemented.

Typically consistency checks are used in projects which take part in code/data separation and EOL variant handling.

If no consistency check should be available for the current block, the “interface number” fields must be turned to zero (this block will not consume or provide any interface data then).

The consistency/interface checksum of a distinct hexblock is calculated by using a defined amount of data which specifies the interface provided by the current block (e.g. name/address/datatype of all parameters within a calblock).

All consistency/interface checksums are calculated and patched by Gen9ProDB to the respective addresses within current hexfile.

An MTC external tool which has to perform a consistency check (e.g. XFlash) must do the following:

- **walk all CHECK structures with EXTENDED type in current hexconfiguration (hexconfiguration = an sum of hexblocks which are combined to be flashed together to ECU)**
- **search for InterfaceRef (consume) there.**
- **The value InterfaceRef points to, must be identical to the InterfaceChecksum directly below**

6.5. Examples

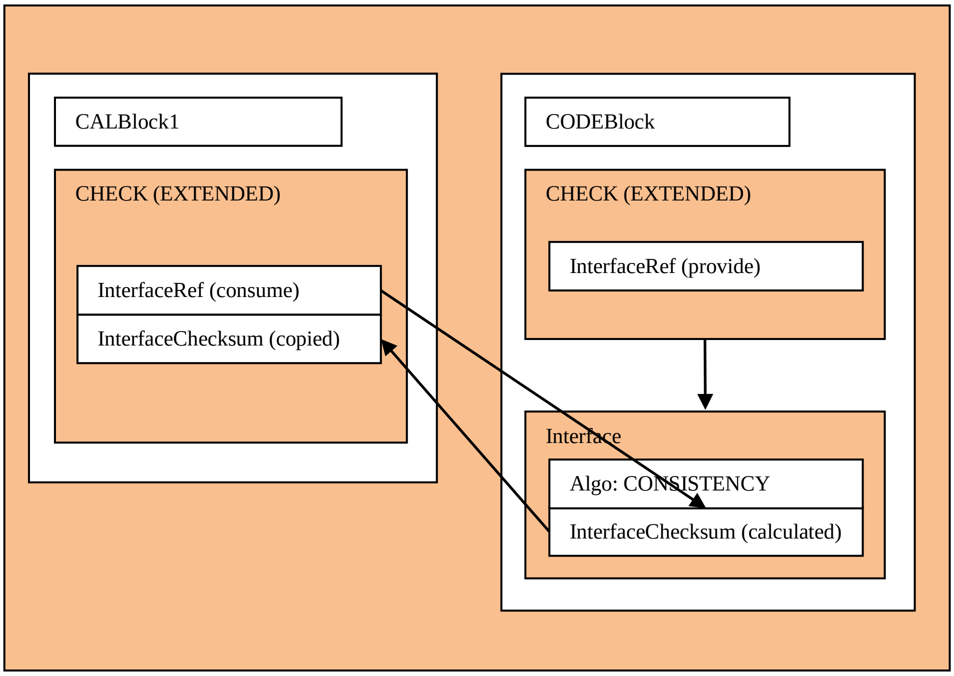
Currently there are only two examples available. This chapter will be amended as soon as additional usecases for consistency checking in gen9.3 are known.

6.5.1. Standard consistency check

This was the only possibility to handle hexblock consistency in gen8, gen9.1 and gen9.2. It is typically used to ensure that a new CalBlock file (with different parameter values inside) fits to an existing CodeBlock.

A CRC32 consistency checksum will be calculated on the complete hexdata in the calblock (except the area behind CheckStart).

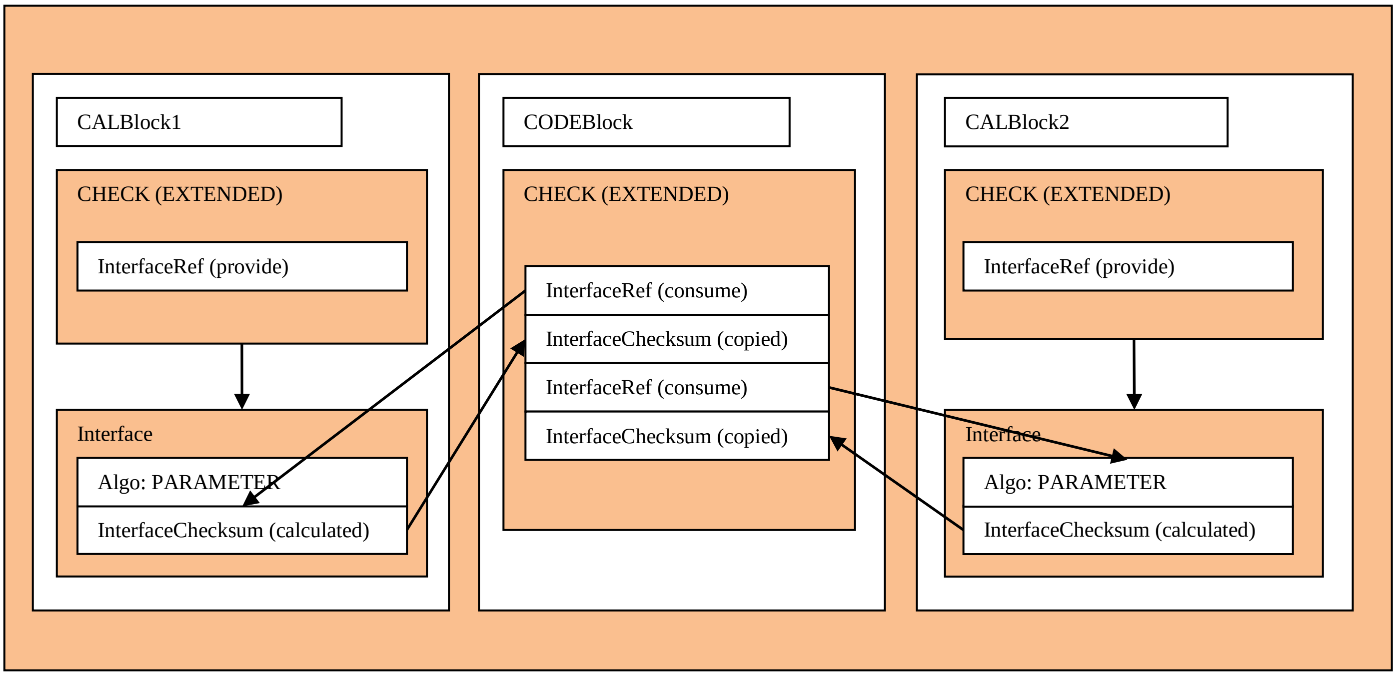
The standard consistency check will fail if the data in CodeBlock has been changed between two builds and the user wants to combine the newly generated CalBlock with the original CodeBlock. It will succed if the CodeBlock data is functionally still identical. The standard consistency check will work fine only in one direction. It is not possible to combine a newly generated CodeBlock with an existing CalBlock.



6.5.2. Extended interface check

While using extended interface check, the checksum will not be calculated on the complete hexdata in the CalBlock, but on the related parameter data collected by MTC build.

The extended interface check will only fail if the internal structure of parameters has been changed (modelname, address, datatype). The check will work fine in each direction, no matter if a newly generated FSW should be combined with an existing CAL, or a newly generated CAL should be combined with an existing FSW.



7. Signature Handling

In the HexBlock structure there is a pointer to the signature area, with the possibility to sign hexblocks in order to ensure their integrity of origin. Independent of possible customer requirements there is the concept of RB-signature for generic security aspects.

7.1. RB-signature

The entry point for the signature information has the structure:

```
typedef struct
{
    uint32                  NumOfSigStructs;
    RBLCF_PKISigStruct_t    SigInfoStruct[RBLCF_NUMSIGSTRUCTS];
}RBLCF_PKISigInfoStruct_t;
```

The following structure is used for the RB-signature (ID = 0x11):

```
typedef struct
{
    uint8                   StructID;       // = 0x11 (certificate + signature structure)
    uint8                   CertType;       // = 0x01 (CVC1)
    uint8                   SigType;        // = 0x01 (PKI signature)
    uint8                   Auth;           // index to content of authorization flags table
    uint8                   ObjectID[32];   // ObjectID -> points to project
    RBLCF_PKICertificate_t  CertStruct;     // length of and pointer to certificate
    RBLCF_PKISignature_t    SigStruct;      // length of and pointer to signature
}RBLCF_PKISigStruct_t;
```

with:

```
typedef struct
{
    uint32          CertificateLength;  // length of the certificate
    const uint8*    Certificate;        // pointer to the certificate
}RBLCF_PKICertificate_t;
```

and for signature accordingly.

For more details on signatures and their handling refer to the dedicated document:

\\bosch.com\dfsrb\DfsDE\DIV\CS\DE_CS$\Tools\PlantIF\SignX\PKI-Structures.pdf

The SW in RBLCF has to provide these structures for the different hexblocks. According to a decision by the security team all hexblocks of the different types (HSM exempted) shall contain a signature.

If a standard hexblock configuration is used the signature relevant fields are automatically provided by RBLCF_Hexblocks_Allocation_Signature.c. For extraordinary configurations (RBFS_HexBlockConfig_Customized, where the other hexblock description blocks have to be defined manually in RBLCF_Customized_Hexblocks_Allocation.c) the required signature parts also have to be provided manually in RBLCF_Customized_Hexblocks_Allocation_Signature.c.

Some notes on the different fields:

- ObjectID (OID)

  It is project specific and has to be obtained from the security-responsible for that project. It consists of a length-Byte and up to 31 content-Bytes coded in ASN1. A tool to translate the dot-representation of an OID into the correct ASN1-sequence can be found here:

  http://abt-ismtwiki.abt.de.bosch.com/twiki/bin/view/Tools/SignX#OID_Calculation_Tool

  The ASN1-representation (together with the correct length as Byte0) has to be filled into the project specific file RBLCF_OID4Signature.h derived from the template in RBLCF.

- Certificate and Signature

  These fields are provided in a dummy way in order to be filled by a signing tool (e.g. SignX or the MTC mechanism for development builds).

Design Decisions:

- Only structures with exactly 1 certificate and exactly 1 signature are used for the RB-signature, i.e. not more than one signature etc. This means that the size of the signature info (ID=0x11 above) is 56 Bytes.

- The content of the signature info, the certificate and the signature (which are referenced by the pointers) are separate parts and shall be located AFTER the Checkstruct and Checksum to avoid a specific order in tools execution (-> no new checksum calculation needed after the certificate/signature is filled)

- It was decided that the RBLCF has to put into the dummy length field the maximum allowed length as an information for the signing tool, which then replaces it by the 'real', i.e. used, length. The maximum lengths are currently:

  1024 Bytes for the certificate

  256 Bytes for the signature

8. Device Description file

The device description file (DDF) contains all data that is necessary to manipulate / interpret a hexfile for a given device.

8.1. Definition

The DDF has the Windows INI file format, with the following entries:

```
[ID]
ProductName=<identifier>
Supplier=<Renesas|STM>
AliasName=<identifier>
FlashSectionStart<n>=<address>
FlashSectionCount<n>=<count>
FlashSectionLength<n>=<length>
FlashOptionBytesStart<n>=<address>
FlashOptionBytesLength<n>=<length>
RAMSectionStart<n>=<address>
RAMSectionLength<n>=<length>
```

<ID>:	hexadecimal number (8 digits)
<identifier>:	name
<address>:	hexadecimal number (8 digits)
<length>:	hexadecimal number (8 digits)
<count>:	hexadecimal number (8 digits)
<n>:	1..n

Areas within a device which are not grouped within DDF do not belong to the usable flash area and are protected from writing (hardware gaps). These areas will stay gaps even within the hexfile generated by MTC HexFileWriter.

8.2. Example

```
Supplier=Renesas
AliasName=D1
ProductName=R7F7xxxx
DeviceFamily=RH850/P1x-C
DeviceShortName=RH850/P1L-C
FlashSectionStart1=0x000000000
FlashSectionCount1=0x0008
FlashSectionLength1=0x00002000
FlashSectionStart2=0x00010000
FlashSectionCount2=0x000E
FlashSectionLength2=0x00008000
FlashOptionBytesStart1=
FlashOptionBytesLength1=
FlashOptionBytesStart2=
FlashOptionBytesLength2=
FlashOptionBytesStart3=
FlashOptionBytesLength3=
RAMSectionStart1=0xFEBF1800
RAMSectionLength1=0x0000E800
```

9. Tools

9.1. HexEditor

There is a tool called HawCC which can be used to compare and edit hexfiles.

Location: C:\MTC10Base\HexfileHandling\HawCC

Functional range HawCCEdit:

- Edit hexfiles
- Full address support
- ASCII equivalent to hexcode
- Highlighting of RB specific areas
- Jump directly to all RB specific areas

Functional range HawCCCompare:

- Compare hexfiles
- Exclude RB specific areas from compare
- Exclude configurable via INI file
- Interface to BeyondCompare
- Compare ASCII equivalent and full address

See: TwikiPages of HawCC tool
See: C:\MTC10Base\HexfileHandling\HawCC\hawcc_documentation.pdf

10. HSW

10.1. ld-file handling

10.1.1. DeviceID

The device ID can be found in ld-file within a comment at the beginning of the file. The device ID is read by Gen9ProDB tool.

Gen9ProDB checks if the device ID read from ld-file fits the device ID read directly from RawHexfile within the hexblock struct.

10.1.2. Memory regions

The memory regions are defined in ld-file as well.

10.1.3. Important Links

Linker / Assembler description can be found here:

C:\MTC10Tools\greenhills_rel\[used version]\manuals\build_v800.pdf

10.2. RBLCF (path: <project>/rb/as/core/hwp/hsw/ucbase/functional/rblcf)

The SW-component RBLCF supports the generation of hexblocks with the help of structure types, macros and also default configurations for the simple usecases of hexblock order.

10.2.1. HexFile Configuration

10.2.1.1. General Switch Settings

Configurations for separate bootblock builds:

RBFS_HexBlockConfig_BMGR, RBFS_HexBlockConfig_BLDR, RBFS_HexBlockConfig_RBBLDR.

Configurations for overall hexfiles:

RBFS_HexBlockConfig_FSW,
RBFS_HexBlockConfig_BLDRxFSW,
RBFS_HexBlockConfig_BLDRxFSWxCAL,
RBFS_HexBlockConfig_BMGRxBLDRxFSW,
RBFS_HexBlockConfig_BMGRxRBBLDRxFSW,
RBFS_HexBlockConfig_BMGRxRBBLDRxBLDRxFSW,

RBFS_HexBlockConfig_BMGRxBLDRxFSWxCAL,
RBFS_HexBlockConfig_BMGRxRBBLDRxBLDRxFSWxCAL,
RBFS_HexBlockConfig_FSWxHSM,
RBFS_HexBlockConfig_BMGRxRBBLDRxFSWxHSM,
RBFS_HexBlockConfig_BMGRxRBBLDRxBLDRxFSWxHSM,
RBFS_HexBlockConfig_BMGRxRBBLDRxBLDRxFSWxCALxHSM,
RBFS_HexBlockConfig_FSWxFSW,
RBFS_HexBlockConfig_BLDRxFSWxFSW,
RBFS_HexBlockConfig_BLDRxFSWxFSWxCAL,
RBFS_HexBlockConfig_BMGRxBLDRxFSWxFSW,
RBFS_HexBlockConfig_BMGRxRBBLDRxFSWxFSW,
RBFS_HexBlockConfig_BMGRxRBBLDRxBLDRxFSWxFSW,
RBFS_HexBlockConfig_BMGRxBLDRxFSWxFSWxCAL,
RBFS_HexBlockConfig_BMGRxRBBLDRxBLDRxFSWxFSWxCAL,
RBFS_HexBlockConfig_FSWxFSWxHSM,
RBFS_HexBlockConfig_BMGRxRBBLDRxFSWxFSWxHSM,
RBFS_HexBlockConfig_BMGRxRBBLDRxBLDRxFSWxFSWxHSM,
RBFS_HexBlockConfig_BMGRxRBBLDRxBLDRxFSWxFSWxCALxHSM,
RBFS_HexBlockConfig_Customized.

While the standard configurations automatically provide the parts they need, in Customized configuration the HexFileConfig –which is now flexible– has to be created manually in the project specific RBLCF_Customized_Hexblocks_Allocation.c (with the help of macros).

10.2.1.2. Additional Switch Settings

Size of different HexBlocks:

RBFS_BootloaderSize_0KB, (for OEM-bootloader)
RBFS_BootloaderSize_32KB,
RBFS_BootloaderSize_40KB,
RBFS_BootloaderSize_48KB,
RBFS_BootloaderSize_56KB,
RBFS_BootloaderSize_64KB,
RBFS_BootloaderSize_96KB,
RBFS_BootloaderSize_128KB,
RBFS_BootloaderSize_160KB,
RBFS_BootloaderSize_176KB,
RBFS_BootloaderSize_192KB,
RBFS_BootloaderSize_Customized.

RBFS_RBBootloaderSize_0KB, (for generic RB-bootloader)
RBFS_RBBootloaderSize_32KB,
RBFS_RBBootloaderSize_64KB,
RBFS_RBBootloaderSize_96KB,
RBFS_RBBootloaderSize_Customized.

RBFS_CalibrationBlockSize_0KB,
RBFS_CalibrationBlockSize_8KB,
RBFS_CalibrationBlockSize_32KB,
RBFS_CalibrationBlockSize_Customized.

Defining Bootblock type presence – defaulted for standard hexblock configurations if not set in CSWPRsettings:

RBFS_RBBootloader_Yes,
RBFS_RBBootloader_No,

RBFS_OEMBootloader_Yes,
RBFS_OEMBootloader_No.

Placement of RBBootloader, OEMBootloader, CALBlock in case of two Flash Banks (like we have in D4 and D5 devices) and one of the standard configurations – defaulted if not set in CSWPRsettings:

RBFS_RBBootloaderAlloc_Bank1,
RBFS_RBBootloaderAlloc_Bank2, (default)

RBFS_OEMBootloaderAlloc_Bank1, (default – for reasons of backwards compatibility)
RBFS_OEMBootloaderAlloc_Bank2,

RBFS_CALBlockAlloc_Bank1, (default as small sectors are located in bank 1)
RBFS_CALBlockAlloc_Bank2.

Presence of EOL feature for CALBlocks:

(for the different associated structures refer to chapter 5.1.3 and following)

RBFS_CalBlockEOLhandling_OFF,
RBFS_CalBlockEOLhandling_ON.

10.2.1.3. HexBlock Structure (path: …/rblcf/api/csw/RBLCF_Hexblocks_Structs.h)

Types

RBLCF_HexBlockStruct_t

Initializer Macros

RBLCF_HEX_BLOCK_STRUCTURE(…), (without extended BlockType)

RBLCF_HEX_BLOCK_STRUCTURE2(…). (including extended BlockType)

10.2.1.4. HexInfo Structure (path: …/rblcf/api/csw/RBLCF_Hexblocks_Structs.h)

Types

RBLCF_StdHexInfoStruct_t,

RBLCF_FSWHexInfoStruct_t,

RBLCF_CALHexInfoStructNoEOL_t,

RBLCF_CALHexInfoStructEOL_t,

RBLCF_CompleteHexInfoStruct_t.

Initializer Macros

RBLCF_STD_HEX_INFO_STRUCT

(Standard Hexinfo e.g. used for Bootblocks, extra FSW-Blocks, HSM-Block),

RBLCF_FSW_HEX_INFO_STRUCT

(Hexinfo for main/first FSWBlock in multi Hexblock builds),

RBLCF_CAL_HEX_INFO_STRUCT_NO_EOL

(Hexinfo for CALBlock NOT supporting EOL feature),

RBLCF_CAL_HEX_INFO_STRUCT_EOL

(Hexinfo for CALBlock supporting EOL feature),

RBLCF_COMPLETE_HEX_INFO_STRUCT

(typically for CSW hexfiles without calblocks, but still supporting parameter calibration).

10.2.1.5. Check Structure (path: …/rblcf/api/csw/RBLCF_Hexblocks_Structs.h)

Types

RBLCF_StdCheckStruct_t, (for standard check structure)

RBLCF_InterfaceCheckStruct_t, (for interface/consistency check)

Extended check structure = standard + m*providedITF + n*consumedITF:

RBLCF_ExtCheckStruct_t,

RBLCF_ProvidedInterface_t,

RBLCF_ConsumedInterface_t.

Initializer Macros (path: …/rblcf/api/csw/RBLCF_Hexblocks_Structs.h)

RBLCF_STD_CHECK_STRUCT

(standard check structure – used whenever HW-CRC32 algorithm applicable and no extra interface checks –like provided/used– needed),

RBLCF_INTERFACE_CHECK_STRUCT_CONSISTENCY_CHECKSUM(…)

(create a consistency interface using HW-CRC32 algorithm for a given address range),

RBLCF_INTERFACE_CHECK_STRUCT_INTERFACE_CHECKSUM1(…)

(create a consistency interface calculating a Hash over the Ascet-Interface),

RBLCF_INTERFACE_CHECK_STRUCT_INTERFACE_CHECKSUM2(…)

(future algorithm – not yet supported by MTC)

Extended check structure = standard + m*providedITF + n*consumedITF:

RBLCF_EXTENDED_CHECK_STRUCTURE_HEADER(m, n),

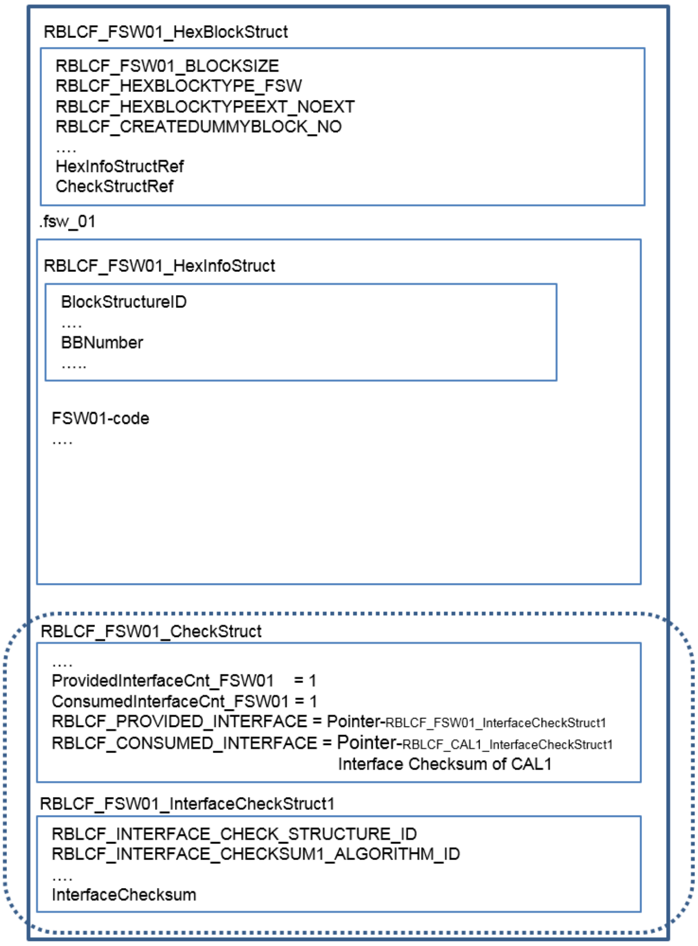
RBLCF_PROVIDED_INTERFACE(…),

RBLCF_CONSUMED_INTERFACE(…).

10.2.1.6. Example picture (allocation of items in example FSW)

Graphical display of EV7135xD4xCustHex mainstream config (CEVT)

FSW01



usually:

RBLCF_FSW01_**HexBlockStruct** in .hexblockst_fsw_01 re-mapped by macro: RBSECTION_START_HEXBLOCKST_FSW_01_RODATA

RBLCF_FSW01_**HexInfoStruct** in .fsw_01 re-mapped by macro: RBSECTION_START_FSW_01_RODATA

RBLCF_FSW01_**CheckStruct** in .checkst_fsw_01 re-mapped by macro: RBSECTION_START_CHECKST_FSW_01_RODATA

RBLCF_FSW01_**InterfaceCheckStruct1** also in .checkst_fsw_01 re-mapped by macro: RBSECTION_START_CHECKST_FSW_01_RODATA

These labels are re-mapped through RBLCF_MemoryRemap.h – to the sections in the project's ld-file.

10.2.2. Memory Remapping macros

For reasons of convenience and safer use the RBLCF component provides Memory Remapping macros in RBLCF_MemoryRemap.h (path: …/rblcf/api/csw/RBLCF_MemoryRemap.h). So memory configuration is possible without using linker symbols directly and also –to a certain extent– without an unintended sda-allocation (**s**mall **d**ata **a**rea feature) of RAM- or ROM-items.

To avoid direct usage of linker symbols in the most common (simple) usecases and to make direct use of #pragma statements not necessary in most cases, a set of macros is provided, which allow data to be placed into specific memory regions.

10.2.2.1. Basic macros

RBSECTION_START_REMAP_DATA(x)

RBSECTION_END_REMAP_DATA

RBSECTION_START_REMAP_BSS(x)

RBSECTION_END_REMAP_BSS

RBSECTION_START_REMAP_RODATA(x)

RBSECTION_END_REMAP_RODATA

RBSECTION_START_REMAP_TEXT(x)

RBSECTION_END_REMAP_TEXT

Where x is a defined linker section (e.g. .checkst_fsw_01).

Note: all the remap macros implicitly avoid sda-allocation because it is assumed that if somebody wants to deliberately place something at a certain location, automatic placement of parts and pieces according to sda-rules is normally unwanted.

10.2.2.2. Specific macros

For the standard memory items specific macros are provided (for readability reasons only the START-macros are listed, while of course all have their corresponding END-macros – for exact linker section names refer to RBLCF_MemoryRemap.h. They are only given here as example for the BMGR case.):

Bootblock

RBSECTION_START_BMGR_RODATA (maps to .bmgr)

RBSECTION_START_CHECKST_BMGR_RODATA (maps to .checkst_bmgr)

```
RBSECTION_START_HEXBLOCKST_BMGR_RODATA   (maps to .hexblockst_bmgr)  etc.
RBSECTION_START_BLDR_RODATA
RBSECTION_START_CHECKST_BLDR_RODATA
RBSECTION_START_HEXBLOCKST_BLDR_RODATA
RBSECTION_START_RBBLDR_RODATA
RBSECTION_START_CHECKST_RBBLDR_RODATA
RBSECTION_START_HEXBLOCKST_RBBLDR_RODATA
```

Back-up RAM Bootblock (RAM not cleared during application reset)

```
RBSECTION_START_BURAM_BB_FSW_INTERFACE_BSS_NOCLEAR
RBSECTION_START_BURAM_BB_BSS_NOCLEAR
```

RAM

In case of Stack/LocalRam Core0

```
RBSECTION_START_STACK_CPU0_NOCLEAR
RBSECTION_START_LRAM0_BSS_CLEAR
RBSECTION_START_LRAM0_BSS_NOCLEAR
RBSECTION_START_LRAM0_DATA
```

In case of Stack/LocalRam Core1

```
RBSECTION_START_STACK_CPU1_NOCLEAR
RBSECTION_START_LRAM1_BSS_CLEAR
RBSECTION_START_LRAM1_BSS_NOCLEAR
RBSECTION_START_LRAM1_DATA
```

In case of GlobalRam (existing in D3, D4 and D5 devices)

```
RBSECTION_START_GRAM_BSS_NOCLEAR
RBSECTION_START_GRAM_BANK_A_BSS_CLEAR
RBSECTION_START_GRAM_BANK_B_BSS_CLEAR
```

```
RBSECTION_START_ASW_RAM_INIT
```

(remapping for initialized ASW C parameters - initialized ASCET parameters are also mapped to this section using the raw GHS pragmas within CGenCorr)

RBSECTION_START_ASW_RAM_NOINIT

(remapping for uninitialized ASW C parameters - uninitialized ASCET parameters are also mapped to this section using the raw GHS pragmas within CGenCorr)

RBSECTION_START_DLM_BSS_NOCLEAR

(remapping of DLM structure (DLM area must be a noclear section, because downloaded reprog driver must be still available after software reset))

RBSECTION_START_GRAM_TEXT

(remapping of text section for special startup-code executed out of RAM)

FSW

RBSECTION_START_FSW_01_RODATA

RBSECTION_START_CHECKST_FSW_01_RODATA

RBSECTION_START_HEXBLOCKST_FSW_01_RODATA

RBSECTION_START_FSW_02_RODATA

RBSECTION_START_CHECKST_FSW_02_RODATA

RBSECTION_START_HEXBLOCKST_FSW_02_RODATA

Back-up RAM FSW

RBSECTION_START_BURAM_FSW_BSS_NOCLEAR

TSI

RBSECTION_START_TSI_RODATA

RBSECTION_START_CHECKST_TSI_RODATA

RBSECTION_START_HEXBLOCKST_TSI_RODATA

Emulation areas (D5ed, D3ed)

RBSECTION_START_XCP_BSS_NOCLEAR

RBSECTION_START_XCPPERF_BSS_NOCLEAR

RBSECTION_START_VARIABLE_RAM_BSS_CLEAR

RBSECTION_START_VARIABLE_RAM_BSS_NOCLEAR

RBSECTION_START_VARIABLE_RAM_DATA

CAL Blocks

RBSECTION_START_CAL1_RODATA

RBSECTION_START_CHECKST_CAL1_RODATA

RBSECTION_START_HEXBLOCKST_CAL1_RODATA

RBSECTION_START_CAL_RODATA (same as CAL1)

RBSECTION_START_CHECKST_CAL_RODATA

RBSECTION_START_HEXBLOCKST_CAL_RODATA

RBSECTION_START_CAL2_RODATA

RBSECTION_START_CHECKST_CAL2_RODATA

RBSECTION_START_HEXBLOCKST_CAL2_RODATA

HSM (Hardware Security Module is realized on Renesas controllers as ICUM - Intelligent Cryptographic Unit Master to provide HW support for security features)

RBSECTION_START_HSM_RODATA

RBSECTION_START_CHECKST_HSM_RODATA

RBSECTION_START_HEXBLOCKST_HSM_RODATA

RBSECTION_START_HSMBOOTCODE_RODATA

Flash Option Bytes (Hardware configuration for the microcontroller)

RBSECTION_START_HCU_CONFIG

RBSECTION_START_HCU_OTP

RBSECTION_START_HCU_LOCKBITS

10.2.3. Manual placement of code/data

The individual location code or data can be managed by specific entries in the ld-file.

10.2.3.1. Putting some code into flash bank B

If some specific code shall be put into flash bank B (if such a flash bank exists in the device), it can be done like in this example:

```
/*******************************************************/
/* Program Flash - Bank B                              */
/*******************************************************/
.hexblockst_fsw_02                       align(4) ABS : > HEXBLOCKST_FSW_02
.fsw_02                                  align(4) ABS : > PFLASH_FSW_02
.ascet_const                             align(4) ABS : > .    /* ASW code */
.text_02                                 align(4) ABS :
{
   // place text sections from all below matching object files into text_02
   "RBLCF_*.o(.text)"
   "RBCLMA_*.o(.text)"
   "rba_*.o(.text)"
} > .
.checkst_fsw_02                               align(4) ABS : > CHECKST_FSW_02
```

10.2.3.2. Moving some data from LRAM to GRAM (e.g. in D3 context)

If some specific data shall be put someplace different from the default, e.g. into GRAM instead of LRAM, if LRAM is the default, this can be done as follows:

```
/*******************************************************/
/* Local RAM CPU0                                      */
/*******************************************************/
.lram0_stack_noclear     NOCLEAR align(4) ABS : > LRAM_CPU0  /* stack at beginning of LRAM due to overflow */
.lram0_data                      align(4) ABS : > .
.lram0_bss_clear         CLEAR   align(4) ABS : > .
.lram0_bss_noclear       NOCLEAR align(4) ABS : > .
.data                            align(4) ABS : > .    /* explicitly initialized data */
.sdata                           align(4) ABS : > .    /* explicitly initialized data in SDA area */
. . . . .

/*******************************************************/
/* Global RAM                                          */
/*******************************************************/
 .gram_bank_a_bss_clear  CLEAR   align(4) ABS : > GRAM_BANK_A    /* zero initialized data for uC safety tests
                                                                    allocated on GRAM BANK A */
 .gram_bank_b_bss_clear  CLEAR   align(4) ABS : > GRAM_BANK_B    /* zero initialized data for uC safety tests
                                                                    allocated on GRAM BANK B */
 .gram_text                      align(4) ABS : > GRAM_BANK_A, GRAM_BANK_B   /* text section for special
                                                             startup-code executed out of RAM */
 .gram_bss_noclear       NOCLEAR align(4) ABS : > .              /* non zero initialized data section */

 .data_from_LRAM align(4) ABS :
 {
  // place .data sections from all below matching object files into .data_from_LRAM which is placed into GRAM
     "RBLCF_*.o(.data)"
     "RBCLMA_*.o(.data)"
     "rba_*.o(.data)"
 } > .

 .sdata_from_LRAM align(4) ABS :
 {
// place .sdata sections from all below matching object files into .sdata_from_LRAM which is placed into GRAM
     "RBLCF_*.o(.sdata)"
     "RBCLMA_*.o(.sdata)"
     "rba_*.o(.sdata)"
 } > .
```

Notes:

- It is important to reallocate the small data areas (e.g. .sdata) along with the regular areas to be independent of compiler/linker settings.
- In case of reallocating parts which have ROM-initialization (like in the example .data/.sdata) the new sections require their corresponding .CROM sections to be allocated specifically in the flash.
  In the above example, this means to add in the flash the following lines:

  ```
  .CROM.data_from_LRAM     CROM(.data_from_LRAM)    align(4) ABS : > .
  .CROM.sdata              CROM(.sdata)             align(4) ABS : > .
  ```

- In this example the parts were arbitrarily chosen, i.e. the wildcards used do not necessarily provide a likely usecase. Also the fact that .bss/.sbss was not reallocated while .data/.sdata was, has been chosen only to see a difference between them in the mapfile, not because it makes so much sense as a usecase.

10.2.4. Checking of Hexblocks

Certain applications (e.g. ROM-checking, Diagnosis) have to go over the different Hexblocks in order to check them or to find a specific Hexblock to extract some data. For this purpose the RBLCF-SW-component offers a generic **iterator.**

10.2.4.1. Iterator interfaces

The following interfaces are provided:

- `void RBLCF_GetFirstHexBlock(RBLCF_HexBlockLayout_t* nHexBlock);`
- `void RBLCF_GetNextHexBlock(RBLCF_HexBlockLayout_t* nHexBlock);`

which pass a pointer to the first/next Hexblock to the caller who can interpret it using:

```
typedef struct
{
  uint32 HexBlockStartAdr;                    // start address of current HexBlock
  const RBLCF_HexBlockStruct_t* HexBlockRef;  // pointer to HexBlock structure
  const RBLCF_StdHexInfoStruct_t* HexInfoRef; // pointer to HexInfo structure
  const RBLCF_StdCheckStruct_t* CheckRef;     // pointer to Check structure
  boolean hasNext;                            // successor of this HexBlock available

} RBLCF_HexBlockLayout_t;
```

and the pointers to the further structures (Hexblock-, Hexinfo-, Check-structures). In case there is no next hexblock, the Boolean value of hasNext is FALSE. When calling RBLCF_GetNextHexBlock the caller has to pass the pointer to the current Hexblock as parameter and when the function returns it is filled with the pointer to the next Hexblock. Calling RBLCF_GetNextHexBlock when hasNext is FALSE leads to an assert in development environment and the pointer will then remain on the current Hexblock.

10.2.4.2. Iterator examples

Iterating over all Hexblocks can be done in the following manner:

```
RBLCF_HexBlockLayout_t myHexBlock;
...
RBLCF_GetFirstHexBlock(&myHexBlock);
...
while (myHexBlock.hasNext)
{
  RBLCF_GetNextHexBlock(&myHexBlock);
  ...
}
```

The following example illustrates how to find the first FSW-Hexblock and then stop:

```
RBLCF_HexBlockLayout_t myHexBlock;

RBLCF_GetFirstHexBlock(&myHexBlock);

while (myHexBlock.hasNext) {
    if((myHexBlock.HexInfoRef->BlockStructureID == RBLCF_HEXINFOTYPE_FSW)
     ||(myHexBlock.HexInfoRef->BlockStructureID == RBLCF_HEXINFOTYPE_COMPLETE)){
        //we have found an FSW or Complete block - we take the data from that block
        break;
    }
    RBLCF_GetNextHexBlock(&myHexBlock);
}
...
```