Confidential

2nd Generation AURIX™ TC3xx Hardware Security Module

HSM Version 2.0

HSM Subsystem

Hardware Security Module

Target Specification

Revision 2.0

2016-03-04

Chip Card & Security

2nd Generation AURIX™ TC3xx
Hardware Security Module
Confidential
Revision History: 2016-03-04, Revision 2.0

Page	Subjects
	PKC: Commands **EXP**, **SMULT25519**, **XRECOVER**, **SMULTED25519**, **CHECKVALID** added.

Previous Revision: 2015-09-28, Rev. 1.4

Page	Subjects (major changes compared to AURIX™ V1.0)
	Local RAM increased to 96 KB
	Timer module exchanged.
	Watchdog timer module added.
	Address map updated
	AES: Correction of GCM example.
	Hash module added.
	PKC module added.
	Bridge Module: **ERRCTRL**: Error Bits added.
	Bridge Module: **ERRIE**: Additional Interrupt Enable Bits added
	Bridge Module: DTSTSEL: Register and DTS trigger removed.
	Firmware Architecture: Procedure for User OS Code Check revised
	HAR will be ignored if debug mode is not enabled (no exception will be triggered).

Table of Contents

Table of Contents **3**

List of Tables **9**

List of Figures **10**

Preface **11**

1 Introduction **12**

2 Core System **16**

2.1 Address Map 16

2.2 CPU 20

2.2.1 Multiprocessor Communication 20

2.2.2 The SysTick Timer 20

2.2.3 Power Management 20

2.2.4 Byte Order 20

2.2.5 Application Interrupt and Reset Control Register 20

2.2.6 Code Prefetch 20

2.2.7 Exclusive Access Instructions 21

2.2.8 Instruction Timing 21

2.3 Interrupt Unit 21

2.3.1 Interrupt mapping 21

2.3.2 Expansion of VTOR 22

2.4 Memory Protection Unit 22

2.5 Cache 23

2.5.1 Overview 23

2.5.2 Functional Description 25

2.5.2.1 Parallel Cache Accesses 26

2.5.3 Cache SFRs 26

2.5.3.1 Configuration SFRs 27

2.5.3.2 Command SFRs 29

3 True Random Number Generator **35**

3.1 Concept Overview 35

3.2 Performance 36

3.3 Functional Overview 37

3.3.1 Sleep Mode / Disable 37

3.3.2 Interrupt Behavior 37

3.4 TRNG SFRs 38

3.4.1 Status and Data Registers 39

3.4.2 Control Registers 42

4 AES 128 Encryption / Decryption Device **43**

4.1 Introduction 43

4.2 Algorithms 43

4.2.1 Plain AES 44

4.2.2 ECB Mode 45

4.2.3 CBC Mode 46

4.2.4 CTR Mode 47

4.2.5 OFB Mode ... 49
4.2.6 CFB Mode ... 51
4.2.7 GCM Mode ... 53
4.2.8 XTS Mode ... 55
4.3 Register Architecture ... 58
4.4 AES SFRs ... 60
4.4.1 Control and Status Registers ... 61
4.4.2 Input Registers ... 66
4.4.3 Output Registers ... 70
4.5 Pseudo Coding Example with Test Data ... 78

5 Hash Module ... 104
5.1 Features ... 104
5.2 Functional Description ... 105
5.2.1 Data Transfers and FIFO Flag ... 105
5.2.2 Output Data Size ... 105
5.2.3 User-Defined Initialization Vector (Multiapplication Support) ... 106
5.2.4 SHA-224 Hash Support ... 107
5.2.5 Endianness ... 107
5.3 Hash Registers ... 111
5.3.1 Configuration SFR ... 112
5.3.2 Status and Control ... 114
5.3.3 Data SFRs ... 117
5.4 Programming Model ... 119
5.4.1 Input Data Size and Padding ... 119
5.4.2 Calculating an MD-5 Hash ... 119
5.4.3 Calculating a SHA-1 or SHA-256 Hash ... 119
5.4.4 Calculating a SHA-224 Hash ... 120
5.4.5 Calculating Hashes in Different Applications ... 120
5.5 Debug Support ... 121

6 PKC Module ... 122
6.1 Introduction ... 122
6.2 Algorithms ... 123
6.2.1 Arithmetic Operations over Z, Z/n, and F_p. ... 124
6.2.1.1 Addition in Z ... 124
6.2.1.2 Subtraction in Z ... 125
6.2.1.3 Multiplication in Z ... 125
6.2.1.4 Modular Reduction in Z ... 125
6.2.1.5 Modular Addition in Z, Addition in Z/n ... 125
6.2.1.6 Modular Subtraction in Z, Subtraction in Z/n ... 125
6.2.1.7 Modular Multiplication in Z, Multiplication in Z/n ... 126
6.2.1.8 Modular Inversion over Z ... 126
6.2.1.9 Modular Division over Z ... 126
6.2.1.10 Modular Exponentiation ... 126
6.2.2 Elliptic Curve Operations over F_p. ... 127
6.2.2.1 Point Doubling on an Elliptic Curve over F_p ... 128
6.2.2.2 Point Addition on an Elliptic Curve over F_p ... 128
6.2.2.3 Scalar Multiplication on an Elliptic Curve over F_p ... 128

6.2.2.4 ECDSA Signature Generation on an Elliptic Curve over F_p . . . 128
6.2.2.5 ECDSA Signature Verification on an Elliptic Curve over F_p . . . 129
6.2.3 Elliptic Curve Operations on Curve25519. . . . 130
6.2.3.1 Point Doubling on Curve25519 . . . 131
6.2.3.2 Point Addition on Curve25519 . . . 131
6.2.3.3 Scalar Multiplication on Curve25519 . . . 132
6.2.4 Elliptic Curve Operations on the Twisted Edwards Curve Ed25519. . . . 132
6.2.4.1 Point Addition on Ed25519 . . . 133
6.2.4.2 Scalar Multiplication on Ed25519 . . . 133
6.2.4.3 X Coordinate Recovery on Ed25519 . . . 134
6.2.5 Arithmetic Operations over $F_2[X]$ and $F_2[X]/(f)=GF(2^d)$. . . . 134
6.2.5.1 Addition in $F_2[X]$ . . . 134
6.2.5.2 Subtraction in $F_2[X]$ . . . 134
6.2.5.3 Multiplication in $F_2[X]$ . . . 134
6.2.5.4 Modular Reduction in $F_2[X]$ . . . 135
6.2.5.5 Modular Addition in $F_2[X]$, Addition in $F_2[X]/(f)$ . . . 135
6.2.5.6 Modular Subtraction in $F_2[X]$, Subtraction in $F_2[X]/(f)$ . . . 135
6.2.5.7 Modular Multiplication in $F_2[X]$, Multiplication in $F_2[X]/(f)$ . . . 136
6.2.5.8 Modular Inversion over $F_2[X]$ . . . 136
6.2.5.9 Modular Division over $F_2[X]$ . . . 136
6.2.6 Elliptic Curve Operations over $F_2[X]/(f)=GF(2^d)$. . . . 136
6.2.6.1 Point Doubling on an Elliptic Curve over $F_2[X]/(f)=GF(2^d)$ . . . 137
6.2.6.2 Point Addition on an Elliptic Curve over $F_2[X]/(f)=GF(2^d)$ . . . 138
6.2.6.3 Scalar Multiplication on an Elliptic Curve over $F_2[X]/(f)=GF(2^d)$ . . . 138
6.2.6.4 ECDSA Signature Generation on an Elliptic Curve over $F_2[X]/(f)=GF(2^d)$ . . . 138
6.2.6.5 ECDSA Signature Verification on an Elliptic Curve over $F_2[X]/(f)=GF(2^d)$ . . . 139
6.3 Register Architecture and Programming Model . . . 140
6.3.1 Interrupts . . . 140
6.4 PKC SFRs . . . 142
6.4.1 Control and Status Registers . . . 144
6.4.2 Shared Crypto Memory . . . 152
6.5 Further detailed definition/specification of commands . . . 156
6.5.1 Arithmetic Operations. . . . 156
6.5.1.1 Modular Addition Command **ADDN** . . . 156
6.5.1.2 Modular Subtraction Command **SUBN** . . . 157
6.5.1.3 Modular Multiplication Command **MULTN** . . . 158
6.5.1.4 Modular Reduction Command **RED** . . . 159
6.5.1.5 Modular Division Command **DIVN** . . . 159
6.5.1.6 Modular Inversion Command **INV** . . . 160
6.5.1.7 Multiplication Command **MULT** . . . 160
6.5.1.8 Modular Inversion Command **INV2** . . . 161
6.5.1.9 Modular Reduction Command **RED2** . . . 162
6.5.1.10 Modular Exponentiation Command **EXP** . . . 162
6.5.2 Elliptic Curve Operations. . . . 163
6.5.2.1 Point Doubling Command **PDBL** . . . 164
6.5.2.2 Point Addition Command **PADD** . . . 165
6.5.2.3 Scalar Multiplication Command **SMULT** . . . 166

6.5.2.4 Parameter Check **CHECKAB** ... 167
6.5.2.5 Parameter Check **CHECKN** ... 168
6.5.2.6 Parameter Check **CHECKPXY** ... 168
6.5.2.7 Scalar Multiplication Command **SMULT25519** ... 169
6.5.2.8 X-Coordinate Recovery **XRECOVER** ... 170
6.5.2.9 Scalar Multiplication Command **SMULTED25519** ... 170
6.5.2.10 Signature Verification **CHECKVALID** ... 171
6.5.2.11 ECDSA Signature generation **ECDSASIG** ... 173
6.5.2.12 ECDSA Signature Verification **ECDSAVER** ... 175
6.6 Pseudo Coding Example with Test Data ... 180

7 Timer Module ... 193
7.1 Introduction ... 193
7.2 Overview ... 193
7.3 Functional Description ... 193
7.3.1 Timer Clock Selection ... 194
7.3.1.1 Timer Clocks in Sleep and Halt Mode ... 194
7.3.2 Prescaler ... 194
7.3.3 Timer Reload ... 195
7.3.4 Timer Overflow and Interrupts ... 195
7.4 Timer Register Description ... 196
7.4.1 Timer Registers ... 197
7.5 Programming Model ... 202
7.5.1 Timer Resolution and Period ... 202
7.5.2 Polling vs. Interrupt Handling ... 202
7.5.3 Single Shot Event Triggered by Software ... 203
7.5.4 Periodic Multi-shot Events ... 203
7.5.5 Software Triggered Watchdog ... 204
7.5.6 Cascading of Timers ... 204

8 Watchdog Timer ... 205
8.1 Introduction ... 205
8.2 Overview ... 205
8.3 Functional Description ... 205
8.3.1 Clock Selection ... 206
8.3.1.1 Timer Clocks in Sleep and Halt Mode ... 206
8.3.2 Watchdog Timer Operation ... 206
8.3.3 Watchdog Service ... 206
8.3.4 Service Counter Period ... 207
8.4 Watchdog Timer Register Description ... 208
8.4.1 Watchdog Timer Registers ... 209
8.5 Programming Model ... 215
8.5.1 Watchdog Timer Resolution and Period ... 215
8.5.2 Watchdog Operation ... 215
8.5.3 Checkpoint Operation ... 216

9 Bridge Module ... 217
9.1 Introduction ... 217
9.2 Functional Description ... 219

9.2.1 Bridging Functionality . . . 219
9.2.2 Communication Unit . . . 219
9.2.3 Clock Control . . . 220
9.2.4 Error and Event Handling . . . 220
9.2.5 Debug Access . . . 221
9.2.6 External Interrupts . . . 222
9.2.7 HSM Reset . . . 223
9.2.8 Control of Pin Outputs . . . 223
9.2.9 Sensor Interrupts and Chip Resets . . . 224
9.3 Bridge Registers . . . 226
9.3.1 Communication Unit . . . 228
9.3.2 Clock Divider . . . 236
9.3.3 System Control . . . 237
9.3.4 Error Unit . . . 239
9.3.5 External Interrupts . . . 245
9.3.6 Single Access to Host Memories . . . 247
9.3.7 Sensor Interrupts and Reset . . . 249
9.4 Bridge Registers (Host side access only) . . . 255
9.4.1 HSM Reset . . . 256
9.4.2 Debug Registers . . . 257

10 Memories . . . 260
10.1 ROM . . . 260
10.1.1 Boot ROM . . . 260
10.1.2 PKC µ-Code ROM . . . 260
10.2 RAM . . . 260
10.2.1 Local RAM . . . 260
10.2.2 PKC RAMs . . . 261

11 Firmware Architecture . . . 262
11.1 HSM Firmware Overview . . . 262
11.2 Boot System . . . 263
11.2.1 Introduction . . . 263
11.2.2 Boot Initialization . . . 264
11.2.3 User Mode . . . 268
11.2.3.1 User-OS Code Check . . . 271
11.2.3.2 RAM Memory Layout in User Mode . . . 274
11.2.4 Test Mode . . . 274
11.3 Information Exchange between the Host and BOS . . . 275
11.4 Information Exchange between Host SSW and HSM User Code . . . 277

12 System Debug . . . 278
12.1 Debug Modules . . . 280
12.1.1 Flash Patch & Breakpoint unit (FPB) . . . 280
12.1.2 Data Watchpoint and Trace Unit (DWT) . . . 280
12.1.3 ROM Table . . . 280
12.2 OCDS Interface . . . 281
12.3 Debug System . . . 282
12.3.1 Reset Handling . . . 282

12.3.2 Clock and Sleep Mode Handling 282
12.3.3 Debug Access / Authentication 282
12.3.4 Halt-After-Reset Implementation 282
12.3.5 Peripheral Behavior in HALT Mode 283
12.3.6 SPB Bus Conflict Triggered by HSM Debugging 283
12.4 Debug Modules Register Extension 284
12.4.1 Flash Patch Registers 285
12.4.2 Debug Halting Control and Status Register 287
12.5 SW Tool Extensions 288
References 289
Terminology 290

List of Tables

Table 2-1 Memory Map ... 16
Table 2-2 Interrupt Mapping ... 21
Table 2-3 Cache Configuration ... 23
Table 2-4 Register Address Space ... 26
Table 2-5 Register Overview ... 26
Table 3-1 Generation Time for One Random Byte ... 36
Table 3-2 Register Address Space ... 38
Table 3-3 Register Overview ... 38
Table 4-1 Register Address Space ... 60
Table 4-2 Register Overview ... 60
Table 5-1 Hash Algorithms and Performance ... 104
Table 5-2 Register Address Space ... 111
Table 5-3 Register Overview ... 111
Table 6-1 Register Address Space ... 142
Table 6-2 Register Overview ... 142
Table 6-3 Shared Crypto Memory Registers 1 to 31 ... 154
Table 6-4 Overview over Arithmetic Operations ... 156
Table 6-5 Overview over Elliptic Curve Operations ... 163
Table 7-1 Clock Sources for Timer 0 ... 194
Table 7-2 Clock Sources for Timer 1 ... 194
Table 7-3 Register Address Space ... 196
Table 7-4 Register Overview ... 196
Table 7-5 Prescaler Adjustment ... 199
Table 8-1 Clock Sources for Watchdog Timer Module ... 206
Table 8-2 Register Address Space ... 208
Table 8-3 Register Overview ... 208
Table 8-4 Watchdog Checkpoint Operations ... 215
Table 9-1 External Interrupt mapping ... 222
Table 9-2 Sensor mapping ... 224
Table 9-3 Register Address Space ... 226
Table 9-4 Register Overview ... 226
Table 9-5 Register Access Types ... 227
Table 9-6 Register Address Space ... 255
Table 9-7 Register Overview ... 255
Table 9-8 Register Access Types ... 255
Table 11-1 Configuration of Interrupts and Exception Handling ... 265
Table 11-2 Boot Sector Selection ... 271
Table 11-3 Information Exchange Format from the Host to the BOS (via HT2HSMS) ... 275
Table 11-4 Information Exchange Format from the BOS to the Host (via HSM2HTS) ... 276
Table 12-1 Register Address Space ... 284
Table 12-2 Register Overview ... 284
Table 12-3 Flash Patch Comparator High Registers ... 285
Table 12-4 Peripheral IDentification Registers ... 286

List of Figures

Figure 1-1 Architecture Overview . . . 12
Figure 2-1 Cache Overview . . . 24
Figure 2-2 Cache Principle . . . 25
Figure 3-1 Principle of TRNG operation . . . 35
Figure 4-1 AES . . . 45
Figure 4-2 ECB mode . . . 46
Figure 4-3 CBC mode . . . 47
Figure 4-4 CTR mode . . . 49
Figure 4-5 OFB mode . . . 51
Figure 4-6 CFB mode . . . 52
Figure 4-7 GCM mode . . . 55
Figure 4-8 XTS mode . . . 57
Figure 5-1 Word Order of Hash Digest Output . . . 106
Figure 5-2 Hash Data Format Big Endian System (BEND_IN = 0) . . . 109
Figure 5-3 Hash Data Format Little Endian System (BEND_IN = 1) . . . 110
Figure 7-1 Timer Overview . . . 193
Figure 8-1 Watchdog timer overview . . . 205
Figure 9-1 HSM bridge overview . . . 218
Figure 11-1 HSM-embedded software overview . . . 262
Figure 11-2 BOS startup overview . . . 264
Figure 11-3 Boot initialization . . . 267
Figure 11-4 BOS Usermode flow (part 1) . . . 269
Figure 11-5 BOS Usermode flow (part 2) . . . 270
Figure 11-6 BOS User-OS code check . . . 273
Figure 11-7 RAM Memory Layout (User Mode) . . . 274
Figure 12-1 HSM debug block diagram . . . 279

Preface

This preface introduces the HSM Target Specification.

Scope of document

This Target Specification is intended to describe the architecture and functionality of the Hardware Security Module (HSM).

Using this book

This book is organized into the following chapters:

- **Chapter 1 Introduction**
- **Chapter 2 Core System**
- **Chapter 3 True Random Number Generator**
- **Chapter 4 AES 128 Encryption / Decryption Device**
- **Chapter 5 Hash Module**
- **Chapter 6 PKC Module**
- **Chapter 7 Timer Module**
- **Chapter 8 Watchdog Timer**
- **Chapter 9 Bridge Module**
- **Chapter 10 Memories**
- **Chapter 11 Firmware Architecture**
- **Chapter 12 System Debug**
- **References**
- **Terminology**

1 Introduction

The Hardware Security Module (HSM) is an optional peripheral module of the AURIX™ Controller family. Its main applications are:

- Secure boot
- Tuning protection (e.g. integrity of calibration data)
- Secure sensor communication (authentication and integrity of sensor data)
- Authentication
- Secure flash load
- Immobilizer (theft protection)
- Secure log and
- Secure debug authentication

The HSM contains all necessary elements to allow software to implement the Secure Hardware Extension (SHE) as defined by HIS, see [1].

Figure 1-1 below shows a general overview of the HSM subsystem and its integration into the host system.

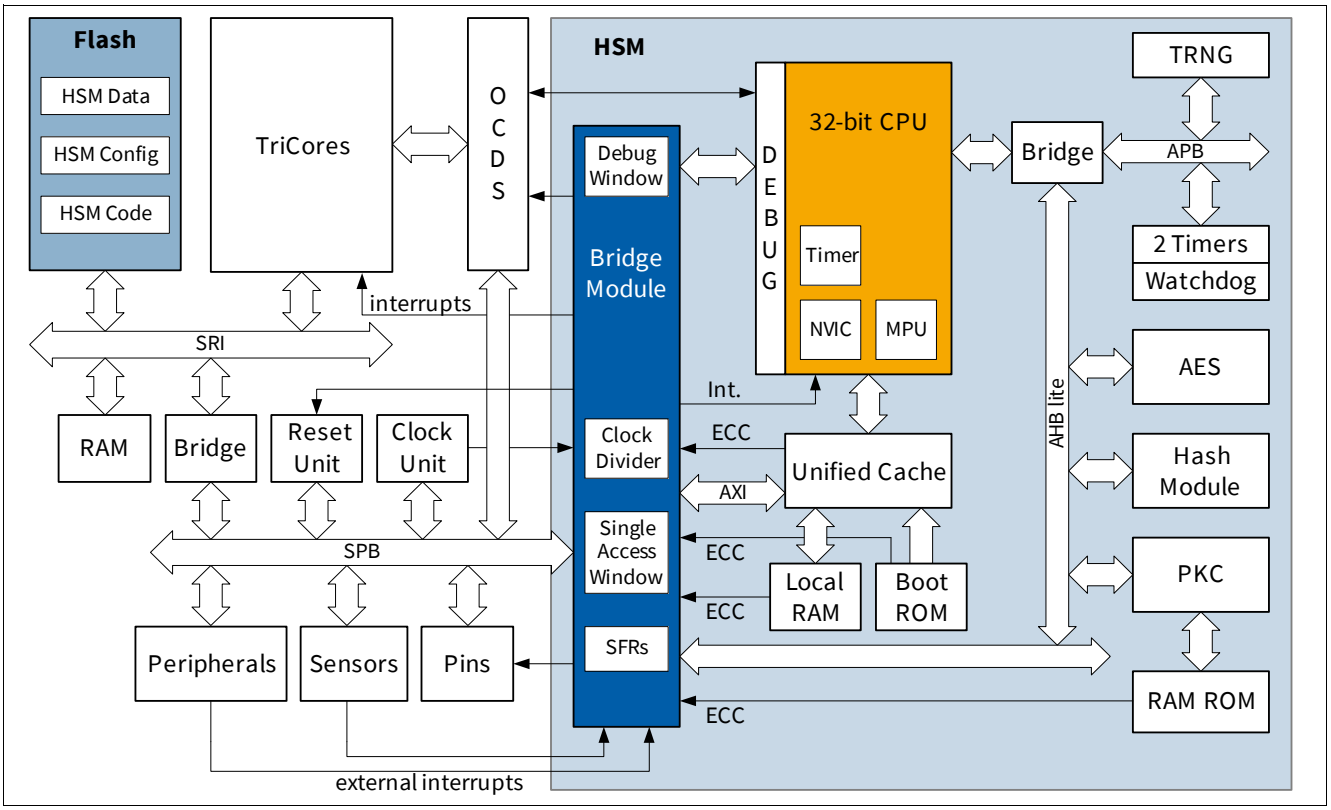


Figure 1-1 Architecture Overview

The HSM module comprises the following functional submodules:

- CPU based on ARM™ Cortex™ M3 including
  - Debug support
  - 24-bit SysTick Timer
  - Nested Vector Interrupt Controller
  - Memory Protection Unit (MPU)
- 4 KB unified data and instruction cache
- Local Boot ROM
- 96 KB RAM
- AES module with local key and context storage
- Hash module executing either the MD-5, SHA-1 or SHA-224/SHA-256 function
- Public Key Crypto supporting fast signature generation and verification based on Elliptic Curve Cryptography
- True Random Number Generator (TRNG)
- Timer module with two 16-bit timers
- Watchdog timer with checkpoint functionality
- HSM bridge including
  - Clock divider unit
  - Debug access
  - Special Function Registers (SFRs) for communication with the host system

Features

- CPU based on ARM Cortex-M3.
  - 32-bit CPU with Thumb-2 instruction set.
  - Privilege/user mode support.
  - Illegal opcode detection.
  - Internal 24-bit timer.
- HSM debug features:
  - Compatible to ARM.
  - 6 hardware breakpoints, no flash patch support.
  - 4 triggers for watchpoints.
  - Full visibility of all software visible HSM registers and memories (excluding e.g. AES keys).
  - HSM controls debug access to the system and separately to the HSM itself.
- 4 KB unified cache (4-way set associative).
- Internal memory protection unit (MPU).
  - ARM compatible.
  - 8 address regions with sub-regions.
  - Used for code execution and data accesses.
  - Granularity 32 Bytes to 4 GB.
  - Regions aligned to size.

  - Access permissions: no access, read-only, executable, read/write.
- 96 KB local RAM.
- Local boot ROM.
- True random number generator (TRNG).
- Timer unit with two 16-bit timers.
- Watchdog timer with checkpoint functionality triggering an NMI upon expiry or checkpoint mismatch.
- High-performance AES-128 unit with private key storage:
  - Local storage for 8 128-bit keys, 2 of them are non-alterable and “use-only”.
  - Local storage of 5 contexts (including 1 for pseudo-random number generation).
  - Modes of operation: ECB, CBC, CTR, OFB, GCM, XTS.
- Hash module for signature generation, verification and generic data integrity checks. It supports the following hashing algorithms:
  - MD-5.
  - SHA-1.
  - SHA-256.
  - SHA-224 (derived using SHA-256 with some software support).
- Public Key Crypto Engine (PKC) that supports fast signature generation and verification based on Elliptic Curve Digital Signature Algorithms (ECDSA). It enables complex algorithms on elliptic curves of bitlength up to 256, like:
  - Operations with up to 256-bit operands.
  - Supports elliptic curves over fields F(p) and F(2^m).
  - Primitive arithmetic operations over F(p) and F(2^m): modular addition, modular subtraction, modular multiplication, modular reduction, modular division, modular inversion, multiplication.
  - Primitive ECC and Check Point operations over F(p) and F(2^m): point doubling, point addition, point multiplication (with projective coordinates), Check_AB, and Check_n.
  - Addition of two points in affine coordinates.
  - Doubling of a point in affine coordinates.
  - Scalar multiplication.
  - ECDSA signature generation
  - ECDSA signature verification.
- Bridge module:
  - “Firewall” functionality: HSM internals are protected from accesses by other masters.
  - Bus master on the SPB. Accesses possible to complete system memory map.
  - Communication SFRs for exchanging data.
  - SFRs for triggering an interrupt in the HSM CPU.
  - 32 HSM external interrupt inputs.
  - 2 interrupt signals from HSM to the system interrupt controller.
  - Inputs for up to 10 sensors.
  - Control of 2 pins.
  - Option for triggering application and system reset of the chip.

- HSM code and constant data are stored in system PFlash. In the UCB it can be configured which sectors are searched for a valid boot code.
- HSM EEPROM data is stored in a dedicated DFlash bank (not in all products).
- HSM “use-only” keys and the unique chip identifier are supplied in UCBs.
- HSM supports low-power mode (internal clock reduced to SPB clock / 2 and SPB clock / 4), and sleep mode.

Each submodule is described in more detail in the following chapters.

In combination with security features implemented in the host system – mainly in the flash memories, PMU (Program Memory Unit) and debug logic – and adequate software the HSM offers protection against logical attacks. For this purpose it relies on the following features implemented in the host system:

- Adequate protection and locking of flash memory areas that are reserved for the HSM (both code and data).
- One-Time-Programmable (OTP) pages in the config area of the flash.
- A memory region in the config area that is reserved for the HSM.
- Locking of this region in the config area against modification (OTP).
- HSM region in the config area can be switched off after the boot process – i.e. the read-only AES keys and other configuration data is no longer accessible in the flash.
- The test must not allow observation of data/information used during normal HSM operation.
- The test must not allow infiltration of data/information into normal HSM operation.
- Debugging of the HSM must not be possible if not enabled by HSM. This includes debugging of the HSM reserved memory regions and tracing of HSM bus transfers on the host system.
- Configuration of MBISTs in HSM module after reset, start of HSM after MBIST configuration.
- Adequate test mode protection.

The HSM contains no hardware countermeasures against physical attacks, neither non-invasive, semi-invasive nor invasive attacks. Countermeasures against side-channel attacks and failure attacks have to be implemented by software.

2 Core System

This section covers the core system. The core system contains:

- A 32-bit CPU based on ARM™ Cortex™ M3 including
  - Nested Vector Interrupt Controller (NVIC)
  - SysTick Timer
- Memory Protection Unit (MPU)
- Unified cache

The debug modules are defined in **Chapter 12**.

2.1 Address Map

The following table defines the memory map of the HSM (for a detailed description of the host memory map see the AURIX™ TC3xx Target Specification):

Table 2-1 Memory Map

Seg.	Address Range	Size	Attributes	System	Description
0	$0000\ 0000_H$ - $0000\ 0FFF_H$	4 KB	B,C,MPU	HSM	Boot ROM
	$0000\ 1000_H$ - $0FFF\ FFFF_H$		B,C,MPU	Host	Reserved
1	$1000\ 0000_H$ - $1FFF\ FFFF_H$		B,C,MPU	Host	SRAMs CPU5
2	$2000\ 0000_H$ - $2001\ 7FFF_H$	96 KB	B,C,MPU	HSM	Local RAM
	$2001\ 8000_H$ - $200F\ FFFF_H$		B,C,MPU	Host	Reserved
	$2010\ 0000_H$ - $21FF\ FFFF_H$		B,C,MPU	Host	Reserved
	$2200\ 0000_H$ - $23FF\ FFFF_H$	32 MB	B,C,MPU	HSM	Bit banding alias region. Addresses are mapped to $2000\ 0000_H$ - $200F\ FFFF_H$
	$2400\ 0000_H$ - $2FFF\ FFFF_H$		B,C,MPU	Host	Reserved
3	$3000\ 0000_H$ - $3FFF\ FFFF_H$		B,C,MPU	Host	SRAMs CPU4
4	$4000\ 0000_H$ - $41FF\ FFFF_H$		B,C,MPU	Host	SRAMs CPU3
	$4200\ 0000_H$ - $43FF\ FFFF_H$	32 MB	B,C,MPU	Host	Bit banding alias region. Addresses are mapped to $4000\ 0000_H$ - $400F\ FFFF_H$
	$4400\ 0000_H$ - $4FFF\ FFFF_H$		B,C,MPU	Host	Reserved
5	$5000\ 0000_H$ - $5FFF\ FFFF_H$		B,C,MPU	Host	SRAMs CPU2
6	$6000\ 0000_H$ - $6FFF\ FFFF_H$		B,C,MPU	Host	SRAMs CPU1
7	$7000\ 0000_H$ - $7FFF\ FFFF_H$		B,C,MPU	Host	SRAMs CPU0
8	$8000\ 0000_H$ - $802F\ FFFF_H$	3 MB	B,C,MPU	Host	Program Flash 0
	$8030\ 0000_H$ - $805F\ FFFF_H$	3 MB	B,C,MPU	Host	Program Flash 1
	$8060\ 0000_H$ - $808F\ FFFF_H$	3 MB	B,C,MPU	Host	Program Flash 2
	$8090\ 0000_H$ - $80BF\ FFFF_H$	3 MB	B,C,MPU	Host	Program Flash 3
	$80C0\ 0000_H$ - $80EF\ FFFF_H$	3 MB	B,C,MPU	Host	Program Flash 4
	$80F0\ 0000_H$ - $80FF\ FFFF_H$	1 MB	B,C,MPU	Host	Program Flash 5
	$8100\ 0000_H$ - $8FFF\ FFFF_H$		B,C,MPU	Host	Reserved for burst access memories

Table 2-1 Memory Map (cont'd)

Seg.	Address Range	Size	Attributes	System	Description
9	9000 0000$_H$ - 9FFF FFFF$_H$		B,C,MPU	Host	Reserved for burst access memories
A	A000 0000$_H$ - A02F FFFF$_H$	3 MB	XN,C,MPU	Host	Program Flash 0
	A030 0000$_H$ - A05F FFFF$_H$	3 MB	XN,C,MPU	Host	Program Flash 1
	A060 0000$_H$ - A08F FFFF$_H$	3 MB	XN,C,MPU	Host	Program Flash 2
	A090 0000$_H$ - A0BF FFFF$_H$	3 MB	XN,C,MPU	Host	Program Flash 3
	A0C0 0000$_H$ - A0EF FFFF$_H$	3 MB	XN,C,MPU	Host	Program Flash 4
	A0F0 0000$_H$ - A0FF FFFF$_H$	1 MB	XN,C,MPU	Host	Program Flash 5
	A100 0000$_H$ - AF3F FFFF$_H$		XN,C,MPU	Host	Reserved for non-burst access memories
	AF40 0000$_H$ - AF40 0DFF$_H$		XN,C,MPU	Host	UCB00 - UCB06
	AF40 0E00$_H$ - AF40 0FFF$_H$	512 B	XN,C,MPU	Host	UCB07 - HSM Config Area
	AF40 1000$_H$ - AF40 5FFF$_H$		XN,C,MPU	Host	UCB08 - UCB47
	AF40 6000$_H$ - AFBF FFFF$_H$		XN,C,MPU	Host	Reserved for non-burst access memories
	AFC0 0000$_H$ - AFC1 FFFF$_H$	128 KB	XN,C,MPU	Host	HSM Data Flash 1
	AFC2 0000$_H$ - AFFF FFFF$_H$		XN,C,MPU	Host	Reserved for non-burst access memories
B	B000 0000$_H$ - BFFF FFFF$_H$		XN,C,MPU	Host	Reserved for non-burst access memories
C	C000 0000$_H$ - CFFF FFFF$_H$		XN,B,C,MPU	Host	Reserved for non-burst access memories
D	D000 0000$_H$ - DFFF FFFF$_H$		XN,B,C,MPU	Host	Reserved for non-burst access memories

Table 2-1 Memory Map (cont'd)

Seg.	Address Range	Size	Attributes	System	Description
E	E000 0000$_H$ - E000 0FFF$_H$		XN,PRIV	HSM	Reserved for Instrumentation Trace Module (ITM)
	E000 1000$_H$ - E000 1FFF$_H$	4 KB	XN,PRIV	HSM	Data Watchpoint and Trace (DWT)
	E000 2000$_H$ - E000 2FFF$_H$	4 KB	XN,PRIV	HSM	Flash Breakpoint and Patch (FPB)
	E000 3000$_H$ - E000 DFFF$_H$		XN,PRIV	HSM	Reserved for private peripherals
	E000 E000$_H$ - E000 EFFF$_H$	4 KB	XN,PRIV	HSM	System Control Space
	E000 F000$_H$ - E008 9FFF$_H$		XN,PRIV	HSM	Reserved for private peripherals
	E008 A000$_H$ - E008 BFFF$_H$	8 KB	XN,PRIV	HSM	Cache Control
	E008 C000$_H$ - E00F EFFF$_H$		XN,PRIV	HSM	Reserved for private peripherals
	E00F F000$_H$ - E00F FFFF$_H$	4 KB	XN,PRIV	HSM	ROM table
	E010 0000$_H$ - E012 BFFF$_H$		XN,MPU	HSM	Reserved for AHB peripherals
	E012 C000$_H$ - E012 C7FF$_H$		XN,MPU	HSM	Reserved
	E012 C800$_H$ - E7FF FFFF$_H$		XN,MPU	HSM	Reserved for AHB peripherals
	E800 0000$_H$ - E800 03FF$_H$	1 KB	XN,MPU	HSM	AES module
	E800 0400$_H$ - E800 07FF$_H$	1 KB	XN,MPU	HSM	Hash module
	E800 0800$_H$ - E800 3BFF$_H$		XN,MPU	HSM	Reserved for AHB peripherals
	E800 3C00$_H$ - E800 3FFF$_H$	1 KB	XN,MPU	HSM	PKC module - SFR region
	E800 4000$_H$ - E800 7FFF$_H$	16 KB	XN,MPU	HSM	PKC module - Memory region
	E800 8000$_H$ - EBFF FFFF$_H$		XN,MPU	HSM	Reserved for AHB peripherals
	EC00 0000$_H$ - EC00 00FF$_H$	256 B	XN,MPU	HSM	Timer 0, Timer 1
	EC00 0100$_H$ - EC00 01FF$_H$	256 B	XN,MPU	HSM	Watchdog Timer
	EC00 0200$_H$ - EC00 02FF$_H$	256 B	XN,MPU	HSM	TRNG
	EC00 0300$_H$ - EFFF FFFF$_H$		XN,MPU	HSM	Reserved for APB peripherals
F	F000 0000$_H$ - F003 FFFF$_H$		XN,MPU	Host	Reserved for host peripherals
	F004 0000$_H$ - F005 FFFF$_H$	128 KB	XN,MPU	Host	HSM bridge
	F006 0000$_H$ - F803 FFFF$_H$		XN,MPU	Host	Reserved for host peripherals
	F804 0000$_H$ - F804 FFFF$_H$	64 KB	XN,MPU	Host	Host Command Interface (DMU)
	F805 0000$_H$ - F805 FFFF$_H$	64 KB	XN,MPU	Host	Host Protection Configuration (DMU)
	F806 0000$_H$ - F806 FFFF$_H$	64 KB	XN,MPU	Host	HSM Command Interface (DMU)
	F807 0000$_H$ - F807 FFFF$_H$	64 KB	XN,MPU	Host	HSM Protection Configuration (DMU)
	F808 0000$_H$ - FFBF FFFF$_H$		XN,MPU	Host	Reserved for host peripherals
	FFC0 0000$_H$ - FFC1 FFFF$_H$	128 KB	XN,MPU	Host	Data Flash 1 (HSM Command Sequence Interpreter)
	FFC2 0000$_H$ - FFFF FFFF$_H$		XN,MPU	Host	Reserved for host peripherals

B	Burst mode access
C	Cached
XN	eXecute Never
MPU	Under control of MPU
PRIV	Accessible in privilege mode only

In general memories are accessed via the unified cache and peripheral registers are directly accessed (not through the unified cache). For exchange of data via (cached) shared host RAM the cache clean registers shall be used, see **"Cache SFRs" on Page 26**.

The ranges $0000\ 0000_H$ – $DFFF\ FFFF_H$ and $E010\ 0000_H$ – $FFFF\ FFFF_H$ are controlled by the Core MPU, see **Chapter 2.4**.

The range $E000\ 0000_H$ – $E00F\ FFFF_H$ is accessible in privilege mode (see ARM Cortex M3 Technical Reference Manual **[2]**) only.

The range $0000\ 0000_H$ – $9FFF\ FFFF_H$ uses burst access. The granularity of an access is 16 bytes.

The range $A000\ 0000_H$ – $FFFF\ FFFF_H$ uses word access. Unaligned accesses are split in several word aligned accesses. Code cannot be executed from this address range, see **[2]**.

According to the ARMv7M architecture unaligned accesses (e.g. word access on a address that is not dividable by four) are split in several aligned accesses. It is also possible that some peripherals do only support register access by word-aligned addresses. Therefore it is strongly recommended to access Special Function Registers (SFRs) word-aligned to avoid unintended side effects.

Also SFRs shall be accessed 32-bit wise or by their register size. Using e.g. 8-bit or 16-bit accesses for a 32-bit reigster lead to undefined behavior.

The ARM processor memory map includes two bit-band regions. These bit-band regions map each word in a 32 MB alias region of memory to a bit in a 1 MB bit-band region of memory (for details see the ARM Technical Reference Manual). Accesses to the 32 MB alias region ($2200\ 0000_H$ - $23FF\ FFFF_H$) map to the 1 MB bit-band region ($2000\ 0000_H$ - $2010\ 0000_H$) and accesses to the 32 MB alias region ($4200\ 0000_H$ - $43FF\ FFFF_H$) map to the 1 MB bit-band region ($4000\ 0000_H$ - $4010\ 0000_H$). This address mapping is performed regardless of whether there is a physical memory present or not. Note, that a bus exception may be triggered by the host system in case there is no target memory present at the re-mapped address.

Accesses to reserved address regions in the range of $0000\ 0000_H$ – $DFFF\ FFFF_H$ or $F000\ 0000_H$ – $FFFF\ FFFF_H$ – i.e. addresses that are not part of any memory or host peripheral – cause a bus exception by the host system and will be signaled via **ERRCTRL** / **ERRADDR**.

Accesses to reserved address regions in the range of $E000\ 0000_H$ – $E00F\ FFFF_H$ – i.e. addresses in the Private Peripheral Bus region the ARM default behavior applies.

Accesses to reserved address regions in the range of $E010\ 0000_H$ – $EFFF\ FFFF_H$ – i.e. addresses that are not part of any HSM peripheral – cause a Bus Fault.

If not otherwise noted, unused addresses within a peripheral region return zero and write data is lost without triggering an exception.

A faulty Core MPU setup resulting in a Cache write-back to ROM is silently ignored, and in this case the data is lost.

2.2 CPU

The HSM includes an Central Processing Unit based on ARM Cortex M3 Revision r2p0. Please refer to ARM Cortex M3 Technical Reference Manual [2] and ARM Cortex M3 Errata Notice [3] for information on the processor architecture and programming model.

The following subsections describe HSM-specific implementation options.

2.2.1 Multiprocessor Communication

The instruction SEV (Send EVent) cannot be used to synchronize execution with the host system. The communication registers of the bridge module (see **Chapter 9.2.2** and **Chapter 9.3.1**) have to be used for synchronization.

2.2.2 The SysTick Timer

The SysTick Timer uses the HSM system clock as core clock, see **Chapter 9.2.3**. As reference clock SPB clock frequency / 16 is provided. The SysTick Calibration Value register holds the value 62500_D which corresponds to a SPB clock frequency of 100 MHz and a reference clock frequency of 6.25 MHz.

For a periodic 10 ms interrupt the value 0000 $F423_H$ (62500 - 1) has to be written into the SysTick Reload Value register.

For a periodic 1 s interrupt the value 005F $5E0F_H$ (62500 x 100 - 1) has to be written into the SysTick Reload Value register.

2.2.3 Power Management

A dedicated Sleepdeep mode is not implemented. Setting of bit SLEEPDEEP in the System Control Register has no effect besides that the SysTick Timer is not running in Sleepdeep mode.

The optional Wake-up Interrupt Controller (WIC) as described in section 7.2.4 in [2] is not implemented.

2.2.4 Byte Order

The byte order of the HSM is little endian.

2.2.5 Application Interrupt and Reset Control Register

Setting the bits SYSRESETREQ or VECTRESET in the Application Interrupt and Reset Control Register (see [2] for details) lead to undefined behavior. Therefore these bits shall not be set.

2.2.6 Code Prefetch

The ARM Cortex M3 utilizes a Prefetch Unit (PFU) with the purpose:

- Fetch instructions in advance and forward PC relative branch instructions. Fetches are speculative in the case of conditional branches
- Detect Thumb 32-bit instructions and present these as a single instruction word.
- Perform vector loads.

The PFU fetches instructions from the memory system that can supply one word each cycle. The PFU buffers up to three word fetches in its FIFO, which means that it can buffer up to three Thumb 32-bit instructions or six Thumb instructions.

The majority of branches that are generated as the ALU addition of PC plus immediate are generated no later than the decode phase of the branch opcode. In the case of conditionally executed branches, the address is speculatively presented (consuming a fetch slot on the bus), and the forwarded result determines if the branch path flushes the fetch queue or is preserved.

The prefetch mechanism has to be considered at the end of a code section. In order to avoid traps or ECC errors it is advisable to pad the end of code by up to 32 bytes (size of two cache blocks).

2.2.7 Exclusive Access Instructions

The exclusive access instructions STREX, LDEX and CLREX are supported and can be used for task synchronization within the HSM. Synchronization with the host system is not possible with these instructions.

The exclusive access instructions shall be used for the HSM local RAM only. The behavior for host memories is not deterministic as these memories can be modified by the host system.

2.2.8 Instruction Timing

The HSM uses a cache to relieve the load on host memories, particularly code memory and, in this way, accelerates access to commonly used data.

Through this cache almost every code fetch and load/store behave as if they would access single cycle memories. Contention on the fetch and data cache interface occurs when both use the same word offset. In this case the code fetch is stalled for one clock cycle.

A cache miss adds up to four cycles delay for local RAM and ROM. With a targeted hit rate of above 90% this can be neglected.

The cache itself is described in its own chapter.

2.3 Interrupt Unit

The HSM uses the Nested Vectored Interrupt Controller (NVIC) from ARM as an interrupt unit. Please refer to chapter 8 of [2] for details.

2.3.1 Interrupt mapping

The NVIC allows design-specific number of interrupt nodes and priority levels. The HSM implements 10 interrupt nodes and 8 priority levels. The watchdog timer interrupt is connected to the NMI.

The following table provides a summary of all interrupts and the mapping to interrupt number. For details please refer to the corresponding description of the module.

Table 2-2 Interrupt Mapping

Exception Number	Exception Type	Priority	Description
2	NMI	-2	Watchdog Timer interrupt
16	Timer 0 Interrupt	Programmable	Timer 0 overflow interrupt
17	Timer 1 Interrupt	Programmable	Timer 1 overflow interrupt
18	TRNG Interrupt	Programmable	True Random Number Generator
19	Bridge Service Interrupt	Programmable	HSM Bridge Service interrupt
20	Bridge Error Interrupt	Programmable	HSM Bridge Error interrupt

Table 2-2 Interrupt Mapping (cont'd)

Exception Number	Exception Type	Priority	Description
21	Sensor Interrupt	Programmable	Sensor interrupt
22	External Interrupt	Programmable	External Peripheral interrupt
23	-		
24	PKC Interrupt	Programmable	PKC Ready interrupt
25	PKC Interrupt	Programmable	PKC Error interrupt

Due to the high performance of the AES module it has no interrupt.

2.3.2 Expansion of VTOR

Bit 30 and 31 of the Vector Table Offset Register (VTOR) are added to allow to set the vector table base to flash addresses. The reset value of VTOR is $0000\ 0000_H$.

2.4 Memory Protection Unit

With the Core MPU the HSM implements an (almost fully) ARMv7-M compatible Protected Memory System Architecture (PMSAv7). Please refer to chapter 9 of [2] and especially [4] for details.

Some comments may be necessary for using the core MPU:

- The address range $A000\ 0000_H$ - $FFFF\ FFFF_H$ is Execute Never (XN) in the ARMv7-M architecture. The MPU cannot change this.
- An MPU region set to the bit band alias region is ignored. Use the corresponding address range in Local RAM instead.
- The C (Cacheable) and S (Shareable) bits and the TEX fields of the MPU regions are ignored, i.e. they have no functionality.
- Overlapping protection regions are supported with ascending region priority, i.e. region number 7 has highest priority, see [2].

2.5 Cache

The following chapter describes the Cache module. It contains the following sections:

- **Overview**.
- **Functional Description**.
- **Cache SFRs**.

2.5.1 Overview

The major features of the cache can be summarized as follows:

- Unified cache for memory.
- Multi-way set associativity.
- Least Recently Used (LRU) replacement policy for each set.
- Support for *lock* and *unlock*, to keep a block from being automatically replaced.
- Cache block-, set-, and all-clean (including invalidation of tags).
- Caching policy is read/write allocate and write-back - no write-through.
- One-cycle access for read and execute hits.
- One-cycle access for write hits, employing two write buffers.
- No cache line wrap penalty.

Cache memory – or simply, a cache – is a high-speed buffer located between the CPU and the (external) main memories holding a copy of some of the memory contents to enable access to the copy, which is considerably faster than retrieving the information from the main memory. In addition to its fast access speed, the cache also consumes less power than the main memories. All cache systems owe their usefulness to the principle of locality, meaning that programs are inclined to utilize a particular section of the address space for their processing over a period of time. By including most or all of such a specific area in the cache, system performance can be dramatically enhanced.

The cache in this product has the following hardware configuration.

Table 2-3 Cache Configuration

Size	Sets (n)	Ways (m)	Block Size (k)
4 KB	64	4	16
Address Decomposition			
tag = ADDR[31:10]	*set* = ADDR[9:4]		*block* = ADDR[3:0]

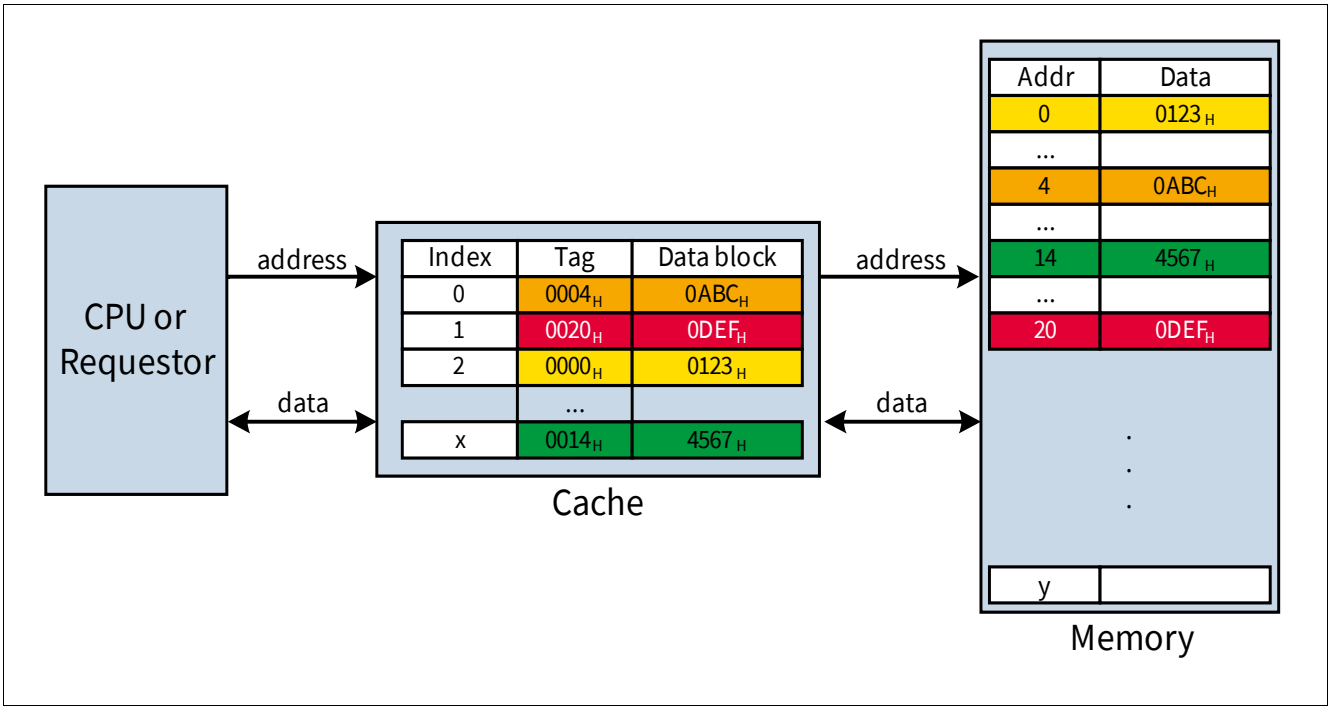


Figure 2-1 Cache Overview

The effectiveness of a cache is measured in terms of its so-called hit ratio – the probability of detecting a requested address in the cache so that the appropriate address can be directly provided. The absence of a corresponding address entry in the cache is referred to as a miss, with the consequence that the relevant information has to be procured from the main memory. The cache contents are refreshed by a LRU or Round Robin replacement policy, which selects entries that have been unused the longest (in a set) for replacement by the most recent memory retrievals. However, the cache also provides an option for locking cache entries against such overwriting.

As caches only cover a subset of all addresses held in the address space, they must necessarily represent a compromise in their realization. They should be adequately dimensioned and mapped to main memory to enable satisfactory hit results at acceptable cost. The cache is based on a set-associative design employing a so-called tag section and a data section. Part of a requested address is compared against the tag entries for a match (hit). If found, the corresponding block in the data section returns the corresponding data of the input address. If the access results in a miss, however, a tag and data block in the cache is then replaced by the new address and data. The former data block in the cache is known as a victim. If this line was modified by a write operation (even if the value remains identical), it must also be written back to the main memory to preserve consistency. A modified line in the cache is called dirty (indicated by an internal flag) until it has been written back to main memory (write-back).

2.5.2 Functional Description

The cache is **m**-way set associative for the placement of its entries. The cache is divided into **n** sets (n is a power of 2), each containing **m** data lines. These sets are referred to as **m**-way sets since **m** tag entries are associated with each set (**m** ways to find the relevant data entry). The number of cache data lines employed by the cache is therefore **n x m**, each of which carries **k** bytes (k is a power of 2). The data line size is referred to as the block size. For example: A 4-way cache with 32 sets and a block size of 16 bytes results in a cache size of 2 KB (2048 bytes).

The relevant data element gives the contents of the address corresponding to the tagged address entry – i.e., the contents of the mapped address.

A set is referenced using the *set* bits in the address (refer to **Table 2-3** for a description of the address as used by the cache). The ways in this set are then searched for a hit with the tag information using the *tag_address* bits in the address. If a hit is found, the cache uses further information (the *block* bits in the address) to locate the appropriate data in the block.

The cache peripheral provides a number of SFR controlled functions for optimized handling of the cache. For example, the user can clean a block specified by an address, or lock a block specified by its address so that those cache entries cannot be replaced. If the lock is no longer needed an unlock function is provided. The complete set of SFRs is described in **Section 2.5.3**.

Figure 2-2 illustrates the cache principle for detecting a specified address to return the data for its assigned address in the address space. A hit is shown in this instance.

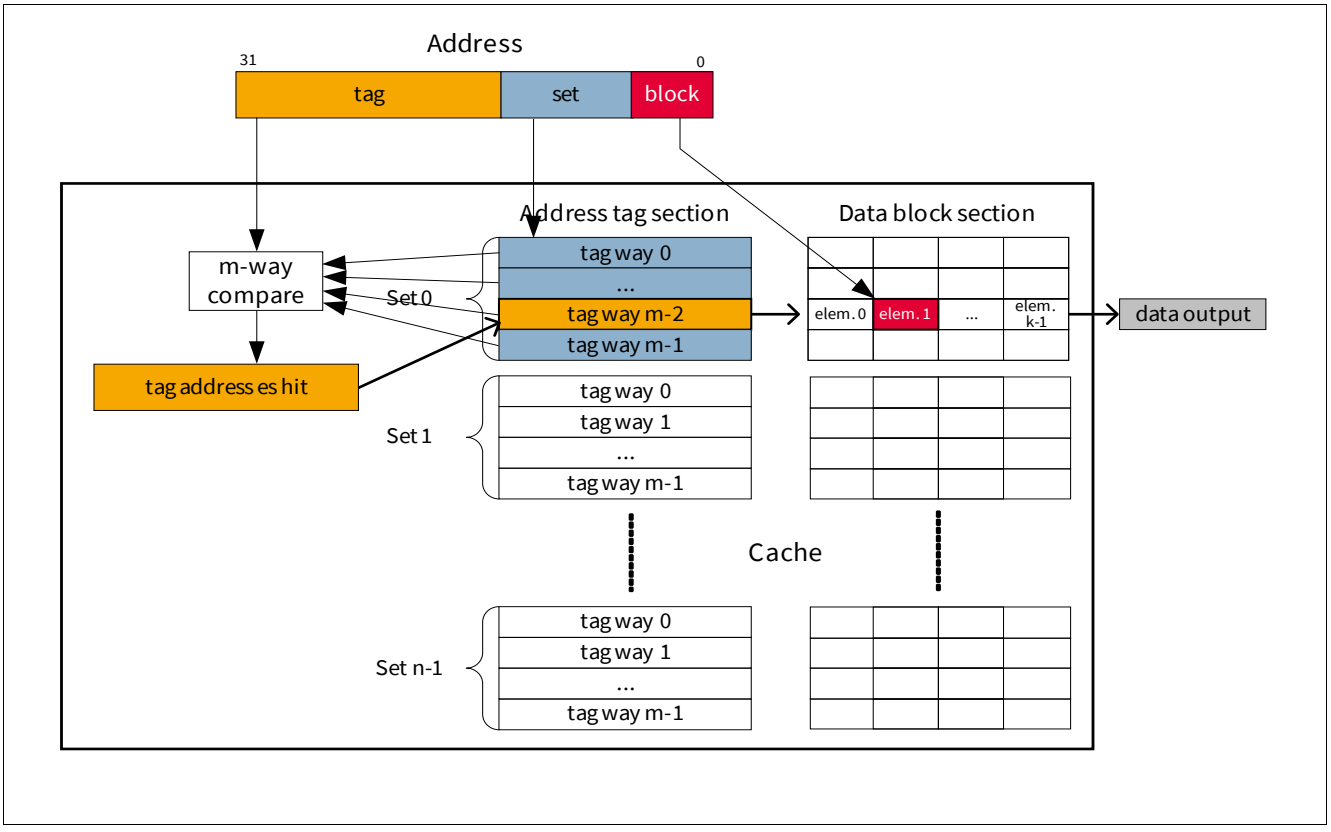


Figure 2-2 Cache Principle

2.5.2.1 Parallel Cache Accesses

The words of a data block are stored in segregated memories (one data word per memory). These memories have several interfaces: memory bus, instruction interface and data interface. The interfaces can be used in parallel, which enhances performance. However, if different data elements in the same memory are accessed by interfaces at the same time, one access will be given priority and the subsequent access(es) will be delayed, meaning that a relationship between address and operation type exists. A bit named DAPCA (DisAble Parallel Cache Accesses) in **CACHE_CTRL** is provided to disable simultaneous accesses to data elements. Some reduction in performance is experienced by disabling parallel accesses.

2.5.3 Cache SFRs

A number of special function registers are implemented to control the Cache operation, as listed below.

Table 2-4 Register Address Space

Module	Base Address	End Address	Note
Cache	E008 A000$_H$	E008 BFFF$_H$	

Table 2-5 Register Overview

Register Short Name	Register Long Name	Offset Address	Reset Value
Cache SFRs, Configuration SFRs			
CACHE_CONFIG	Cache Configuration Register	0000$_H$	0006 0404$_H$
CACHE_CTRL	Cache Control Register	0004$_H$	0000 0000$_H$
Cache SFRs, Command SFRs			
CACHE_AC	Cache All Clean Register	0100$_H$	0000 0000$_H$
CACHE_SC	Cache Set Clean Register	0104$_H$	0000 0000$_H$
CACHE_BC	Cache Block Clean Register	0110$_H$	0000 0000$_H$
CACHE_BT	Cache Block Touch Register	0114$_H$	0000 0000$_H$
CACHE_BL	Cache Block Lock Register	0118$_H$	0000 0000$_H$
CACHE_BU	Cache Block Unlock Register	011C$_H$	0000 0000$_H$

The registers are addressed wordwise.

All Cache module SFRs are accessible in privileged mode, user access will cause a BusFault.

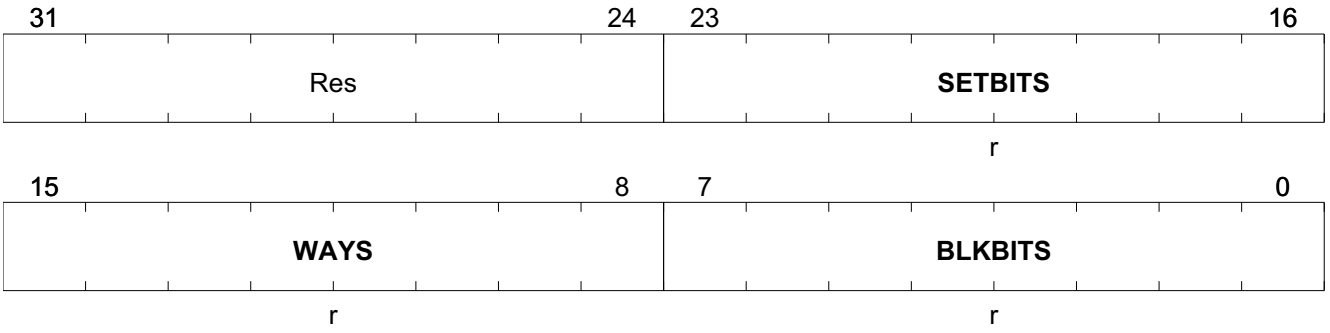
2.5.3.1 Configuration SFRs

The configuration registers control the behavior of the cache.

Cache Configuration Register

The cache configuration register allows the cache configuration details to be viewed.

CACHE_CONFIG	Offset	Reset Value
Cache Configuration Register	0000_H	$0006\ 0404_H$

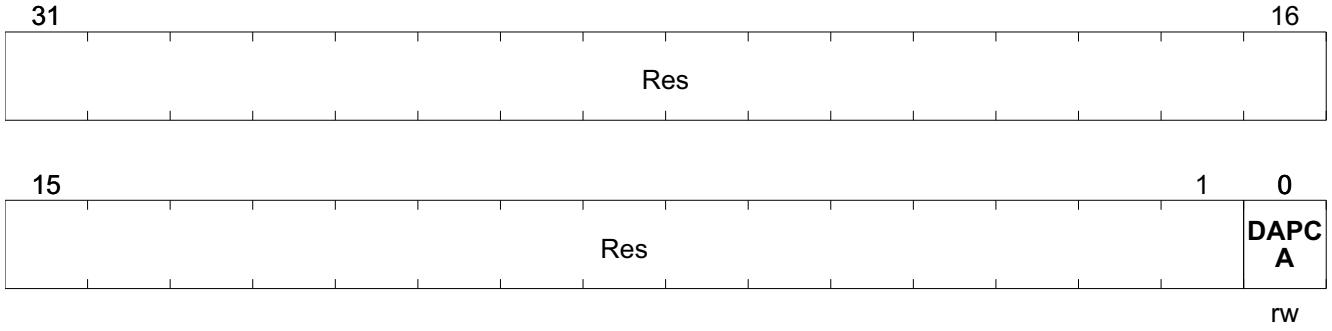


Field	Bits	Type	Description
SETBITS	23:16	r	**Number of Set Bits** This field controls the number of sets within the cache. The cache uses $2^{SETBITS}$ sets. The maximum cache configuration uses 64 sets.
WAYS	15:8	r	**Number of Ways** This field controls the number of ways within each set. A set contains WAYS number of ways. The maximum cache configuration uses 4 ways.
BLKBITS	7:0	r	**Number of Block Bits** This field shows the number of bytes within a cache line. The following values are defined, all others are *rfu*. 04_H **BW4**, A cache line contains $2^{BLKBITS}$ = 16 bytes.

Cache Control Register

The cache control register allows configuration of the parallel cache access functions.

CACHE_CTRL	Offset	Reset Value
Cache Control Register	0004_H	$0000\ 0000_H$



Field	Bits	Type	Description
DAPCA	0	rw	**Disable Parallel Cache Accesses** See **Section 2.5.2.1** for more details regarding this bit. 0_B **EN**, Parallel accesses to the cache are possible. 1_B **DIS**, Parallel accesses to the cache are disabled.

2.5.3.2 Command SFRs

The command registers are used to perform cache operations:

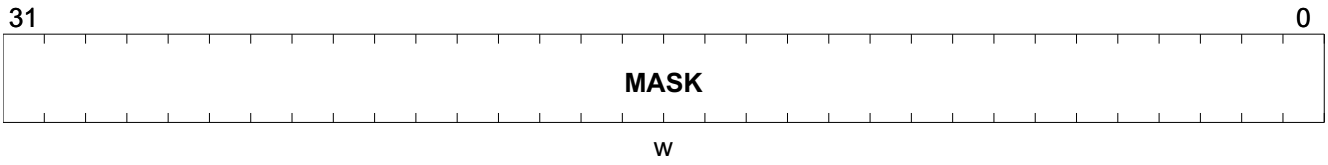
- Cache clean – with different granularities
- Block touch.
- Block lock and unlock.

This group of registers are write only registers. By writing to one of these registers, an operation on the cache is triggered. Reading out the registers returns 0000 0000_H.

Cache All Clean Register

Writing to this register cleans all valid blocks from the cache. If a data block is marked as dirty the block is written back to memory. All valid blocks are unlocked and invalidated.

CACHE_AC	Offset	Reset Value
Cache All Clean Register	0100_H	0000 0000_H

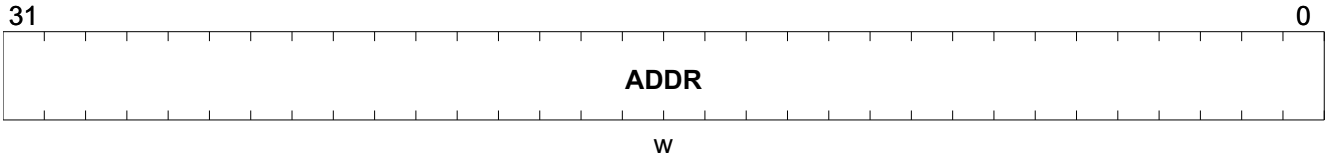


Field	Bits	Type	Description
MASK	31:0	w	**Mask** Writing to this SFR cleans the cache. A read returns zero.

Cache Set Clean Register

Writing to this register cleans all valid blocks of one *SET* from the cache specified by the address (*ADDR*). If a data block in this set is marked as dirty the block is written back to memory. All valid blocks in this set are unlocked and invalidated.

CACHE_SC	Offset	Reset Value
Cache Set Clean Register	0104_H	$0000\ 0000_H$

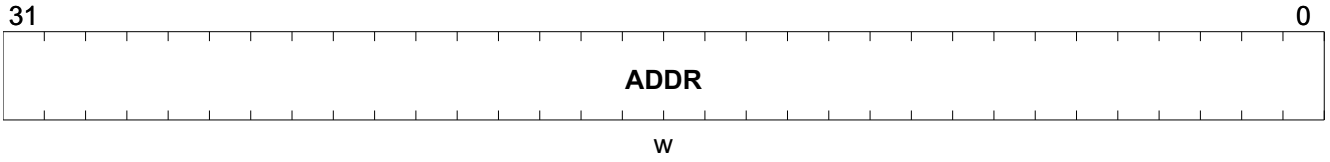


Field	Bits	Type	Description
ADDR	31:0	w	**Address** The address defines which set to clean from the cache. The address bits used depend on the cache size configuration.The maximum cache configuration uses ADDR[9:4]. **CACHE_CONFIG**.SETBITS contains the current configuration. A read returns zero.

Cache Block Clean

A memory block specified by the address (*ADDR*) is cleaned from the cache. If dirty the block will be written back. It is also unlocked, invalidated. If that block is not valid or not in the cache, then no operation is performed. Specifically, a refill is not performed.

CACHE_BC	Offset	Reset Value
Cache Block Clean Register	0110_H	$0000\ 0000_H$



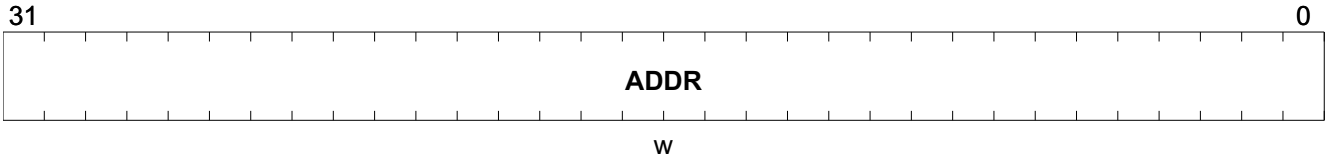
Field	Bits	Type	Description
ADDR	31:0	w	**Address** The address of the block to be cleaned. ADDR[3:0] are ignored. A read returns zero.

Cache Block Touch Register

A memory block specified by the address (ADDR) written to this register is loaded into the cache. If that address has a hit in the cache, then no load is performed. In both cases this block is marked as most recently used.

If the load from memory fails no BusFault is triggered.

CACHE_BT	Offset	Reset Value
Cache Block Touch Register	0114_H	$0000\ 0000_H$



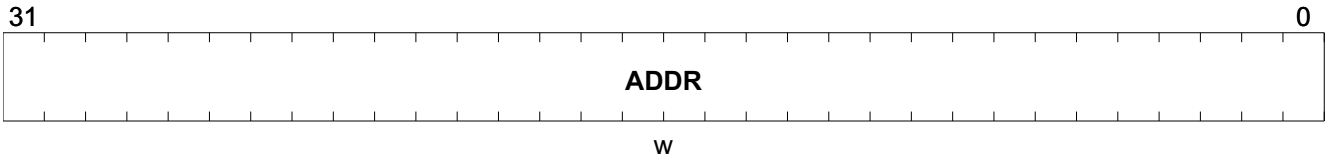
Field	Bits	Type	Description
ADDR	31:0	w	**Address** The address to be touched. ADDR[3:0] are ignored. A read returns zero.

Cache Block Lock Register

A memory block specified by the address (ADDR) is locked in the cache. If that address has no hit in the cache, then the block is fetched and locked. If the maximum number of lockable ways are already locked, i.e. **CACHE_CONFIG**.WAYS - 2, then the cache lock operation is not performed and no fetch is executed. In this case a BusFault exception is thrown.

Two ways per set will always remain unlocked – i.e. a cache configuration with less than 4 ways results in one or even zero lockable ways (see **CACHE_CONFIG**.WAYS).

CACHE_BL	Offset	Reset Value
Cache Block Lock Register	0118_H	$0000\ 0000_H$

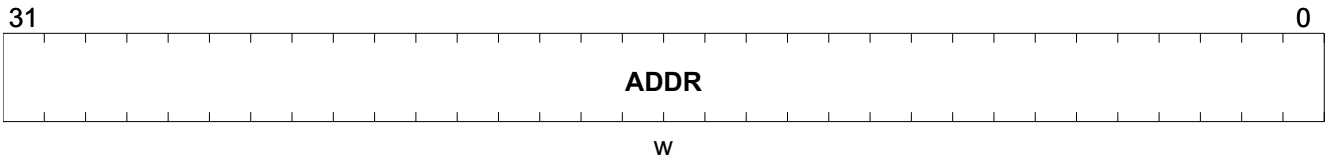


Field	Bits	Type	Description
ADDR	31:0	w	**Address** The address to be locked. ADDR[3:0] are ignored. A read returns zero.

Cache Block Unlock

A memory block specified by the address is unlocked and marked as most recently used. If that address has no hit in the cache, then no operation is performed. Specifically, a fill is not performed.

CACHE_BU	Offset	Reset Value
Cache Block Unlock Register	$011C_H$	$0000\ 0000_H$



Field	Bits	Type	Description
ADDR	31:0	w	**Address** The address to be unlocked. The bits ADDR[3:0] are ignored. A read returns zero.

3 True Random Number Generator

This chapter describes the True Random Number Generator (TRNG) module. It contains the following sections:

- **Concept Overview**.
- **Performance**.
- **Functional Overview**.
- **TRNG SFRs**.

3.1 Concept Overview

The TRNG module provides random data for cryptographic algorithms (keys), protocols (challenges, blinding values, padding bytes, etc.).

A non-deterministic natural phenomenon is exploited to generate a data stream which has to fulfill the quality criteria (minimum entropy requirement) defined in the AIS-20/31 publication by the German BSI **[12]**. In particular, requirements for devices belonging to the functionality class PTG.2 (strength of mechanism: high) are considered .

Every noise source, even if well-designed, produces a bit stream that usually shows statistical defects due to bandwidth limitation, ageing and temperature drifts, fabrication tolerances and deterministic disturbances. As a consequence, the noise source is always followed by a digital postprocessing (DPP) device, as shown in **Figure 3-1**, where the definitions stated in AIS-20/31 have been adopted.

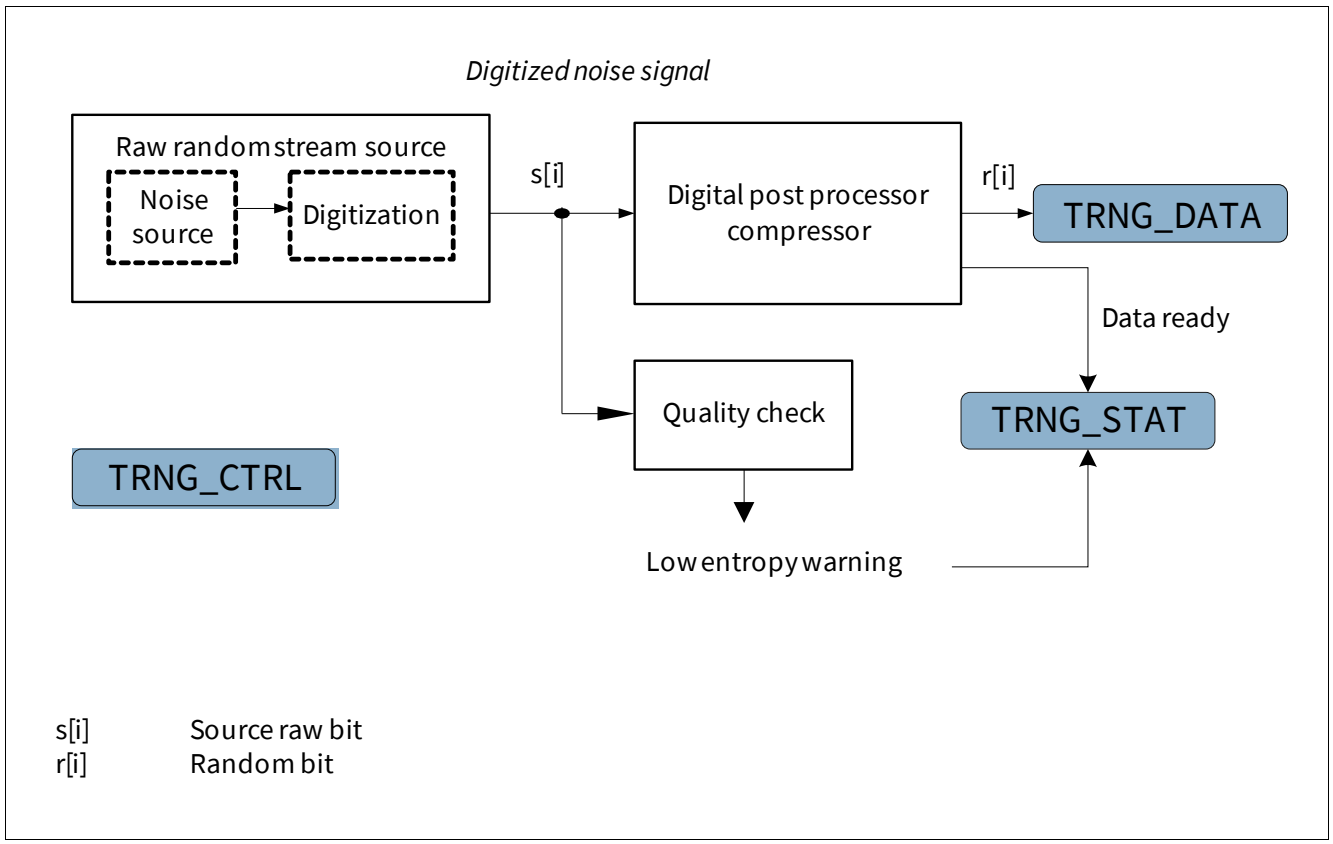


Figure 3-1 Principle of TRNG operation

The postprocessor has the following two functions: it is requested to adjust the statistical distribution of the raw bit stream s[i] (referred to as a digitized noise signal, DNS) and, above all, to increase the entropy per bit of the output stream r[i] (internal random sequence, IRS) collecting entropy from the input sequence. Entropy in information theory is a measure of the uncertainty associated with a random variable used to quantify the information in a segment of data: it represents the minimum average number of bits that must be sent in order to relay the true value of a random variable. The lower the entropy, the more predictable the bit values and therefore the poorer the randomization.

3.2 Performance

The data rate of the TRNG after postprocessing is variable and depends on the bits of entropy per second generated by the noise source. Furthermore, it depends on the clock frequency and the configuration settings. Nontheless the following appoximations for the minimum, typical, and maximum avarage throughput are accurate to ± 15%. The throughput R is given in units of kilobits per second and the clock frequency f in MHz.

$$R\left[\frac{kbit}{s}\right] = \frac{f[MHz]\cdot 8000}{f[MHz]\cdot T[\mu s] + C} \tag{3.1}$$

The constants T and C are given in the following table. They correspond to the constant and dynamic time components, respectively, for generating one byte. Hence, the number of clock cycles needed for generating one output byte is given by t = f×T+C.

Table 3-1 Generation Time for One Random Byte

	Minimum	Typical	Maximum
Constant time T [μs]	9	18	71
Dynamic time C [clock cycles]	335	414	758

According to this formula and for a clock frequency of 33 MHz we have a typical generation time for one byte of t_{typ} = 1008 clock cycles, corresponding to a throughput of R_{typ} = 260 kb/s. For a clock frequency of 100 MHz the typical throughput is approximately R_{typ} = 360 kb/s . Should the maximum number of clock cycles elapse before generation of a data block (with a size specified by **TRNG_CTRL**.DBS), a warning is indicated (**TRNG_STAT**.WARN).

After a reset, the bias tuning loop needs a setup time during which a lower entropy is expected from the noise source. The setup time is less than the time required to generate one output byte (i.e. typically less than 1000 clock cycles for 33 MHz). Due to the adaptive compression, the output quality is practically assured in every condition and the user does not have to discard any output byte after a reset or after returning from sleep mode.

3.3 Functional Overview

The TRNG module continuously generates true random bytes for the system operation once enabled (**TRNG_CTRL**.DIS = 0). The availability of new true random data is signaled by setting the DTA_RDY bit in the **TRNG_STAT** register and triggering the corresponding “output buffer not empty” interrupt. The TRNG waits until the data has been read before recommencing generation. Once the data has been read (from **TRNG_DATA32**) the flag **TRNG_STAT**.DTA_RDY is cleared by hardware. (Note that the TRNG will, in certain cases, not generate new data if the DTA_RDY flag is cleared: these cases relate to poor quality data as indicated by warning and/or error bits. Please refer to **Additional FIPS_ERR and WARN Information** for a description of these cases.)

3.3.1 Sleep Mode / Disable

The TRNG enters sleep mode when the module is disabled by setting the control bit**TRNG_CTRL**.DIS.

In sleep mode, the noise source and other components are reset, the digital post-processor is disabled and its state is maintained.

3.3.2 Interrupt Behavior

The true random number generator triggers an interrupt whenever a random data block is ready to be output from **TRNG_DATA32** (also indicated by **TRNG_STAT**.DTA_RDY = 1_B).

An interrupt is triggered in the event of an error or a warning occurring, as indicated by the FIPS_ERR bit and/or the WARN bit in the **TRNG_STAT** register. The interrupt is handled by the interrupt controller.

Note: Remember that an interrupt is only triggered once, which means that a 1_B for a warning or error bit in the TRNG_STAT SFR will not cause continuously new interrupt requests.

3.4 TRNG SFRs

The interface of the TRNG, comprises the registers shown in **Table 3-3** below.

Table 3-2 Register Address Space

Module	Base Address	End Address	Note
TRNG	EC00 0200_H	EC00 $02FF_H$	

Table 3-3 Register Overview

Register Short Name	Register Long Name	Offset Address	Reset Value
TRNG SFRs, Status and Data Registers			
TRNG_DATA32	True RNG Data Register	00_H	XXXX $XXXX_H$
TRNG_STAT	TRNG status register	$0C_H$	$0X0X_H$
TRNG SFRs, Control Registers			
TRNG_CTRL	TRNG control register	10_H	0002_H

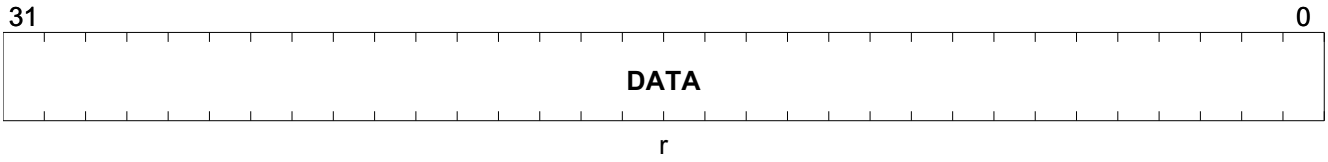
The registers are addressed wordwise.

3.4.1 Status and Data Registers

Note: A read access to TRNG_DATA when a random word is not available (TRNG_STAT.DTA_RDY = 0) retrieves the last value. Therefore it is necessary to check TRNG_STAT for WARN, FIPS_ERR and DTA_RDY before TRNG_DATA32 is accessed to ensure that a new random value can be read.

True RNG Data Register

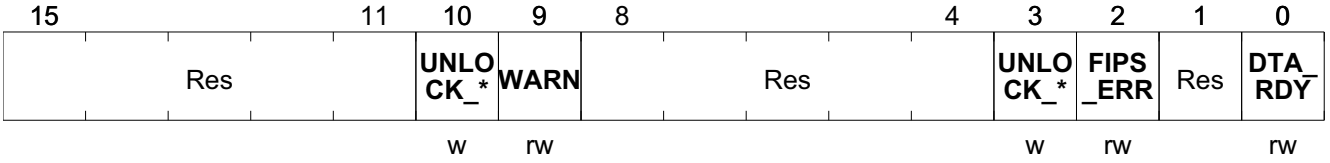
TRNG_DATA32	Offset	Reset Value
True RNG Data Register	00_H	$XXXX\ XXXX_H$



Field	Bits	Type	Description
DATA	31:0	r	**Random Data** Random bit block to read *Note: If the TRNG operates in byte mode, the randomly generated data is provided in bits 7...0; all other bits return zero.*

TRNG status register, TRNG_STAT

TRNG_STAT	Offset	Reset Value
TRNG status register	$0C_H$	$0X0X_H$



Field	Bits	Type	Description
UNLOCK_WARN	10	w	**Unlock Bit for WARN** 0_B **LOCK**, The bit WARN remains unchanged in the event of this SFR being written to. This setting prevents unintentional changes to that bit. 1_B **WE**, The bit WARN may be changed by writing to the SFR.
WARN	9	rw	**Quality Warning** The number of transitions from the value 0_B to the value 1_B and from 1_B to 0_B is monitored according to the concept outlined in **Section 3.1**. A warning is issued if too few transitions within a sequence of random bits occur. The compression does not have adequate quality. See also **Additional FIPS_ERR and WARN Information**. 0_B **GOOD**, No warning. The quality of the random data is satisfactory. 1_B **LOW**, Warning. The quality of the random data is inadequate.
UNLOCK_FIPS_ ERR	3	w	**Unlock Bit for FIPS_ERR** 0_B **LOCK**, The bit FIPS_ERR remains unchanged in the event of this SFR being written to. This setting prevents unintentional changes to that bit. 1_B **WE**, The bit FIPS_ERR may be changed by writing to the SFR.
FIPS_ERR	2	rw	**FIPS Error** An error is indicated if the current 32 bits of data are identical to the 32 bits of data generated before the current data. See also **Additional FIPS_ERR and WARN Information**. 0_B **OK**, No error detected. 1_B **ERROR**, Data is identical.

Field	Bits	Type	Description
DTA_RDY	0	rw	**Random Data Ready** The bit is cleared by hardware during generation of a random word. The bit is set by hardware when a new result is available and the generation has run without warning or error . The bit is cleared by hardware once the random data has been read. See also **Additional FIPS_ERR and WARN Information**. 0_B **BUSY**, No data ready for reading. 1_B **RDY**, Data ready for reading.

Additional FIPS_ERR and WARN Information

If the hardware sets FIPS_ERR or WARN, the corresponding “error has occurred” interrupt is triggered. Data generation is stopped because low quality data has been produced. The software must then decide what to do with this low quality data, namely to validate or discard the data by writing to TRNG_STAT in either of the ways explained below:

- Discard the generated data by writing 0408_H (unlock_warn = 1, warn = 0, unlock_fips_err = 1, fips_err = 0, dta_rdy = 0). The TRNG will immediately start to generate new random data, or
- Validate the data by writing 0409_H (unlock_warn = 1, warn = 0, unlock_fips_err = 1, fips_err = 0, dta_rdy = 1). This causes an “output buffer not empty” interrupt to be triggered, and the data can be read out of **TRNG_DATA32** (remember that the data is of low quality). The TRNG will recommence generation of new random data after **TRNG_DATA32** has been read.

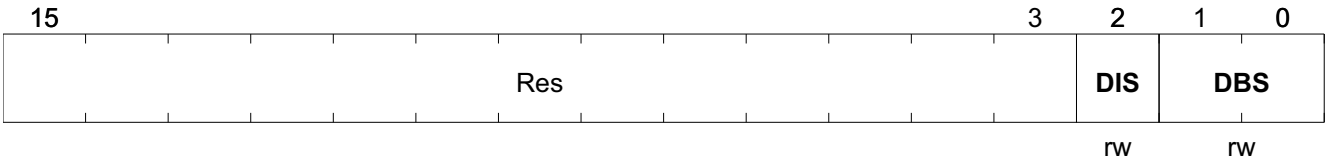
Each of the two operations must be carried out with a single access. The TRNG will behave unpredictably if the software attempts to update any flag in **TRNG_STAT** while the TRNG is generating data, or if good quality data was generated.

Attempting to read the **TRNG_DATA32** register after the hardware has detected and indicated low quality data without first validating the data (as described above) causes the controller to retrieve the last value.

3.4.2 Control Registers

TRNG control register, TRNG_CTRL

TRNG_CTRL	Offset	Reset Value
TRNG control register	10_H	0002_H



Field	Bits	Type	Description
DIS	2	rw	**Disable** It stops oscillators and digital logic. 0_B **EN**, TRNG starts generating new data if no data is ready at the time (as indicated by **TRNG_STAT**.DTA_RDY) and the warning/error flags are not set (see **TRNG_STAT**). 1_B **DIS**, TRNG is stopped even if no random data is available at the time. Reset: 0_B
DBS	1:0	rw	**Data Block Size** 00_B **BYTE**, Size of data block is 8 bits. 01_B **HW**, Size of data block is 16 bits. 10_B **WORD**, Size of data block is 32 bits. Reset: 10_B

Note: The data block size can be changed (using the DBS bit) while the TRNG is generating data without first having to disable it . Depending on the internal status of the TRNG, the new data block size is applied on the current data generation or on the following generation of data.

4 AES 128 Encryption / Decryption Device

This chapter describes the HSM's Advanced Encryption Standard module. It contains the following sections:

- **Introduction**.
- **Algorithms**.
- **Register Architecture**.
- **AES SFRs**.
- **Pseudo Coding Example with Test Data**.

4.1 Introduction

The AES module is a fast hardware device that supports fast encryption and decryption via a 128-bit-key AES [5]. It enables plain/simple encryption and decryption of a single 128-bit data (i.e., plain text or cipher text) block as well as encryption or decryption of a multitude of data blocks of 128 bits each. For these, several so-called modes of operation are implemented

- ECB (electronic code book mode)
- CBC (cipher block chaining mode)
- CTR (32-bit counter mode)
- OFB (output feedback mode)
- CFB (cipher feedback mode)

This enables also the additional modes

- GCM (Galois counter mode)
- XTS (XEX-based Tweaked CodeBook mode (TCB) with Cipher Text Stealing (CTS))

Furthermore, the AES module supports the following features:

- Internal storage for 8 AES keys which are not readable.
- Key 0 and 1 are lockable, i.e., not writable nor readable.
- Minimum: 25 MB/s @ 100 MHz.
- Maximal latency of 1 μs @ 100 MHz.
- 5 contexts for modes of operation.

4.2 Algorithms

The AES module enables the efficient implementation of the following encryption and decryption algorithms. All the following algorithms can be implemented using the basic instructions defined via the control register AESCTRL.

For the following we will use the abbreviations:

BS is the byte space of the hexadecimal symbols $\{00_H, 01_H, ..., FF_H\}$. It contains 256 elements.

DS is the space of the 16-byte data blocks BS^{16}. It contains 2^{128} elements.

KS is the space of the 16-byte AES-keys BS^{16}. It contains 2^{128} elements.

DS and KS are identical spaces. Distinction is made for clarity reasons.

An element P or C of $DS=BS^{16}$ is written as $P=(x_0,...,x_{15})$ or $C=(y_0,...,y_{15})$. Similarly, for a k in KS.

In the GCM mode, as well as the XTS mode, polynomial arithmetic is used: The input and output values of the plain AES are interpreted as binary polynomials of degree < 128, i.e., elements of the polynomial residue ring

$R := F_2[X]/(f(x))$. (4.1)

Here, f is the binary polynomial of degree 128:

$f(X) = X^{128} + X^7 + X^2 + X^1 + 1$. (4.2)

In particular, the AES input or output will be interpreted as a 128-bit string

$a_0, a_1, a_2, \ldots, a_{126}, a_{127}$, (4.3)

and this defines the polynomial:

$a_{127} X^{127} + a_{126} X^{126} + \ldots + a_2 X^2 + a_1 X + a_0$. (4.4)

In XTS mode, the mapping between the 16-byte data blocks $(x_0,\ldots,x_{15})$ and the 128-bit strings $a_0, a_1, a_2, \ldots, a_{126}, a_{127}$, is as follows: For i=0,1,...,15 and j:=8*i

$x_i = a_{j+7}\, 2^7 + a_{j+6}\, 2^6 + \ldots + a_{j+2}\, 2^2 + a_{j+1}\, 2 + a_j = (a_{j+7} \ldots a_{j+1} a_j)_2$. (4.5)

In GCM mode, the mapping between the 16-byte data blocks $(x_0,\ldots,x_{15})$ and the 128-bit strings $a_0, a_1, a_2, \ldots, a_{126}, a_{127}$, is as follows: For i=0,1,...,15 and j:=8*i

$x_i = a_j\, 2^7 + a_{j+1}\, 2^6 + \ldots + a_{j+5}\, 2^2 + a_{j+6}\, 2 + a_{j+7} = (a_j\, a_{j+1} \ldots a_{j+7})_2$. (4.6)

Note that all symbols like P_i, C_i, IV_i, CV_i, Y_i,... denote elements of DS or KS. They do **not** denote any registers which will be introduced later and do not have an index but a numeration within the name.

4.2.1 Plain AES

The plain AES encryption algorithm AES-ENC is a map:

$C := \text{AES-ENC}_k(P)$ (4.7)

where P is a 16-byte plain text from DS, k is a 16-byte key from KS, and C is a 16-byte cipher text from DS.

The mapping to the standard [5] is given as follows:

$P = (in_0,\ldots, in_{15})$, as in 3.4 of [5], or P = (in[0],..., in[15]) as in 5.1 of [5]

$C = (out_0,\ldots, out_{15})$, as in 3.4 of [5], or C = (out[0],..., out[15]) as in 5.1 of [5]

k = (k[0],..., k[15]), as in 5.2 of [5]

The plain AES decryption algorithm AES-DEC is a map:

$P := \text{AES-DEC}_k(C)$ (4.8)

where C is a 16-byte cipher text from DS, k is a 16-byte key from KS, and P is a 16-byte plain text from DS.

The mapping to the standard [5] is given as follows:

$C = (in_0,\ldots, in_{15})$, as in 3.4 of [5], or C = (in[0],..., in[15]) as in 5.3 of [5]

$P = (out_0,\ldots, out_{15})$, as in 3.4 of [5], or P = (out[0],..., out[15]) as in 5.3 of [5]

k = (k[0],..., k[15]), as in 5.2 of [5]

The main functional property of these two maps is

$P = \text{AES-DEC}_k(\text{AES-ENC}_k(P))$, for any k in KS and P in DS.

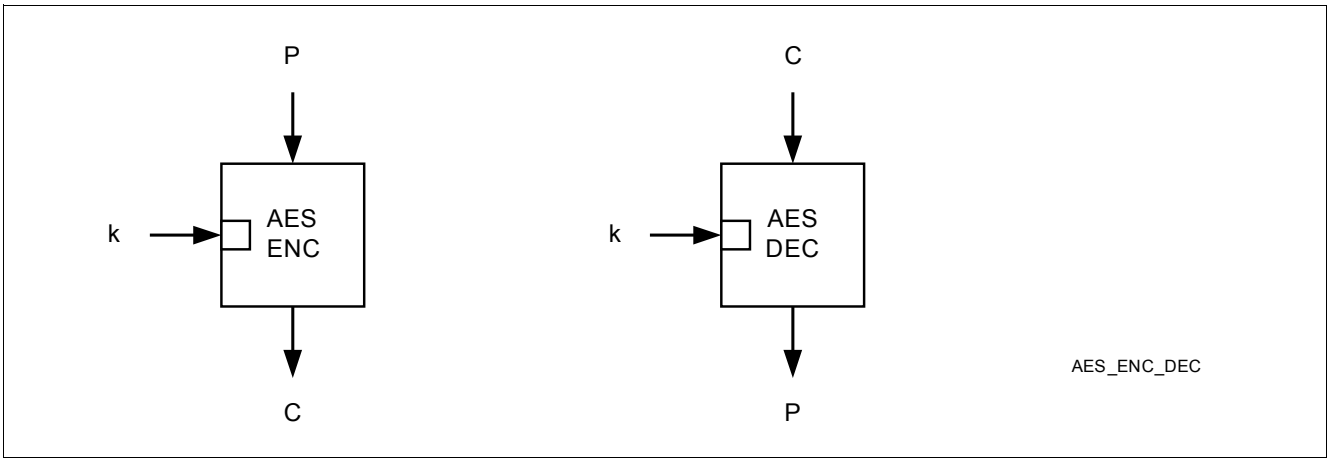


Figure 4-1 AES

4.2.2 ECB Mode

In the ECB mode, AES encrypts and decrypts not only one 128-bit data block P out of DS but rather several consecutive data blocks P_1,..., P_n into the cipher text blocks C_1,..., C_n or decrypts them back.

The ECB encryption, denoted by AES-ENC_{ECB} is defined by:

$$(C_1,..., C_n) := \text{AES-ENC}_{EBC,k}\,(P_1,..., P_n), \tag{4.9}$$

where n is an arbitrary positive integer and

$C_1 := \text{AES-ENC}_k\,(P_1)$,

...

$C_n := \text{AES-ENC}_k\,(P_n)$.

The ECB decryption, denoted by AES-DEC_{ECB} is defined by:

$$(P_1,..., P_n) := \text{AES-DEC}_{ECB,k}\,(C_1,..., C_n), \tag{4.10}$$

where n is an arbitrary positive integer and

$P_1 := \text{AES-DEC}_k\,(C_1)$,

...

$P_n := \text{AES-DEC}_k\,(C_n)$.

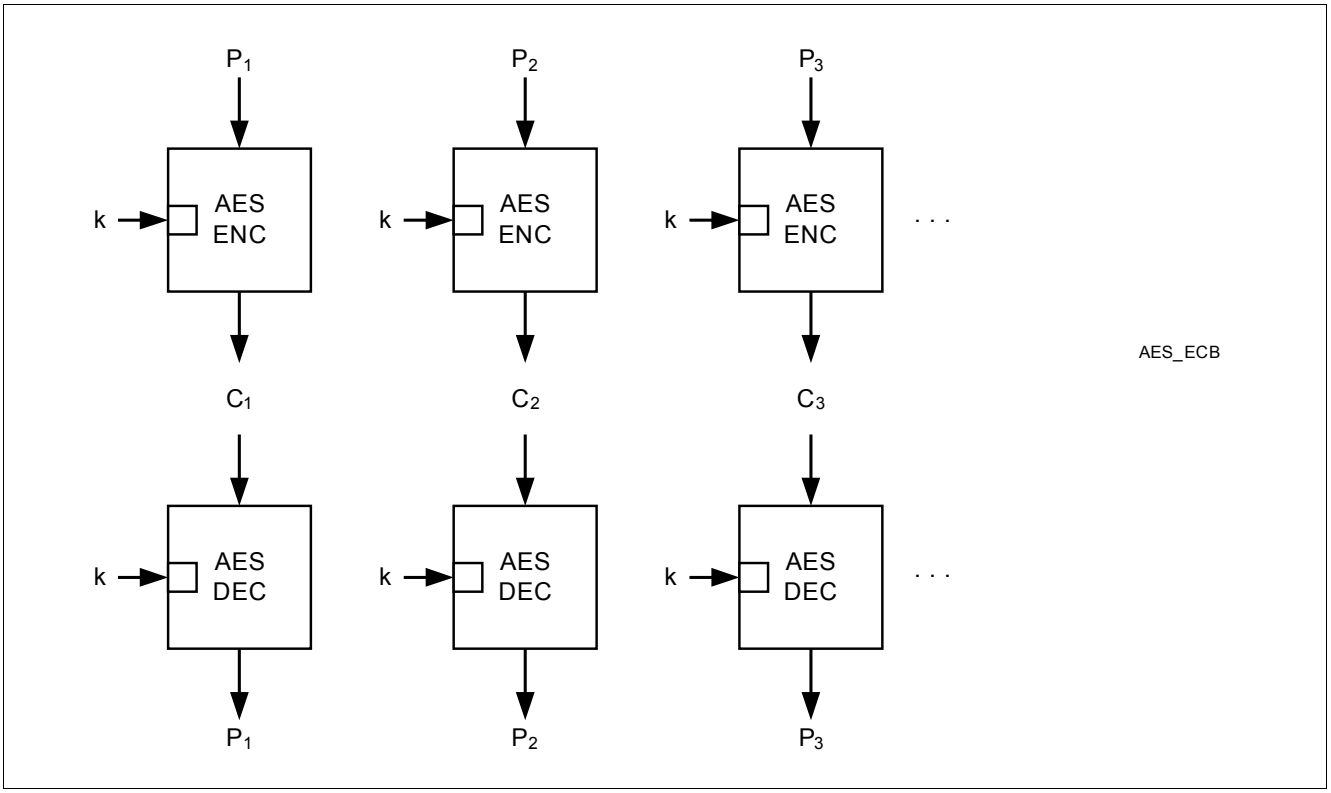


Figure 4-2 ECB mode

4.2.3 CBC Mode

In the CBC mode with initial value IV, AES encrypts and decrypts not only one 128-bit data block P out of DS but rather several consecutive data blocks P_1,..., P_n into the cipher text blocks C_1,..., C_n or decrypts them back. It depends not only on the private key k, but also on the initial value IV, which is a value from DS.

The CBC encryption, denoted by AES-ENC_{CBC} is defined by:

$$(C_1,..., C_n) := \text{AES-ENC}_{CBC,k,IV}\,(P_1,..., P_n), \quad (4.11)$$

where n is an arbitrary positive integer and

$C_0 := IV.$

$C_1 := \text{AES-ENC}_k\,(P_1 \text{ XOR } C_0),$

$C_2 := \text{AES-ENC}_k\,(P_2 \text{ XOR } C_1),$

...

$C_n := \text{AES-ENC}_k\,(P_n \text{ XOR } C_{n-1}).$

The CBC decryption, denoted by AES-DEC_{CBC} is defined by:

$$(P_1,..., P_n) := \text{AES-DEC}_{CBC,k,IV}\,(C_1,..., C_n), \quad (4.12)$$

where n is an arbitrary positive integer and

$C_0 := IV.$

P_1 := AES-DEC_k (C_1) XOR C_0,

P_2 := AES-DEC_k (C_2) XOR C_1,

...

P_n := AES-DEC_k (C_n) XOR C_{n-1}.

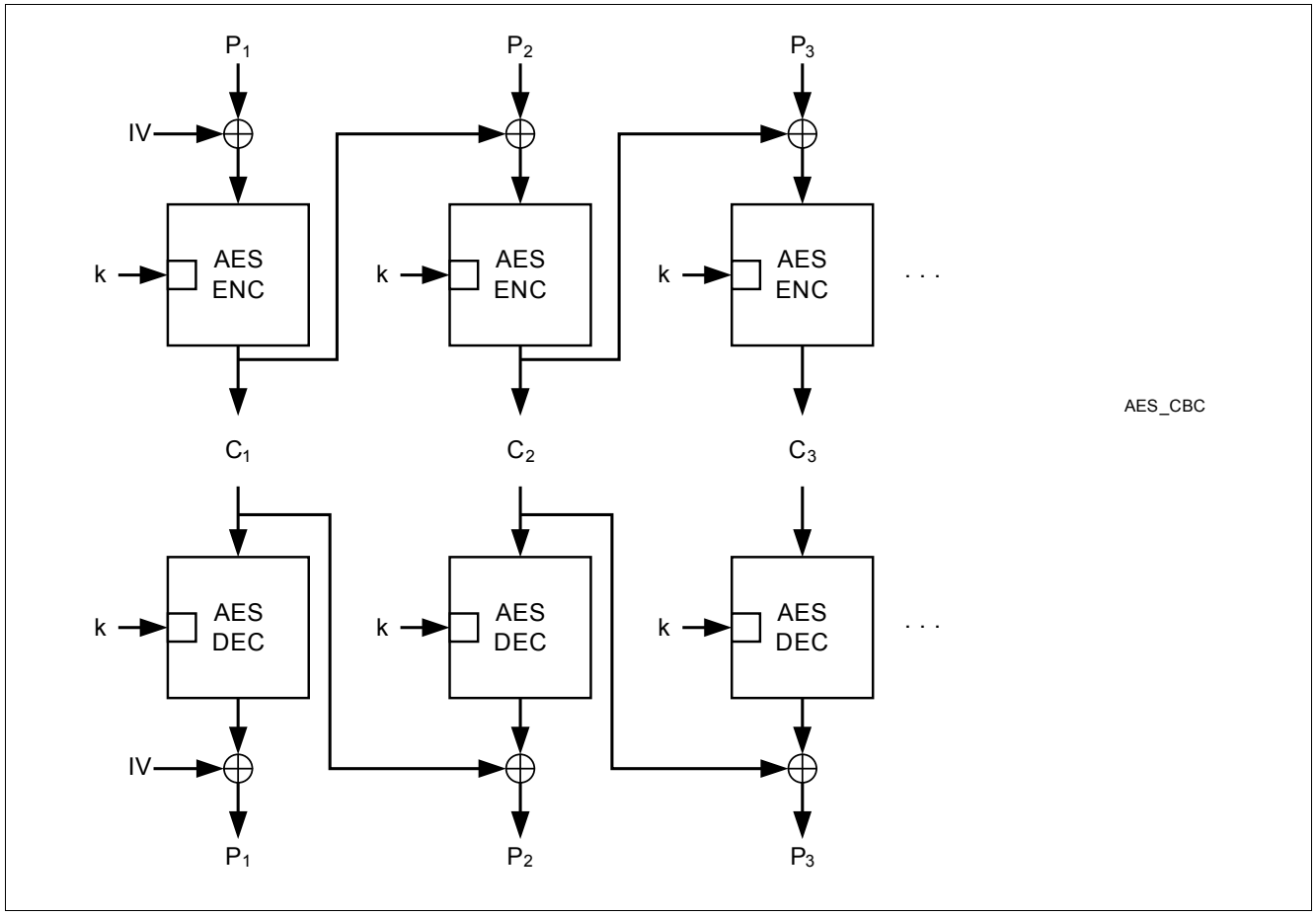


Figure 4-3 CBC mode

4.2.4 CTR Mode

In the CTR mode with initial value IV, AES encrypts and decrypts not only one 128-bit data block P out of DS but rather several consecutive data blocks P_1,..., P_n into the cipher text blocks C_1,..., C_n or decrypts them back. It depends not only on the private key k, but also on the initial value IV, which is a value from DS.

The CTR encryption, denoted by AES-ENC_{CTR} is defined by:

(C_1,..., C_n) := AES-$ENC_{CTR,k,IV}$ (P_1,..., P_n), (4.13)

where n is an arbitrary positive integer and

IV_1 := IV.

C_1 := AES-ENC_k (IV_1) XOR P_1,

IV_2 := INC32(IV_1),

C_2 := AES-ENC_k (IV_2) XOR P_2,

IV_3 := INC32(IV_2),

...

C_n := AES-ENC$_k$ (IV_n) XOR P_n.

INC_{32} denotes the incrementation in the lowest 32 bits modulo 2^{32}.

The CTR decryption, denoted by AES-DEC_{CTR} is defined by:

(P_1,..., P_n) := AES-$DEC_{CTR,k,IV}$ (C_1,..., C_n), (4.14)

where n is an arbitrary positive integer and

IV_1 := IV.

P_1 := AES-ENC$_k$ (IV_1) XOR C_1,

IV_2 := INC32(IV_1),

P_2 := AES-ENC$_k$ (IV_2) XOR C_2,

IV_3 := INC32(IV_2),

...

P_n := AES-ENC$_k$ (IV_n) XOR C_n.

INC_{32} denotes the incrementation in the lowest 32 bits modulo 2^{32}.

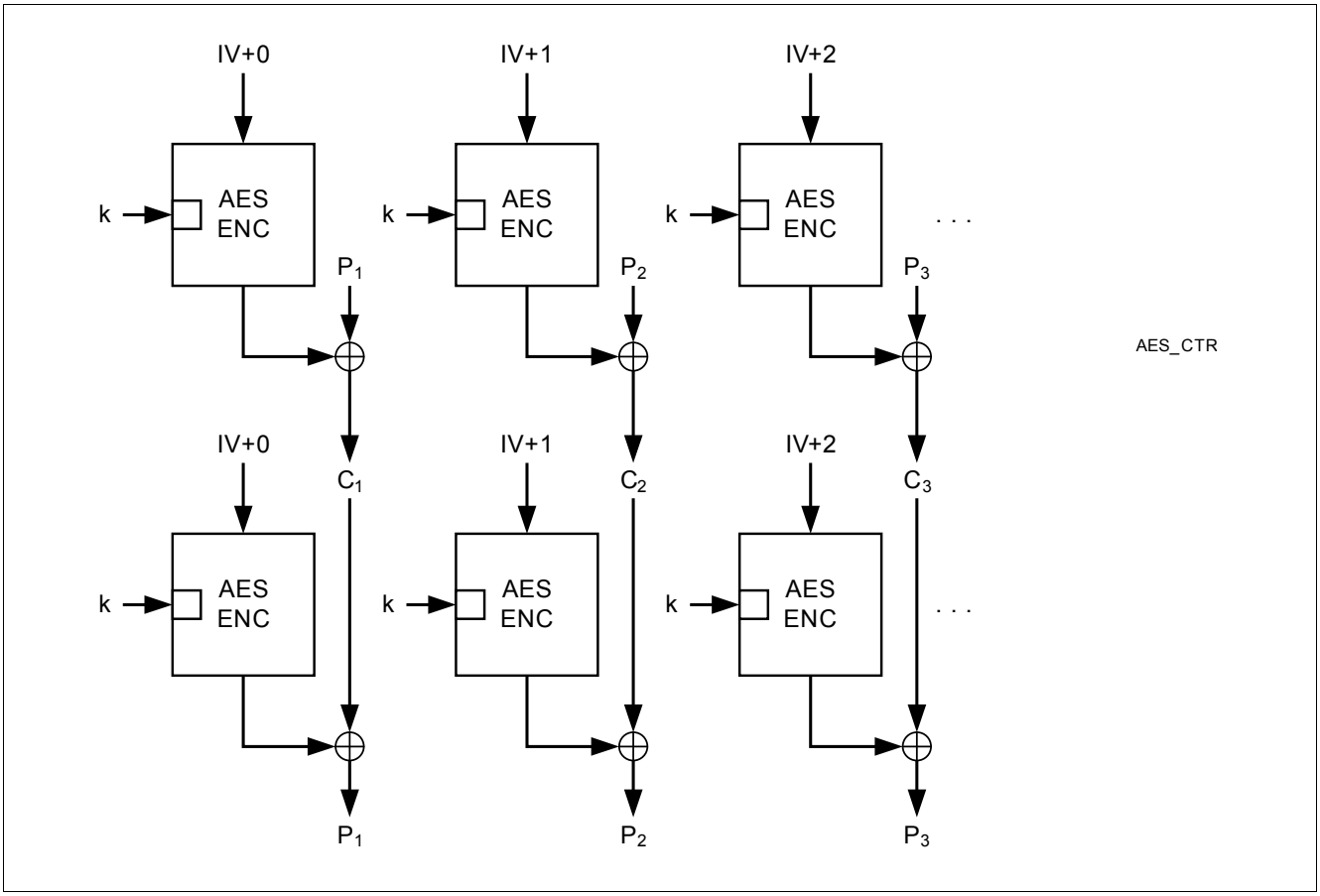


Figure 4-4 CTR mode

4.2.5 OFB Mode

In the OFB mode with initial value IV, AES encrypts and decrypts not only one 128-bit data block P out of DS but rather several consecutive data blocks P_1,..., P_n into the cipher text blocks C_1,..., C_n or decrypts them back. It depends not only on the private key k, but also on the initial value IV, which is a value from DS.

The OFB encryption, denoted by AES-ENC_{OFB} is defined by:

$$(C_1,..., C_n) := \text{AES-ENC}_{OFB,k,IV}\,(P_1,..., P_n), \quad (4.15)$$

where n is an arbitrary positive integer and

$CV_0 := IV,$
$CV_1 := \text{AES-ENC}_k\,(CV_0),$
$C_1 := CV_1 \text{ XOR } P_1,$
$CV_2 := \text{AES-ENC}_k\,(CV_1),$
$C_2 := CV_2 \text{ XOR } P_2,$
$CV_3 := \text{INC}(CV_2),$
...
$CV_n := \text{AES-ENC}_k\,(CV_{n-1}),$
$C_n := CV_n \text{ XOR } P_n,$

Attention: Do not mix up the values CV_i with the later defined registers CV[0],...! The latter ones might contain the former ones, but one has to distinguish between both notations!

The OFB decryption, denoted by $AES\text{-}DEC_{OFB}$ is defined by:

$$(P_1,..., P_n) := AES\text{-}DEC_{OFB,k,IV}\,(C_1,..., C_n), \quad (4.16)$$

where n is an arbitrary positive integer and

$CV_0 := IV,$
$CV_1 := AES\text{-}ENC_k\,(CV_0),$
$P_1 := CV_1 \text{ XOR } C_1,$
$CV_2 := AES\text{-}ENC_k\,(CV_1),$
$P_2 := CV_2 \text{ XOR } C_2,$
$CV_3 := AES\text{-}ENC_k\,(CV_2),$
...
$CV_n := AES\text{-}ENC_k\,(CV_{n-1}),$
$P_n := CV_n \text{ XOR } C_n,$

Attention: Do not mix up the values CV_i with the later defined registers CV[0],...! The latter ones might contain the former ones, but one has to distinguish between both notations!

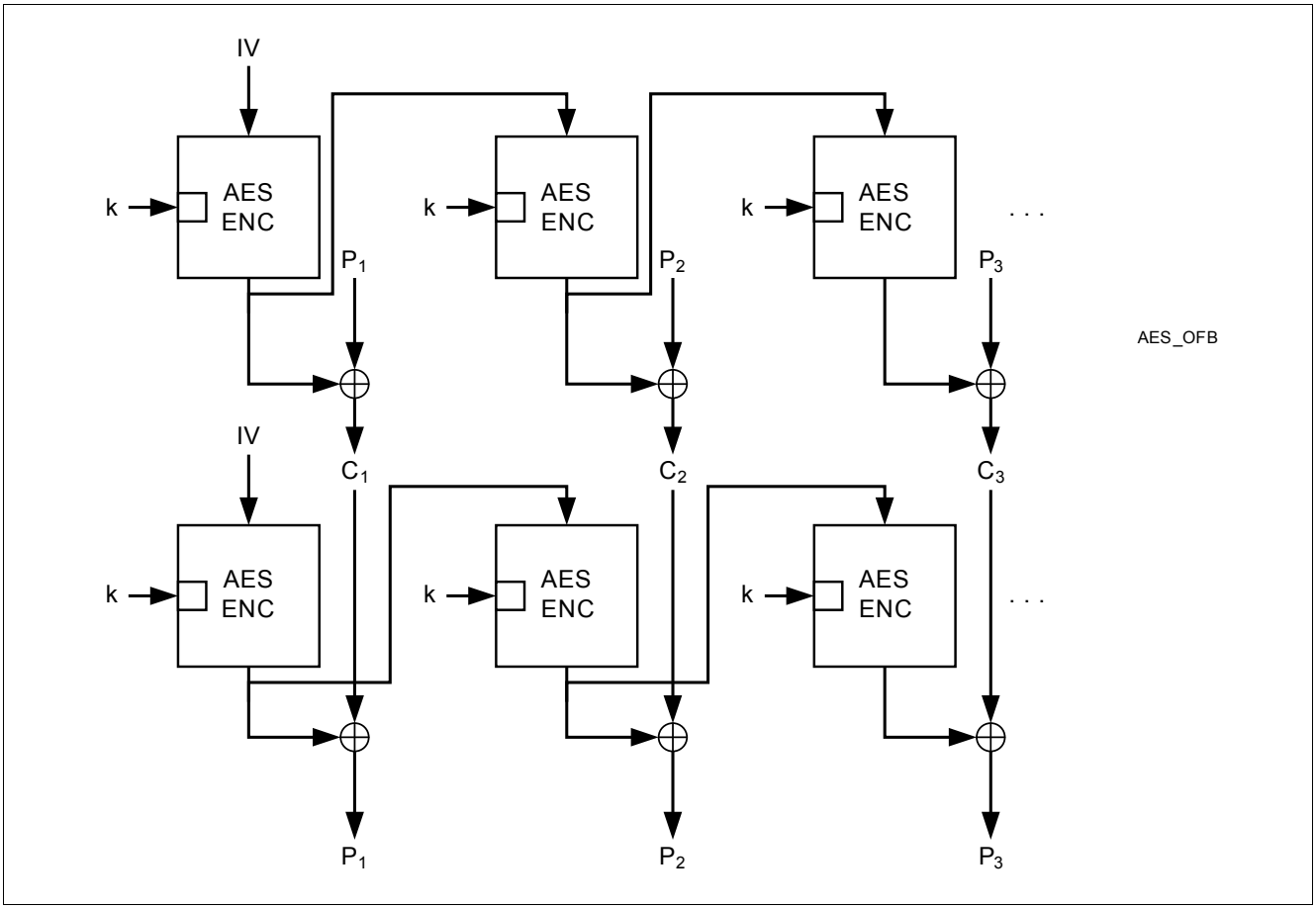


Figure 4-5 OFB mode

4.2.6 CFB Mode

In the CFB mode with initial value IV, AES encrypts and decrypts not only one 128-bit data block P out of DS but rather several consecutive data blocks P_1,..., P_n into the cipher text blocks C_1,..., C_n or decrypts them back. It depends not only on the private key k, but also on the initial value IV, which is a value from DS.

The CFB encryption, denoted by AES-ENC_{CFB} is defined by:

$$(C_1,..., C_n) := \text{AES-ENC}_{CFB,k,IV}\,(P_1,..., P_n), \tag{4.17}$$

where n is an arbitrary positive integer and

$CV_0 := IV$,

$C_1 := CV_1 := \text{AES-ENC}_k\,(CV_0)$ XOR P_1,

$C_2 := CV_2 := \text{AES-ENC}_k\,(CV_1)$ XOR P_2,

...

$C_n := CV_n := \text{AES-ENC}_k\,(CV_{n-1})$ XOR P_n.

Attention: Do not mix up the values CV_i with the later defined registers CV[0],...! The latter ones might contain the former ones, but one has to distinguish between both notations!

The CFB decryption, denoted by AES-DEC_{CFB} is defined by:

$(P_1,..., P_n) := \text{AES-DEC}_{CFB,k,IV}(C_1,..., C_n)$, (4.18)

where n is an arbitrary positive integer and

$C_0 := IV$,
$CV_1 := C_0$,
$P_1 := \text{AES-ENC}_k(CV_1)$ XOR C_1,
$CV_2 := C_1$,
$P_2 := \text{AES-ENC}_k(CV_2)$ XOR C_2,
$CV_3 := C_2$,
...
$P_n := \text{AES-ENC}_k(CV_n)$ XOR C_n,

Attention: Do not mix up the values CV_i with the later defined registers CV[0],...! The latter ones might contain the former ones, but one has to distinguish between both notations!

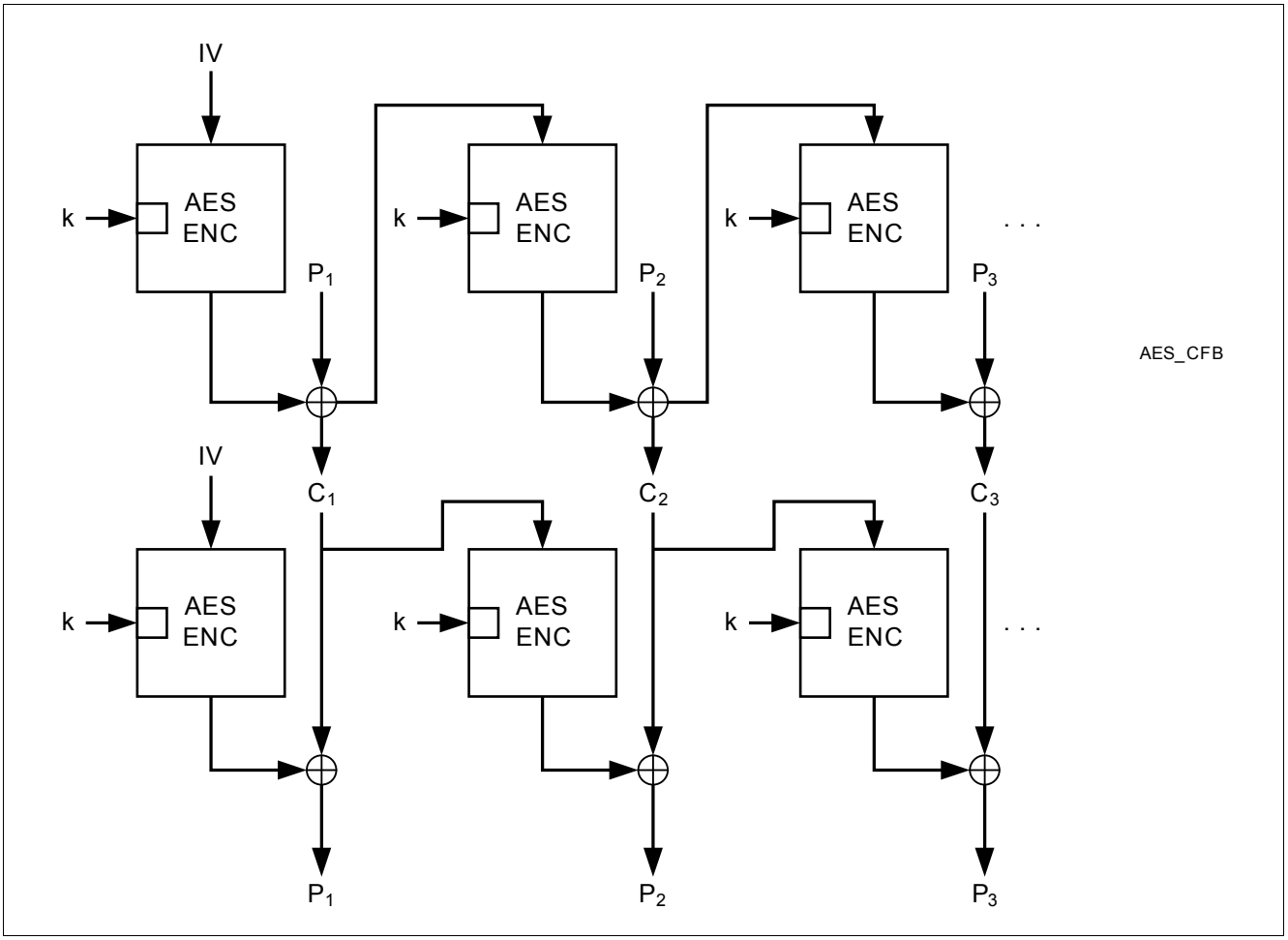


Figure 4-6 CFB mode

4.2.7 GCM Mode

The GCM mode [8] is a combination of the CTR mode together with the GHASH. In the GCM mode with initial value IV, AES encrypts and decrypts not only one 128-bit data block P out of DS but rather several consecutive data blocks P_1,..., P_n into the cipher text blocks C_1,..., C_n or decrypts them back. It depends not only on the private key k, but also on the initial value IV, which is a value from DS. Additionally, the encryption operation computes an authentication tag T which additionally depends on additional data blocks A_1,..., A_m which will not be encrypted. The decryption not only decrypts the cipher text blocks C_1,..., C_n but also checks the authentication tag information T. It is assumed, that the blocks P_1,..., P_n and A_1,..., A_m are already full 128-bit blocks (maybe by the padding described in [8]).

The GCM encryption, denoted by AES-ENC_{GCM} is defined by:

$$(C_1,..., C_n,T) := \text{AES-ENC}_{GCM, k, IV}\ (A_1,..., A_m, P_1,..., P_n), \tag{4.19}$$

where n, m are arbitrary non-negative integers and the blocks are computed in the following way:

$H := \text{AES-ENC}_k\ (0^{128})$

J_0 := IV (or the preparation according to [8])

$Y_0 := 0$

$Y_1 := (Y_0 \text{ XOR } A_1)*H \bmod f$

$Y_2 := (Y_1 \text{ XOR } A_2)*H \bmod f$

. . .

$Y_m := (Y_{m-1} \text{ XOR } A_m)*H \bmod f$

$CV_1 := \text{inc}(J_0)$

$C_1 := P_1 \text{ XOR AES-ENC}_k\ (CV_1)$

$Y_{m+1} := (Y_m \text{ XOR } C_1)*H \bmod f$

$CV_2 := \text{inc}(CV_1)$

$C_2 := P_2 \text{ XOR AES-ENC}_k\ (CV_2)$

$Y_{m+2} := (Y_{m+1} \text{ XOR } C_2)*H \bmod f$

. . .

$CV_n := \text{inc}(CV_{n-1})$

$C_n := P_n \text{ XOR AES-ENC}_k\ (CV_n)$

$Y_{m+n} := (Y_{m+n-1} \text{ XOR } C_n)*H \bmod f$

$S := (Y_{m+n} \text{ XOR } L)*H \bmod f$

$T := S \text{ XOR AES-ENC}_k\ (J_0)$

Here, the block L consists of length information about the blocks A_1, ..., A_m and P_1, ..., P_n.

The GCM decryption, denoted by AES-DEC_{GCM} is defined by:

$$(P_1,..., P_n,T') := \text{AES-DEC}_{GCM, k, IV}\ (A_1,..., A_m, C_1,..., C_n), \tag{4.20}$$

where n, m are arbitrary positive integers and the blocks are computed in the following way:

$H := \text{AES-ENC}_k\ (0^{128})$

J_0 := IV (or the preparation according to [8])

Y_0 := 0

Y_1 := (Y_0 XOR A_1)*H mod f

Y_2 := (Y_1 XOR A_2)*H mod f

. . .

Y_m := (Y_{m-1} XOR A_m)*H mod f

Y_{m+1} := (Y_m XOR C_1)*H mod f

CV_1 := inc(J_0)

P_1 := C_1 XOR AES-ENC$_k$ (CV_1)

Y_{m+2} := (Y_{m+1} XOR C_2)*H mod f

CV_2 := inc(CV_1)

P_2 := C_2 XOR AES-ENC$_k$ (CV_2)

. . .

Y_{m+n} := (Y_{m+n-1} XOR C_n)*H mod f

CV_n := inc(CV_{n-1})

P_n := C_n XOR AES-ENC$_k$ (CV_n)

S := (Y_{m+n} XOR L)*H mod f

T' := S XOR AES-ENC$_k$ (J_0)

Here, the block L consists of length information about the blocks A_1, ..., A_m and C_1, ..., C_n.

Now, T can be compared with T'. If the values are equal, the cipher text blocks are authenticated, otherwise rejected.

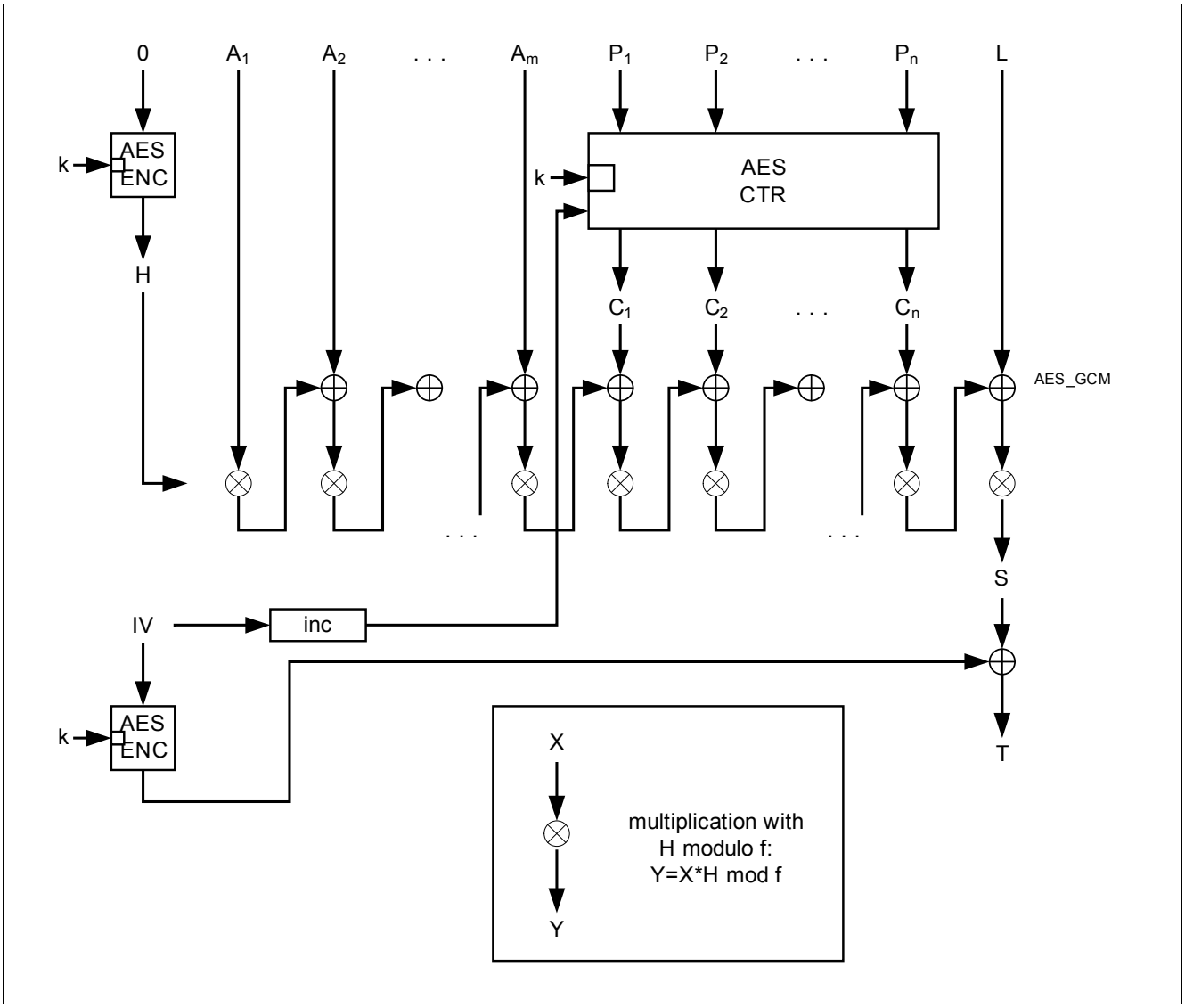


Figure 4-7 GCM mode

Attention: Note that the GCM-Mode usually is also defined in the case that the last Block P_n is not a complete 128 bit block. Then P_n has to be padded with zeros, and also C_n has to be padded with zeros. This more general mode is not covered in this specificationof the AES module.

4.2.8 XTS Mode

The XTS encryption **[9]** of a single 128-bit data block P, which has the sequential number j within a data unit, using a 256-bit key k=k1||k_2 and the (128-bit) tweak value i, is denoted by:

C := AES-$ENC_{XTS,k,i,j}$ (P),

also in **[9]** written as

C := XTS-AES-blockEnc(k, P, i, j).

C is computed by the following way:

T := (AES-ENC_{k2}(i) * (X^j)) mod f,

C := AES-ENC$_{k1}$ (P XOR T) XOR T,

The XTS decryption of a single 128-bit data block C, which has the sequential number j within a data unit, using a 256-bit key k=k1||k$_2$ and the tweak value i, is denoted by:

P := AES-DEC$_{XTS,k,i,j}$ (C).

also in **[9]** written as

P := XTS-AES-blockDec(k, C, i, j).

P is computed by the following way:

T := (AES-ENC$_{k2}$(i) * (X^j)) mod f,

P := AES-DEC$_{k1}$ (C XOR T) XOR T,

Note that for consecutive blocks, i.e., with sequential number j and j+1 within the data unit, the tweak value T for the block j+1 can be computed from the tweak value of block j by simply multiplying it with X mod f.

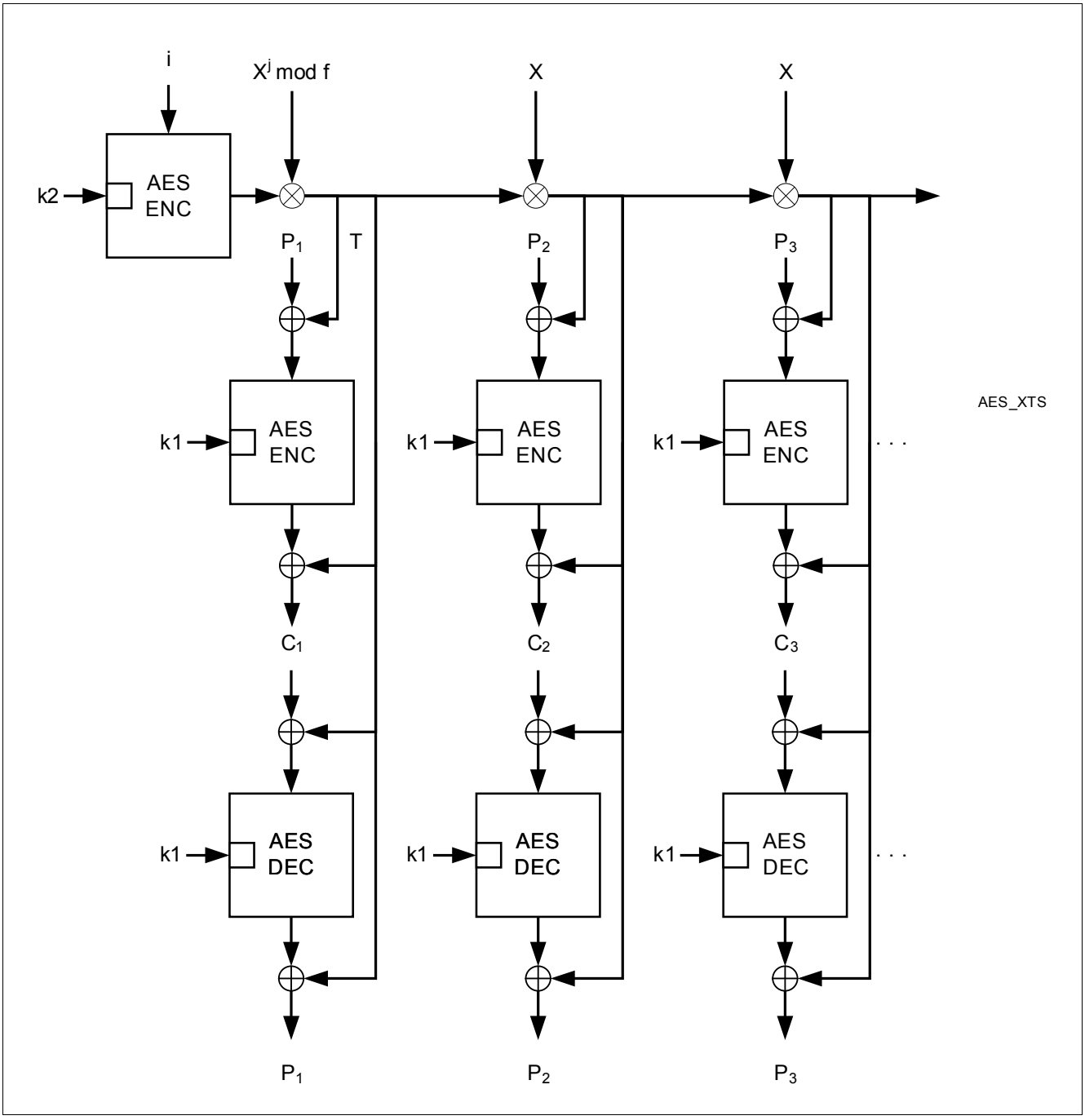


Figure 4-8 XTS mode

4.3 Register Architecture

The registers of the AES module, which are relevant for the programer, can be classified into two categories, namely the SFR-registers which are directly accessible by the CPU and the internal registers which are only accessible indirectly via the SFR-registers.

The SFR-registers are:

- The control register AESCTRL which is a 32-bit word register.
- The status register AESSTAT which is a 32-bit word register.
- The input register AESIN which consists of 4 32-bit word registers.
- The output register AESOUT which consists of 4 32-bit word registers.
- The second output register AESOUTSAVE which consists of 4 32-bit word registers.

The internal registers are:

- The 8 key registers K0, K1, ..., K7 which are each 128 bits wide.
- The 5 chaining variable registers CV0, CV1, CV2, CV3, CV4 which are each 128 bits wide.

After reset, these 8+5 registers have the value 0.

The operation of the AES module will be controlled solely by writing to the AESCTRL register. By virtue of this the following operations are manageable:

- Copying of a key value from AESIN to some Kx (x=0,...,7) register.
- Locking of the keys K0 or K1, such that they are not writable any more.
- Copying of an IV value from AESIN to some CVy (y=0,1,2,3,4).
- Copying of a CV value from some CVy to AESOUT.
- Cryptographically processing of the input in AESIN, using at most one of the specified keys and chaining variables and outputting it in AESOUT.
- Saving of the content of AESOUT into AESOUTSAVE. This is only an indirect process which will happen automatically if a process is triggered that will overwrite AESOUT.

Usually an en-/decryption operation is done in the following way: The key is loaded into some Ky, an IV is loaded into some CVy, then AESIN is filled with the input for the cryptographic operation. By writing AESCTRL the operation is triggered. Waiting for AES to finish the operation and then reading the output out of AESOUT.

Important for pipelined operation of the module are the following facts:

- The AESCTRL register is always accessible. If the system (CPU) writes to the AESCTRL register while an operation takes place, i.e., while AESSTAT.BSY is set, then the AES module will generate a wait signal for the accessing bus, halting the write process until the operation is finished and the new value can be written into AESCTRL.
- The AESSTAT register is always accessible.
- The input register is always accessible. After the start of an encryption operation, the input register AESIN is idle and ready for being filled with the next input value.
- After the start of an encryption operation and before the end of this operation (indicated by the busy flag in AESSTAT) the AESOUT is not accessible. A read access during this time will be stalled until AESSTAT.BSY is cleared. For pipelined usage, the AESOUTSAVE register has to be used: The first action of a cryptographic operation will be the copying of the content of AESOUT to AESOUTSAVE.

Furthermore note: If interrupts may occur that also access the AES module, the software must make sure that these interrupts may not disturb the former AES operation by overwriting input or output values. So, either

these interrupts will be disabled, or the it is assured that the old values will be restored after such an intermediary interrupt.

4.4 AES SFRs

There are 14 registers in the AES module, the AESCTRL, AESSTAT, AESIN0, AESIN1, AESIN2, AESIN3, AESOUT0, AESOUT1, AESOUT2, AESOUT3, AESOUTSAVE0, AESOUTSAVE1, AESOUTSAVE2, AESOUTSAVE3.

Table 4-1 Register Address Space

Module	Base Address	End Address	Note
AES	E800 0000$_H$	E800 03FF$_H$	

Table 4-2 Register Overview

Register Short Name	Register Long Name	Offset Address	Reset Value
AES SFRs, Control and Status Registers			
AESCTRL	AES Control Register	10$_H$	0001 3000$_H$
AESSTAT	AES Status Register	14$_H$	0000 0006$_H$
AES SFRs, Input Registers			
AESIN0	AES INPUT register 0	00$_H$	0000 0000$_H$
AESIN1	AES INPUT register 1	04$_H$	0000 0000$_H$
AESIN2	AES INPUT register 2	08$_H$	0000 0000$_H$
AESIN3	AES INPUT register 3	0C$_H$	0000 0000$_H$
AES SFRs, Output Registers			
AESOUT0	AES OUTPUT register 0	20$_H$	0000 0000$_H$
AESOUT1	AES OUTPUT register 1	24$_H$	0000 0000$_H$
AESOUT2	AES OUTPUT register 2	28$_H$	0000 0000$_H$
AESOUT3	AES OUTPUT register 3	2C$_H$	0000 0000$_H$
AESOUTSAVE0	AES OUTPUT save register 0	30$_H$	0000 0000$_H$
AESOUTSAVE1	AES OUTPUT save register 1	34$_H$	0000 0000$_H$
AESOUTSAVE2	AES OUTPUT save register 2	38$_H$	0000 0000$_H$
AESOUTSAVE3	AES OUTPUT save register 3	3C$_H$	0000 0000$_H$

The registers are addressed wordwise.

Registers can only be accessed 32-bit wise and 32-bit aligned. Using 8-bit, 16-bit or un-aligned accesses lead to a bus error.

Access to unused addresses results in a bus error.

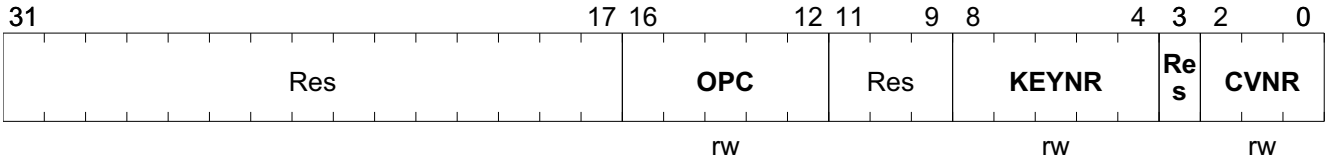
Write access to read only registers results in a bus error.

4.4.1 Control and Status Registers

AESCTRL

This register triggers the operations of the AES module. In the description, IN denotes the input to the AES-operation which was loaded to the registers AESIN0,..., AESIN3. OUT denotes the output of the AES-operation which will be stored in AESOUT0,...,AESOUT3. OUTSAVE denotes the old Output of the former operation. Commands that are not defined, will trigger no operation (NOP).

AESCTRL	Offset	Reset Value
AES Control Register	10_H	$0001\ 3000_H$



Field	Bits	Type	Description
OPC	16:12	rw	**Operation Code** This field defines the operation of the AES module. For an in-depth description of the commands, see the table below. *Note: An opcode with unspecified OPC will be ignored.* 00_H **ECB-ENC**, encrypts input with ECB mode 01_H **ECB-DEC**, decrypts input with ECB mode 02_H **CBC-ENC**, encrypts input with CBC mode 03_H **CBC-DEC**, decrypts input with CBC mode 04_H **CTR**, en- and decrypts input with CTR mode 05_H **OFB**, en- and decrypts input with OFB mode 06_H **CFB-ENC**, encrypts input with CFB mode 07_H **CFB-DEC**, decrypts input with CFB mode 08_H **GCM-ENC**, encrypts input with GCTR mode and G-hashes OUT 09_H **GCM-DEC**, decrypts input with GCTR mode and G-hashes IN $0A_H$ **GCM-MAC**, G-hashes the input $0B_H$ **XTS-ENC**, encrypts with XTS mode $0C_H$ **XTS-DEC**, decrypts with XTS mode 10_H **WK**, writes input to selected key register and resets IN 11_H **LK**, locks the key registers K0 or K1 such that they are not writable anymore: Key is specified by KEYNR and works only with KEYNR=0 or 1, otherwise the command will be ignored. Furthermore LOCK0 or LOCK1 (in AESSTAT) is set to 1. 12_H **WCV**, writes input to the selected chaining variable register 13_H **RCV**, copies selected CV register to output register

Field	Bits	Type	Description
KEYNR	8:4	rw	**Key Number** This bit field defines the number of the key register for the selected operation. Valid keys are represented by 00_H up to 07_H. All values between 08_H and $1E_H$ are rfu. *Note: An opcode with invalid KEYNR will be ignored.* $1F_H$ **OLDKEY**, uses the key of the last AES operation, but only for the cryptographic operations – i.e., only for OPC < $0D_H$, with the exception of $0A_H$. This functionality can only work properly provided it is preceded by one of the operations referred to above (using a valid key). Moreover, the key must not be overwritten since the associated operation was performed.
CVNR	2:0	rw	**Chaining Variable Number** This bit field defines the number of the chaining variable register for the selected operation. *Note: An opcode with invalid CVNR will be ignored.* 0_D **CVNR0**, Depending on the OpCode, Chaining Variable CV[0] will be used 1_D **CVNR1**, Depending on the OpCode, Chaining Variable CV[1] will be used 2_D **CVNR2**, Depending on the OpCode, Chaining Variable CV[2] will be used 3_D **CVNR3**, Depending on the OpCode, Chaining Variable CV[3] will be used 4_D **CVNR4**, Depending on the OpCode, Chaining Variable CV[4] will be used

ECB-ENC

OUTSAVE <-- OUT,
OUT <-- AES-$ENC_{K[KEYNR]}$(IN)

ECB-DEC

OUTSAVE <-- OUT,
OUT <-- AES-$DEC_{K[KEYNR]}$(IN)

CBC-ENC

OUTSAVE <-- OUT,
OUT <-- AES-$ENC_{K[KEYNR]}$(IN XOR CV[CVNR])
CV[CVNR] <-- OUT

CBC-DEC

OUTSAVE <-- OUT,
OUT <-- AES-DEC$_{K[KEYNR]}$(IN) XOR CV[CVNR],
CV[CVNR] <-- IN

CTR

OUTSAVE <-- OUT,
OUT <-- AES-ENC$_{K[KEYNR]}$(CV[CVNR]) XOR IN
CV[CVNR] <-- INC$_{32}$(CV[CVNR]).

OFB

OUTSAVE <-- OUT,
CV[CVNR] <-- AES-ENC$_{K[KEYNR]}$(CV[CVNR])
OUT <-- CV[CVNR] XOR IN.

CFB-ENC

OUTSAVE <-- OUT
OUT <--CV[CVNR] <-- AES-ENC$_{K[KEYNR]}$(CV[CVNR]) XOR IN

CFB-DEC

OUTSAVE <-- OUT,
OUT <-- AES-ENC$_{K[KEYNR]}$(CV[CVNR]) XOR IN,
CV[CVNR] <-- IN.

GCM-ENC

OUTSAVE <-- OUT,
OUT <-- AES-ENC$_{K[KEYNR]}$(CV[0]) XOR IN
CV[0] <-- INC$_{32}$(CV[0]),
CV[2] <-- (CV[2] XOR OUT) * CV[1] mod f(X).

GCM-DEC

OUTSAVE <-- OUT,
CV[2] <-- (CV[2] XOR IN) * CV[1] mod f(X),
OUT <-- AES-ENC$_{K[KEYNR]}$(CV[0]) XOR IN,
CV[0] <-- INC$_{32}$(CV[0]).

GCM-MAC

OUTSAVE <-- OUT,
OUT <-- IN,
CV[2] <-- (CV[2] XOR IN) * CV[1] mod f(X) .

XTS-ENC

OUTSAVE <-- OUT,
OUT <-- AES-ENC$_{K[KEYNR]}$(IN XOR CV[CVNR]) XOR CV[CVNR],
CV[CVNR] <-- CV[CVNR] * X mod f(X).

XTS-DEC

OUTSAVE <-- OUT,
OUT <-- AES-DEC$_{K[KEYNR]}$(IN XOR CV[CVNR]) XOR CV[CVNR],
CV[CVNR] <-- CV[CVNR] * X mod f(X).

WK

if (KEYNR>2) or (KEYNR=1 and AESSTAT.LOCK1=0) or (KEYNR=0 and AESSTAT.LOCK0=0) then
K[KEYNR] <-- IN
end if
IN <-- 0

LK

-

WCV

CV[CVNR] <-- IN

RCV

OUTSAVE <-- OUT
OUT <-- CV[CVNR]

AESSTAT

The AESSTAT register contains information whether AES is busy and the keys K0 and K1 are locked..

AESSTAT	Offset	Reset Value
AES Status Register	14_H	$0000\ 0006_H$

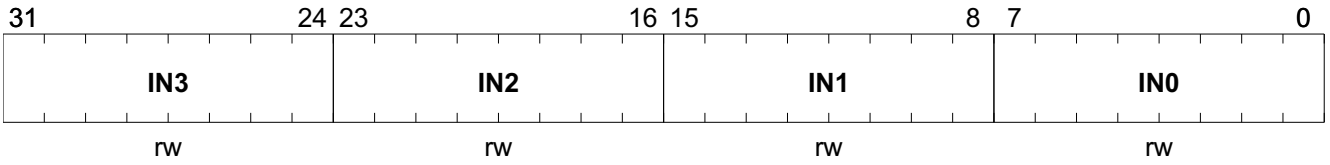
Field	Bits	Type	Description
LOCK1	2	r	**Lock Key1** 0_H **K1WENABLED**, Key K1 can be written again. 1_H **K1LOCKED**, Key K1 is locked, i.e., K1 can not be written again.
LOCK0	1	r	**Lock Key0** 0_H **K0WENABLED**, Key K0 can be written again. 1_H **K0LOCKED**, Key K0 is locked, i.e., K0 can not be written again.
BSY	0	r	**Busy flag** 0_H **idle**, The AES module is idle. Input data can be sent to AESIN, Output data can be read out of AESOUT or AESOUTSAVE, or an operation can be triggered by sending a command to the AESCTRL register. 1_H **busy**, The AES module is working. Future Input data can be sent to AESIN. A command sent to the AESCRTL and reading of the AESOUTx registers will be stalled until the running operation is finished.

4.4.2 Input Registers

AESIN0

The AESIN0 register contains part of a key, IV, plain text or cipher text.

AESIN0	Offset	Reset Value
AES INPUT register 0	00_H	$0000\ 0000_H$

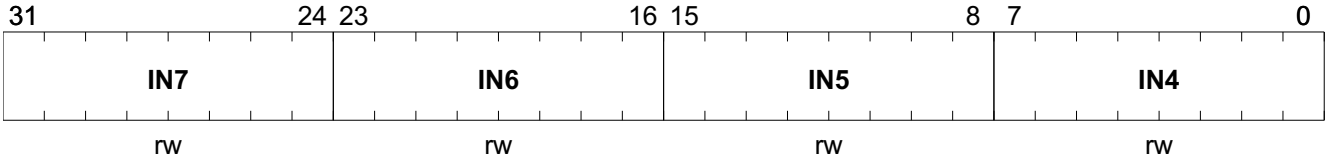


Field	Bits	Type	Description
IN3	31:24	rw	**Input Byte 3** This byte represents the byte 3 of a key, IV, plain text or cipher text.
IN2	23:16	rw	**Input Byte 2** This byte represents the byte 2 of a key, IV, plain text or cipher text.
IN1	15:8	rw	**Input Byte 1** This byte represents the byte 1 of a key, IV, plain text or cipher text.
IN0	7:0	rw	**Input Byte 0** This byte represents the byte 0 of a key, IV, plain text or cipher text.

AESIN1

The AESIN1 register contains part of a key, IV, plain text or cipher text.

AESIN1	Offset	Reset Value
AES INPUT register 1	04_H	$0000\ 0000_H$

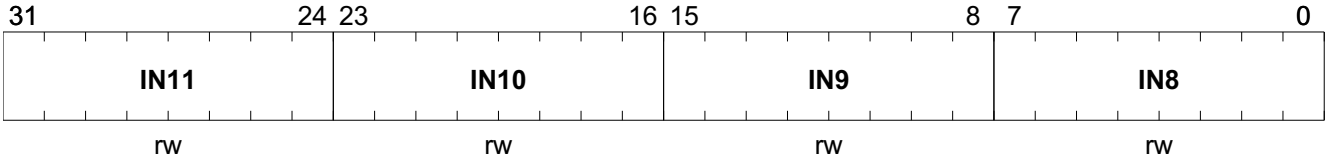


Field	Bits	Type	Description
IN7	31:24	rw	**Input Byte 7** This byte represents the byte 7 of a key, IV, plain text or cipher text.
IN6	23:16	rw	**Input Byte 6** This byte represents the byte 6 of a key, IV, plain text or cipher text.
IN5	15:8	rw	**Input Byte 5** This byte represents the byte 5 of a key, IV, plain text or cipher text.
IN4	7:0	rw	**Input Byte 4** This byte represents the byte 4 of a key, IV, plain text or cipher text.

AESIN2

The AESIN2 register contains part of a key, IV, plain text or cipher text.

AESIN2	Offset	Reset Value
AES INPUT register 2	08_H	$0000\ 0000_H$

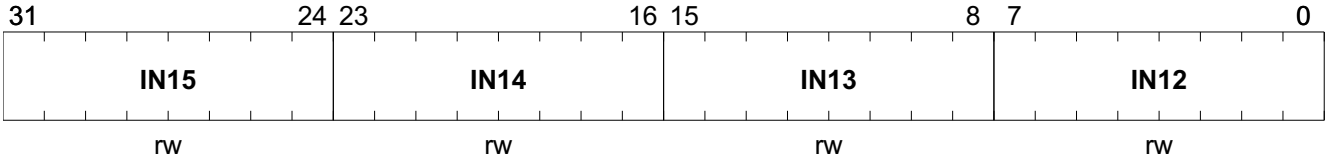


Field	Bits	Type	Description
IN11	31:24	rw	**Input Byte 11** This byte represents the byte 11 of a key, IV, plain text or cipher text.
IN10	23:16	rw	**Input Byte 10** This byte represents the byte 10 of a key, IV, plain text or cipher text.
IN9	15:8	rw	**Input Byte 9** This byte represents the byte 9 of a key, IV, plain text or cipher text.
IN8	7:0	rw	**Input Byte 8** This byte represents the byte 8 of a key, IV, plain text or cipher text.

AESIN3

The AESIN2 register contains part of a key, IV, plain text or cipher text.

AESIN3	Offset	Reset Value
AES INPUT register 3	$0C_H$	$0000\ 0000_H$



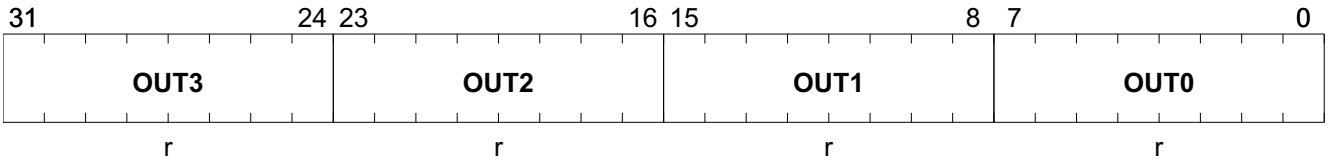
Field	Bits	Type	Description
IN15	31:24	rw	**Input Byte 15** This byte represents the byte 15 of a key, IV, plain text or cipher text.
IN14	23:16	rw	**Input Byte 14** This byte represents the byte 14 of a key, IV, plain text or cipher text.
IN13	15:8	rw	**Input Byte 13** This byte represents the byte 13 of a key, IV, plain text or cipher text.
IN12	7:0	rw	**Input Byte 12** This byte represents the byte 12 of a key, IV, plain text or cipher text.

4.4.3 Output Registers

AESOUT0

The AESOUT0 register contains part of a plain text or cipher text.

AESOUT0	Offset	Reset Value
AES OUTPUT register 0	20_H	$0000\ 0000_H$

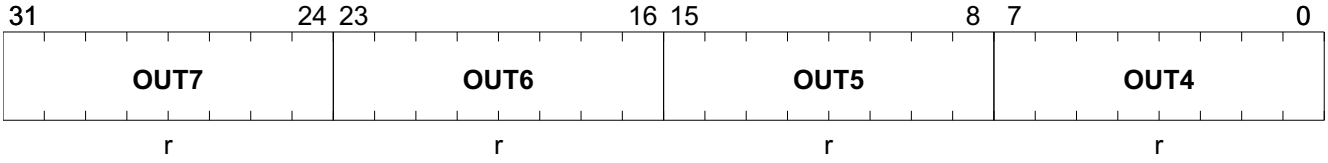


Field	Bits	Type	Description
OUT3	31:24	r	**Output Byte 3** This byte represents the byte 3 of the output.
OUT2	23:16	r	**Output Byte 2** This byte represents the byte 2 of the output.
OUT1	15:8	r	**Output Byte 1** This byte represents the byte 1 of the output.
OUT0	7:0	r	**Output Byte 0** This byte represents the byte 0 of the output.

AESOUT1

The AESOUT1 register contains part of a plain text or cipher text.

AESOUT1	Offset	Reset Value
AES OUTPUT register 1	24_H	$0000\ 0000_H$

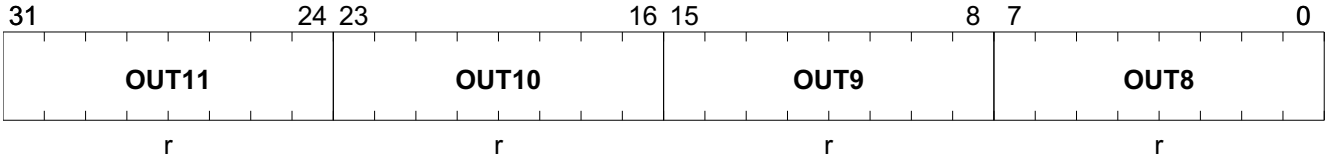


Field	Bits	Type	Description
OUT7	31:24	r	**Output Byte 7** This byte represents the byte 7 of the output.
OUT6	23:16	r	**Output Byte 6** This byte represents the byte 6 of the output.
OUT5	15:8	r	**Output Byte 5** This byte represents the byte 5 of the output.
OUT4	7:0	r	**Output Byte 4** This byte represents the byte 4 of the output.

AESOUT2

The AESOUT2 register contains part of a plain text or cipher text.

AESOUT2	Offset	Reset Value
AES OUTPUT register 2	28_H	$0000\ 0000_H$

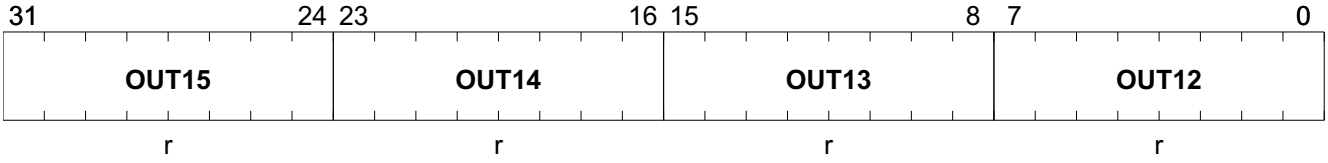


Field	Bits	Type	Description
OUT11	31:24	r	**Output Byte 11** This byte represents the byte 11 of the output.
OUT10	23:16	r	**Output Byte 10** This byte represents the byte 10 of the output.
OUT9	15:8	r	**Output Byte 9** This byte represents the byte 9 of the output.
OUT8	7:0	r	**Output Byte 8** This byte represents the byte 8 of the output.

AESOUT3

The AESOUT3 register contains part of plain text or cipher text.

AESOUT3	Offset	Reset Value
AES OUTPUT register 3	$2C_H$	$0000\ 0000_H$

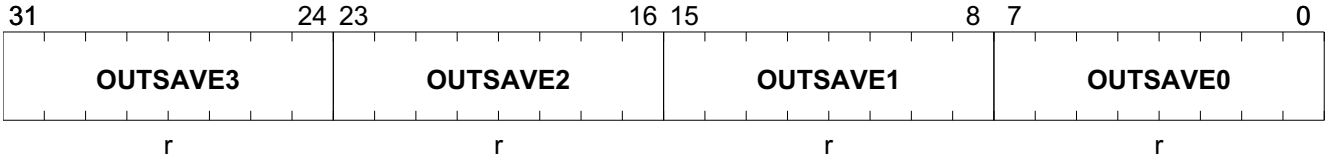


Field	Bits	Type	Description
OUT15	31:24	r	**Output Byte 15** This byte represents the byte 15 of the output.
OUT14	23:16	r	**Output Byte 14** This byte represents the byte 14 of the output.
OUT13	15:8	r	**Output Byte 13** This byte represents the byte 13 of the output.
OUT12	7:0	r	**Output Byte 12** This byte represents the byte 12 of the output.

AESOUTSAVE0

The AESOUTSAVE0 register contains part of a plain text or cipher text.

AESOUTSAVE0	Offset	Reset Value
AES OUTPUT save register 0	30_H	$0000\ 0000_H$

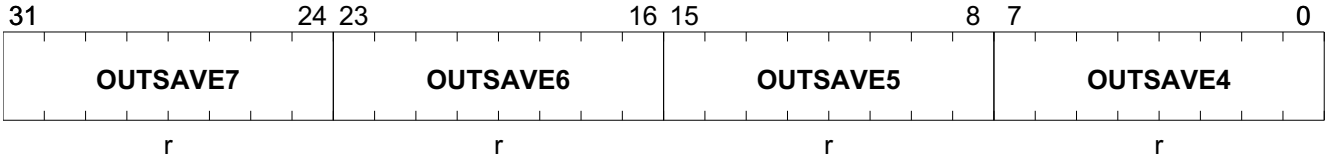


Field	Bits	Type	Description
OUTSAVE3	31:24	r	**Output Byte 3** This byte represents the byte 3 of the output.
OUTSAVE2	23:16	r	**Output Byte 2** This byte represents the byte 2 of the output.
OUTSAVE1	15:8	r	**Output Byte 1** This byte represents the byte 1 of the output.
OUTSAVE0	7:0	r	**Output Byte 0** This byte represents the byte 0 of the output.

AESOUTSAVE1

The AESOUTSAVE1 register contains part of a plain text or cipher text.

AESOUTSAVE1	Offset	Reset Value
AES OUTPUT save register 1	34_H	$0000\ 0000_H$

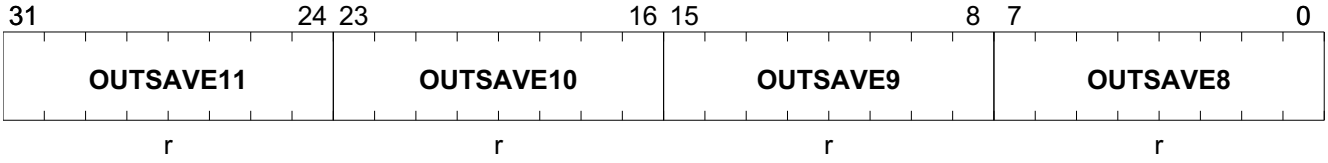


Field	Bits	Type	Description
OUTSAVE7	31:24	r	**Output Byte 7** This byte represents the byte 7 of the output.
OUTSAVE6	23:16	r	**Output Byte 6** This byte represents the byte 6 of the output.
OUTSAVE5	15:8	r	**Output Byte 5** This byte represents the byte 5 of the output.
OUTSAVE4	7:0	r	**Output Byte 4** This byte represents the byte 4 of the output.

AESOUTSAVE2

The AESOUTSAVE2 register contains part of a plain text or cipher text.

AESOUTSAVE2	Offset	Reset Value
AES OUTPUT save register 2	38_H	$0000\ 0000_H$

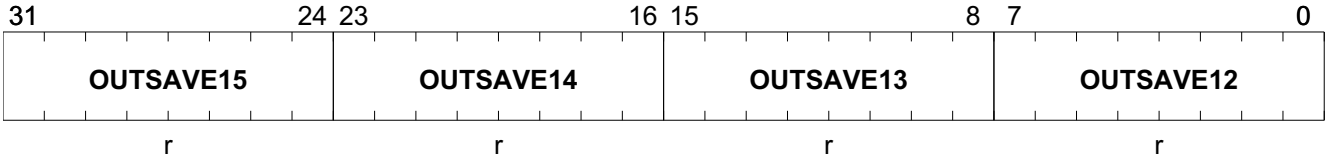


Field	Bits	Type	Description
OUTSAVE11	31:24	r	**Output Byte 11** This byte represents the byte 11 of the output.
OUTSAVE10	23:16	r	**Output Byte 10** This byte represents the byte 10 of the output.
OUTSAVE9	15:8	r	**Output Byte 9** This byte represents the byte 9 of the output.
OUTSAVE8	7:0	r	**Output Byte 8** This byte represents the byte 8 of the output.

AESOUTSAVE3

The AESOUTSAVE3 register contains part of a plain text or cipher text.

AESOUTSAVE3	Offset	Reset Value
AES OUTPUT save register 3	$3C_H$	$0000\ 0000_H$



Field	Bits	Type	Description
OUTSAVE15	31:24	r	**Output Byte 15** This byte represents the byte 15 of the output.
OUTSAVE14	23:16	r	**Output Byte 14** This byte represents the byte 14 of the output.
OUTSAVE13	15:8	r	**Output Byte 13** This byte represents the byte 13 of the output.
OUTSAVE12	7:0	r	**Output Byte 12** This byte represents the byte 12 of the output.

4.5 Pseudo Coding Example with Test Data

```
All 32 and 128 bit values are given in hexadecimal, omitting the prefix “0x”

identify:
loadkey = 0x10
loadCV  = 0x12
ecbenc  = 0x00
cbcenc  = 0x02
ctrenc  = 0x04
ofbenc  = 0x05
cfbenc  = 0x06
gcmenc  = 0x08
gcmmac  = 0x0a
xtsenc  = 0x0b

Default words are given in LITTLE ENDIAN MODE!

For all following encryption examples, the following data are fixed:

key = 2b7e151628aed2a6abf7158809cf4f3c
IV  = 000102030405060708090a0b0c0d0e0f (except for the CTR mode)
P1  = 6bc1bee22e409f96e93d7e117393172a
P2  = ae2d8a571e03ac9c9eb76fac45af8e51
P3  = 30c81c46a35ce411e5fbc1191a0a52ef
P4  = f69f2445df4f9b17ad2b417be66c3710
where these hexadecimal strings have to be interpreted, e.g., in the way that the 16
bytes are concatenated as:
B = b_0|b_1|b_2|b_3|b_4|b_5|b_6|b_7|b_8|b_9|b_10|b_11|b_12|b_13|b_14|b_15
On a 32-bit platform, this will be written as
arrays of 32-bit words:
key    k[4]
IV     IV[4]
P1     P1[4]
P2     P2[4]
P3     P3[4]
P4     P4[4]
C1     P1[4]
C2     P2[4]
C3     P3[4]
C4     P4[4]
with
k[0]  <- 16157e2b
k[1]  <- a6d2ae28
k[2]  <- 8815f7ab
k[3]  <- 3c4fcf09
IV[0] <- 03020100
IV[1] <- 07060504
IV[2] <- 0b0a0908
IV[3] <- 0f0e0d0c
```

```
P1[0] <- e2bec16b
P1[1] <- 969f402e
P1[2] <- 117e3de9
P1[3] <- 2a179373
P2[0] <- 578a2dae
P2[1] <- 9cac031e
P2[2] <- ac6fb79e
P2[3] <- 518eaf45
P3[0] <- 461cc830
P3[1] <- 11e45ca3
P3[2] <- 19c1fbe5
P3[3] <- ef520a1a
P4[0] <- 45249ff6
P4[1] <- 179b4fdf
P4[2] <- 7b412bad
P4[3] <- 10376ce6


1. ECB-mode

NIST SP800-38A [6]:
"F.1.1 ECB-AES128.Encrypt
Key           2b7e151628aed2a6abf7158809cf4f3c
Block #1
Plaintext     6bc1bee22e409f96e93d7e117393172a
Input Block   6bc1bee22e409f96e93d7e117393172a
Output Block 3ad77bb40d7a3660a89ecaf32466ef97
Ciphertext    3ad77bb40d7a3660a89ecaf32466ef97
Block #2
Plaintext     ae2d8a571e03ac9c9eb76fac45af8e51
Input Block   ae2d8a571e03ac9c9eb76fac45af8e51
Output Block f5d3d58503b9699de785895a96fdbaaf
Ciphertext    f5d3d58503b9699de785895a96fdbaaf
Block #3
Plaintext     30c81c46a35ce411e5fbc1191a0a52ef
Input Block   30c81c46a35ce411e5fbc1191a0a52ef
Output Block 43b1cd7f598ece23881b00e3ed030688
Ciphertext    43b1cd7f598ece23881b00e3ed030688
Block #4
Plaintext     f69f2445df4f9b17ad2b417be66c3710
Input Block   f69f2445df4f9b17ad2b417be66c3710
Output Block 7b0c785e27e8ad3f8223207104725dd4
Ciphertext    7b0c785e27e8ad3f8223207104725dd4"

key = 2b7e151628aed2a6abf7158809cf4f3c
P1  = 6bc1bee22e409f96e93d7e117393172a
P2  = ae2d8a571e03ac9c9eb76fac45af8e51
P3  = 30c81c46a35ce411e5fbc1191a0a52ef
P4  = f69f2445df4f9b17ad2b417be66c3710
C1  = 3ad77bb40d7a3660a89ecaf32466ef97
C2  = f5d3d58503b9699de785895a96fdbaaf
```

```
C3  = 43b1cd7f598ece23881b00e3ed030688
C4  = 7b0c785e27e8ad3f8223207104725dd4

Encryption with ECB mode with key in K5:

AESIN0  <- k[0]
AESIN1  <- k[1]
AESIN2  <- k[2]
AESIN3  <- k[3]
AESCTRL <- 0000loadkey050
AESIN0  <- P1[0]
AESIN1  <- P1[1]
AESIN2  <- P1[2]
AESIN3  <- P1[3]
AESCTRL <- 0000ecbenc050
wait until AESSTAT AND 00000001 = 0
C1[0]   <- AESOUT0
C1[1]   <- AESOUT1
C1[2]   <- AESOUT2
C1[3]   <- AESOUT3
AESIN0  <- P2[0]
AESIN1  <- P2[1]
AESIN2  <- P2[2]
AESIN3  <- P2[3]
AESCTRL <- 0000ecbenc050
wait until AESSTAT AND 00000001 = 0
C2[0]   <- AESOUT0
C2[1]   <- AESOUT1
C2[2]   <- AESOUT2
C2[3]   <- AESOUT3
AESIN0  <- P3[0]
AESIN1  <- P3[1]
AESIN2  <- P3[2]
AESIN3  <- P3[3]
AESCTRL <- 0000ecbenc050
wait until AESSTAT AND 00000001 = 0
C3[0]   <- AESOUT0
C3[1]   <- AESOUT1
C3[2]   <- AESOUT2
C3[3]   <- AESOUT3
AESIN0  <- P4[0]
AESIN1  <- P4[1]
AESIN2  <- P4[2]
AESIN3  <- P4[3]
AESCTRL <- 0000ecbenc050
wait until AESSTAT AND 00000001 = 0
C4[0]   <- AESOUT0
C4[1]   <- AESOUT1
C4[2]   <- AESOUT2
C4[3]   <- AESOUT3
```

```
now, the output is

C1[0] = B47BD73A
C1[1] = 60367A0D
C1[2] = F3CA9EA8
C1[3] = 97EF6624
C2[0] = 85D5D3F5
C2[1] = 9D69B903
C2[2] = 5A8985E7
C2[3] = AFBAFD96
C3[0] = 7FCDB143
C3[1] = 23CE8E59
C3[2] = E3001B88
C3[3] = 880603ED
C4[0] = 5E780C7B
C4[1] = 3FADE827
C4[2] = 71202382
C4[3] = D45D7204

Encryption with ECB mode with key in K5, in a pipelined version:

AESIN0  <- k[0]
AESIN1  <- k[1]
AESIN2  <- k[2]
AESIN3  <- k[3]
AESCTRL <- 0000loadkey050
AESIN0  <- P1[0]
AESIN1  <- P1[1]
AESIN2  <- P1[2]
AESIN3  <- P1[3]
AESCTRL <- 0000ecbenc050
AESIN0  <- P2[0]
AESIN1  <- P2[1]
AESIN2  <- P2[2]
AESIN3  <- P2[3]
[wait until AESSTAT AND 00000001 = 0]
AESCTRL <- 0000ecbenc050
C1[0]   <- AESOUTSAVE0
C1[1]   <- AESOUTSAVE1
C1[2]   <- AESOUTSAVE2
C1[3]   <- AESOUTSAVE3
AESIN0  <- P3[0]
AESIN1  <- P3[1]
AESIN2  <- P3[2]
AESIN3  <- P3[3]
[wait until AESSTAT AND 00000001 = 0]
AESCTRL <- 0000ecbenc050
C2[0]   <- AESOUTSAVE0
C2[1]   <- AESOUTSAVE1
C2[2]   <- AESOUTSAVE2
C2[3]   <- AESOUTSAVE3
```

```
AESIN0  <- P4[0]
AESIN1  <- P4[1]
AESIN2  <- P4[2]
AESIN3  <- P4[3]
[wait until AESSTAT AND 00000001 = 0]
AESCTRL <- 0000ecbenc050
C3[0]   <- AESOUTSAVE0
C3[1]   <- AESOUTSAVE1
C3[2]   <- AESOUTSAVE2
C3[3]   <- AESOUTSAVE3
[wait until AESSTAT AND 00000001 = 0]
C4[0]   <- AESOUT0
C4[1]   <- AESOUT1
C4[2]   <- AESOUT2
C4[3]   <- AESOUT3

now, the output is again the same as above.

2. CBC-mode

NIST SP800-38A [6]:
"F.2.1 CBC-AES128.Encrypt
Key           2b7e151628aed2a6abf7158809cf4f3c
IV            000102030405060708090a0b0c0d0e0f
Block #1
Plaintext     6bc1bee22e409f96e93d7e117393172a
Input Block   6bc0bce12a459991e134741a7f9e1925
Output Block  7649abac8119b246cee98e9b12e9197d
Ciphertext    7649abac8119b246cee98e9b12e9197d
Block #2
Plaintext     ae2d8a571e03ac9c9eb76fac45af8e51
Input Block   d86421fb9f1a1eda505ee1375746972c
Output Block  5086cb9b507219ee95db113a917678b2
Ciphertext    5086cb9b507219ee95db113a917678b2
Block #3
Plaintext     30c81c46a35ce411e5fbc1191a0a52ef
Input Block   604ed7ddf32efdff7020d0238b7c2a5d
Output Block  73bed6b8e3c1743b7116e69e22229516
Ciphertext    73bed6b8e3c1743b7116e69e22229516
Block #4
Plaintext     f69f2445df4f9b17ad2b417be66c3710
Input Block   8521f2fd3c8eef2cdc3da7e5c44ea206
Output Block  3ff1caa1681fac09120eca307586e1a7
Ciphertext    3ff1caa1681fac09120eca307586e1a7"


key = 2b7e151628aed2a6abf7158809cf4f3c
IV  = 000102030405060708090a0b0c0d0e0f
P1  = 6bc1bee22e409f96e93d7e117393172a
P2  = ae2d8a571e03ac9c9eb76fac45af8e51
P3  = 30c81c46a35ce411e5fbc1191a0a52ef
```

```
P4  = f69f2445df4f9b17ad2b417be66c3710
C1  = 7649abac8119b246cee98e9b12e9197d
C2  = 5086cb9b507219ee95db113a917678b2
C3  = 73bed6b8e3c1743b7116e69e22229516
C4  = 3ff1caa1681fac09120eca307586e1a7

Encryption with CBC mode with key in K7 and IV in CV1:

AESIN0  <- k[0]
AESIN1  <- k[1]
AESIN2  <- k[2]
AESIN3  <- k[3]
AESCTRL <- 0000loadkey070
AESIN0  <- IV[0]
AESIN1  <- IV[1]
AESIN2  <- IV[2]
AESIN3  <- IV[3]
AESCTRL <- 0000loadCV001
AESIN0  <- P1[0]
AESIN1  <- P1[1]
AESIN2  <- P1[2]
AESIN3  <- P1[3]
AESCTRL <- 0000cbcenc071
wait until AESSTAT AND 00000001 = 0
// Now CV1 = 7649abac8119b246cee98e9b12e9197d
C1[0]   <- AESOUT0
C1[1]   <- AESOUT1
C1[2]   <- AESOUT2
C1[3]   <- AESOUT3
AESIN0  <- P2[0]
AESIN1  <- P2[1]
AESIN2  <- P2[2]
AESIN3  <- P2[3]
AESCTRL <- 0000cbcenc071
wait until AESSTAT AND 00000001 = 0
// Now CV1 = 5086cb9b507219ee95db113a917678b2
C2[0]   <- AESOUT0
C2[1]   <- AESOUT1
C2[2]   <- AESOUT2
C2[3]   <- AESOUT3
AESIN0  <- P3[0]
AESIN1  <- P3[1]
AESIN2  <- P3[2]
AESIN3  <- P3[3]
AESCTRL <- 0000cbcenc071
wait until AESSTAT AND 00000001 = 0
// Now CV1 = 73bed6b8e3c1743b7116e69e22229516
C3[0]   <- AESOUT0
C3[1]   <- AESOUT1
C3[2]   <- AESOUT2
C3[3]   <- AESOUT3
```

```
AESIN0  <- P4[0]
AESIN1  <- P4[1]
AESIN2  <- P4[2]
AESIN3  <- P4[3]
AESCTRL <- 0000cbcenc071
wait until AESSTAT AND 00000001 = 0
// Now CV1 = 3ff1caa1681fac09120eca307586e1a7
C4[0]   <- AESOUT0
C4[1]   <- AESOUT1
C4[2]   <- AESOUT2
C4[3]   <- AESOUT3

now, the output is:

C1[0] = ACAB4976
C1[1] = 46B21981
C1[2] = 9B8EE9CE
C1[3] = 7D19E912
C2[0] = 9BCB8650
C2[1] = EE197250
C2[2] = 3A11DB95
C2[3] = B2787691
C3[0] = B8D6BE73
C3[1] = 3B74C1E3
C3[2] = 9EE61671
C3[3] = 16952222
C4[0] = A1CAF13F
C4[1] = 09AC1F68
C4[2] = 30CA0E12
C4[3] = A7E18675

Encryption with CBC mode with key in K7, in a pipelined version:

AESIN0  <- k[0]
AESIN1  <- k[1]
AESIN2  <- k[2]
AESIN3  <- k[3]
AESCTRL <- 0000loadkey050
AESIN0  <- IV[0]
AESIN1  <- IV[1]
AESIN2  <- IV[2]
AESIN3  <- IV[3]
AESCTRL <- 0000loadCV001
AESIN0  <- P1[0]
AESIN1  <- P1[1]
AESIN2  <- P1[2]
AESIN3  <- P1[3]
AESCTRL <- 0000cbcenc071
AESIN0  <- P2[0]
AESIN1  <- P2[1]
AESIN2  <- P2[2]
```

```
AESIN3  <- P2[3]
[wait until AESSTAT AND 00000001 = 0]
// Now CV1 = 7649abac8119b246cee98e9b12e9197d
AESCTRL <- 0000cbcenc071
C1[0]   <- AESOUTSAVE0
C1[1]   <- AESOUTSAVE1
C1[2]   <- AESOUTSAVE2
C1[3]   <- AESOUTSAVE3
AESIN0  <- P3[0]
AESIN1  <- P3[1]
AESIN2  <- P3[2]
AESIN3  <- P3[3]
[wait until AESSTAT AND 00000001 = 0]
// Now CV1 = 5086cb9b507219ee95db113a917678b2
AESCTRL <- 0000cbcenc071
C2[0]   <- AESOUTSAVE0
C2[1]   <- AESOUTSAVE1
C2[2]   <- AESOUTSAVE2
C2[3]   <- AESOUTSAVE3
AESIN0  <- P4[0]
AESIN1  <- P4[1]
AESIN2  <- P4[2]
AESIN3  <- P4[3]
[wait until AESSTAT AND 00000001 = 0]
// Now CV1 = 73bed6b8e3c1743b7116e69e22229516
AESCTRL <- 0000cbcenc071
C3[0]   <- AESOUTSAVE0
C3[1]   <- AESOUTSAVE1
C3[2]   <- AESOUTSAVE2
C3[3]   <- AESOUTSAVE3
[wait until AESSTAT AND 00000001 = 0]
// Now CV1 = 3ff1caa1681fac09120eca307586e1a7
C4[0]   <- AESOUT0
C4[1]   <- AESOUT1
C4[2]   <- AESOUT2
C4[3]   <- AESOUT3

now, the output is again the same as above.



3. CTR mode

NIST SP800-38A [6]:
"F.5.1 CTR-AES128.Encrypt
Key           2b7e151628aed2a6abf7158809cf4f3c
Init. Counter f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff
Block #1
Input Block   f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff
Output Block  ec8cdf7398607cb0f2d21675ea9ea1e4
Plaintext     6bc1bee22e409f96e93d7e117393172a
```

```
Ciphertext    874d6191b620e3261bef6864990db6ce
Block #2
Input Block   f0f1f2f3f4f5f6f7f8f9fafbfcfdff00
Output Block  362b7c3c6773516318a077d7fc5073ae
Plaintext     ae2d8a571e03ac9c9eb76fac45af8e51
Ciphertext    9806f66b7970fdff8617187bb9fffdff
Block #3
Input Block   f0f1f2f3f4f5f6f7f8f9fafbfcfdff01
Output Block  6a2cc3787889374fbeb4c81b17ba6c44
Plaintext     30c81c46a35ce411e5fbc1191a0a52ef
Ciphertext    5ae4df3edbd5d35e5b4f09020db03eab
Block #4
Input Block   f0f1f2f3f4f5f6f7f8f9fafbfcfdff02
Output Block  e89c399ff0f198c6d40a31db156cabfe
Plaintext     f69f2445df4f9b17ad2b417be66c3710
Ciphertext    1e031dda2fbe03d1792170a0f3009cee"


key = 2b7e151628aed2a6abf7158809cf4f3c
IV  = f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff
P1  = 6bc1bee22e409f96e93d7e117393172a
P2  = ae2d8a571e03ac9c9eb76fac45af8e51
P3  = 30c81c46a35ce411e5fbc1191a0a52ef
P4  = f69f2445df4f9b17ad2b417be66c3710
C1  = 874d6191b620e3261bef6864990db6ce
C2  = 9806f66b7970fdff8617187bb9fffdff
C3  = 5ae4df3edbd5d35e5b4f09020db03eab
C4  = 1e031dda2fbe03d1792170a0f3009cee

On a 32-bit platform, the counter will be written as
array:
IV     IV[4]
with
IV[0] <- F3F2F1F0
IV[1] <- F7F6F5F4
IV[2] <- FBFAF9F8
IV[3] <- FFFEFDFC

Encryption with CTR mode with key in K3 and IV in CV3:

AESIN0  <- k[0]
AESIN1  <- k[1]
AESIN2  <- k[2]
AESIN3  <- k[3]
AESCTRL <- 0000loadkey030
AESIN0  <- IV[0]
AESIN1  <- IV[1]
AESIN2  <- IV[2]
AESIN3  <- IV[3]
AESCTRL <- 0000loadCV003
// Now CV3 = f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff
```

```
AESIN0  <- P1[0]
AESIN1  <- P1[1]
AESIN2  <- P1[2]
AESIN3  <- P1[3]
AESCTRL <- 0000ctrenc033
wait until AESSTAT AND 00000001 = 0
C1[0]   <- AESOUT0
C1[1]   <- AESOUT1
C1[2]   <- AESOUT2
C1[3]   <- AESOUT3
// Now CV3 = f0f1f2f3f4f5f6f7f8f9fafbfcfdff00
AESIN0  <- P2[0]
AESIN1  <- P2[1]
AESIN2  <- P2[2]
AESIN3  <- P2[3]
AESCTRL <- 0000ctrenc033
wait until AESSTAT AND 00000001 = 0
C2[0]   <- AESOUT0
C2[1]   <- AESOUT1
C2[2]   <- AESOUT2
C2[3]   <- AESOUT3
// Now CV3 = f0f1f2f3f4f5f6f7f8f9fafbfcfdff01
AESIN0  <- P3[0]
AESIN1  <- P3[1]
AESIN2  <- P3[2]
AESIN3  <- P3[3]
AESCTRL <- 0000ctrenc033
wait until AESSTAT AND 00000001 = 0
C3[0]   <- AESOUT0
C3[1]   <- AESOUT1
C3[2]   <- AESOUT2
C3[3]   <- AESOUT3
// Now CV3 = f0f1f2f3f4f5f6f7f8f9fafbfcfdff002
AESIN0  <- P4[0]
AESIN1  <- P4[1]
AESIN2  <- P4[2]
AESIN3  <- P4[3]
AESCTRL <- 0000ctrenc033
wait until AESSTAT AND 00000001 = 0
C4[0]   <- AESOUT0
C4[1]   <- AESOUT1
C4[2]   <- AESOUT2
C4[3]   <- AESOUT3
// Now CV3 = f0f1f2f3f4f5f6f7f8f9fafbfcfdff03

now, the output is:

C1[0] = 91614D87
C1[1] = 26E320B6
C1[2] = 6468EF1B
C1[3] = CEB60D99
```

```
C2[0] = 6BF60698
C2[1] = FFFD7079
C2[2] = 7B181786
C2[3] = FFFDFFB9
C3[0] = 3EDFE45A
C3[1] = 5ED3D5DB
C3[2] = 02094F5B
C3[3] = AB3EB00D
C4[0] = DA1D031E
C4[1] = D103BE2F
C4[2] = A0702179
C4[3] = EE9C00F3
```

Encryption with CTR mode with key in K3, in a pipelined version:

```
AESIN0  <- k[0]
AESIN1  <- k[1]
AESIN2  <- k[2]
AESIN3  <- k[3]
AESCTRL <- 0000loadkey030
AESIN0  <- IV[0]
AESIN1  <- IV[1]
AESIN2  <- IV[2]
AESIN3  <- IV[3]
AESCTRL <- 0000loadCV003
// Now CV3 = f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff
AESIN0  <- P1[0]
AESIN1  <- P1[1]
AESIN2  <- P1[2]
AESIN3  <- P1[3]
AESCTRL <- 0000ctrenc033
AESIN0  <- P2[0]
AESIN1  <- P2[1]
AESIN2  <- P2[2]
AESIN3  <- P2[3]
// Now CV3 = f0f1f2f3f4f5f6f7f8f9fafbfcfdff00
AESCTRL <- 0000ctrenc033
C1[0]   <- AESOUTSAVE0
C1[1]   <- AESOUTSAVE1
C1[2]   <- AESOUTSAVE2
C1[3]   <- AESOUTSAVE3
AESIN0  <- P3[0]
AESIN1  <- P3[1]
AESIN2  <- P3[2]
AESIN3  <- P3[3]
// Now CV3 = f0f1f2f3f4f5f6f7f8f9fafbfcfdff01
AESCTRL <- 0000ctrenc033
C2[0]   <- AESOUTSAVE0
C2[1]   <- AESOUTSAVE1
C2[2]   <- AESOUTSAVE2
```

```
C2[3]   <- AESOUTSAVE3
AESIN0  <- P4[0]
AESIN1  <- P4[1]
AESIN2  <- P4[2]
AESIN3  <- P4[3]
// Now CV3 = f0f1f2f3f4f5f6f7f8f9fafbfcfdff02
AESCTRL <- 0000ctrenc033
C3[0]   <- AESOUTSAVE0
C3[1]   <- AESOUTSAVE1
C3[2]   <- AESOUTSAVE2
C3[3]   <- AESOUTSAVE3
// Now CV3 = f0f1f2f3f4f5f6f7f8f9fafbfcfdff03
C4[0]   <- AESOUT0
C4[1]   <- AESOUT1
C4[2]   <- AESOUT2
C4[3]   <- AESOUT3

now, the output is again the same as above.

4. OFB mode

NIST SP800-38A [6]:
"F.4.1 OFB-AES128.Encrypt
Key           2b7e151628aed2a6abf7158809cf4f3c
IV            000102030405060708090a0b0c0d0e0f
Block #1
Input Block   000102030405060708090a0b0c0d0e0f
Output Block  50fe67cc996d32b6da0937e99bafec60
Plaintext     6bc1bee22e409f96e93d7e117393172a
Ciphertext    3b3fd92eb72dad20333449f8e83cfb4a
Block #2
Input Block   50fe67cc996d32b6da0937e99bafec60
Output Block  d9a4dada0892239f6b8b3d7680e15674
Plaintext     ae2d8a571e03ac9c9eb76fac45af8e51
Ciphertext    7789508d16918f03f53c52dac54ed825
Block #3
Input Block   d9a4dada0892239f6b8b3d7680e15674
Output Block  a78819583f0308e7a6bf36b1386abf23
Plaintext     30c81c46a35ce411e5fbc1191a0a52ef
Ciphertext    9740051e9c5fecf64344f7a82260edcc
Block #4
Input Block   a78819583f0308e7a6bf36b1386abf23
Output Block  c6d3416d29165c6fcb8e51a227ba994e
Plaintext     f69f2445df4f9b17ad2b417be66c3710
Ciphertext    304c6528f659c77866a510d9c1d6ae5e"

key = 2b7e151628aed2a6abf7158809cf4f3c
IV  = 000102030405060708090a0b0c0d0e0f
P1  = 6bc1bee22e409f96e93d7e117393172a
P2  = ae2d8a571e03ac9c9eb76fac45af8e51
P3  = 30c81c46a35ce411e5fbc1191a0a52ef
```

```
P4  = f69f2445df4f9b17ad2b417be66c3710
C1  = 3b3fd92eb72dad20333449f8e83cfb4a
C2  = 7789508d16918f03f53c52dac54ed825
C3  = 9740051e9c5fecf64344f7a82260edcc
C4  = 304c6528f659c77866a510d9c1d6ae5e

Encryption with OFB mode with key in K3 and IV in CV2:

AESIN0  <- k[0]
AESIN1  <- k[1]
AESIN2  <- k[2]
AESIN3  <- k[3]
AESCTRL <- 0000loadkey030
AESIN0  <- IV[0]
AESIN1  <- IV[1]
AESIN2  <- IV[2]
AESIN3  <- IV[3]
AESCTRL <- 0000loadCV002
// Now CV2 = 000102030405060708090a0b0c0d0e0f
AESIN0  <- P1[0]
AESIN1  <- P1[1]
AESIN2  <- P1[2]
AESIN3  <- P1[3]
AESCTRL <- 0000ofbenc032
wait until AESSTAT AND 00000001 = 0
C1[0]   <- AESOUT0
C1[1]   <- AESOUT1
C1[2]   <- AESOUT2
C1[3]   <- AESOUT3
// Now CV2 = 50fe67cc996d32b6da0937e99bafec60
AESIN0  <- P2[0]
AESIN1  <- P2[1]
AESIN2  <- P2[2]
AESIN3  <- P2[3]
AESCTRL <- 0000ofbenc032
wait until AESSTAT AND 00000001 = 0
C2[0]   <- AESOUT0
C2[1]   <- AESOUT1
C2[2]   <- AESOUT2
C2[3]   <- AESOUT3
// Now CV2 = d9a4dada0892239f6b8b3d7680e15674
AESIN0  <- P3[0]
AESIN1  <- P3[1]
AESIN2  <- P3[2]
AESIN3  <- P3[3]
AESCTRL <- 0000ofbenc032
wait until AESSTAT AND 00000001 = 0
C3[0]   <- AESOUT0
C3[1]   <- AESOUT1
C3[2]   <- AESOUT2
C3[3]   <- AESOUT3
```

```
// Now CV2 = a78819583f0308e7a6bf36b1386abf23
AESIN0  <- P4[0]
AESIN1  <- P4[1]
AESIN2  <- P4[2]
AESIN3  <- P4[3]
AESCTRL <- 0000ofbenc032
wait until AESSTAT AND 00000001 = 0
C4[0]   <- AESOUT0
C4[1]   <- AESOUT1
C4[2]   <- AESOUT2
C4[3]   <- AESOUT3
// Now CV0 = c6d3416d29165c6fcb8e51a227ba994e

now, the output is:

C1[0] = 2ED93F3B
C1[1] = 20AD2DB7
C1[2] = F8493433
C1[3] = 4AFB3CE8
C2[0] = 8D508977
C2[1] = 038F9116
C2[2] = DA523CF5
C2[3] = 25D84EC5
C3[0] = 1E054097
C3[1] = F6EC5F9C
C3[2] = A8F74443
C3[3] = CCED6022
C4[0] = 28654C30
C4[1] = 78C759F6
C4[2] = D910A566
C4[3] = 5EAED6C1


Encryption with OFB mode with key in K3, in a pipelined version:

AESIN0  <- k[0]
AESIN1  <- k[1]
AESIN2  <- k[2]
AESIN3  <- k[3]
AESCTRL <- 0000loadkey030
AESIN0  <- IV[0]
AESIN1  <- IV[1]
AESIN2  <- IV[2]
AESIN3  <- IV[3]
AESCTRL <- 0000loadCV002
// Now CV2 = 000102030405060708090a0b0c0d0e0f
AESIN0  <- P1[0]
AESIN1  <- P1[1]
AESIN2  <- P1[2]
AESIN3  <- P1[3]
AESCTRL <- 0000ofbenc032
```

```
AESIN0  <- P2[0]
AESIN1  <- P2[1]
AESIN2  <- P2[2]
AESIN3  <- P2[3]
// Now CV2 = 50fe67cc996d32b6da0937e99bafec60
AESCTRL <- 0000ofbenc032
C1[0]   <- AESOUTSAVE0
C1[1]   <- AESOUTSAVE1
C1[2]   <- AESOUTSAVE2
C1[3]   <- AESOUTSAVE3
AESIN0  <- P3[0]
AESIN1  <- P3[1]
AESIN2  <- P3[2]
AESIN3  <- P3[3]
// Now CV2 = d9a4dada0892239f6b8b3d7680e15674
AESCTRL <- 0000ofbenc032
C2[0]   <- AESOUTSAVE0
C2[1]   <- AESOUTSAVE1
C2[2]   <- AESOUTSAVE2
C2[3]   <- AESOUTSAVE3
AESIN0  <- P4[0]
AESIN1  <- P4[1]
AESIN2  <- P4[2]
AESIN3  <- P4[3]
// Now CV2 = a78819583f0308e7a6bf36b1386abf23
AESCTRL <- 0000ofbenc032
C3[0]   <- AESOUTSAVE0
C3[1]   <- AESOUTSAVE1
C3[2]   <- AESOUTSAVE2
C3[3]   <- AESOUTSAVE3
// Now CV0 = c6d3416d29165c6fcb8e51a227ba994e
C4[0]   <- AESOUT0
C4[1]   <- AESOUT1
C4[2]   <- AESOUT2
C4[3]   <- AESOUT3

now, the output is again the same as above.

5. CFB mode

NIST SP800-38A [6]:
"CFB128-AES192.Encrypt
Key          2b7e151628aed2a6abf7158809cf4f3c
IV           000102030405060708090a0b0c0d0e0f
Segment #1
Input Block  000102030405060708090a0b0c0d0e0f
Output Block 50fe67cc996d32b6da0937e99bafec60
Plaintext    6bc1bee22e409f96e93d7e117393172a
Ciphertext   3b3fd92eb72dad20333449f8e83cfb4a
Segment #2
Input Block  3b3fd92eb72dad20333449f8e83cfb4a
```

```
Output Block 668bcf60beb005a35354a201dab36bda
Plaintext    ae2d8a571e03ac9c9eb76fac45af8e51
Ciphertext   c8a64537a0b3a93fcde3cdad9f1ce58b
Segment #3
Input Block  c8a64537a0b3a93fcde3cdad9f1ce58b
Output Block 16bd032100975551547b4de89daea630
Plaintext    30c81c46a35ce411e5fbc1191a0a52ef
Ciphertext   26751f67a3cbb140b1808cf187a4f4df
Segment #4
Input Block  26751f67a3cbb140b1808cf187a4f4df
Output Block 36d42170a312871947ef8714799bc5f6
Plaintext    f69f2445df4f9b17ad2b417be66c3710
Ciphertext   c04b05357c5d1c0eeac4c66f9ff7f2e6"


key = 2b7e151628aed2a6abf7158809cf4f3c
IV  = 000102030405060708090a0b0c0d0e0f
P1  = 6bc1bee22e409f96e93d7e117393172a
P2  = ae2d8a571e03ac9c9eb76fac45af8e51
P3  = 30c81c46a35ce411e5fbc1191a0a52ef
P4  = f69f2445df4f9b17ad2b417be66c3710
C1  = 3b3fd92eb72dad20333449f8e83cfb4a
C2  = c8a64537a0b3a93fcde3cdad9f1ce58b
C3  = 26751f67a3cbb140b1808cf187a4f4df
C4  = c04b05357c5d1c0eeac4c66f9ff7f2e6

Encryption with CFB mode with key in K5 and IV in CV0:

AESIN0  <- k[0]
AESIN1  <- k[1]
AESIN2  <- k[2]
AESIN3  <- k[3]
AESCTRL <- 0000loadkey050
AESIN0  <- IV[0]
AESIN1  <- IV[1]
AESIN2  <- IV[2]
AESIN3  <- IV[3]
AESCTRL <- 0000loadCV000
// Now CV0 = 000102030405060708090a0b0c0d0e0f
AESIN0  <- P1[0]
AESIN1  <- P1[1]
AESIN2  <- P1[2]
AESIN3  <- P1[3]
AESCTRL <- 0000cfbenc050
wait until AESSTAT AND 00000001 = 0
C1[0]   <- AESOUT0
C1[1]   <- AESOUT1
C1[2]   <- AESOUT2
C1[3]   <- AESOUT3
// Now CV0 = 3b3fd92eb72dad20333449f8e83cfb4a
AESIN0  <- P2[0]
```

```
AESIN1  <- P2[1]
AESIN2  <- P2[2]
AESIN3  <- P2[3]
AESCTRL <- 0000cfbenc050
wait until AESSTAT AND 00000001 = 0
C2[0]   <- AESOUT0
C2[1]   <- AESOUT1
C2[2]   <- AESOUT2
C2[3]   <- AESOUT3
// Now CV0 = c8a64537a0b3a93fcde3cdad9f1ce58b
AESIN0  <- P3[0]
AESIN1  <- P3[1]
AESIN2  <- P3[2]
AESIN3  <- P3[3]
AESCTRL <- 0000cfbenc050
wait until AESSTAT AND 00000001 = 0
C3[0]   <- AESOUT0
C3[1]   <- AESOUT1
C3[2]   <- AESOUT2
C3[3]   <- AESOUT3
// Now CV0 = 26751f67a3cbb140b1808cf187a4f4df
AESIN0  <- P4[0]
AESIN1  <- P4[1]
AESIN2  <- P4[2]
AESIN3  <- P4[3]
AESCTRL <- 0000cfbenc050
wait until AESSTAT AND 00000001 = 0
C4[0]   <- AESOUT0
C4[1]   <- AESOUT1
C4[2]   <- AESOUT2
C4[3]   <- AESOUT3
// Now CV0 = c04b05357c5d1c0eeac4c66f9ff7f2e6

now, the output is:

C1[0] = 2ED93F3B
C1[1] = 20AD2DB7
C1[2] = F8493433
C1[3] = 4AFB3CE8
C2[0] = 3745A6C8
C2[1] = 3FA9B3A0
C2[2] = ADCDE3CD
C2[3] = 8BE51C9F
C3[0] = 671F7526
C3[1] = 40B1CBA3
C3[2] = F18C80B1
C3[3] = DFF4A487
C4[0] = 35054BC0
C4[1] = 0E1C5D7C
C4[2] = 6FC6C4EA
C4[3] = E6F2F79F
```

```
Encryption with CFB mode with key in K5, in a pipelined version:

AESIN0  <- k[0]
AESIN1  <- k[1]
AESIN2  <- k[2]
AESIN3  <- k[3]
AESCTRL <- 0000loadkey050
AESIN0  <- IV[0]
AESIN1  <- IV[1]
AESIN2  <- IV[2]
AESIN3  <- IV[3]
AESCTRL <- 0000loadCV000
// Now CV0 = 000102030405060708090a0b0c0d0e0f
AESIN0  <- P1[0]
AESIN1  <- P1[1]
AESIN2  <- P1[2]
AESIN3  <- P1[3]
AESCTRL <- 0000cfbenc050
AESIN0  <- P2[0]
AESIN1  <- P2[1]
AESIN2  <- P2[2]
AESIN3  <- P2[3]
// Now CV0 = 3b3fd92eb72dad20333449f8e83cfb4a
AESCTRL <- 0000cfbenc050
C1[0]   <- AESOUTSAVE0
C1[1]   <- AESOUTSAVE1
C1[2]   <- AESOUTSAVE2
C1[3]   <- AESOUTSAVE3
AESIN0  <- P3[0]
AESIN1  <- P3[1]
AESIN2  <- P3[2]
AESIN3  <- P3[3]
// Now CV0 = c8a64537a0b3a93fcde3cdad9f1ce58b
AESCTRL <- 0000cfbenc050
C2[0]   <- AESOUTSAVE0
C2[1]   <- AESOUTSAVE1
C2[2]   <- AESOUTSAVE2
C2[3]   <- AESOUTSAVE3
AESIN0  <- P4[0]
AESIN1  <- P4[1]
AESIN2  <- P4[2]
AESIN3  <- P4[3]
// Now CV0 = 26751f67a3cbb140b1808cf187a4f4df
AESCTRL <- 0000cfbenc050
C3[0]   <- AESOUTSAVE0
C3[1]   <- AESOUTSAVE1
C3[2]   <- AESOUTSAVE2
C3[3]   <- AESOUTSAVE3
// Now CV0 = c04b05357c5d1c0eeac4c66f9ff7f2e6
C4[0]   <- AESOUT0
```

```
C4[1]   <- AESOUT1
C4[2]   <- AESOUT2
C4[3]   <- AESOUT3

now, the output is again the same as above.

6. XTS-mode

IEEE 1619-2007:
"Annex B" Vector 8
Key1 = 27182818284590452353602874713526
Key2 = 31415926535897932384626433832795
i    = fe000000000000000000000000000000
P1   = d55f684f81f4426e9fde92a5ff02df2a
P2   = c896af63962888a97910c1379e20b0a3
P3   = b1db613fb7fe2e07004329ea5c22bfd3
P4   = 3e3dbe4cf58cc608c2c26c19a2e2fe22
where these hexadecimal strings have to be interpreted, e.g., in the way that the 16
bytes are concatenated as:
B = b_0|b_1|b_2|b_3|b_4|b_5|b_6|b_7|b_8|b_9|b_10|b_11|b_12|b_13|b_14|b_15
On a 32-bit platform, this will be written as
arrays of 32-bit words:
key1 k1[4]
key2 k2[4]
i    I[4]
P1   P1[4]
P2   P2[4]
P3   P3[4]
P4   P4[4]
C1   P1[4]
C2   P2[4]
C3   P3[4]
C4   P4[4]
with
k1[0] <- 18281827
k1[1] <- 45904528
k1[2] <- 28605323
k1[3] <- 26357174

k2[0] <- 26594131
k2[1] <- 93975853
k2[2] <- 64628423
k2[3] <- 95278333

I[0] <- 000000FE
I[1] <- 00000000
I[2] <- 00000000
I[3] <- 00000000

P1[0] <- 4F685FD5
P1[1] <- 6E42F481
```

```
P1[2] <- A592DE9F
P1[3] <- 2ADF02FF
P2[0] <- 63AF96C8
P2[1] <- A9882896
P2[2] <- 37C11079
P2[3] <- A3B0209E
P3[0] <- 3F61DBB1
P3[1] <- 072EFEB7
P3[2] <- EA294300
P3[3] <- D3BF225C
P4[0] <- 4CBE3D3E
P4[1] <- 08C68CF5
P4[2] <- 196CC2C2
P4[3] <- 22FEE2A2

Key1          271828182845904523536028747135 26
Key2          314159265358979323846264338327 95
i             fe000000000000000000000000000000
j             start with 0
Block #1
Plaintext     d55f684f81f4426e9fde92a5ff02df2a
Ciphertext    72efc1ebfe1ee25975a6eb3aa8589dda
Block #2
Plaintext     c896af63962888a97910c1379e20b0a3
Ciphertext    2b261f1c85bdab442a9e5b2dd1d7c395
Block #3
Plaintext     b1db613fb7fe2e07004329ea5c22bfd3
Ciphertext    7a16fc08e526d4b1223f1b1232a11af2
Block #4
Plaintext     3e3dbe4cf58cc608c2c26c19a2e2fe22
Ciphertext    74c3d70dac57f83e0983c498f1a6f1ae

Encryption with XTS mode with key1 in K5 and key2 in K4:

AESIN0  <- k2[0]
AESIN1  <- k2[1]
AESIN2  <- k2[2]
AESIN3  <- k2[3]
AESCTRL <- 0000loadkey040
AESIN0  <- I[0]
AESIN1  <- I[1]
AESIN2  <- I[2]
AESIN3  <- I[3]
AESCTRL <- 0000ecbenc040
wait until AESSTAT AND 00000001 = 0
AESIN0  <- AESOUT0
AESIN1  <- AESOUT1
AESIN2  <- AESOUT2
AESIN3  <- AESOUT3
AESCTRL <- 0000loadCV002
// Now CV2 = AES-enc(key2,i)
```

```
AESIN0  <- k1[0]
AESIN1  <- k1[1]
AESIN2  <- k1[2]
AESIN3  <- k1[3]
AESCTRL <- 0000loadkey050
// Now K5 = key1
AESIN0  <- P1[0]
AESIN1  <- P1[1]
AESIN2  <- P1[2]
AESIN3  <- P1[3]
AESCTRL <- 000xtsenc052
wait until AESSTAT AND 00000001 = 0
C1[0]   <- AESOUT0
C1[1]   <- AESOUT1
C1[2]   <- AESOUT2
C1[3]   <- AESOUT3
AESIN0  <- P2[0]
AESIN1  <- P2[1]
AESIN2  <- P2[2]
AESIN3  <- P2[3]
AESCTRL <- 000xtsenc052
wait until AESSTAT AND 00000001 = 0
C2[0]   <- AESOUT0
C2[1]   <- AESOUT1
C2[2]   <- AESOUT2
C2[3]   <- AESOUT3
AESIN0  <- P3[0]
AESIN1  <- P3[1]
AESIN2  <- P3[2]
AESIN3  <- P3[3]
AESCTRL <- 000xtsenc052
wait until AESSTAT AND 00000001 = 0
C3[0]   <- AESOUT0
C3[1]   <- AESOUT1
C3[2]   <- AESOUT2
C3[3]   <- AESOUT3
AESIN0  <- P4[0]
AESIN1  <- P4[1]
AESIN2  <- P4[2]
AESIN3  <- P4[3]
AESCTRL <- 000xtsenc052
wait until AESSTAT AND 00000001 = 0
C4[0]   <- AESOUT0
C4[1]   <- AESOUT1
C4[2]   <- AESOUT2
C4[3]   <- AESOUT3

now, the output is

C1[0] <- EBC1EF72
C1[1] <- 59E21EFE
```

```
C1[2] <- 3AEBA675
C1[3] <- DA9D58A8
C2[0] <- 1C1F262B
C2[1] <- 44ABBD85
C2[2] <- 2D5B9E2A
C2[3] <- 95C3D7D1
C3[0] <- 08FC167A
C3[1] <- B1D426E5
C3[2] <- 121B3F22
C3[3] <- F21AA132
C4[0] <- 0DD7C374
C4[1] <- 3EF857AC
C4[2] <- 98C48309
C4[3] <- AEF1A6F1


8. GCM-mode
Reference
 McGrew, after Test Case 4 (slightly changed)
Key  = feffe9928665731c6d6a8f9467308308
IV   = cafebabefacedbaddecaf888
A1   = feedfacedeadbeeffeedfacedeadbeef
A2   = abaddad2
P1   = d9313225f88406e5a55909c5aff5269a
P2   = 86a7a9531534f7da2e4c303d8a318a72
P3   = 1c3c0c95956809532fcf0e2449a6b525
P4   = b16aedf5aa0de657ba637b3900000000
L    = 00000000000000a00000000000000200
where these hexadecimal strings have to be interpreted, e.g., in the way that the 16
bytes are concatenated as:
B = b_0|b_1|b_2|b_3|b_4|b_5|b_6|b_7|b_8|b_9|b_10|b_11|b_12|b_13|b_14|b_15
On a 32-bit platform, this will be written as
arrays of 32-bit words:
key      k[4]
IV       I[4]
A1      A1[4]
A2      A2[4]
P1      P1[4]
P2      P2[4]
P3      P3[4]
P4      P4[4]
L        L[4]
C1      C1[4]
C2      C2[4]
C3      C3[4]
C4      C4[4]
T        T[4]
with

k[0] <- 92E9FFFE
k[1] <- 1C736586
```

```
k[2] <- 948F6A6D
k[3] <- 08833067
I[0] <- BEBAFECA
I[1] <- ADDBCEFA
I[2] <- 88F8CADE
I[3] <- 01000000
A1[0] <- CEFAEDFE
A1[1] <- EFBEADDE
A1[2] <- CEFAEDFE
A1[3] <- EFBEADDE
A2[0] <- D2DAADAB
A2[1] <- 00000000
A2[2] <- 00000000
A2[3] <- 00000000
P1[0] <- 253231D9
P1[1] <- E50684F8
P1[2] <- C50959A5
P1[3] <- 9A26F5AF
P2[0] <- 53A9A786
P2[1] <- DAF73415
P2[2] <- 3D304C2E
P2[3] <- 728A318A
P3[0] <- 950C3C1C
P3[1] <- 53096895
P3[2] <- 240ECF2F
P3[3] <- 25B5A649
P4[0] <- F5ED6AB1
P4[1] <- 57E60DAA
P4[2] <- 397B63BA
P4[3] <- 00000000
L[0] <- 00000000
L[1] <- A0000000
L[2] <- 00000000
L[3] <- 00020000

Key          feffe9928665731c6d6a8f9467308308
IV           cafebabefacedbaddecaf88800000001
H            b83b533708bf535d0aa6e52980d53b78

Encryption with GCM mode with key in K5, IV in CV0, H in CV1:

AESIN0  <- k[0]
AESIN1  <- k[1]
AESIN2  <- k[2]
AESIN3  <- k[3]
AESCTRL <- 0000loadkey050
// Now k in K5
AESIN0  <- 00000000
AESIN1  <- 00000000
AESIN2  <- 00000000
AESIN3  <- 00000000
```

```
AESCTRL <- 0000loadCV002
// Clear CV2 = 0
AESCTRL <- 0000ecbenc050
wait until AESSTAT AND 00000001 = 0
// Now H in AESOUT
AESIN0  <- AESOUT0
AESIN1  <- AESOUT1
AESIN2  <- AESOUT2
AESIN3  <- AESOUT3
AESCTRL <- 0000loadCV001
// Now H in CV1 = b83b533708bf535d0aa6e52980d53b78
AESIN0  <- I[0]
AESIN1  <- I[1]
AESIN2  <- I[2]
AESIN3  <- I[3]
AESCTRL <- 0000loadCV000
// Now IV in CV0 = cafebabefacedbaddecaf88800000001
AESCTRL <- 0000ctrenc050
// Dummy encryption for incrementation of CV0
// Now inc32(IV) in CV0
AESIN0  <- A1[0]
AESIN1  <- A1[1]
AESIN2  <- A1[2]
AESIN3  <- A1[3]
AESCTRL <- 000gcmmac000
wait until AESSTAT AND 00000001 = 0
// Now CV2 = ED56AAF8A72D67049FDB9228EDBA1322
AESIN0  <- A2[0]
AESIN1  <- A2[1]
AESIN2  <- A2[2]
AESIN3  <- A2[3]
AESCTRL <- 000gcmmac000
wait until AESSTAT AND 00000001 = 0
// Now CV2 = CD47221CCEF0554EE4BB044C88150352
AESIN0  <- P1[0]
AESIN1  <- P1[1]
AESIN2  <- P1[2]
AESIN3  <- P1[3]
AESCTRL <- 000gcmenc050
wait until AESSTAT AND 00000001 = 0
// Now CV2 = 54F5E1B2B5A8F9525C23924751A3CA51
C1[0]   <- AESOUT0
C1[1]   <- AESOUT1
C1[2]   <- AESOUT2
C1[3]   <- AESOUT3
AESIN0  <- P2[0]
AESIN1  <- P2[1]
AESIN2  <- P2[2]
AESIN3  <- P2[3]
AESCTRL <- 000gcmenc050
wait until AESSTAT AND 00000001 = 0
```

```
// Now CV2 = 324F585C6FFC1359AB371565D6C45F93
C2[0]   <- AESOUT0
C2[1]   <- AESOUT1
C2[2]   <- AESOUT2
C2[3]   <- AESOUT3
AESIN0  <- P3[0]
AESIN1  <- P3[1]
AESIN2  <- P3[2]
AESIN3  <- P3[3]
AESCTRL <- 000gcmenc050
wait until AESSTAT AND 00000001 = 0
// Now CV2 = CA7DD446AF4AA70CC3C0CD5ABBA6AA1C
C3[0]   <- AESOUT0
C3[1]   <- AESOUT1
C3[2]   <- AESOUT2
C3[3]   <- AESOUT3
AESIN0  <- P4[0]
AESIN1  <- P4[1]
AESIN2  <- P4[2]
AESIN3  <- P4[3]
AESCTRL <- 000gcmenc050
wait until AESSTAT AND 00000001 = 0
// Now CV2 = 8E8C2F3823398AD4C475594827899155
// CV2 = 1590DF9B2EB6768289E57D56274C8570 is the value according to unchanged case.
C4[0]   <- AESOUT0
C4[1]   <- AESOUT1
C4[2]   <- AESOUT2
C4[3]   <- AESOUT3
// C4[3] = 5d908bd0
AESIN0  <- L[0]
AESIN1  <- L[1]
AESIN2  <- L[2]
AESIN3  <- L[3]
AESCTRL <- 000gcmmac000
wait until AESSTAT AND 00000001 = 0
// Now CV2 = 57AAFAF870DDDF6EB1CE6E77B798_A5DF
// CV2 = 698E57F70E6ECC7FD9463B7260A9AE5F is the value according to unchanged case.
AESIN0  <- I[0]
AESIN1  <- I[1]
AESIN2  <- I[2]
AESIN3  <- I[3]
AESCTRL <- 0000loadCV000
// Now IV in CV0
AESCTRL <- 0000readCV002
wait until AESSTAT AND 00000001 = 0
AESIN0  <- AESOUT0
AESIN1  <- AESOUT1
AESIN2  <- AESOUT2
AESIN3  <- AESOUT3
AESCTRL <- 0000ctrenc050
wait until AESSTAT AND 00000001 = 0
```

```
T[0]   <- AESOUT0
T[1]   <- AESOUT1
T[2]   <- AESOUT2
T[3]   <- AESOUT3
now, the output is

C1[0] = C21E8342
C1[1] = 24747721
C1[2] = B721724B
C1[3] = 9CD4D084
C2[0] = 2F21AAE3
C2[1] = E0A4022C
C2[2] = 237EC135
C2[3] = 2EA1AC29
C3[0] = B214D521
C3[1] = 1C936654
C3[2] = 5A6A8F7D
C3[3] = 05AA84AC
C4[1] = 390BA31B
C4[2] = 97AC0A6A
C4[3] = 91E0583D
T[0] = B3E2ED65
T[1] = CAB6924C
T[2] = 5FBC72FC
T[3] = C7112330

T             65ED_E2B3_4C92_B6CA_FC72_BC5F_3023_11C7
// 5BC9_4FBC_3221_A5DB_94FA_E95A_E712_1A47 is the value according to unchanged case.
```

5 Hash Module

5.1 Features

The hash module supports different hashing algorithms. It works in a block-oriented way and uses a FIFO and an internal block buffer to maximize data throughput. To support different protocols, it also includes an endianness converter.

The module is intended to be used for signature generation, verification and generic data integrity checks. The following table summarizes the supported hash algorithms and the performance of the module.

Table 5-1 Hash Algorithms and Performance

Algorithm	Clock Cycles per 512-bit Data Block	Notes
MD-5	65	—
SHA-1	81	—
SHA-224	65	Special software handling needed, see **Section 5.2.4**
SHA-256	65	—

The module supports multi-tasking environments; however, for preemptive multi-tasking, it will be necessary to use a software abstraction layer as the module does not support stopping or resuming hash calculations at arbitrary points in time.

5.2 Functional Description

The hash module works in a block-oriented way. To achieve maximum performance, a doublebuffering scheme is implemented. A 64-byte internal buffer stores 512 bits of data to be hashed. If this buffer is filled, the hash calculation is started. While the hash module is busy, the next 512 bits of data can already be written into the module.

The module itself can be used in a word-oriented way or in block mode. In word mode, the **HASH_STAT.DF_NF** bit must be polled after each 32-bit data write. In block mode, 512 bits of data must be written before the bit is polled. Obviously, the performance in block mode is much higher.

Data is written into a 32-bit wide FIFO which has 16 entries (this equals the 512-bit block size needed by the hash algorithms). When this FIFO is full, its contents is copied into an internal buffer which is then hashed. After all data to be hashed has been written and the hashing operation of the last written data block has finished (indicated by **HASH_STAT.BSY** == 0), the result can be read from the hash result value register **HASH_VAL**. Since the length of the result depends on the chosen hash algorithm, a counter in the status register indicates the amount of 32-bit words which are available in the output register.

If the result is read from **HASH_VAL** when the **HASH_STAT.BSY** bit is still 1, zero values are read. The read operation of the hash value is destructive; that means the result can be read only once from the module. When **HASH_STAT.CNT** reached 0, the result cannot be read again; reading further data returns zero values.

5.2.1 Data Transfers and FIFO Flag

After reset or after a write to the **HASH_CFG** register, the internal buffer and the FIFO are empty. Bit **HASH_STAT.DF_NF** is set which indicates that 64 bytes can be written into the FIFO. After 64 bytes have been written, this bit will change from 1 to 0.

Note: The busy flag HASH_STAT.BSY will already change from 0 to 1 after the first 32 bits are written to the data register.

When the FIFO is full, its contents is copied into the internal buffer; after the first word has been copied, **HASH_STAT.DF_NF** will already be read as 1 again. The whole copy operation needs 16 clock cycles. The **DF_NF** bit will only be read as 0 when the software writes 64 bytes (512 bits) while the hash engine is still busy and cannot receive new data.

The hashing operation starts after the FIFO contents has been copied into the internal buffer; when hashing of the internal buffer is done, the **HASH_STAT.BSY** flag will change back from 1 to 0. If the FIFO has been filled again already when the hashing of the internal buffer finishes, it is immediately copied again and hashed (which implies that **HASH_STAT.BSY** will be read as 1 again).

The word order of the data which is sent to the module is fixed; the most significant word (MSW) should always be sent first. The word order of the result which is read from the module can be programmed. The byte order can be changed for both input and output data (see **Section 5.2.5**).

5.2.2 Output Data Size

The length of the data which is read from the module (the hash digest) depends on the chosen algorithm. When calculation of the hash value is finished, the **HASH_STAT.CNT** field holds the number of output words. The word order of the digest output can be configured using **HASH_CFG.ORDER_OUT** as shown in **Figure 5-1** (note that this bit has to be set as needed before data is sent to the module since writing **HASH_CFG** resets the hash engine and clears the data FIFO).

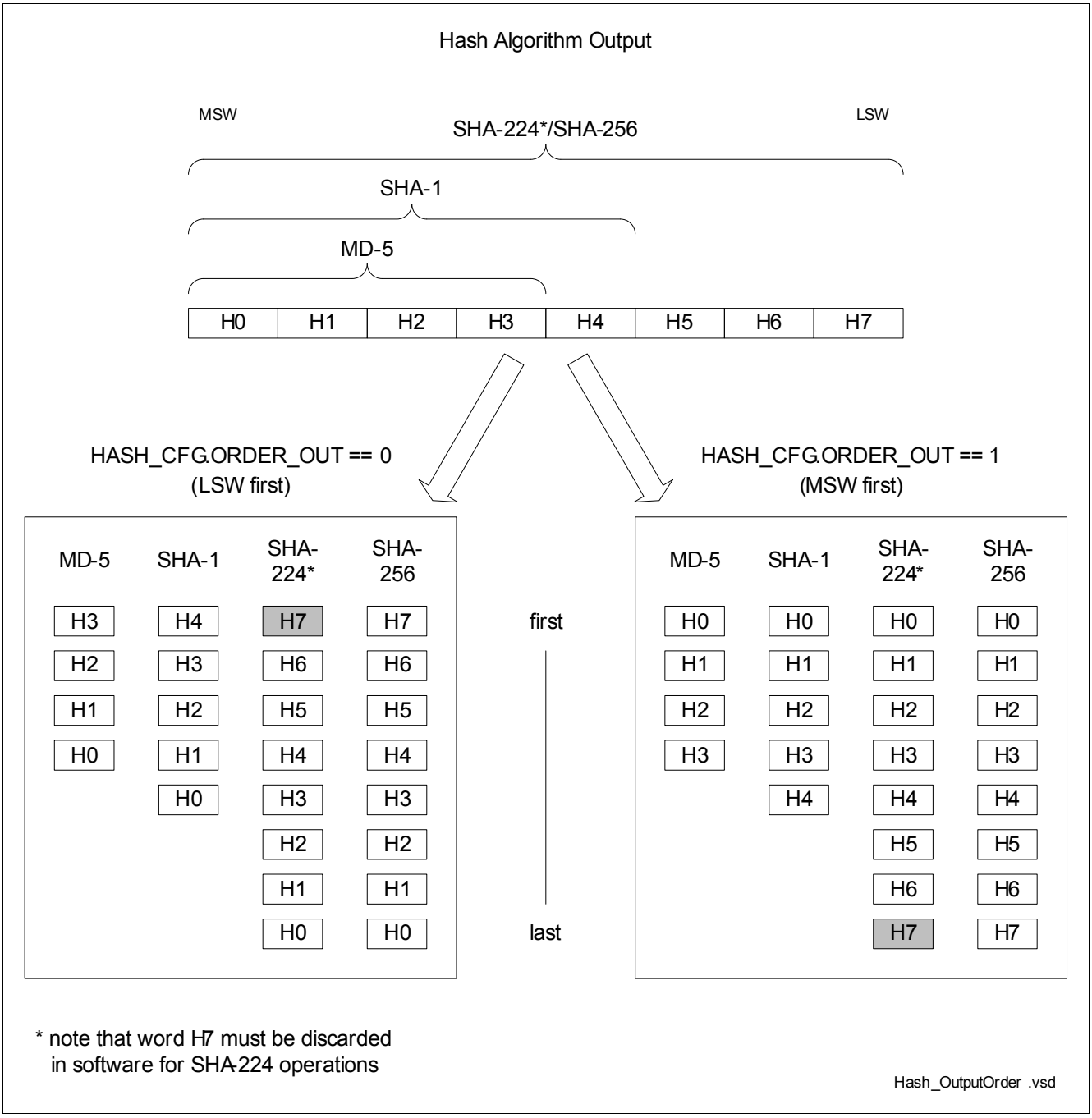


Figure 5-1 Word Order of Hash Digest Output

5.2.3 User-Defined Initialization Vector (Multiapplication Support)

The hash module supports user-defined initialization vectors. This allows support of multitasking as well: when one task uses the hash module and is interrupted, it just has to save the current hash value and reload it as initialization value after the task runs again (note that saving and restoring the current hash value is only possible after a running hash operation has finished which is indicated by **HASH_STAT.BSY** == 0).

As interleaved reading and writing of data is not possible (for instance, writes to the initialization vector register **HASH_IVIN** must not be interleaved with writes to **HASH_DATA** or reads from **HASH_VAL**), a software

abstraction layer is necessary to fully support preemptive multitasking. This layer must also assure that writes to **HASH_DATA** and reads from **HASH_VAL** are not interleaved.

The length of the initialization vector depends on the selected hash algorithm. After selecting the hash algorithm using the **HASH_CFG.ALGO** bitfield and enabling the use of a predefined initialization vector by setting the **HASH_CFG.IV_MODE** bit to 1, the initialization vector to be used is written into the **HASH_IVIN** register (with the most significant word first).

Writing **HASH_CFG.IV_MODE** to 1 does not change the current initialization vector (which also implies that the vector is not automatically reset to the default value of the currently selected hash algorithm). This is only done by writing the new vector into the **HASH_IVIN** register. If **HASH_CFG.IV_MODE** is set but no vector is written into **HASH_IVIN**, the hash digest from the previous calculation is used as initialization vector.

The length of the initialization vector for MD-5 is 128 bits, that means that four write accesses to **HASH_IVIN** are necessary. If SHA-1 is selected, five accesses are necessary to write 160 bits; for SHA-224 and SHA-256, eight write accesses are needed to write 256 bits (also refer to **Section 5.2.4** for SHA-224).

If sufficient bits have been written into the **HASH_IVIN** register, the bit **HASH_STAT.IV_OK** will be set to 1 by the hardware which indicates that the initialization vector is ok. If more data is written to the **HASH_IVIN** register, this bit will be reset to 0 until a full initialization vector has been written again.

As long as no settings have to be changed in **HASH_CFG**, it is possible to just write a different initialization vector to **HASH_IVIN** after a hash operation finished without writing **HASH_CFG** again (that means, the **HASH_CFG.IV_MODE** bit is not reset by hardware once an initialization vector has been loaded into the module).

Note: If the HASH_STAT.BSY bit is set, writes to HASH_IVIN are ignored.

5.2.4 SHA-224 Hash Support

The module can also be used to calculate SHA-224 hashes. This is done using a user-defined initialization vector and selecting the SHA-256 algorithm in the **HASH_CFG.ALGO** bitfield (this can be done since the calculation of SHA-256 and SHA-224 is identical; only the initialization vector is different and 32 bits of the result are discarded). For a detailed flow, please refer to **Section 5.4.4**.

If the calculation of the hash value has to be interrupted after some data blocks have been written (also see **Section 5.2.3**), software has to read the current value from **HASH_VAL** and later re-enter it as user-defined initialization vector into the **HASH_IVIN** register. This is the same flow used for other hash algorithms as well; however, note that all eight 32-bit values have to be read and restored into the initialization value register.

5.2.5 Endianness

The data format employed by each hashing algorithm is specified in the respective specification documents. Regarding byte endianness, MD-5 expects little-endian data (most-significant-byte right), while SHA-x expect big-endian data (most-significant-byte left).

To ease the usage of the hash module, the hardware takes care of arranging the bytes in the order needed by the target algorithm. Furthermore, the expected byte endianness can be programmed via configuration bits **BEND_IN** / **BEND_OUT** in the **HASH_CFG** register. **BEND_IN** defines the byte order interpretation for registers **HASH_DATA** and **HASH_IVIN**, while **BEND_OUT** defines the endianness of the **HASH_VAL** register.

The resulting 16-word input block given to the internal hash engine is shown in **Figure 5-2** for **BEND_IN** == 0 and in **Figure 5-3** for **BEND_IN** == 1. The hardware takes care of feeding each algorithm (MD-5 / SHA-1 / SHA-224 / SHA-256) with the proper byte order according to the standard.

Attention: ***Byte swapping is performed in HW for MD-5 when BEND_IN = 0 and for SHA-x when BEND_IN = 1 (i.e. when a hash function is executed with its non-default endianness). The SW must take into account HW byte swapping, so that padding data must be pre-swapped in such cases.***

Similarly, algorithm results are sent out according to the **BEND_OUT** configuration. Note that, according to the MD-5 specifications, the resulting string should be read out in little-endian format (hence **BEND_OUT** should be set to 1), while in the case of SHA-x, the result should be interpreted as big-endian (**BEND_OUT** = 0). However, the SW has full control over the desired output format.

Attention: ***Output data byte swapping is performed in HW for MD-5 when BEND_OUT = 0 and for SHA-x when BEND_OUT = 1.***

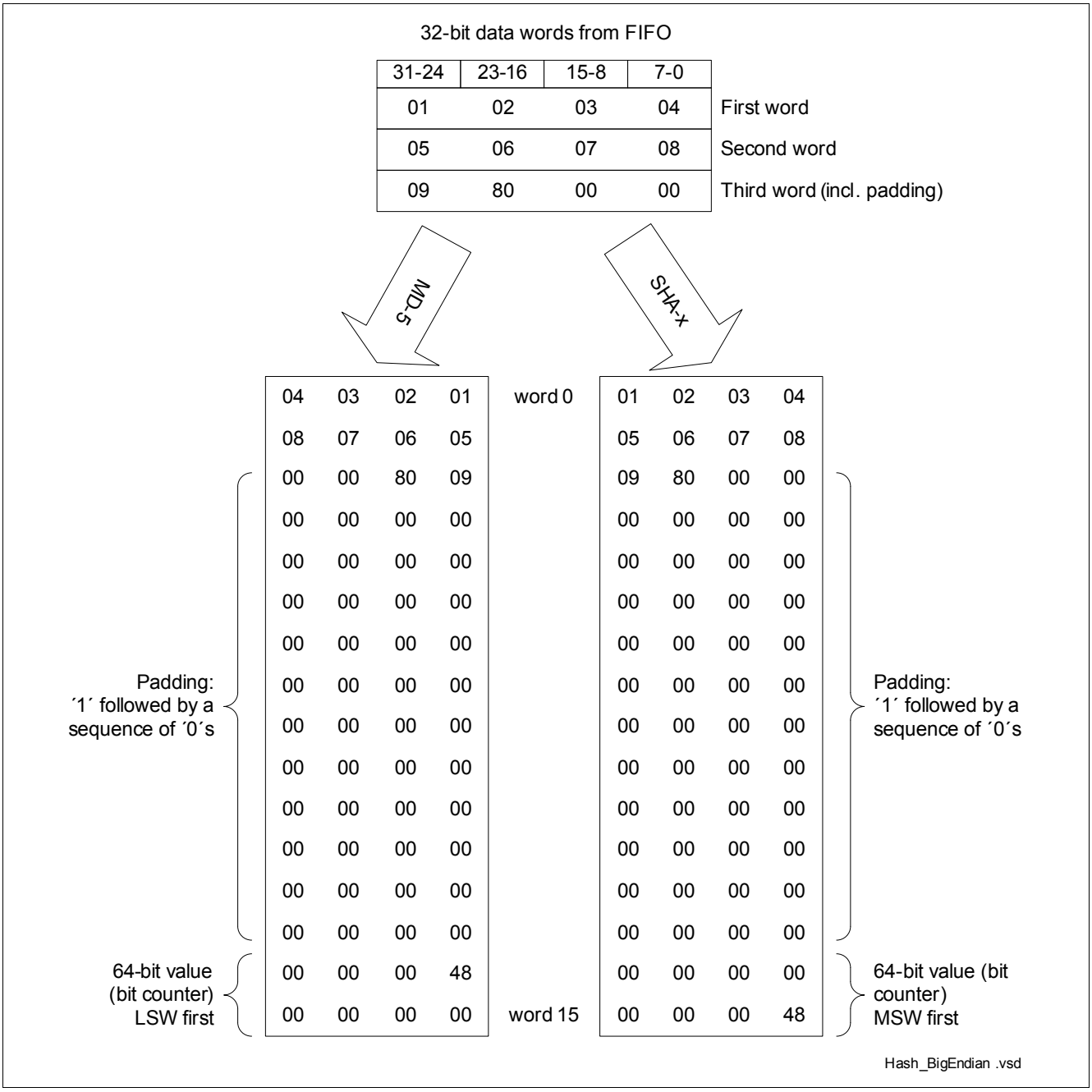


Figure 5-2 Hash Data Format Big Endian System (BEND_IN = 0)

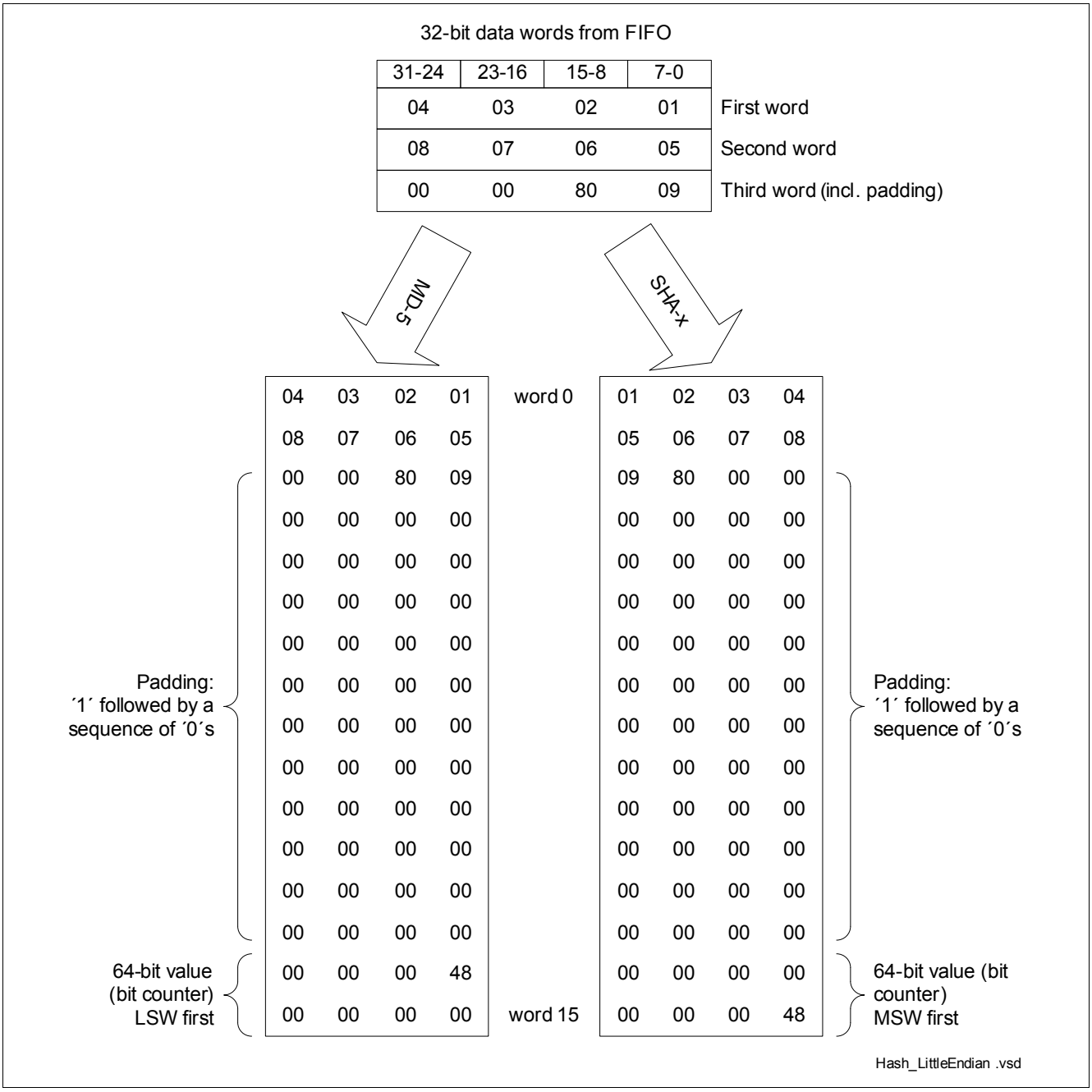


Figure 5-3 Hash Data Format Little Endian System (BEND_IN = 1)

5.3 Hash Registers

This section describes the special function registers of the hash module. All bits markes as reserved (Res) are ignored if written; when read, these bits return a value of 0.

Table 5-2 Register Address Space

Module	Base Address	End Address	Note
HASH	E800 0400$_H$	E800 07FF$_H$	Hash module

Table 5-3 Register Overview

Register Short Name	Register Long Name	Offset Address	Reset Value
Hash Registers, Configuration SFR			
HASH_CFG	HASH Configuration Register	00$_H$	0000 0000$_H$
Hash Registers, Status and Control			
HASH_STAT	HASH Status Register	04$_H$	0000 0004$_H$
HASH_IVIN	Hash Initialization Value Register	08$_H$	0000 0000$_H$
Hash Registers, Data SFRs			
HASH_VAL	Hash Output Value Register	0C$_H$	0000 0000$_H$
HASH_DATA	Hash Data Input Register	10$_H$	0000 0000$_H$

The registers are addressed wordwise.

Accesses to unused addresses result in a bus error.

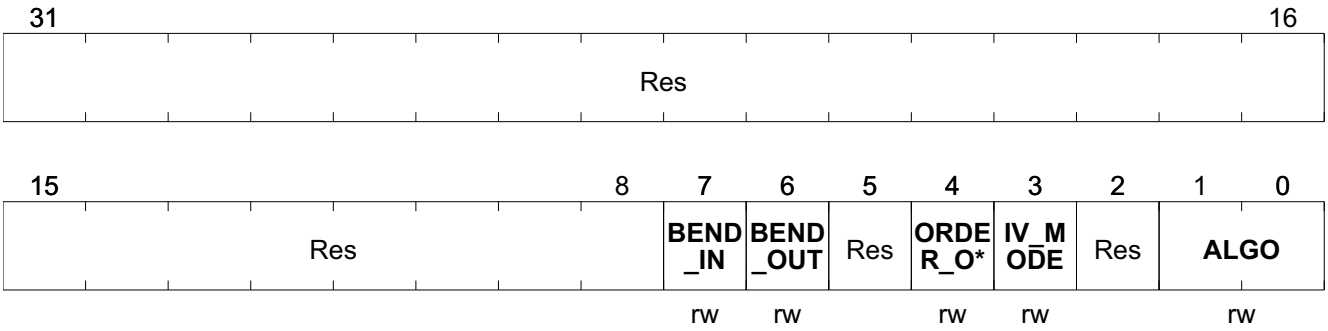
Read accesses to write-only registers or write accesses to read-only registers result in a bus error.
Registers can only be accessed 32-bit wise and 32-bit aligned. Using 8-bit, 16-bit or un-aligned accesses results in a bus error.

5.3.1 Configuration SFR

HASH Configuration Register

Writes to this register reset the hash engine. Ongoing hash operations are aborted, the internal buffer is cleared and the FIFO is cleared as well.

HASH_CFG	Offset	Reset Value
HASH Configuration Register	00_H	$0000\ 0000_H$



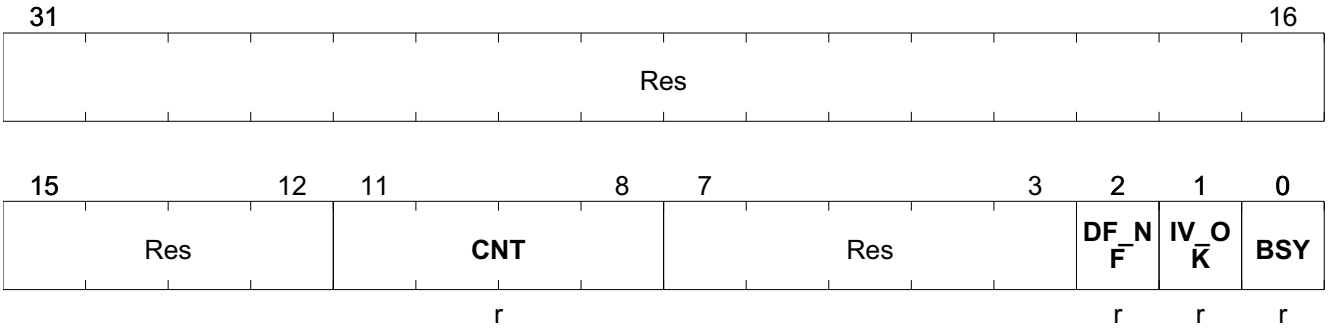
Field	Bits	Type	Description
BEND_IN	7	rw	**Input Byte Endianness** This bit defines the byte order interpretation of the 32-bit input words to the hash engine. This bit affects the byte order when data is written to **HASH_DATA** or **HASH_IVIN** (see **Section 5.2.5**). 0_B **BIG**, The MSB is the left-most byte. 1_B **LIT**, The MSB is the right-most byte.
BEND_OUT	6	rw	**Output Byte Endianness** This bit defines the byte order interpretation of the 32-bit output words of the hash engine. This bit affects the byte order when data is read from **HASH_VAL** (see **Section 5.2.5**). 0_B **BIG**, The MSB is the left-most byte. 1_B **LIT**, The MSB is the right-most byte.
ORDER_OUT	4	rw	**Output Word Sequence Order** This bit defines whether the most significant word or the least significant word is output first when the hash digest is read from **HASH_VAL** (see **Section 5.2.2**). 0_B **LSW_FIRST**, First word of the output packet is the least significant word 1_B **MSW_FIRST**, First word of the output packet is the most significant word

Field	Bits	Type	Description
IV_MODE	3	rw	**Initialization Value Mode** Selects whether the starting state of the hash algorithm will be the initialization value programmed to the **HASH_IVIN** port rather than the default starting state. *Note: This bit is not automatically reset after a user-defined initialization vector has been written; also refer to Section 5.2.3.* 0_B **NO_IV**, Hashing starts from scratch. The initialization value is ignored. 1_B **USE_IV**, The starting state of the hash algorithm is determined by the previously written initialization value.
ALGO	1:0	rw	**Hash Algorithm** Selects the hash function to be executed. To calculate SHA-224 hash values, please refer to **Section 5.2.4**. 00_B **NONE**, No hash function is executed. 01_B **MD5**, MD-5 hash function is executed. 10_B **SHA1**, SHA-1 hash function is executed. 11_B **SHA256**, SHA-256 hash function is executed.

5.3.2 Status and Control

HASH Status Register

HASH_STAT	Offset	Reset Value
HASH Status Register	04_H	$0000\ 0004_H$

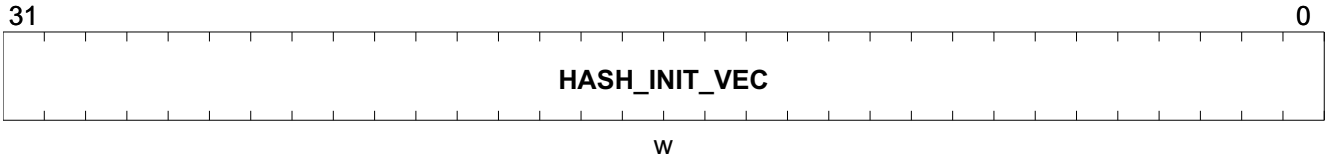


Field	Bits	Type	Description
CNT	11:8	r	**Result Counter** After a hash operation finishes (bit **BSY** indicates idle state), this bit field is set to the number of 32-bit words available in the hash result register **HASH_VAL**. When the result register is read, each read of a 32-bit value decrements this field. If this bit field is 0 and data is read from **HASH_VAL**, zeroes will be returned and the CNT field is not updated anymore.
DF_NF	2	r	**Data FIFO Not Full** If this bit is set, the data FIFO is not full, that means, additional data can be written into the **HASH_DATA** register. 0_B **FIFO_FULL**, Data FIFO currently full. 1_B **FIFO_NOTFULL**, Data can be written into **HASH_DATA**.
IV_OK	1	r	**Initialization Vector OK** This bit indicates that a valid initialization vector has been written into the **HASH_IVIN** register. Depending on the chosen hash algorithm, this bit is set to 1 by hardware after the needed amount of bits has been written to **HASH_IVIN** (see **Section 5.2.3**). If this bit is set and write accesses to **HASH_IVIN** occur, it is reset until a full initialization vector was written again. This bit is also reset if the hash algorithm is changed in **HASH_CFG**. 0_B **IV_INVALID**, Custom initialization vector not valid. 1_B **IV_VALID**, Custom initialization vector is valid.

Field	Bits	Type	Description
BSY	0	r	**Busy Flag** This bit indicates the busy status of the hash module. Note that this bit is set as soon as data is written into **HASH_DATA**. 0_B **HASH_IDLE**, No operation. 1_B **HASH_BUSY**, A hashing operation or initialization is ongoing.

Hash Initialization Value Register

HASH_IVIN	Offset	Reset Value
Hash Initialization Value Register	08_H	$0000\ 0000_H$

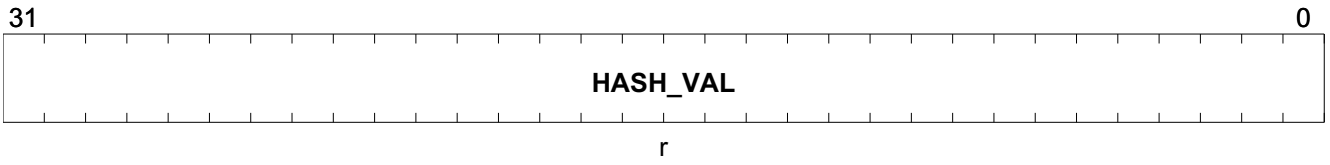


Field	Bits	Type	Description
HASH_INIT_VEC	31:0	w	**Hash Initialization Value** If bit **HASH_CFG.IV_MODE** is set, the initialization value for the hash calculation is taken from this register. Otherwise, this register is ignored and the default initialization value for the selected hash algorithm is used. Note that the amount of data this register holds is dependent on the selected hash algorithm; be sure to set the algorithm before writing this register. MD-5 needs 4 words (128 bit) while for the SHA-1 algorithm, 5 words (that means, 160 bits) have to be written. For SHA-224 or SHA-256, 8 words (256 bits) must be written. Write operations to this register have to occur as consecutive accesses (that means, no accesses to other registers with the exception of reading **HASH_STAT** must occur). If an init value for SHA-256 shall be written, this means 8 consecutive write accesses to this register.

5.3.3 Data SFRs

Hash Output Value Register

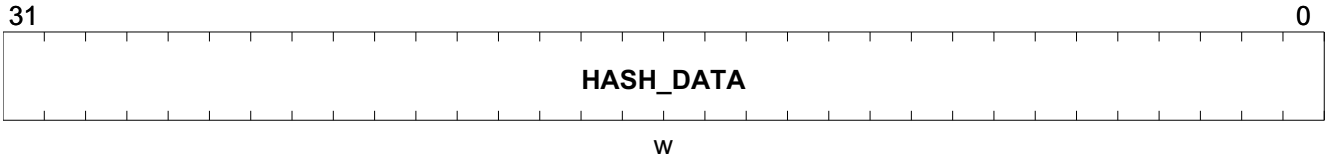
HASH_VAL	Offset	Reset Value
Hash Output Value Register	$0C_H$	$0000\ 0000_H$



Field	Bits	Type	Description
HASH_VAL	31:0	r	**Hash Output Value** After hashing has finished, the output value can be read from this register. Depending on the selected hash algorithm, this needs several consecutive read accesses (4 reads for MD-5, 5 reads for SHA-1 and 8 reads for SHA-256; SHA-224 needs 7 or 8 reads depending on the output word order, see **Section 5.2.2**). Each read decrements the **HASH_STAT.CNT** bit field (this bit field counts the number of 32-bit words available as result) until that field is 0.

Hash Data Input Register

HASH_DATA	Offset	Reset Value
Hash Data Input Register	10_H	$0000\ 0000_H$



Field	Bits	Type	Description
HASH_DATA	31:0	w	**Hash Input Data** Data to be hashed is written into this register. This fills the internal data FIFO; after 512 bits have been written, the whole data block is copied from the FIFO into the hash buffer and is hashed. In the meantime, the data FIFO can be filled again. Note that hashing only starts after exactly 512 bits have been written into this register (that means 16 write accesses). No automatic padding of the data block is done in the module.

5.4 Programming Model

This section describes how to program the module for various hash operations.

5.4.1 Input Data Size and Padding

For all the algorithms (MD-5 and SHA-x), the input message is split into blocks of 512 bits (that means 16 words), then each block is processed one after the other. A padding string is appended to the message in such a way that:

- A bit with value 1 is appended to the message;
- A sequence of bits with value 0 is appended;
- At the end of the 16-words block, a 64-bit representation of the effective message length in bits (without any padding) is appended.

The message size is appended as two 32-bit words. For MD-5, the low order word is appended first while for SHA-x, the high-order word is given first. If the message plus the padding sequence and the bit counter do not fit into the current 512-bit block, an additional 512-bit block is added which includes as many padding 0-valued bits as needed. The bit counter value must always end on a 512-bit block boundary regardless of the number of padding bits needed (which implies that if the message size is a multiple of 512 bits, another 512-bit block containing only the padding has to be added).

5.4.2 Calculating an MD-5 Hash

To calculate an MD-5 hash digest of a given data stream, follow these steps:

1. Write **HASH_CFG.ALGO** to 01_B and **HASH_CFG.IV_MODE** to 0_B (set the other bits of this register as appropriate for the endianness of the software/platform; for a little-endian platform, the BEND_* bits are usually set to 1 since MD-5 expects little-endian data; also see **Section 5.2.5**).
2. Write 64 bytes of data to be hashed to **HASH_DATA** (if less than 64 bytes of data are left, append the padding and the bit counter; this always gives a full 64-byte data block again).
3. If all data (including padding and bit counter) has been written, goto step 5.
4. Poll until **HASH_STAT.DF_NF** is 1, then goto step 2.
5. Poll until **HASH_STAT.BSY** is 0.
6. While **HASH_STAT.CNT** is not 0, read the hash result from register **HASH_VAL** (note that the output order bit **HASH_CFG.ORDER_OUT** which was set in step 1 influences whether the most significant word or the least significant word of the result is read first).

5.4.3 Calculating a SHA-1 or SHA-256 Hash

To calculate a SHA-1 or SHA-256 hash digest of a given data stream, follow these steps:

1. Write **HASH_CFG.ALGO** to 10_B (for SHA-1) or 11_B (for SHA-256) and **HASH_CFG.IV_MODE** to 0_B (set the other bits of this register as appropriate for the endianness of the software/platform; for a big-endian platform, the BEND_* bits are usually set to 0 since SHA-x expects big-endian data; also see **Section 5.2.5**).
2. Write 64 bytes of data to be hashed to **HASH_DATA** (if less than 64 bytes of data are left, append the padding and the bit counter; this always gives a full 64-byte data block again).
3. If all data (including padding and bit counter) has been written, goto step 5.
4. Poll until **HASH_STAT.DF_NF** is 1, then goto step 2.
5. Poll until **HASH_STAT.BSY** is 0.

6. While **HASH_STAT.CNT** is not 0, read the hash result from register **HASH_VAL** (note that the output order bit **HASH_CFG.ORDER_OUT** which was set in step 1 influences whether the most significant word or the least significant word of the result is read first).

5.4.4 Calculating a SHA-224 Hash

To calculate a SHA-224 hash digest of a given data stream, the SHA-256 mode with a different initialization vector is used. Follow these steps:

1. Write **HASH_CFG.ALGO** to 11_B (for SHA-256) and **HASH_CFG.IV_MODE** to 1_B (set the other bits of this register as appropriate for the endianness of the software/platform; for a big-endian platform, the BEND_* bits are usually set to 0 since SHA-x expects big-endian data; also see **Section 5.2.5**).
2. Write the following initialization vector to **HASH_IVIN** (first word is the most significant word and has to be written first as well, all eight words are hexadecimal 32-bit values):
   C1059ED8 367CD507 3070DD17 F70E5939 FFC00B31 68581511 64F98FA7 BEFA4FA4
3. (optional) Check that **HASH_STAT.IV_OK** is 1; if not, goto step 1.
4. Write 64 bytes of data to be hashed to **HASH_DATA** (if less than 64 bytes of data are left, append the padding and the bit counter; this always gives a full 64-byte data block again).
5. If all data (including padding and bit counter) has been written, goto step 7.
6. Poll until **HASH_STAT.DF_NF** is 1, then goto step 4.
7. Poll until **HASH_STAT.BSY** is 0.
8. While **HASH_STAT.CNT** is not 0, read the hash result from register **HASH_VAL** (note that the output order bit **HASH_CFG.ORDER_OUT** which was set in step 1 influences whether the most significant word or the least significant word of the result is read first).
9. Depending on the set output order, the SHA-224 hash digest of the input data either consists of the first seven words or the last seven words read from the **HASH_VAL** register (see **Section 5.2.2**).

5.4.5 Calculating Hashes in Different Applications

As outlined in **Section 5.2.3**, the module supports user defined initialization vectors. Assume that two independent applications A and B want to use the module concurrently to hash their respective data streams; both applications use the same settings for **HASH_CFG** (that means, the same byte order, word order and the same hash algorithm). This can be done in the following way:

1. Initialize the hash digest values in both applications to the initialization vector of the used algorithm (for instance, SHA-1 uses the initial vector 67452301 EFCDAB89 98BADCFE 10325476 C3D2E1F0; the first word is the most significant one).
2. Initialize the hash module by writing **HASH_CFG** appropriately with **HASH_CFG.IV_MODE** set to 1_B.
3. When application A is scheduled to use the module:
   a) Write current hash digest of A to **HASH_IVIN**.
   b) Check that **HASH_STAT.IV_OK** is 1.
   c) Write 64-byte data block to **HASH_DATA**.
   d) Poll until **HASH_STAT.BSY** is 0.
   e) Read the hash digest from **HASH_VAL** and save it as current digest of application A.
4. When application B is scheduled to use the module:
   a) Write current hash digest of B to **HASH_IVIN**.

b) Check that **HASH_STAT.IV_OK** is 1.

c) Write 64-byte data block to **HASH_DATA**.

d) Poll until **HASH_STAT.BSY** is 0.

e) Read the hash digest from **HASH_VAL** and save it as current digest of application B.

5. Repeat steps 3 and 4 until all data of the applications has been hashed.

Obviously, the applications can hash more than one block of data when they are scheduled to use the module (but steps 3a...3e and 4a...4e of the above description must not be interrupted by a task switch). If the applications use different settings or algorithms, each application has to initialize the module by itself (that means, step 2 of the description above would have to be done by each application whenever it is scheduled to use the hash module).

Note that for SHA-224, it is mandatory that all eight words of the intermediate digest are read out and later written back as initialization vector (even though for the final result, only seven words are valid, see **Section 5.2.4**).

5.5 Debug Support

Due to the fact that most accesses to the module are either destructive read operations or change the internal state of the hash engine, the debug support of this module is limited.

If a user-defined initialization vector is used and is written into **HASH_IVIN**, it is not possible to read back that vector (or any part of it) since **HASH_IVIN** is a write-only register. Similarly, hash data written to the data FIFO via the **HASH_DATA** register cannot be read back; it is not possible to read the FIFO fill level.

The hash digest register **HASH_VAL** is read-only. The result can only be read once from this register (destructive read), it is not possible in any way to read the result (or parts of it) in a debug break environment and then restore that result to be available again in a runtime environment.

Note that it is possible to read the **HASH_STAT** register while an initialization vector is written into **HASH_IVIN**; however, no other register accesses must occur during that write of the user-defined vector.

6 PKC Module

This chapter describes the HSM's Public Key Cryptography module PKC. It contains the following sections:

- **Introduction**.
- **Algorithms**.
- **Register Architecture and Programming Model**.
- **PKC SFRs**.
- **Shared Crypto Memory**.
- **Further detailed definition/specification of commands**
- **Pseudo Coding Example with Test Data**

6.1 Introduction

The PKC module is a hardware module that supports fast signature generation and verification with ECDSA.

In particular, it enables modular and non-modular operations on integers and binary polynomials up to 256 bit length:

- Multiplication
- Modular addition and subtraction
- Modular multiplication
- Modular inversion and division
- Modular exponentiation

It enables also complex algorithms on all common elliptic curves of bitlength up to 256, like

- Addition of two points in affine coordinates
- Doubling of a point in affine coordinates
- Scalar multiplication

The supported curves are all curves defined over finite fields of the type $\mathbf{F}_p$ and GF(2^d)= $\mathbf{F}_2$[X]/f of bitlength up to 256 bit length. This includes the NIST curves P-192, P-224, P-256, K-163, B-163, K-233, B-233, as well as the Brainpool curves brainpoolP160r1, brainpoolP192r1, brainpoolP224r1, brainpoolP256r1.

Additionally support for operations on Curve25519 and Ed25519 is included.

Furthermore, the PKC module supports the following features:

- Storage for 32 values (integers or binary polynomials) of up to 256 bit length.
- Generation of 200 ECDSA-signature/s @100MHz for elliptic curves of key-length 256.
- Verification of 100 ECDSA-signature/s @100MHz for elliptic curves of key-length 256.

6.2 Algorithms

The PKC module enables the efficient implementation of the operations and algorithms described below.

For the following, the notations, concepts, and abbreviations are used:

- **Z**:
  - **Ring of integers**: **Z** = {..., -3, -2, -1, 0, 1, 2, 3, ...}
  - The elements are usually represented as signed or unsigned integers, as known from, e.g., C-language. Note that on a real physical device, the representation of integers is always restricted to a certain subset of **Z**, e.g., $[0,2^n[$ or $[-2^n,2^n[$.
  - Furthermore addition (ADD), subtraction (SUB), and multiplication (MULT) operations are defined on the elements of **Z**. They are the well known operations on integers.
- $\mathbf{F}_2$:
  - **Field of two elements**, also sometimes written as GF(2).
  - The elements are the values 0 and 1, and are usually represented by a single bit.
  - Furthermore addition (ADD), subtraction (SUB), and multiplication (MULT) operations are defined on the elements of $\mathbf{F}_2$. Addition and Subtraction are the same operation and correspond to the logic XOR operation. Multiplication corresponds to the logic AND operation.
- $\mathbf{F}_2[X]$:
  - **Ring of binary polynomials**.
  - The elements f are polynomials in X with coefficients a_i in $\mathbf{F}_2$:
    $f(X) = a_d X^d + a_{d-1} X^{d-1} + ... + a_1 X^1 + a_0 X^0$.
    If a_d=1, then d is called the **degree** deg(f) of f. Additionally one sets deg(0) := -1.
    Usually the element f is represented as the bit-string
    $(a_d\ a_{d-1} ... a_1\ a_0)$
    and therefore often identified with the corresponding (unsigned) integer of the value f(2), i.e.,
    $(a_d\ a_{d-1} ... a_1\ a_0)_2 = a_d\ 2^d + a_{d-1}\ 2^{d-1} + ... + a_1\ 2^1 + a_0$.
    Hence f is usually stored the same way the integer $f(2)=(a_d\ a_{d-1} ... a_1\ a_0)_2$ would be stored.
  - Here, addition (ADD), subtraction (SUB), and multiplication (MULT) operations are defined: They differ from the well known operations on integers. An addition and subtraction consists of a bitwise XOR operation on the usual corresponding integer representation. The multiplication is more complex and built-up by additions.
- mod:
  - for any positive integer n, an integer or expression followed by “mod n” denotes the **unique** integer in the interval [0,n[which has the same modulo-n-remainder as the expression itself:
    For a fixed positive integer n, every integer a can uniquely be written in the form
    $a = q * n + r$,
    with r lying in the interval [0,n[. (This means that **Z** is a so called Euclidean Domain.) Then r = a mod n.
  - for any binary polynomial f of positive degree, a binary polynomial or expression followed by “mod f” denotes the **unique** binary polynomial of degree < deg(f) which has the same modulo-f-remainder as the expression itself.
    for a fixed non-zero polynomial f, every binary polynomial a can **uniquely** be written in the form
    $a = q * f + r$,
    with deg(r)<deg(f). (This means that $\mathbf{F}_2[X]$ is a so called Euclidean Domain.) Then r = a mod f.
- **Z**/n:

- **Residue Class Ring**: For any positive integer n, **Z**/n is a mathematical object that usually is represented by the set of n symbols
  {(0 mod n), (1 mod n),...,(n-1 mod n)}.
- In practical application, the n elements of **Z**/n are usually represented by the n integers 0, 1, ..., n-1.
- An addition, subtraction and multiplication is defined on this set, making it to a commutative ring.
- Using the usual representation with the integers 0, 1, ..., n-1, these three operations are the modular addition (ADDN), modular subtraction (SUBN), and modular multiplication (MULTN).

- $\mathbf{F}_p$:
  - **Finite Field** of p elements. p has to be a prime number.
  - $\mathbf{F}_p := \mathbf{Z}/p$.
- $\mathbf{F}_2[X]/(f)$:
  - **Residue Polynomial Ring**: For any non-zero binary polynomial f, $\mathbf{F}_2[X]/(f)$ is a mathematical object that usually is represented by the set of symbols
    { (a mod n) : for all a in $\mathbf{F}_2[X]$ with deg(a)<deg(f) }.
  - In practical application, the elements of $\mathbf{F}_2[X]/(f)$ are usually represented by the binary polynomials a of degree less than the degree of f.
  - An addition, subtraction and multiplication is defined on this set, making it to a commutative ring.
  - Using the usual representation with binary polynomials a of degree less than deg(f), these three operations are the modular addition (ADDN), modular subtraction (SUBN), and modular multiplication (MULTN).
  - If the polynomial is irreducible, i.e., if there is not non-trivial product f = g*h, then this object is a finite field and will often be denoted by $GF(2^{deg(f)})$. Also the notation $\mathbf{F}_2[X]/f$ is used.

Note: Also, not directly obvious by the notation, the following objects correspond to each other, i.e, are used in a similar way:
- Z and $F_2[X]$
- Z/n and $F_2[X]/(f)$
- F_p and $F_2[X]/(f)$ for an irreducible polynomial f, or $GF(2^d)$. However the last notation does not specify the module-polynomial f.

6.2.1 Arithmetic Operations over Z, Z/n, and F_p.

6.2.1.1 Addition in Z

For any integer a and b, the addition in **Z**, denoted as

c := ADD(a,b) (6.1)

is mathematically defined in **Z** in the well known way:

ADD(a,b) := a + b. (6.2)

6.2.1.2 Subtraction in Z

For any integer a and b, the subtraction in **Z**, denoted as

c := SUB(a,b) (6.3)

is mathematically defined in **Z** in the well known way:

SUB(a,b) := a - b. (6.4)

6.2.1.3 Multiplication in Z

For any integer a and b, the addition in **Z**, denoted as

c := MULT(a,b) (6.5)

is mathematically defined in **Z** in the well known way:

MULT(a,b) := a * b. (6.6)

6.2.1.4 Modular Reduction in Z

For any integer b and positive integer n, the modular reduction, denoted as

c := RED(b,n) (6.7)

is mathematically defined in **Z** in the following way:

RED(b,n) := b mod n. (6.8)

Note: Although, the input values may be arbitrary, the result always lies in the interval [0,n[.

6.2.1.5 Modular Addition in Z, Addition in Z/n

For any integer a and b and positive integer n, the modular addition, denoted as

c := ADDN(a,b,n) (6.9)

is mathematically defined in **Z** in the following way:

ADDN(a,b,n) := (a + b) mod n. (6.10)

This also induces a well defined addition on the ring **Z**/n.

Note: Although, the input values may be arbitrary, the result always lies in the interval [0,n[.

6.2.1.6 Modular Subtraction in Z, Subtraction in Z/n

For any integer a and b and positive integer n, the modular subtraction, denoted as

c := SUBN(a,b,n) (6.11)

is mathematically defined in **Z** in the following way:

SUBN(a,b,n) := (a - b) mod n. (6.12)

This also induces a well defined subtraction on the ring **Z**/n.

Note: Although, the input values may be arbitrary, the result always lies in the interval [0,n[.

6.2.1.7 Modular Multiplication in Z, Multiplication in Z/n

For any integer a and b and positive integer n, the modular multiplication, denoted as

c := MULTN(a,b,n) (6.13)

is mathematically defined in **Z** in the following way:

MULTN(a,b,n) := (a * b) mod n (6.14)

This also induces a well defined multiplication on the ring **Z**/n.

Note: Although, the input values may be arbitrary, the result always lies in the interval [0,n[.

6.2.1.8 Modular Inversion over Z

For any positive integer n and any integer b such that the greatest common divisor of b and n is 1, the modular inversion computation, denoted as

c := INV(b,n) (6.15)

is mathematically defined in **Z** in the following way:

INV(b,n) := b^{-1} mod n. (6.16)

c is the unique integer in the interval [0,n[that has the property 1=c * b mod n. This inverse does not exist if there exists an integer d>1 which divides b as well as n.

This also induces a well defined inverse operation in the ring **Z**/n - but not for all of its elements.

Note: In Z this can be realized by the extended Euclidean algorithm.

Note: If n is a prime, than the inverse exists for all b with b mod n > 0.

Note: Although, the input values may be arbitrary, the result always lies in the interval [0,n[.

6.2.1.9 Modular Division over Z

For any positive integer n, any integer a, and any integer b such that the greatest common division of b and n is 1, the modular division computation, denoted as

c := DIVN(a,b,n) (6.17)

is mathematically defined in **Z** in the following way:

DIVN(a,b,n) := a * b^{-1} mod n = MULTN(a,INV(b,n),n). (6.18)

This quotient does not exist if there exists an integer d>1 which divides b as well as n.

This also induces a well defined division operation in the ring **Z**/n - but not for all of its elements.

Note: Although, the input values may be arbitrary, the result always lies in the interval [0,n[.

6.2.1.10 Modular Exponentiation

For any integer a, any non-negative integer b and any positive integer n, the modular exponentiation computation, denoted as

c := EXP(a,b,n) (6.19)

is mathematically defined in **Z** in the following way:

$$EXP(a,b,n) := a^b \bmod n. \quad (6.20)$$

This also induces a well defined exponentiation operation in the ring **Z**/n.

Note: This can be realized by the usual square and multiply algorithms.

Note: Although, the input values of this mathematical operation may be arbitrary, the result always lies in the interval [0,n[.

6.2.2 Elliptic Curve Operations over F_p.

If p>3 is a prime integer and a,b, are two integers, preferably in $\mathbf{F}_p$={0,...,p-1} or in]-p,p[, such that the **Discriminant** is not zero in $\mathbf{F}_p$,

$$4 * a^3 + 27 * b^2 \bmod p \neq 0 \quad (6.21)$$

then an **elliptic curve** E(a,b,p) is defined. E(a,b,p) is a mathematical object which can be represented by a subset of ($\mathbf{F}_p \times \mathbf{F}_p$) u { **0** } via

$$E(a,b,p) := \{ (x,y) \text{ in } (F_p \times F_p) : (x^3 + a * x + b - y^2) \bmod p = 0\} \cup \{0\} \quad (6.22)$$

Additionally a formal element **0** will be added to this set. This element is called the **point at infinity**, or also the **zero-point**.

The number of elements of the set E(a,b,p) is called the **order** of the elliptic curve and is often denoted by n.

An element of the set E(a,b,p) is called a **point** on the elliptic curve E(a,b,p). On this set, an addition of points P_A,P_B is defined. There are three cases for an addition $P_C = P_A + P_B$:

- If P_A or P_B is **0** then $P_C := P_B$ or $P_C := P_A$, respectively.
- If $P_A = P_B$, the addition is a doubling operation.(**Section 6.2.2.1**)
- Else if $P_A \neq P_B$, then this is a (generic) addition operation. (**Section 6.2.2.2**)

This addition operation makes E(a,b,p) to an abelian group. Here the **inverse** point (or negative) of a point $P_A=(x_A,y_A)$ is given by:

$$- P_A = (x_A, - y_A \bmod p). \quad (6.23)$$

Every abelian group is canonically a so called **Z**-module. I.e., the integers k operate on the points P of the elliptic curve viz:

$$k * P := (...((P + P) + P) +...) + P \quad (k \text{ times}), \text{ if } k > 0 \quad (6.24)$$

$$k * P := 0, \quad \text{if } k = 0 \quad (6.25)$$

$$k * P := 0 - ((-k) P), \quad \text{if } k < 0 \quad (6.26)$$

This operation is called **scalar multiplication**.

Let P be a point on the elliptic curve E(a,b,p). The smallest positive integer n such that n*P=**0** is called the **order** of the point P. It is known, that the order always exists (and is finite). Furthermore, the order of a point divides the order of the elliptic curve.

Additionally, there are cryptographic operations on elliptic curves like the Elliptic Curve Digital Signature Algorithm (ECDSA). See **Section 6.2.2.4**

6.2.2.1 Point Doubling on an Elliptic Curve over F_p

The point doubling computation, denoted as

$$P_C := PDBL(P_A,a,b,p) \quad (6.27)$$

is mathematically defined in the following way:

If $P_A = (x_A,y_A)$ is a point on the elliptic curve E(a,b,p), represented by its affine coordinates, then the double $P_C=(x_C,y_C)$ is given by:

$$x_C := c^2 - 2* x_A \bmod p \quad (6.28)$$

$$y_C := c * (x_A-x_C) - y_A \bmod p. \quad (6.29)$$

with

$$c := (3 * x_A^2 + a) * (2 * y_A)^{-1} \bmod p. \quad (6.30)$$

If P_A lies on the curve E(a,b,p) then the double of the point, P_C, also lies on the curve.

If it happens that $y_A=0$ and a division is not possible then P_C is defined to be **0**.

6.2.2.2 Point Addition on an Elliptic Curve over F_p

The point addition computation, denoted as

$$P_C := PADD(P_A, P_B, a, b, p) \quad (6.31)$$

is mathematically defined in the following way:

If $P_A = (x_A,y_A)$, $P_B = (x_B,y_B)$ are different points on the elliptic curve E(a,b,p), represented by their affine coordinates, then the sum point $P_C=(x_C,y_C)$ is given by:

$$x_C := c^2 - x_A- x_B \bmod p \quad (6.32)$$

$$y_C := c * (x_A-x_C) - y_A \bmod p. \quad (6.33)$$

with

$$c := (y_B-y_A) * (x_B-x_A)^{-1} \bmod p. \quad (6.34)$$

If P_A and P_B lie on the curve E(a,b,p) then the sum of the point, P_C, also lies on the curve.

If it happens that $x_B = x_A$ and $y_A \neq y_B$ and therefore a division is not possible then P_C is defined to be **0**.

6.2.2.3 Scalar Multiplication on an Elliptic Curve over F_p

For any integer Point P_A on an elliptic curve and any integer k, the scalar multiplication, denoted as

$$P_C := SMULT(k,P_A,a,b,p) \quad (6.35)$$

is mathematically defined in the following way:

$$SMULT(k,P_A,a,b,p) := k * P_A. \quad (6.36)$$

6.2.2.4 ECDSA Signature Generation on an Elliptic Curve over F_p

Given the following situation:

- Let E(a,b,p) be an elliptic curve, as shown above.

- Let further P=(x,y) be a Point on E(a,b,p); the order of the point is n (this is a prime).
- Let d be a random positive integer; this will be the **(long term) private key** for the signature.
- [Let Q := d * P be the scalar product of d with the point P; this will be the **(long term) public key**.]
- Let ek be a positive integer, chosen uniformly random from the interval]1,n-1[; this will be the **(ephemeral) private key** for the signature.
- Let h be the hash of a message m for which the signature should be generated.

Then the ECDSA signature, denoted as

(r,s) := ECDSASIG(h,ek,d,n,P,a,b,p) (6.37)

is mathematically defined in the following way:

$(x_1,y_1) := P_1 := ek * P.$ (6.38)

$r := x_1 \bmod n,$ (6.39)

$kinv := ek^{-1} \bmod n$ (6.40)

$s := kinv * (h + d * r) \bmod n$ (6.41)

If it happens that r=0 or s=0 then no signature is generated with this ek. Then a new value for ek has to be chosen and this process has to be done again.

Attention: The above description does not reflect a certain standard. It just states the mathematical formula. E.g., which hash algorithm is used and how it is padded is ignored for this description.

6.2.2.5 ECDSA Signature Verification on an Elliptic Curve over F_p

Given the following situation:

- Let E(a,b,p) be an elliptic curve, as shown above.
- Let further P=(x,y) be a Point on E(a,b,p); the order of the point is n (usually this is a prime).
- [Let d be a random positive integer; this will be the **(long term) private** key for the signature.]
- Let Q := d * P be the scalar product of d with the point P; this will be the **(long term) public key**.
- [Let ek be a positive integer, chosen uniformly random from the interval]1,n-1[; this will be the **(ephemeral) private key** for the signature.]
- Let h be the hash of a message m for which the signature should be verified.
- Given a pair (r,s) of integers from **Z**/n pretending to be a signature for the message m.

Then the ECDSA signature verification, denoted as

boolean := ECDSAVER(h,r,s,Q,n,P,a,b,p) (6.42)

is mathematically defined in the following way:

$w = s^{-1} \bmod n.$ (6.43)

$u_2 := h * w \bmod n$ (6.44)

$u_3 := r * w \bmod n$ (6.45)

$P_2 := u_2 * P \bmod n$ (6.46)

$P_3 := u_3 * Q \bmod n$ (6.47)

$(x_1,y_1) = P_1 := P_2 + P_3$ (6.48)

if $P_1 = 0$ (point at infinity) then return false, (6.49)

$v := x_1 \bmod n$ (6.50)

if r = v then return true else return false. (6.51)

If it happens that r, s are not in [1,n-1] (in particular, if r=0 or s=0), then the signature is also rejected (false).

Attention: The above description does not reflect a certain standard. It just states the mathematical formula. E.g., which hash algorithm is used and how it is padded is ignored for this description.

6.2.3 Elliptic Curve Operations on Curve25519.

If p>3 is a prime integer, and A, B, are two integers, preferably in $\mathbf{F}_p$={0,...,p-1} or in]-p,p[, such that the **Discriminant** is not zero in $\mathbf{F}_p$, i.e.,

$B * (A^2 - 4) \bmod p \neq 0$ (6.52)

then an **elliptic curve in Montgomery form** (short **Montgomery curve**) M(A,B,p) is defined. M(A,B,p) is an alternative representation for some elliptic curve which can be represented by a subset of $(\mathbf{F}_p \times \mathbf{F}_p) \cup \{ \mathbf{0} \}$ via

$M(A,B,p) := \{ (x,y) \text{ in } (F_p \times F_p) : (x^3 + A * x^2 + x - B*y^2) \bmod p = 0\} \cup \{0\}.$ (6.53)

Additionally a formal element **0** will be added to this set. This element is called the **point at infinity**, or also the **zero-point**.

The number of elements of the set M(a,b,p) is called the **order** of the elliptic curve and is often denoted by n.

An element of the set M(A,B,p) is called a **point** on the elliptic curve M(A,B,p). On this set, an addition of points P_A,P_B is defined. There are three cases for an addition $P_C = P_A + P_B$:

- If P_A or P_B is **0** then $P_C := P_B$ or $P_C := P_A$, respectively.
- If $P_A = P_B$, the addition is a doubling operation.(**Section 6.2.3.1**)
- Else if $P_A \neq P_B$, then this is a (generic) addition operation. (**Section 6.2.3.2**)

This addition operation makes M(A,B,p) to an abelian group. Here the **inverse** point (or negative) of a point $P_A=(x_A,y_A)$ is given by:

$- P_A = (x_A, - y_A \bmod p).$ (6.54)

Every abelian group is canonically a so called **Z**-module. I.e., the integers k operate on the points P of the elliptic curve viz:

$$k * P := (...((P + P) + P) + ...) + P \quad (k \text{ times}), \text{ if } k > 0 \tag{6.55}$$

$$k * P := 0, \quad \text{if } k = 0 \tag{6.56}$$

$$k * P := 0 - ((-k) P), \quad \text{if } k < 0 \tag{6.57}$$

This operation is called **scalar multiplication**.

Let P be a point on the elliptic curve E(a,b,p). The smallest positive integer n such that n*P=**0** is called the **order** of the point P. It is known, that the order always exists (and is finite). Furthermore, the order of a point divides the order of the elliptic curve.

For the special case of the values:

$$p := 2^{255} - 19 \tag{6.58}$$

$$A := 486662 \tag{6.59}$$

$$B := 1 \tag{6.60}$$

this curve is called **Curve25519**. The standard generator point P=(x,y) for applications is given by

$$x := 9 \tag{6.61}$$

$$y := ... \text{ (undefined)} \tag{6.62}$$

the order of the point is 2^{252}+27742317777372353535851937790883648493.

6.2.3.1 Point Doubling on Curve25519

The point doubling computation on Curve25519 or Montgomery curves in general, denoted as

$$P_C := \text{M-PDBL}(P_A,A,B,p) \tag{6.63}$$

is mathematically defined in the following way:

If $P_A = (x_A,y_A)$ is a point on the Montgomery curve M(A,B,p), represented by its affine coordinates, then the double $P_C=(x_C,y_C)$ is given by:

$$x_C := B * c^2 - A - x_A - x_A \bmod p \tag{6.64}$$

$$y_C := (2x_A+x_A+A) *c - B * c^3 - y_A \bmod p. \tag{6.65}$$

with

$$c := (3 x_A^2 + 2 * A + 1) * (2 * B * y_A)^{-1} \bmod p. \tag{6.66}$$

If P_A lies on the curve M(A,B,p) then the double of the point, P_C, also lies on the curve.

If it happens that $y_A=0$ and a division is not possible then P_C is defined to be **0**.

6.2.3.2 Point Addition on Curve25519

The point addition computation on Curve25519 or Montgomery curves in general, denoted as

$$P_C := \text{M-PADD}(P_A, P_B, A, B, p) \tag{6.67}$$

is mathematically defined in the following way:

If $P_A = (x_A,y_A)$, $P_B = (x_B,y_B)$ are different points on the elliptic curve M(A,B,p), represented by their affine coordinates, then the sum point $P_C=(x_C,y_C)$ is given by:

$$x_C := B * c^2 - A - x_A - x_B \bmod p \quad (6.68)$$

$$y_C := (2x_A+x_B+A) * c - B * c^3 - y_A \bmod p. \quad (6.69)$$

with

$$c := (y_B-y_A) * (x_B-x_A)^{-1} \bmod p. \quad (6.70)$$

If P_A and P_B lie on the curve M(a,b,p) then the sum of the point, P_C, also lies on the curve.

If it happens that $x_B = x_A$ and $y_A \neq y_B$ and therefore a division is not possible then P_C is defined to be **0**.

6.2.3.3 Scalar Multiplication on Curve25519

For any integer Point P_A on Curve25519 or Montgomery curves in general and any integer k, the scalar multiplication, denoted as

$$P_C := \text{SMULTcurve25519}(k,P_A) \quad (6.71)$$

or in general

$$P_C := \text{M-SMULT}(k,P_A,A,B,p) \quad (6.72)$$

is mathematically defined in the following way:

$$\text{SMULT}(k,P_A,A,B,p) := k * P_A. \quad (6.73)$$

6.2.4 Elliptic Curve Operations on the Twisted Edwards Curve Ed25519.

If p>3 is a prime integer, and a, d, are two integers, preferably in $\mathbf{F}_p$={0,...,p-1} or in]-p,p[, such that in $\mathbf{F}_p$

$$a * d * (a - d) \bmod p \neq 0 \quad (6.74)$$

then a **twisted Edwards curve** Ed(a,d,p) is defined. Ed(a,d,p) is an alternative representation for some elliptic curve which can be represented by a subset of ($\mathbf{F}_p \times \mathbf{F}_p$)} via

$$\text{Ed}(a,d,p) := \{ (x,y) \text{ in } (F_p \times F_p) : (a * x^2 + y^2 - 1 - d*x^2*y^2) \bmod p = 0\}. \quad (6.75)$$

The **zero-point** is given by the point (0,1).

The number of elements of the set Ed(a,d,p) is called the **order** of the curve and is often denoted by n.

An element of the set Ed(a,d,p) is called a **point** on the curve Ed(a,d,p). On this set, an addition of points P_A,P_B is defined: $P_C = P_A + P_B$:

This addition operation makes Ed(a,d,p) to an abelian group. Here the **inverse** point (or negative) of a point $P_A=(x_A,y_A)$ is given by:

$$- P_A = (-x_A \bmod p, y_A). \quad (6.76)$$

Every abelian group is canonically a so called **Z**-module. I.e., the integers k operate on the points P of the elliptic curve viz:

$$k * P := (...((P + P) + P) +...) + P \quad (k \text{ times}), \text{ if } k > 0 \quad (6.77)$$

$$k * P := 0, \quad \text{if } k = 0 \quad (6.78)$$

$$k * P := 0 - ((-k) P), \quad \text{if } k < 0 \quad (6.79)$$

This operation is called **scalar multiplication**.

Let P be a point on the elliptic curve Ed(a,b,p). The smallest positive integer n such that n*P=**0** is called the **order** of the point P. It is known, that the order always exists (and is finite). Furthermore, the order of a point divides the order of the elliptic curve.

For the special case of the values:

$$p := 2^{255} - 19 \quad (6.80)$$

$$a := -1 \quad (6.81)$$

$$d := 121666^{-1} - 1 \bmod p \quad (6.82)$$

this curve is called **Ed25519**. The standard generator point P=(x,y) for applications is given by

$$x := 216936D3CD6E53FEC0A4E231FDD6DC5C692CC7609525A7B2C9562D608F25D51A_H \quad (6.83)$$

$$y := 6666666666666666666666666666666666666666666666666666666666666658_H \quad (6.84)$$

the order of the point is 2^{252}+27742317777372353535851937790883648493.

6.2.4.1 Point Addition on Ed25519

The point addition computation on Ed25519 or on twisted Edwards curves in general, denoted as

$$P_C := \text{Ed-PADD}(P_A, P_B, a, d, p) \quad (6.85)$$

is mathematically defined in the following way:

If $P_A = (x_A,y_A)$, $P_B = (x_B,y_B)$ are (not necessarily different) points on the elliptic curve Ed(a,d,p), represented by their affine coordinates, then the sum point $P_C=(x_C,y_C)$ is given by:

$$x_C := (x_Ay_B+y_Ax_B)\,(1+dx_Ax_By_Ay_B)^{-1} \bmod p \quad (6.86)$$

$$y_C := (y_Ay_B-ax_Ax_B)\,(1-dx_Ax_By_Ay_B)^{-1} \bmod p. \quad (6.87)$$

If P_A and P_B lie on the curve Ed(a,d,p) then the sum of the point, P_C, also lies on the curve.

6.2.4.2 Scalar Multiplication on Ed25519

For any integer Point P_A on Ed25519 or twisted Edwards curves in general and any integer k, the scalar multiplication, denoted as

$$P_C := \text{SMULTEd25519}(k,P_A) \quad (6.88)$$

or in general

$$P_C := \text{Ed-SMULT}(k,P_A,a,d,p) \quad (6.89)$$

is mathematically defined in the following way:

$$\text{Ed-SMULT}(k,P_A,a,d,p) := k * P_A. \quad (6.90)$$

6.2.4.3 X Coordinate Recovery on Ed25519

If only the y-coordinate of a Point P_A=(x,y) on Ed25519 is given, then the x-coordinate can be recovered - at least up to the sign modulo p:

x := XrecoverEd25519(y) (6.91)

is mathematically evaluated in the following way:

$u := y^2 - 1 \bmod p.$ (6.92)

$v := d * y^2 + 1 \bmod p$ (6.93)

$e := (p-5)/8$ (6.94)

$x' := u^{1+e} * v^{3+7e} \bmod p$ (6.95)

if $(v * x^2 + u \bmod p = 0)$ then $x := x' * J \bmod p$ (6.96)

else x:= x'. (6.97)

Here J is a square root of -1 mod p. In this case it is:

$J := 2B8324804FC1DF0B2B4D00993DFBD7A72F431806AD2FE478C4EE1B274A0EA0B0_H$. (6.98)

6.2.5 Arithmetic Operations over $F_2[X]$ and $F_2[X]/(f)=GF(2^d)$.

6.2.5.1 Addition in $F_2[X]$

For any binary polynomial a and b, the addition in $\mathbf{F}_2[X]$, denoted as

c := ADD(a,b) (6.99)

is mathematically defined in $\mathbf{F}_2[X]$ as the addition of polynomials in the well known way:

ADD(a,b) := a + b. (6.100)

6.2.5.2 Subtraction in $F_2[X]$

For any binary polynomial a and b, the subtraction in $\mathbf{F}_2[X]$, denoted as

c := SUB(a,b) (6.101)

is mathematically defined in $\mathbf{F}_2[X]$ as the subtraction of polynomials in the well known way:

SUB(a,b) := a - b. (6.102)

Note: In the case of characteristic 2 (as it is here) the subtraction is identical to the addition. More general, the additive inverse (negative) of a polynomial a is the polynomial itself.

6.2.5.3 Multiplication in $F_2[X]$

For any binary polynomial a and b, the addition in $\mathbf{F}_2[X]$, denoted as

c := MULT(a,b) (6.103)

is mathematically defined in $\mathbf{F}_2[X]$ as the multiplication of polynomials in the well known way:

MULT(a,b) := a * b. (6.104)

To be precise: If

$a = a_m X^m + a_{m-1} X^{m-1} + ... + a_1 X^1 + a_0 X^0$

and

$b = b_n X^n + b_{n-1} X^{n-1} + ... + b_1 X^1 + b_0 X^0$

Then

$c = c_{n+m} X^{n+m} + c_{n+m-1} X^{n+m-1} + ... + c_1 X^1 + c_0 X^0$

with

$c_i = a_i * b_0 + a_{i-1} * b_1 + ... + a_1 * b_{i-1} + a_0 * b_i$.

Here the last computation takes place in $\mathbf{F}_2$.

6.2.5.4 Modular Reduction in $F_2[X]$

For any binary polynomial b and non-zero binary polynomial f, the modular reduction, denoted as

c := RED(b,f) (6.105)

is mathematically defined in $\mathbf{F}_2[X]$ as the reduction of polynomials in the following way:

RED(b,f) := b mod f. (6.106)

Note: Although, the input values may be arbitrary, the result always has degree < deg(f).

6.2.5.5 Modular Addition in $F_2[X]$, Addition in $F_2[X]/(f)$

For any binary polynomial a and b and non-zero binary polynomial f, the modular addition, denoted as

c := ADDN(a,b,f) (6.107)

is mathematically defined in $\mathbf{F}_2[X]$ as the modular addition of polynomials in the following way:

ADDN(a,b,f) := (a + b) mod f. (6.108)

This also induces a well defined addition on the ring $\mathbf{F}_2[X]/(f)$.

Note: Although, the input values may be arbitrary, the result always has degree < deg(f).

6.2.5.6 Modular Subtraction in $F_2[X]$, Subtraction in $F_2[X]/(f)$

For any binary polynomial a and b and non-zero binary polynomial f, the modular subtraction, denoted as

c := SUBN(a,b,f) (6.109)

is mathematically defined in $\mathbf{F}_2[X]$ as the modular subtraction of polynomials in the following way:

SUBN(a,b,f) := (a - b) mod f. (6.110)

This also induces a well defined subtraction on the ring $\mathbf{F}_2[X]/(f)$.

Note: Although, the input values may be arbitrary, the result always has degree < deg(f).

Note: Analogously to SUB, also SUBN is identical to ADDN.

6.2.5.7 Modular Multiplication in $F_2[X]$, Multiplication in $F_2[X]/(f)$

For any binary polynomial a and b and non-zero binary polynomial f, the modular addition, denoted as

c := MULTN(a,b,f) (6.111)

is mathematically defined in $\mathbf{F}_2[X]$ as the modular multiplication of polynomials in the following way:

MULTN(a,b,f) := (a * b) mod f (6.112)

This also induces a well defined multiplication on the ring $\mathbf{F}_2[X]/(f)$.

Note: Although, the input values may be arbitrary, the result always has degree < deg(f).

6.2.5.8 Modular Inversion over $F_2[X]$

For any non-zero binary polynomial f and any binary polynomial b such that the greatest common divisor of b and f is 1, the modular inversion computation, denoted as

c := INV(b,f) (6.113)

is mathematically defined in $\mathbf{F}_2[X]$ as the inversion operation of polynomials in the following way:

$INV(b,f) := b^{-1} \bmod f.$ (6.114)

c is the unique binary polynomial of degree < deg(f), that has the property 1=c * b mod f. This inverse does not exist if there exists a non-constant binary polynomial which divides b as well as f.

This also induces a well defined inverse operation in the ring $\mathbf{F}_2[X]/(f)$ - but not for all of its elements.

Note: This can be realized by the extended Euclidean algorithm.

Note: If f is an irreducible polynomial, than the inverse exists for all b with b mod f ≠ 0.

Note: Although, the input values may be arbitrary, the result always has degree < deg(f).

6.2.5.9 Modular Division over $F_2[X]$

For any non-zero binary polynomial f, any binary polynomial a, and any binary polynomial b such that the greatest common divisor of b and f is 1, the modular division computation, denoted as

c := DIVN(a,b,f) (6.115)

is mathematically defined in $\mathbf{F}_2[X]$ as the modular division of polynomials in the following way:

$DIVN(a,b,f) := a * b^{-1} \bmod f = MULTN(a,INV(b,f),f).$ (6.116)

This quotient does not exist if there exists a non-constant binary polynomial which divides b as well as n.

This also induces a well defined division operation in the ring $\mathbf{F}_2[X]/(f)$ - but not for all of its elements.

Note: Although, the input values may be arbitrary, the result always has degree < deg(f).

6.2.6 Elliptic Curve Operations over $F_2[X]/(f)=GF(2^d)$.

If f is an irreducible binary polynomial, and a,b, are two binary polynomials, preferably of degree < deg(f), such that,

b mod f ≠ 0 (6.117)

then an **elliptic curve** E(a,b,f) is defined. E(a,b,f) is a mathematical object which can be represented by a subset of ($\mathbf{F}_2$[X]/(f) × $\mathbf{F}_2$[X]/(f)) u { **0** } via

$$E(a,b,f) := \{ (x,y) \text{ in } (F_2[X]/(f) \times F_2[X]/(f)) : (x^3 + a * x^2 + b - y^2 - x * y) \bmod f = 0\} u \{0\} \quad (6.118)$$

Additionally a formal element **0** will be added to this set. This element is called the **point at infinity**, or also the **zero-point**.

The number of elements of the set E(a,b,f) is called the **order** of the elliptic curve and is often denoted by n.

An element of the set E(a,b,f) is called a **point** on the elliptic curve E(a,b,f). On this set, an addition of points P_A,P_B is defined. There are three cases for an addition $P_C = P_A + P_B$:

- If P_A or P_B is **0** then $P_C := P_B$ or $P_C := P_A$, respectively.
- If $P_A = P_B$, the addition is a doubling operation. (**Section 6.2.6.1**)
- Else if $P_A \neq P_B$, then this is a (generic) addition operation. (**Section 6.2.6.2**)

This addition operation makes E(a,b,f) to an abelian group. Here the **inverse** point (or negative) of a point $P_A=(x_A,y_A)$ is given by:

$$- P_A = (x_A, x_A + y_A \bmod p). \quad (6.119)$$

Every abelian group is canonically a so called **Z**-module. I.e., the integers k operate on the points P of the elliptic curve viz:

$$k * P := (...((P + P) + P) + ...) + P \quad (k \text{ times}), \text{ if } k > 0 \quad (6.120)$$

$$k * P := 0, \quad \text{if } k = 0 \quad (6.121)$$

$$k * P := 0 - ((-k) P), \quad \text{if } k < 0 \quad (6.122)$$

This operation is called **scalar multiplication**.

Let P be a point on the elliptic curve E(a,b,f). The smallest positive integer n such that n*P=**0** is called the **order** of the point P. It is known, that the order always exists (and is finite). Furthermore, the order of a point divides the order of the elliptic curve.

Additionally, there are cryptographic operations on elliptic curves like ECDSA. See **Section 6.2.6.4**.

6.2.6.1 Point Doubling on an Elliptic Curve over F_2[X]/(f)=GF(2^d)

The point doubling computation, denoted as

$$P_C := PDBL(P_A,a,b,f) \quad (6.123)$$

is mathematically defined in the following way:

If $P_A = (x_A,y_A)$ is a point on the elliptic curve E(a,b,f), represented by its affine coordinates, then the double $P_C=(x_C,y_C)$ is given by:

$$x_C := c^2 + c + a \bmod f \quad (6.124)$$

$$y_C := c * x_C + x_C + x_A^2 \bmod f. \quad (6.125)$$

with

$$c := (x_A + y_A) * (x_A)^{-1} \bmod f. \quad (6.126)$$

If P_A lies on the curve E(a,b,f) then the double of the point, P_C, also lies on the curve.

If it happens that x_A=0 and a division is not possible then P_C is defined to be **0**.

6.2.6.2 Point Addition on an Elliptic Curve over $F_2[X]/(f)=GF(2^d)$

The point addition computation, denoted as

$$P_C := PADD(P_A, P_B, a, b, f) \quad (6.127)$$

is mathematically defined in the following way:

If $P_A = (x_A,y_A)$, $P_B = (x_B,y_B)$ are different points on the elliptic curve E(a,b,p), represented by their affine coordinates, then the sum point $P_C=(x_C,y_C)$ is given by:

$$x_C := c^2 + c + x_A + x_B + a \bmod f \quad (6.128)$$

$$y_C := c * (x_A+x_C) + x_C + y_A \bmod f. \quad (6.129)$$

with

$$c := (y_B+y_A) * (x_B+x_A)^{-1} \bmod p. \quad (6.130)$$

If P_A and P_B lie on the curve E(a,b,p) then the sum of the point, P_C, also lies on the curve.

If it happens that $x_B = x_A$ and $y_A \neq y_B$ and therefore a division is not possible then P_C is defined to be **0**.

6.2.6.3 Scalar Multiplication on an Elliptic Curve over $F_2[X]/(f)=GF(2^d)$

For any integer Point P_A on an elliptic curve and any integer k, the scalar multiplication, denoted as

$$P_C := SMULT(k,P_A,a,b,f) \quad (6.131)$$

is mathematically defined in the following way:

$$SMULT(k,P_A,a,b,f) := k * P_A. \quad (6.132)$$

6.2.6.4 ECDSA Signature Generation on an Elliptic Curve over $F_2[X]/(f)=GF(2^d)$

Given the following situation:

- Let E(a,b,f) be an elliptic curve, as shown above.
- Let further P=(x,y) be a Point on E(a,b,f); the order of the point is n (in many cases this is two or four times of a prime).
- Let d be a random positive integer; this will be the **(long term) private key** for the signature.
- [Let Q := d * P be the scalar product of d with the point P; this will be the **(long term) public key**.]
- Let ek be a positive integer, chosen uniformly random from the interval]1,n-1[; this will be the **(ephemeral) private key** for the signature.
- Let h be the hash of a message m for which the signature should be generated.

Then the ECDSA signature, denoted as

$$(r,s) := ECDSASIG(h,ek,d,n,P,a,b,f) \quad (6.133)$$

is mathematically defined in the following way:

$$(x_1,y_1) := P_1 := ek * P. \quad (6.134)$$

$$r := x_1 \bmod n, \quad (6.135)$$

$$kinv := ek^{-1} \bmod n \quad (6.136)$$

$$s := kinv * (h + d * r) \bmod n \quad (6.137)$$

If it happens that r=0 or s=0 then no signature is generated with this ek. Then a new value for ek has to be chosen and this process has to be done again.

Attention: The above description does not reflect a certain standard. It just states the mathematical formula. E.g., which hash algorithm is used and how it is padded is ignored for this description.

6.2.6.5 ECDSA Signature Verification on an Elliptic Curve over $F_2[X]/(f)=GF(2^d)$

Given the following situation:

- Let E(a,b,f) be an elliptic curve, as shown above.
- Let further P=(x,y) be a Point on E(a,b,p); the order of the point is n (usually this is a prime).
- [Let d be a random positive integer; this will be the **(long term) private key** for the signature.]
- Let Q := d * P be the scalar product of d with the point P; this will be the **(long term) public key**.
- [Let ek be a positive integer, chosen uniformly random from the interval]1,n-1[; this will be the **(ephemeral) private key** for the signature.]
- Let h be the hash of a message m for which the signature should be verified.
- Given a pair (r,s) of integers from **Z**/n pretending to be a signature for the message m.

Then the ECDSA signature verification, denoted as

boolean := ECDSAVER(h,r,s,Q,n,P,a,b,f) (6.138)

is mathematically defined in the following way:

$w = s^{-1} \bmod n.$ (6.139)

$u_2 := h * w \bmod n$ (6.140)

$u_3 := r * w \bmod n$ (6.141)

$P_2 := u_2 * P \bmod n$ (6.142)

$P_3 := u_3 * Q \bmod n$ (6.143)

$(x_1,y_1) = P_1 := P_2 + P_3$ (6.144)

if $P_1 = 0$ (point at infinity) then return false, (6.145)

$v := x_1 \bmod n$ (6.146)

if r = v then return true else return false. (6.147)

If it happens that r, s are not in [1,n-1] (in particular, if r=0 or s=0), then the signature is also rejected (false).

Attention: The above description does not reflect a certain standard. It just states the mathematical formula. E.g., which hash algorithm is used and how it is padded is ignored for this description.

6.3 Register Architecture and Programming Model

The registers of the PKC module, which are relevant for the programmer, can be classified into two categories, namely the SFRs which are directly accessible by the CPU and the Shared Crypto Memory [SCM].

The SFRs which are all 32 bit wide, are:

- The **PKC Configuration Register PKC_CONFIG** is used to determine certain input parameter used for the command.
- The **PKC Command Register PKC_CMD** is used to define the command.
- The **PKC Control Register PKC_CTRL** is used to start the operation of the command.
- The **PKC Status Register PKC_STATUS** is used to check whether the operation is finished and to get additional output information. During the operation time no write to **PKC_CMD** and **PKC_CONFIG** is allowed, otherwise this results in an undefined behavior.

The **SCM** is a 1 KB large memory block.

- The CPU as well as the core of the PKC-Module have r/w access to this memory block. However, no simultaneous access is allowed. The CPU has to make sure that no conflicts occur. Cf. **Chapter 6.4.2**.
- The **SCM** is divided in 32 equally long sections, called Registers for long integers [LIR]. Each section can contain a 256 bit long representation of an integer or binary polynomial.

After reset, all **SCM** registers have an undefined value.

The PKC module contains two additional internal memory areas. They are used for data processing during operation of the commands and are not accessible by the CPU:

- The Tightly Coupled Memory [TCM] of size 1 KB.
- The internal µ-Code Memory ROM [µCM] of size 1 k 18-bit words.

The usual way to realize an operation, like the ones described in **Section 6.2**, is:

1. Write operands, parameters, and data to the **SCM**.
2. Write command into **PKC_CMD** and possible parameters to **PKC_CONFIG**.
3. Write **PKC_CTRL** to start the operation.
4. Wait until the operation is finished. E.g. by reading status from **PKC_STATUS**, or by waiting for interrupt.
5. Read data from **SCM**, and, if applicable, status/error information from **PKC_STATUS**. In case **PKC_IRQEnd** was used, it is recommended to read **PKC_STATUS** first and then read data.

If some data or parameters are already present in the PKC-module, e.g., from the last operation, they may be used for the next operation - if applicable.

6.3.1 Interrupts

The PKC module generates two interrupts:

- The IRQEnd [PKC_IRQEnd] indicates that the core has finished processing of the command and the results of the operation are available in the **SCM**.
  - This signal is level-sensitive, active high.
  - This signal is cleared, when reading the status register **PKC_STATUS**.

  - It is also activated when an error has been detected (then **PKC_IRQEnd** and **PKC_IRQErr** are activated simultaneaously.)
- The IRQErr [PKC_IRQErr] indicates that an error has been detected while executing parameter checking procedures.
  - This signal is level-sensitive, active high.
  - This signal is cleared when reading the status register **PKC_STATUS**.

Furthermore note: If an interrupt service routine access the PKC-module, it has to make sure that it does not disturb former PKC operations by overwriting values in **SCM**. So, either these interrupts will be disabled, or it is assured that the old values will be restored after such an intermediary interrupt.

6.4 PKC SFRs

There are 4 SFRs in the PKC module:

- **PKC_CONFIG**,
- **PKC_CMD**,
- **PKC_CTRL**,
- **PKC_STATUS**,

They are always accessible and will be defined in the present section.

Furthermore there is the Shared Crypto Memory (**SCM**). It is 1 KB in size and is structured in 32 registers. The Shared Crypto Memory is accessible by the CPU. It will be described in **Chapter 6.4.2**.

Table 6-1 Register Address Space

Module	Base Address	End Address	Note
PKC	E800 3C00$_H$	E800 3FFF$_H$	

Table 6-2 Register Overview

Register Short Name	Register Long Name	Offset Address	Reset Value
PKC SFRs, Control and Status Registers			
PKC_CONFIG	PKC Configuration Register	00$_H$	0000 0000$_H$
PKC_CMD	PKC Command Register	04$_H$	0000 0000$_H$
PKC_CTRL	PKC Control Register	08$_H$	0000 0000$_H$
PKC_STATUS	PKC Status Register	0C$_H$	0000 0000$_H$
PKC SFRs, Shared Crypto Memory			
PKC_SCM_R0	Shared Crypto Memory Register 0	0400$_H$	X$_H$
PKC_SCM_R1	Shared Crypto Memory Register 1	0420$_H$	X$_H$
PKC_SCM_R2	Shared Crypto Memory Register 2	0440$_H$	X$_H$
PKC_SCM_R3	Shared Crypto Memory Register 3	0460$_H$	X$_H$
PKC_SCM_R4	Shared Crypto Memory Register 4	0480$_H$	X$_H$
PKC_SCM_R5	Shared Crypto Memory Register 5	04A0$_H$	X$_H$
PKC_SCM_R6	Shared Crypto Memory Register 6	04C0$_H$	X$_H$
PKC_SCM_R7	Shared Crypto Memory Register 7	04E0$_H$	X$_H$
PKC_SCM_R8	Shared Crypto Memory Register 8	0500$_H$	X$_H$
PKC_SCM_R9	Shared Crypto Memory Register 9	0520$_H$	X$_H$
PKC_SCM_R10	Shared Crypto Memory Register 10	0540$_H$	X$_H$
PKC_SCM_R11	Shared Crypto Memory Register 11	0560$_H$	X$_H$
PKC_SCM_R12	Shared Crypto Memory Register 12	0580$_H$	X$_H$
PKC_SCM_R13	Shared Crypto Memory Register 13	05A0$_H$	X$_H$
PKC_SCM_R14	Shared Crypto Memory Register 14	05C0$_H$	X$_H$
PKC_SCM_R15	Shared Crypto Memory Register 15	05E0$_H$	X$_H$
PKC_SCM_R16	Shared Crypto Memory Register 16	0600$_H$	X$_H$

Table 6-2 Register Overview (cont'd)

Register Short Name	Register Long Name	Offset Address	Reset Value
PKC_SCM_R17	Shared Crypto Memory Register 17	0620_H	X_H
PKC_SCM_R18	Shared Crypto Memory Register 18	0640_H	X_H
PKC_SCM_R19	Shared Crypto Memory Register 19	0660_H	X_H
PKC_SCM_R20	Shared Crypto Memory Register 20	0680_H	X_H
PKC_SCM_R21	Shared Crypto Memory Register 21	$06A0_H$	X_H
PKC_SCM_R22	Shared Crypto Memory Register 22	$06C0_H$	X_H
PKC_SCM_R23	Shared Crypto Memory Register 23	$06E0_H$	X_H
PKC_SCM_R24	Shared Crypto Memory Register 24	0700_H	X_H
PKC_SCM_R25	Shared Crypto Memory Register 25	0720_H	X_H
PKC_SCM_R26	Shared Crypto Memory Register 26	0740_H	X_H
PKC_SCM_R27	Shared Crypto Memory Register 27	0760_H	X_H
PKC_SCM_R28	Shared Crypto Memory Register 28	0780_H	X_H
PKC_SCM_R29	Shared Crypto Memory Register 29	$07A0_H$	X_H
PKC_SCM_R30	Shared Crypto Memory Register 30	$07C0_H$	X_H
PKC_SCM_R31	Shared Crypto Memory Register 31	$07E0_H$	X_H

The registers are addressed wordwise.

Registers can only be accessed 32-bit wise and 32-bit aligned. Using 8-bit, 16-bit or un-aligned accesses lead to a bus error.

Access to unused addresses results in a bus error.

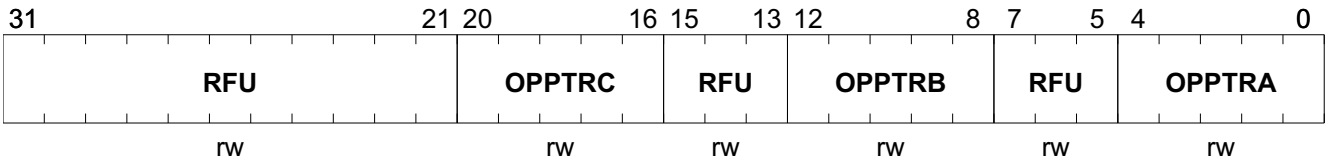
Write access to read only registers results in a bus error.

6.4.1 Control and Status Registers

PKC_CONFIG

When executing primitive arithmetic operations (like c:=a+b) and exponentiation, then the location of the input parameters a and b, as well as the location of the output parameter c will be defined by using this register. Also some points P_A, P_B, P_C, on elliptic curves are defined using this register.

PKC_CONFIG	Offset	Reset Value
PKC Configuration Register	00_H	$0000\ 0000_H$



Field	Bits	Type	Description
RFU	31:21	rw	**RFU**
OPPTRC	20:16	rw	**Pointer for Operand C** When executing primitive arithmetic operations, or certain elliptic curve operations, this pointer defines the location, where the result will be stored in memory **SCM**. i_D , Result c will be written into **PKC_SCM_Ri**. Result Point P_C will be written to **PKC_SCM_Ri** (x-coordinate) and **PKC_SCM_Ri**+1 (y-coordinate). Only values between $1(1_H)$ and $15(F_H)$ are allowed. Further restrictions cf. the descriptions of the single commands. All other values result in an undefined behavior.
RFU	15:13	rw	**RFU**
OPPTRB	12:8	rw	**Pointer for Operand B** When executing primitive arithmetic operations, or certain elliptic curve operations, this pointer defines the location, where the operand B is located in memory **SCM**. i_D , Operand b will be read from **PKC_SCM_Ri**. Point P_B will be read from **PKC_SCM_Ri** (x-coordinate) and **PKC_SCM_Ri**+1 (y-coordinate). Only values between $1(1_H)$ and $15(F_H)$ are allowed. Further restrictions cf. the descriptions of the single commands. All other values result in an undefined behavior.
RFU	7:5	rw	**RFU**

Field	Bits	Type	Description
OPPTRA	4:0	rw	**Pointer for Operand A** When executing primitive arithmetic operations, or certain elliptic curve operations, this pointer defines the location, where the operand A is located in memory **SCM**. i_D , Operand a will be read from **PKC_SCM_Ri**. Point P_A will be read from **PKC_SCM_Ri** (x-coordinate) and **PKC_SCM_Ri**+1 (y-coordinate). Only values between $1(1_H)$ and $15(F_H)$ are allowed. Further restrictions cf. the descriptions of the single commands. All other values result in an undefined behavior.

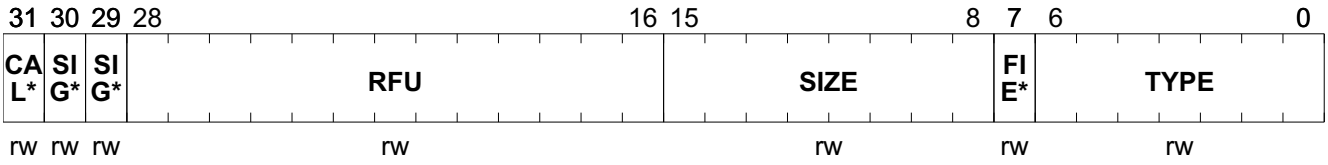
PKC_CMD

The Command Register specifies low level arithmetic operations as well as high-level cryptographic algorithms. The available instructions are listed in the table.

Note the following abbreviations:

- "a = ***PKC_CONFIG.OPPTRA**" stands for:
  a denotes the integer or binary polynomial represented by the **LIRPKC_CONFIG.OPPTRA** is pointing to.
- Similar for other variables instead of a and for **OPPTRB**, **OPPTRC**.
- "a = ***PKC_SCM_R0**" stands for:
  a denotes the integer or binary polynomial represented by the **LIRPKC_SCM_R0**.
- Similar for other variables instead of a and for **PKC_SCM_R1**, ...

PKC_CMD	Offset	Reset Value
PKC Command Register	**04_H**	**$0000\ 0000_H$**



Field	Bits	Type	Description
CALCR2	31	rw	**Calculate R^2mod N** This bit indicates if the PKC-module has to calculate the Montgomery-value (R^2mod N) prior to the next operation (which is defined in this SFR). This bit must be set to 1_B every time a new modulus is loaded into **PKC_SCM_R0**. The bit is automatically cleared as soon as the module has calculated the value (R^2mod N). Once this bit is set to 1_B it can not be set to 0_B by software. Only the commands described below can do it. The operation works for both values of **FIELD**. 0_H , No effect, no calculation will take place. 1_H , The PKC-Module will calculate the value (R^2mod N) if a modular operation **MULTN**, **RED**, **INV2**, or an elliptic curve operation is defined in **TYPE**. After the calculation, the bit is set to 0_B. Otherwise no change in the value takes place.
SIGNB	30	rw	**Sign of Parameter b in equation $Y^2=X^3\pm a*X\pm b$** 0_B , Sign is positive. 1_B , Sign is negative, i.e. the equation is $Y^2=X^3\pm a*X-b$. This is only allowed in case of **FIELD**=0_B. In case of **FIELD**=1_B, this results in an undefined behavior.
SIGNA	29	rw	**Sign of Parameter a in equation $Y^2=X^3\pm a*X\pm b$** 0_B , Sign is positive. 1_B , Sign is negative, i.e. the equation is $Y^2=X^3-a*X\pm b$. This is only allowed in case of **FIELD**=0_B. In case of **FIELD**=1_B, this results in an undefined behavior.

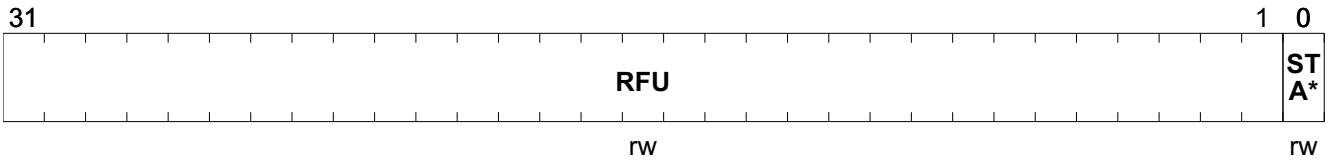
Field	Bits	Type	Description
RFU	28:16	rw	**RFU**
SIZE	15:8	rw	**Size of Operands** This field defines an upper bound of the size of the operands for the operation in multiples of 64. All other values than the allowed ones result in an undefined behavior. 1_H , Operands up to 64 bit length. (Only for arithmetic Operations) 2_H , Operands up to 128 bit length. 3_H , Operands up to 192 bit length. (Only for arithmetic Operations) 4_H , Operands up to 256 bit length.
FIELD	7	rw	**Field Definition** 0_B , Operation computes in Z, or Z/p in case of modular operations, where p is the in **PKC_SCM_R0** defined prime number. 1_B , Operation computes in $F_2[X]$, or $F_2[X]/(f)$ in case of modular operations, where f is the in **PKC_SCM_R0** defined irreducible polynomial.

Field	Bits	Type	Description
TYPE	6:0	rw	**Type of Operation** All constant values that are not explicitly listed are RFU and result in an undefined behavior. For additional information cf. **Chapter 6.5**. **Arithmetic Operations:** The following abbreviations are used for the description: a = ***PKC_CONFIG.OPPTRA**, b = ***PKC_CONFIG.OPPTRB**, c = ***PKC_CONFIG.OPPTRC**, n = ***PKC_SCM_R0**. 01_H **ADDN**, Modular Addition. c := ADDN(a,b,n). 02_H **SUBN**, Modular Subtraction. c := SUBN(a,b,n). 03_H **MULTN**, Modular Multiplication. c := MULTN(a,b,n). 04_H **RED**, Modular Reduction. c := RED(b,n). 05_H **DIVN**, Modular Division. c := DIVN(a,b,n). 06_H **INV**, Modular Inversion. c := INV(b,n). 08_H **MULT**, Multiplication. c := MULT(a,b), only for **PKC_CMD.FIELD**=0_B. 09_H **INV2**, Modular Inversion (for n even). c := INV(b,n), only for **FIELD**=0_B. $0A_H$ **RED2**, Modular Reduction (for n even). c := RED(b,n), only for **FIELD**=0_B. 10_H **EXP**, Modular Exponentiation. c := EXP(a,b,n), only for **FIELD**=0_B. **Elliptic Curve Operations:** The following abbreviations are used for the description (except for ...25519 ops): p/f = ***PKC_SCM_R0**, n = ***PKC_SCM_R1**, x_P = ***PKC_SCM_R2**, y_P = ***PKC_SCM_R3**, a = ***PKC_SCM_R4**, b = ***PKC_SCM_R5**, d = ***PKC_SCM_R6**, ek = ***PKC_SCM_R7**, x_Q = ***PKC_SCM_R8**, y_Q = ***PKC_SCM_R9**,r = ***PKC_SCM_R10**, s = ***PKC_SCM_R11**. h = ***PKC_SCM_R12**, k = ***PKC_CONFIG.OPPTRB**, x_A = ***PKC_CONFIG.OPPTRA**, y_A = *(**PKC_CONFIG.OPPTRA**+1), x_B = ***PKC_CONFIG.OPPTRB**, y_B = *(**PKC_CONFIG.OPPTRB**+1), x_C = ***PKC_CONFIG.OPPTRC**, y_C = *(**PKC_CONFIG.OPPTRC**+1), **OPPTRA**/B/C may only have the values: 6_H,8_H,C_H,E_H, 20_H **PDBL**, Point Doubling. P_C := PDBL(P_A,a,b,p/f) 21_H **PADD**, Point Addition. P_C := PADD(P_A,P_B,a,b,p/f) 22_H **SMULT**, Scalar Multiplication. P_C := SMULT(k,P_A,a,b,p/f). 23_H **CHECKAB**, Check parameters a&b being reduced and the discriminant!=0. 24_H **CHECKN**, Check for order n not equal p. 25_H **CHECKPXY**, Check for point coordinates of P_A being reduced modulo p/f. 28_H **SMULT25519**, Scalar Multiplication. P_C := SMULTcurve25519(k,P_A). 29_H **XRECOVER**, xcoordinate. x_A := XrecoverEd25519(y_A). $2A_H$ **SMULTED25519**, Scalar Multiplication. P_C := SMULTEd25519(k,P_A). $2B_H$ **CHECKVALID**, Check whether S * P_B= h * P_A + P_R. 30_H **ECDSASIG**, ECDSA Signature Generation. (r,s) := ECDSASIG(h,ek,d,n,P,a,b,p/f) (NOT IMPLEMENTED! RFU!) 31_H **ECDSAVER**, ECDSA Signature Verification. ECDSAVER(h,r,s,Q,n,P,a,b,p/f) (NOT IMPLEMENTED! RFU!)

PKC_CTRL

This Register is used to start the execution of the operation specified in **PKC_CMD** and possibly in **PKC_CONFIG**.

PKC_CTRL	Offset	Reset Value
PKC Control Register	08_H	$0000\ 0000_H$

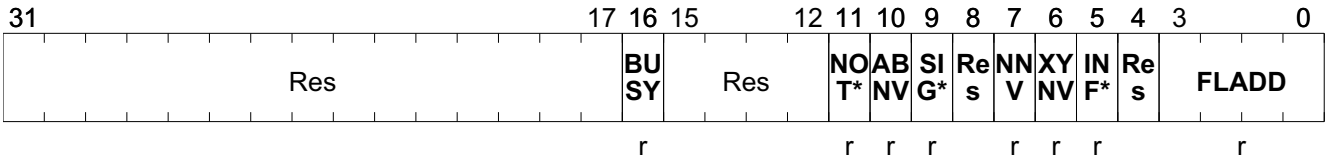


Field	Bits	Type	Description
RFU	31:1	rw	**RFU**
START	0	rw	**Start Signal** The Start Signal can be set when all data and key inputs have been loaded in the **SCM** and are available for processing. When the signal is set, the Command present in the **PKC_CMD** register is initiated and executed. The **START** signal is ignored when the module is already processing data, and it is automatically cleared when the operation is finished.

PKC_STATUS

This Register is used to evaluate the outcome of the operation specified in **PKC_CMD** and possibly in **PKC_CONFIG**.

PKC_STATUS	Offset	Reset Value
PKC Status Register	$0C_H$	$0000\ 0000_H$



Field	Bits	Type	Description
BUSY	16	r	**PKC module busy** 0_B , The PKC module is idle, data can be read from or written to **SCM**. 1_B , The PKC module is processing data. No access to **SCM**.
NOTINV	11	r	**Parameter not invertible** This flag will first be reset at the start of any command. It is then updated after the execution of the command **INV** and **INV2**. It indicates whether the input parameter is invertible or not. 0_B , Input operand of an inversion was successfully inverted, i.e., was invertible. 1_B , Input operand of an inversion was not invertible.
ABNV	10	r	**Parameter A,B not valid** This flag will first be reset at the start of any command. It is then updated after the execution of the command **CHECKAB**. See **Section 6.5.2.4**. 0_B , Parameters a and b are valid. 1_B , Parameters a and b are not valid, i.e., e.g., $4a^3+27b^2=0$.
SIGNV	9	r	**Signature not valid** This flag will first be reset at the start of any command. It is then updated after the execution of the command **CHECKVALID**. It indicates whether the signature can be accepted or must be rejected. See **Section 6.5.2.10**. 0_B , Signature is valid. 1_B , Signature is not valid.
NNV	7	r	**Parameter n (Curve order) not valid** This flag will first be reset at the start of any command. It is then updated after the execution of the command **CHECKN**. It indicates whether the Parameter n is distinct from the module p/f. See **Section 6.5.2.5**. 0_B , Parameter n is valid. 1_B , Parameter n is not valid.

Field	Bits	Type	Description
XYNV	6	r	**Coordinates not reduced** This flag will first be reset at the start of any command. It is then updated after the execution of the command **CHECKPXY**. It indicates whether the coordinates of the point P_A, with x_A = ***PKC_CONFIG.OPPTRA**, y_A = *(**PKC_CONFIG.OPPTRA**+1), are reduced. 0_B , Coordinates are valid, i.e., reduced. 1_B , At least one coordinate is not reduced.
INFTY	5	r	**Point at infinity** This flag will first be reset at the start of any command. It is then updated after the execution of any PKC command. It indicates that a point at infinity occurred. The address of the point at infinity is then given in **FLADD**. 0_B , No point at infinity occurred. 1_B , Point at infinity detected.
FLADD	3:0	r	**Fail Address** The address of the last point detected as not valid, or at the infinity. In this case, the registers holding that point have an undefined value.

6.4.2 Shared Crypto Memory

The Shared Crypto Memory (**SCM**) is 1 KB in size. It is required to support all operations and algorithms. The Shared Crypto Memory is used to program parameters and keys and to upload/download operands/results from the CPU side.

The Shared Crypto Memory is structured in 32 Registers for long integers (**LIR**) of 256 Bits in size. Each register **PKC_SCM_Ri** (i = 0, 1, ..., 31) can also be represented by 8 words of 32-bit each accessible at the 8 addresses:

- **PKC_SCM_Ri** + 0x00
- **PKC_SCM_Ri** + 0x04
- **PKC_SCM_Ri** + 0x08
- ...
- **PKC_SCM_Ri** + 0x1C

where **PKC_SCM_Ri** is identified by base address of **PKC_SCM_Ri**.

Each of the **PKC_SCM_Ri** can represent a non negative integer $a < 2^{256}$ or a binary polynomial f of degree < 256.

- An **integer** a in $[0,2^{256}[$ will be represented in the register **PKC_SCM_Ri** in the following way: Let
  $a = a_{255}\, 2^{255} + ... + a_1\, 2^1 + a_0\, 2^0$, with a_i in {0,1},
  the unique representation of a to the basis 2, then a will be represented in the register **PKC_SCM_Ri** (if **PKC_CMD.FIELD**=0_B) by writing
  - the 32 bit value $(a_{31} ... a_1\, a_0)_2$ into address **PKC_SCM_Ri** + 0x00,
  - the 32 bit value $(a_{63} ... a_{33}\, a_{32})_2$ into address **PKC_SCM_Ri** + 0x04,
  - the 32 bit value $(a_{95} ... a_{65}\, a_{64})_2$ into address **PKC_SCM_Ri** + 0x08,
  - ...
  - the 32 bit value $(a_{255} ... a_{225}\, a_{224})_2$ into address **PKC_SCM_Ri** + 0x1C.
- A **binary polynomial** f of degree < 256 will be represented in the register **PKC_SCM_Ri** in the following way: Let
  $f(X) = a_{255}\, X^{255} + ... + a_1\, X^1 + a_0\, X^0$, with a_i in {0,1},
  the unique representation of f as polynomial in X, then f will be represented in the register **PKC_SCM_Ri** (if **PKC_CMD.FIELD**=1_B) by writing
  - the 32 bit value $(a_{31} ... a_1\, a_0)_2$ into address **PKC_SCM_Ri** + 0x00,
  - the 32 bit value $(a_{63} ... a_{33}\, a_{32})_2$ into address **PKC_SCM_Ri** + 0x04,
  - the 32 bit value $(a_{95} ... a_{65}\, a_{64})_2$ into address **PKC_SCM_Ri** + 0x08,
  - ...
  - the 32 bit value $(a_{255} ... a_{225}\, a_{224})_2$ into address **PKC_SCM_Ri** + 0x1C.

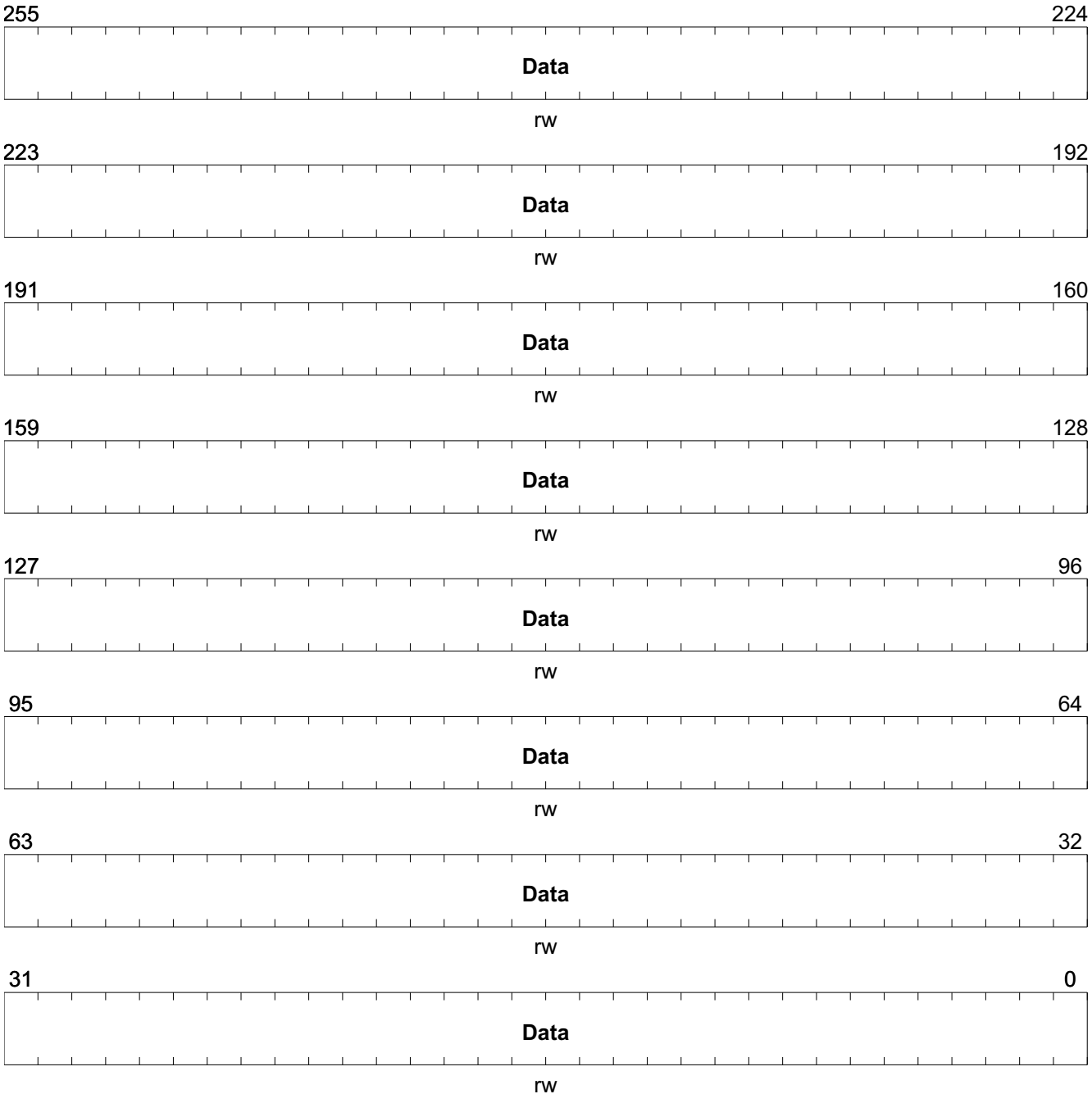
The same holds for reading the integer or binary polynomial out of these registers.

PKC_SCM_Ri

PKC_SCM_R0 — **Shared Crypto Memory Register 0**

Offset 0400_H

Reset Value X_H



Field	Bits	Type	Description
Data	255:0	rw	**Parameters, keys, operands or results.**

Additional Shared Crypto Memory Registers

Additional Shared Crypto Memory Registers have the same layout and access rights as **PKC_SCM_R0**.

Table 6-3 Shared Crypto Memory Registers 1 to 31

Register Short Name	Register Long Name	Offset Address	Reset Value
PKC_SCM_R1	Shared Crypto Memory Register 1	0420_H	X_H
PKC_SCM_R2	Shared Crypto Memory Register 2	0440_H	X_H
PKC_SCM_R3	Shared Crypto Memory Register 3	0460_H	X_H
PKC_SCM_R4	Shared Crypto Memory Register 4	0480_H	X_H
PKC_SCM_R5	Shared Crypto Memory Register 5	$04A0_H$	X_H
PKC_SCM_R6	Shared Crypto Memory Register 6	$04C0_H$	X_H
PKC_SCM_R7	Shared Crypto Memory Register 7	$04E0_H$	X_H
PKC_SCM_R8	Shared Crypto Memory Register 8	0500_H	X_H
PKC_SCM_R9	Shared Crypto Memory Register 9	0520_H	X_H
PKC_SCM_R10	Shared Crypto Memory Register 10	0540_H	X_H
PKC_SCM_R11	Shared Crypto Memory Register 11	0560_H	X_H
PKC_SCM_R12	Shared Crypto Memory Register 12	0580_H	X_H
PKC_SCM_R13	Shared Crypto Memory Register 13	$05A0_H$	X_H
PKC_SCM_R14	Shared Crypto Memory Register 14	$05C0_H$	X_H
PKC_SCM_R15	Shared Crypto Memory Register 15	$05E0_H$	X_H
PKC_SCM_R16	Shared Crypto Memory Register 16	0600_H	X_H
PKC_SCM_R17	Shared Crypto Memory Register 17	0620_H	X_H
PKC_SCM_R18	Shared Crypto Memory Register 18	0640_H	X_H
PKC_SCM_R19	Shared Crypto Memory Register 19	0660_H	X_H
PKC_SCM_R20	Shared Crypto Memory Register 20	0680_H	X_H
PKC_SCM_R21	Shared Crypto Memory Register 21	$06A0_H$	X_H
PKC_SCM_R22	Shared Crypto Memory Register 22	$06C0_H$	X_H
PKC_SCM_R23	Shared Crypto Memory Register 23	$06E0_H$	X_H
PKC_SCM_R24	Shared Crypto Memory Register 24	0700_H	X_H
PKC_SCM_R25	Shared Crypto Memory Register 25	0720_H	X_H
PKC_SCM_R26	Shared Crypto Memory Register 26	0740_H	X_H
PKC_SCM_R27	Shared Crypto Memory Register 27	0760_H	X_H
PKC_SCM_R28	Shared Crypto Memory Register 28	0780_H	X_H
PKC_SCM_R29	Shared Crypto Memory Register 29	$07A0_H$	X_H
PKC_SCM_R30	Shared Crypto Memory Register 30	$07C0_H$	X_H
PKC_SCM_R31	Shared Crypto Memory Register 31	$07E0_H$	X_H

Note

- The **SCM** is r/w.
- The **SCM** can only be accessed 32-bit wise and 32-bit aligned. Using 8-bit, 16-bit or un-aligned accesses lead to a bus error.
- The **SCM** will be accessed by the core of the PKC-Module during operation. No simultaneous access of the core and the CPU is allowed. Hence:
- The **SCM** may only be accessed by the CPU if the PKC-module is in an idle mode, i.e., if **PKC_STATUS.BUSY**=0_B. So, the software has to make sure that this condition holds when the CPU is accessing the **SCM**.

6.5 Further detailed definition/specification of commands

Note:

1. *For all following commands: If the input conditions are not fulfilled this will result in an undefined behavior.*
2. *Notation will be Z-specific: Unlike the notation in Section 6.2.5 and Section 6.2.6, for modular operations, the module will be denoted as “n” or “p”, but not as “f”.*

6.5.1 Arithmetic Operations.

The following table gives a short overview over the arithmetic operations:

Table 6-4 Overview over Arithmetic Operations

	ADDN	SUBN	MULTN	RED	DIVN	INV	MULT	INV2	RED2	EXP
Before Operation:										
CALCR2 has to be set, if modul is new.			X	X				X		X
Allowed values for SIZE	2, 4	2, 4	2, 4	2, 4	2, 4	2, 4	2, 4	2, 4	2, 4	2, 4
Allowed values for FIELD	0/1	0/1	0/1	0/1	0/1	0/1	0	0	0	0
Allowed values for OPPTRA	1...15	1...15	1...15		1...15		1...15			1...15
Allowed values for OPPTRB	1...15	1...15	1...15	1...15	1...15	1...15	1...15	1...15	1...15	1...15
Allowed values for OPPTRC	1...15	1...15	1...15	1...15	1...15	1...15	1...15	1...15	1...15	1...15
module n (In R0) has to be odd.			X	X	X	X				X
Operation:	ADDN (a,b,n)	SUBN (a,b,n)	MULTN (a,b,n)	RED (b,n)	DIV (a,b,n)	INV (b,n)	MULT (a,b)	INV (b,n)	DIV (b,n)	EXP (a,b,n)
After Operation:										
CALCR2 will be reset			X	X				X		X
NOTINV may be set						X		X		

For more detailed information about the commans see the following subsections.

6.5.1.1 Modular Addition Command ADDN

The command **ADDN** realizes the mathematical operation

c := ADDN(a,b,n)

in **Z** if **PKC_CMD.FIELD**=0_B or $\mathbf{F}_2$[X] if **PKC_CMD.FIELD**=1_B respectively.

Input values:

- a: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRA**.
- b: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRB**.
- n: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.

Further Input conditions:

- **PKC_CMD.SIZE** = 2, 4.
- **PKC_CONFIG.OPPTRA** = 1, ..., 15.
- **PKC_CONFIG.OPPTRB** = 1, ..., 15.
- **PKC_CONFIG.OPPTRC** = 1, ..., 15.
- - If **PKC_CMD.FIELD**=0_B then 0 ≤ a,b < n < 2^(64*PKC_CMD.SIZE).
  - If **PKC_CMD.FIELD**=1_B then deg(a),deg(b) < deg(n) < (64*PKC_CMD.SIZE).

Output: (After operation finished)

- c: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.

6.5.1.2 Modular Subtraction Command SUBN

The command **SUBN** realizes the mathematical operation

c := SUBN(a,b,n)

in **Z** if **PKC_CMD.FIELD**=0_B or $\mathbf{F}_2$[X] if **PKC_CMD.FIELD**=1_B respectively.

Input values:

- a: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRA**.
- b: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRB**.
- n: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.

Further Input conditions:

- **PKC_CMD.SIZE** = 2, 4.

- **PKC_CONFIG.OPPTRA** = 1, ..., 15.
- **PKC_CONFIG.OPPTRB** = 1, ..., 15.
- **PKC_CONFIG.OPPTRC** = 1, ..., 15.
- - If **PKC_CMD.FIELD**=0_B then 0 ≤ a,b < n < 2^(64***PKC_CMD.SIZE**).
  - If **PKC_CMD.FIELD**=1_B then deg(a),deg(b) < deg(n) < (64***PKC_CMD.SIZE**).

Output: (After operation finished)

- c: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.

6.5.1.3 Modular Multiplication Command MULTN

The command **MULTN** realizes the mathematical operation

c := MULTN(a,b,n)

in **Z** if **PKC_CMD.FIELD**=0_B or $\mathbf{F}_2[X]$ if **PKC_CMD.FIELD**=1_B respectively.

Input values:

- a: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRA**.
- b: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRB**.
- n: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.

Further Input conditions:

- **PKC_CMD.CALCR2** = 1_B, if (R^2mod N) was not computed yet, i.e., this is the first modular operation using the module n.
- **PKC_CMD.SIZE** = 2, 4.
- **PKC_CONFIG.OPPTRA** = 1, ..., 15.
- **PKC_CONFIG.OPPTRB** = 1, ..., 15.
- **PKC_CONFIG.OPPTRC** = 1, ..., 15.
- - If **PKC_CMD.FIELD**=0_B then 0 ≤ a,b < n < 2^(64***PKC_CMD.SIZE**), n has to be odd.
  - If **PKC_CMD.FIELD**=1_B then deg(a),deg(b) < deg(n) < (64***PKC_CMD.SIZE**), n_0=1_B.

Output: (After operation finished)

- c: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.
- **PKC_CMD.CALCR2** = 0_B

6.5.1.4 Modular Reduction Command RED

The command **RED** realizes the mathematical operation

c := RED(b,n)

in **Z** if **PKC_CMD.FIELD**=0_B or $\mathbf{F}_2[X]$ if **PKC_CMD.FIELD**=1_B respectively.

Input values:

- b: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRB**.
- n: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.

Further Input conditions:

- **PKC_CMD.CALCR2** = 1_B, if (R^2mod N) was not computed yet, i.e., this is the first modular operation using the module n.
- **PKC_CMD.SIZE** = 2,4.
- **PKC_CONFIG.OPPTRB** = 1, ..., 15.
- **PKC_CONFIG.OPPTRC** = 1, ..., 15.
- - If **PKC_CMD.FIELD**=0_B then 0 ≤ b,n < 2^(64*__PKC_CMD.SIZE__), n has to be odd.
  - If **PKC_CMD.FIELD**=1_B then deg(b),deg(n) < (64*__PKC_CMD.SIZE__), n_0=1_B.

Output: (After operation finished)

- c: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.
- **PKC_CMD.CALCR2** = 0_B.

6.5.1.5 Modular Division Command DIVN

The command **DIVN** realizes the mathematical operation

c := DIVN(a,b,n)

in **Z** if **PKC_CMD.FIELD**=0_B or $\mathbf{F}_2[X]$ if **PKC_CMD.FIELD**=1_B respectively.

Input values:

- a: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRA**.
- b: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRB**.
- n: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.

Further Input conditions:

- **PKC_CMD.SIZE** = 2, 4.
- **PKC_CONFIG.OPPTRA** = 1, ..., 15.
- **PKC_CONFIG.OPPTRB** = 1, ..., 15.
- **PKC_CONFIG.OPPTRC** = 1, ..., 15.
- - If **PKC_CMD.FIELD**=0_B then 0 ≤ a,b < n < 2^(64*__PKC_CMD.SIZE__), n has to be odd.
  - If **PKC_CMD.FIELD**=1_B then deg(a),deg(b) < deg(n) < (64*__PKC_CMD.SIZE__), n_0=1_B.

Output: (After operation finished)

- c: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.

6.5.1.6 Modular Inversion Command INV

The command **INV** realizes the mathematical operation

c := INV(b,n)

in **Z** if **PKC_CMD.FIELD**=0_B or $\mathbf{F}_2$[X] if **PKC_CMD.FIELD**=1_B respectively.

Input values:

- b: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRB**.
- n: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.

Further Input conditions:

- **PKC_CMD.SIZE** = 2, 4.
- **PKC_CONFIG.OPPTRB** = 1, ..., 15.
- **PKC_CONFIG.OPPTRC** = 1, ..., 15.
- - If **PKC_CMD.FIELD**=0_B then 0 ≤ b < n < 2^(64*__PKC_CMD.SIZE__), n has to be odd.
  - If **PKC_CMD.FIELD**=1_B then deg(b) < deg(n) < (64*__PKC_CMD.SIZE__), n_0=1_B.

Output: (After operation finished)

- **PKC_STATUS.NOTINV** = 1_B if b is not invertible modulo n, i.e., if the gcd of b and n is not 1. In this case c is in an undefined state. Otherwise:
- c: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.

6.5.1.7 Multiplication Command MULT

The command **MULT** realizes the mathematical operation

c := MULT(a,b)

in **Z** if **PKC_CMD.FIELD**=0_B.

Attention: The case F_2[X] with PKC_CMD.FIELD=1_B is not supportet.

Input values:

- a: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRA**.
- b: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRB**.

Further Input conditions:

- **PKC_CMD.SIZE** = 2, 4.
- **PKC_CONFIG.OPPTRA** = 1, ..., 15.
- **PKC_CONFIG.OPPTRB** = 1, ..., 15.
- **PKC_CONFIG.OPPTRC** = 1, ..., 15.
- 0 ≤ a,b, a*b< 2^(64***PKC_CMD.SIZE**).

Output: (After operation finished)

- c: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.

6.5.1.8 Modular Inversion Command INV2

The command **INV2** realizes the mathematical operation

c := INV(b,n)

in **Z** if **PKC_CMD.FIELD**=0_B. Contrary to **INV**, this command works in the case, when the LSB of n is 0_B.

Attention: The case F_2[X] with PKC_CMD.FIELD=1_B is not supportet.

Input values:

- b: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRB**.
- n: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.

Further Input conditions:

- **PKC_CMD.CALCR2** = 1_B, if (R^2mod N) was not computed yet, i.e., this is the first modular operation using the module n.
- **PKC_CMD.SIZE** = 2, 4.
- **PKC_CONFIG.OPPTRB** = 1, ..., 15.
- **PKC_CONFIG.OPPTRC** = 1, ..., 15.
- 0 ≤ b < n < 2^(64***PKC_CMD.SIZE**), n has to be even, n not 0.

Output: (After operation finished)

- **PKC_STATUS.NOTINV** = 1_B if b is not invertible modulo n, i.e., if the gcd of b and n is not 1. In this case c is in an undefined state. Otherwise:
- c: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.
- **PKC_CMD.CALCR2** = 0_B.

6.5.1.9 Modular Reduction Command RED2

The command **RED2** realizes the mathematical operation

c := RED(b,n)

in **Z** if **PKC_CMD.FIELD**=0_B. Contrary to **RED**, this command works even in the case, when the LSB of n is 0_B.

Attention: The case $F_2[X]$ with PKC_CMD.FIELD=1_B is not supportet.

Input values:

- b: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRB**.
- n: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.

Further Input conditions:

- **PKC_CMD.SIZE** = 2, 4.
- **PKC_CONFIG.OPPTRB** = 1, ..., 15.
- **PKC_CONFIG.OPPTRC** = 1, ..., 15.
- 0 ≤ b,n < 2^(64*__PKC_CMD.SIZE__), n not 0.

Output: (After operation finished)

- c: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.

6.5.1.10 Modular Exponentiation Command EXP

The command **EXP** realizes the mathematical operation

c := EXPN(a,b,n)

in **Z** independent of the value of **PKC_CMD.FIELD**.

Input values:

- a: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRA**.
- b: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRB**.
- n: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.

Further Input conditions:

- **PKC_CMD.CALCR2** = 1_B, if (R^2mod N) was not computed yet, i.e., this is the first modular operation using the module n.
- **PKC_CMD.SIZE** = 2, 4.
- **PKC_CONFIG.OPPTRA** = 1, ..., 15.
- **PKC_CONFIG.OPPTRB** = 1, ..., 15.
- **PKC_CONFIG.OPPTRC** = 1, ..., 15.
- **PKC_CMD.FIELD**=0_B, if **PKC_CMD.FIELD**=1_B this bit will be ignored.
- 0 ≤ a,b,n < 2^(64***PKC_CMD.SIZE**), n has to be odd.

Output: (After operation finished)

- c: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.
- **PKC_CMD.CALCR2** = 0_B

6.5.2 Elliptic Curve Operations.

For the description of elliptic curve comands, the following notation is used:

- a := ((-1)^**PKC_CMD.SIGNA**) * a,
  b := ((-1)^**PKC_CMD.SIGNB**) * b,
  if **PKC_CMD.FIELD**=0_B or, with
- a := a,
  b := b,
  if **PKC_CMD.FIELD**=1_B.

The following table gives a short overview:

Table 6-5 Overview over Elliptic Curve Operations

	PDBL	PADD	SMULT	CHECKAB	CHECKN	CHECKPXY
Before Operation:						
CALCR2 has to be set, if modul is new.	X	X	X	X		
Allowed values for SIZE	4	4	4	4	4	4
Allowed values for FIELD	0/1	0/1	0/1	0/1	0/1	0/1
Allowed values for OPPTRA	6_H, 8_H, C_H, E_H	6_H, 8_H, C_H, E_H	6_H, 8_H, C_H, E_H	6_H, 8_H, C_H, E_H	6_H, 8_H, C_H, E_H	6_H, 8_H, C_H, E_H
Allowed values for OPPTRB		6_H, 8_H, C_H, E_H	6_H, 8_H, C_H, E_H	6_H, 8_H, C_H, E_H	6_H, 8_H, C_H, E_H	6_H, 8_H, C_H, E_H
Allowed values for OPPTRC	6_H, 8_H, C_H, E_H	6_H, 8_H, C_H, E_H	6_H, 8_H, C_H, E_H	6_H, 8_H, C_H, E_H	6_H, 8_H, C_H, E_H	6_H, 8_H, C_H, E_H
p (In R0) has to be odd.	X	X	X	X	X	X
Operation:	PDBL (P_a,a,b,p)	PADD (P_a,P_b,a,b,p)	SMULT (k,P_a,a,b,n)	a < p,b < p,...	n != p	x_A, y_A < p

Table 6-5 Overview over Elliptic Curve Operations (cont'd)

	PDBL	PADD	SMULT	CHECKAB	CHECKN	CHECKPXY
After Operation:						
CALCR2 will be reset	X	X	X	(X)[1]		
SIGNA will be reset and R4 set to (a mod p)	X	X	X	(X)		
SIGNB will be reset and R5 set to (b mod p)	X	X	X	(X)		
ABNV may be set				X		
NNV may be set					X	
XYNV may be set						X
INFTY may be set	X	X	X			
FLADD may be set	X	X	X			

1) The (X) means that this may happen or not. It definitely happens if the output of the test is positive. Otherwise it depends on the error case.

For more detailed information about the commands see the following subsections.

6.5.2.1 Point Doubling Command PDBL

The command **PDBL** realizes the point doubling operation

$P_C := PDBL(P_A,a,b,p)$

on an elliptic curve E(a,b,p).

Input values:

- p: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.
- a: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R4**.
- b: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R5**.
- $P_A=(x_A,y_A)$:
  x_A: The x-coordinate of P_A is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRA**.
  y_A: The y-coordinate of P_A is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**+1, where i is the value in **PKC_CONFIG.OPPTRA**.

Further Input conditions:

- **PKC_CMD.CALCR2** = 1_B, if (R^2mod N) was not computed yet, i.e., this is the first modular operation using the module p.
- **PKC_CMD.SIZE** = 4.

- **PKC_CONFIG.OPPTRA** = 6_H, 8_H, C_H, E_H.
- **PKC_CONFIG.OPPTRC** = 6_H, 8_H, C_H, E_H.
- - If **PKC_CMD.FIELD**=0_B then $0 \le x_A,y_A,a,b < p < 2^{(64*PKC\_CMD.SIZE)}$, p has to be odd.
  - If **PKC_CMD.FIELD**=1_B then $deg(x_A),deg(y_A),deg(a),deg(b) < deg(p) < (64*PKC\_CMD.SIZE)$, $p_0=1_B$.
- a, b, p are valid parameters such that an elliptic curve E(a,b,p) is defined.
- P_A lies on the curve E(a,b,p).

Output: (After operation finished)

- $P_C=(x_C,y_C)$:
  x_C: The x-coordinate of P_C is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.
  y_C: The y-coordinate of P_C is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**+1, where i is the value in **PKC_CONFIG.OPPTRC**.
- **PKC_CMD.CALCR2** = 0_B
- **PKC_CMD.SIGNA** = 0_B, **PKC_CMD.SIGNB** = 0_B, **PKC_SCM_R4** = a, **PKC_SCM_R5** = b,
- **PKC_STATUS.INFTY** = 1_B, if a point at infinity occured.
  In this case **PKC_STATUS.FLADD** shows the address of the point at infinity (i.e., **OPPTRC**).

6.5.2.2 Point Addition Command PADD

The command **PADD** realizes the point addition operation

$P_C := PADD(P_A,P_B,a,b,p)$

on an elliptic curve E(a,b,p).

Input values:

- p: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.
- a: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R4**.
- b: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R5**.
- $P_A=(x_A,y_A)$:
  x_A: The x-coordinate of P_A is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRA**.
  y_A: The y-coordinate of P_A is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**+1, where i is the value in **PKC_CONFIG.OPPTRA**.
- $P_B=(x_B,y_B)$:
  x_B: The x-coordinate of P_B is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRB**.
  y_B: The y-coordinate of P_B is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is

represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**+1, where i is the value in **PKC_CONFIG.OPPTRB**.

Further Input conditions:

- **PKC_CMD.CALCR2** = 1_B, if (R^2mod N) was not computed yet, i.e., this is the first modular operation using the module p.
- **PKC_CMD.SIZE** = 4.
- **PKC_CONFIG.OPPTRA** = 6_H, 8_H, C_H, E_H.
- **PKC_CONFIG.OPPTRB** = 6_H, 8_H, C_H, E_H.
- **PKC_CONFIG.OPPTRC** = 6_H, 8_H, C_H, E_H.
- - If **PKC_CMD.FIELD**=0_B then $0 \le x_A,y_A,x_B,y_B,a,b < p < 2^{(64*PKC\_CMD.SIZE)}$, p has to be odd.
  - If **PKC_CMD.FIELD**=1_B then $deg(x_A),deg(y_A),deg(x_B),deg(y_B),deg(a),deg(b) < deg(p) < (64*PKC\_CMD.SIZE)$, $p_0=1_B$.
- a, b, p are valid parameters such that an elliptic curve E(a,b,p) is defined.
- P_A, P_B lie on the curve E(a,b,p).

Output: (After operation finished)

- $P_C=(x_C,y_C)$:
  x_C: The x-coordinate of P_C is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.
  y_C: The y-coordinate of P_C is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**+1, where i is the value in **PKC_CONFIG.OPPTRC**.
- **PKC_CMD.CALCR2** = 0_B
- **PKC_CMD.SIGNA** = 0_B, **PKC_CMD.SIGNB** = 0_B, **PKC_SCM_R4** = a, **PKC_SCM_R5** = b,
- **PKC_STATUS.INFTY** = 1_B, if a point at infinity occured.
  In this case **PKC_STATUS.FLADD** shows the address of the point at infinity (i.e., **OPPTRC**).

6.5.2.3 Scalar Multiplication Command SMULT

The command **SMULT** realizes the scalar multiplication operation

P_C := SMULT(k,P_A,a,b,p)

on an elliptic curve E(a,b,p).

Input values:

- p: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.
- a: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R4**.
- b: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R5**.
- k: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRB**.

- $P_A=(x_A,y_A)$:
  x_A: The x-coordinate of P_A is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRA**.
  y_A: The y-coordinate of P_A is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**+1, where i is the value in **PKC_CONFIG.OPPTRA**.

Further Input conditions:

- **PKC_CMD.CALCR2** = 1_B, if (R^2mod N) was not computed yet, i.e., this is the first modular operation using the module p.
- **PKC_CMD.SIZE** = 4.
- **PKC_CONFIG.OPPTRA** = 6_H, 8_H, C_H, E_H.
- **PKC_CONFIG.OPPTRC** = 6_H, 8_H, C_H, E_H.
- **PKC_CONFIG.OPPTRB** = 6_H, 8_H, C_H, E_H.
- - If **PKC_CMD.FIELD**=0_B then $0 \le x_A,y_A,a,b < p < 2^{(64*PKC\_CMD.SIZE)}$, p has to be odd.
  - If **PKC_CMD.FIELD**=1_B then $deg(x_A),deg(y_A),deg(a),deg(b) < deg(p) < (64*PKC\_CMD.SIZE)$, $p_0=1_B$.
- $\underline{a}$, $\underline{b}$, p are valid parameters such that an elliptic curve $E(\underline{a},\underline{b},p)$ is defined.
- P_A lies on the curve $E(\underline{a},\underline{b},p)$.

Output: (After operation finished)

- $P_C=(x_C,y_C)$:
  x_C: The x-coordinate of P_C is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.
  y_C: The y-coordinate of P_C is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**+1, where i is the value in **PKC_CONFIG.OPPTRC**.
- **PKC_CMD.CALCR2** = 0_B
- **PKC_CMD.SIGNA** = 0_B, **PKC_CMD.SIGNB** = 0_B, **PKC_SCM_R4** = $\underline{a}$, **PKC_SCM_R5** = $\underline{b}$,
- **PKC_STATUS.INFTY** = 1_B, if a point at infinity occured.
  In this case **PKC_STATUS.FLADD** shows the address of the point at infinity (i.e., = **OPPTRA**, or = **OPPTRC**).

6.5.2.4 Parameter Check CHECKAB

The command **CHECKAB** checks the parameters a and b, resp. $\underline{a}$ and $\underline{b}$, for validity:

- If **PKC_CMD.FIELD** = 0_B the check goes for:
  - a < p AND
  - b < p AND
  - $(4\,\underline{a}^3 + 27\,\underline{b}^2) \bmod p > 0$.
- If **PKC_CMD.FIELD** = 1_B the check is done for integers (i.e., a, b, p are interpreted as integers) and it goes for:
  - a < p AND
  - b < p.

Input values:

- p: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.
- a: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R4**.
- b: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R5**.

Further Input conditions:

- **PKC_CMD.CALCR2** = 1_B, if (R^2mod N) was not computed yet, i.e., this is the first modular operation using the module p.
- **PKC_CMD.SIZE** = 4.

Output: (After operation finished)

- **PKC_CMD.CALCR2** = 0_B (sometimes).
- **PKC_CMD.SIGNA** = 0_B, **PKC_CMD.SIGNB** = 0_B, **PKC_SCM_R4** = a, **PKC_SCM_R5** = b (sometimes).
- **PKC_STATUS.ABNV** = 0_B, if the check generates a TRUE.
- **PKC_STATUS.ABNV** = 1_B, if the check generates a FALSE.

6.5.2.5 Parameter Check CHECKN

The command **CHECKN** checks the parameters n (some point order), for validity. The check goes for: n != p.

Input values:

- p: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.
- n: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R1**.

Output: (After operation finished)

- **PKC_STATUS.NNV** = 0_B, if the check generates a TRUE.
- **PKC_STATUS.NNV** = 1_B, if the check generates a FALSE.

6.5.2.6 Parameter Check CHECKPXY

The command **CHECKPXY** checks the coordinates of point $P_A=(x_A,y_A)$ for validity:

- If **PKC_CMD.FIELD** = 0_B the check goes for:
  - $x_A < p$ AND
  - $y_A < p$.
- If **PKC_CMD.FIELD** = 1_B the check is done for integers (i.e., x_A,y_A, p are interpreted as integers) and it goes for:
  - $x_A < p$ AND
  - $y_A < p$.

Input values:

- p: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.
- $P_A=(x_A,y_A)$:
  x_A: The x-coordinate of P_A is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRA**.
  y_A: The y-coordinate of P_A is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**+1, where i is the value in **PKC_CONFIG.OPPTRA**.

Output: (After operation finished)

- **PKC_STATUS.XYNV** = 0_B, if the check generates a TRUE.
- **PKC_STATUS.XYNV** = 1_B, if the check generates a FALSE.

6.5.2.7 Scalar Multiplication Command SMULT25519

The command **SMULT25519** realizes the scalar multiplication operation

P_C := SMULTcurve25519(k,P_A) = M-SMULT(k,P_A, A, B, p)

on the elliptic curve Montgomery Curve25519=M(A,B,p), for p=2^{255}-19, A = 486662, B = 1.

Input values:

- p: The integer module 2^{255}-19 is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.
- x_A: The x-coordinate of P_A is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRA**.
- (A-2)/4=1DB41$_H$: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**+1, where i is the value in **PKC_CONFIG.OPPTRA**.
- k: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRB**.

Further Input conditions:

- **PKC_CMD.CALCR2** = 1_B, if (R^2mod N) was not computed yet, i.e., this is the first modular operation using the module p.
- **PKC_CMD.SIZE** = 4.
- **PKC_CONFIG.OPPTRA** = 2_H,4_H,6_H, 8_H,A_H,C_H, E_H.
- **PKC_CONFIG.OPPTRC** = 2_H,4_H,6_H, 8_H,A_H,C_H, E_H.
- **PKC_CONFIG.OPPTRB** = 2_H,4_H,6_H, 8_H,A_H,C_H, E_H.
- **PKC_CMD.FIELD**=0_B.
- P_A lies on the curve Curve25519.

Output: (After operation finished)

- x_C: The x-coordinate of P_C is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.

- **PKC_CMD.CALCR2** = 0_B,

6.5.2.8 X-Coordinate Recovery XRECOVER

The command **XRECOVER** realizes the scalar multiplication operation

x := XrecoverEd25519(y).

on the elliptic curve Ed25519=Ed(a,d,p), for $p=2^{255}-19$, a = -1, d = 121666^{-1} - 1 mod p.

Input values:

- p: The integer module $2^{255}-19$ is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.
- (p-5)/8 := 0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD$_H$: This is a special integer used for the computation which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R4**.
- y: The y-coordinate of a point is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R8**.
- d := 52036CEE2B6FFE738CC740797779E89800700A4D4141D8AB75EB4DCA135978A3$_H$: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R10**.
- J := 2B8324804FC1DF0B2B4D00993DFBD7A72F431806AD2FE478C4EE1B274A0EA0B0$_H$: The integer is the square root modulo p and is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R11**.

Further Input conditions:

- **PKC_CMD.CALCR2** = 1_B, if (R^2mod N) was not computed yet, i.e., this is the first modular operation using the module p.
- **PKC_CMD.SIZE** = 4.
- **PKC_CMD.FIELD**=0_B.

Output: (After operation finished)

- x: An x-coordinate of a point with coordinate y is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R6**.
- **PKC_CMD.CALCR2** = 0_B,

6.5.2.9 Scalar Multiplication Command SMULTED25519

The command **SMULTED25519** realizes the scalar multiplication operation

P_C := SMULTEd25519(k,P_A) = Ed-SMULT(k,P_A, a, d, p)

on the elliptic curve Ed25519, for $p=2^{255}-19$, a = -1, d = 121666^{-1} - 1 mod p.

Input values:

- p: The integer module $2^{255}-19$ is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.
- D2 := d*2^{257} mod p = 590456B4E53F8A4DCB27240F78310D2021430EEF5F8C52E701DB17FDBE8FD3F4$_H$ is a special integer used for the computation which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R1**.

- P_A=(x_A,y_A):
  x_A: The x-coordinate of P_A is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRA**.
  y_A: The y-coordinate of P_A is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**+1, where i is the value in **PKC_CONFIG.OPPTRA**. .
- k: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRB**.

Further Input conditions:

- **PKC_CMD.CALCR2** = 1_B, if (R^2mod N) was not computed yet, i.e., this is the first modular operation using the module p.
- **PKC_CMD.SIZE** = 4.
- **PKC_CONFIG.OPPTRA** = 2_H,4_H,6_H, 8_H,A_H,C_H, E_H.
- **PKC_CONFIG.OPPTRC** = 2_H,4_H,6_H, 8_H,A_H,C_H, E_H.
- **PKC_CONFIG.OPPTRB** = 2_H,4_H,6_H, 8_H,A_H,C_H, E_H.
- **PKC_CMD.FIELD**=0_B.
- P_A lies on the curve Ed25519.

Output: (After operation finished)

- P_C=(x_C,y_C):
  x_C: The x-coordinate of P_C is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**, where i is the value in **PKC_CONFIG.OPPTRC**.
  y_C: The y-coordinate of P_C is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_Ri**+1, where i is the value in **PKC_CONFIG.OPPTRC**.
- **PKC_CMD.CALCR2** = 0_B,

6.5.2.10 Signature Verification CHECKVALID

The command **CHECKVALID** checks the validity of the following equation on the curve Ed25519:

$S*P_B = h * P_A + P_R$

on the elliptic curve Ed25519=Ed(a,d,p), for $p=2^{255} - 19$, a = -1, $d = 121666^{-1} - 1$ mod p.

Input values:

- p: The integer module 2^{255}-19 is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.
- D2 := $d*2^{257}$ mod p = 590456B4E53F8A4DCB27240F78310D2021430EEF5F8C52E701DB17FDBE8FD3F4$_H$ is a special integer used for the computation which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R1**.
- P_B=(x_B,y_B):
  x_B: The x-coordinate of P_B is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R2**.
  y_B: The y-coordinate of P_B is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R3**.

- S: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R4**.
- h: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R5**.
- $P_A=(x_A,y_A)$:
  x_A: The x-coordinate of P_A is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R6**.
  y_A: The y-coordinate of P_A is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R7**.
- $P_R=(x_R,y_R)$:
  x_R: The x-coordinate of P_R is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R8**.
  y_R: The y-coordinate of P_R is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R9**.

Further Input conditions:

- **PKC_CMD.CALCR2** = 1_B, if (R^2mod N) was not computed yet, i.e., this is the first modular operation using the module p.
- **PKC_CMD.SIZE** = 4.
- **PKC_CMD.FIELD**=0_B.
- P_A,P_B,P_R lie on the curve Ed25519.

Output: (After operation finished)

- **PKC_CMD.CALCR2** = 0_B,
- **PKC_STATUS.SIGNV** = 1_B, if the equation is not valid.

Additional Output:

- $P_1=(x_1,y_1) := S*P_B$:
  x_1: The x-coordinate of P_1 is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R10**.
  y_1: The y-coordinate of P_1 is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R11**.
- $P_2=(x_2,y_2) := h*P_A$:
  x_2: The x-coordinate of P_2 is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R12**.
  y_2: The y-coordinate of P_2 is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R13**.
- $P_3=(x_3,y_3) := P_R+P_2$:
  x_3: The x-coordinate of P_3 is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R14**.
  y_3: The y-coordinate of P_3 is an integer which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R15**.

6.5.2.11 ECDSA Signature generation ECDSASIG

Attention: This command may not be included in the delivery. Whether it is included, this can be seen in the list of PKC_CMD.TYPE in "Control and Status Registers" on Page 144.

The command **ECDSASIG** realizes the ECDSA signature generation of a message,

(r,s) := ECDSASIG(h,ek,d,n,P,a,b,p)

on an elliptic curve E(a,b,p). See **Chapter 6.2.2.4** and **Chapter 6.2.6.4** for notation and more details.

Input values:

- p: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.
- n: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R1**.
- a: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R4**.
- b: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R5**.
- d: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R6**.
- ek: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R7**.
- h: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R12**.
- $P=(x_P,y_P)$:
  x_P: The x-coordinate of P_A is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R2**.
  y_P: The y-coordinate of P_A is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R3**.

Further Input conditions:

- **PKC_CMD.CALCR2** = 1_B, if (R^2mod N) was not computed yet, i.e., this is the first modular operation using the module p.
- **PKC_CMD.SIZE** = 4.
- - If **PKC_CMD.FIELD**=0_B then 0 ≤ x,y,a,b < p < 2^(64*PKC_CMD.SIZE), p has to be odd.
  - If **PKC_CMD.FIELD**=1_B then deg(x),deg(y),deg(a),deg(b) < deg(p) < (64*PKC_CMD.SIZE), $p_0=1_B$.
- a, b, p are valid parameters such that an elliptic curve E(a,b,p) is defined.
- P lies on the curve E(a,b,p).
- n is the order of the Point P.
- 0 < ek < n-1 is a random number, generated uniformly in the intervall]1, n-1[.
- h is a hash value of the message that has to be signed, generated according to the signature standard which is going to be used. h has to be interpreted as an integer. 0 ≤ h < n.
- d is the secret key, 0 < d < n.

Output: (After operation finished)

- Signature (r,s):
  r: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R10**.
  s: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R11**.
- **PKC_CMD.CALCR2** = 0_B.
- **PKC_CMD.SIGNA** = 0_B, **PKC_CMD.SIGNB** = 0_B, **PKC_SCM_R4** = a, **PKC_SCM_R5** = b.

Additional Output:

- $P_1 = (x_1,y_1)$:= SMULT(ek,P,a,b,p):
  x_1: The x-coordinate of P_1 is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R14**.
  y_1: The y-coordinate of P_1 is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R15**.
- kinv := INV(ek,n) : The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R13**.

Programming example, for ECDSA Signature Generation without this command:

If this command is not included in the delivery then it still can be implemented by using the following code: This is illustrated for the settings with **PKC_CMD.SIZE** = 4 and **PKC_CMD.SIGNA** = **PKC_CMD.SIGNB** = 0_B. If (**SIGNA**,**SIGNB**) is not (0_B, 0_B) then the bits have to be taken into account only for the **SMULT**-operation.

```
Input: p, n, a, b, d, ek, h, P=(xP,yP), as defined above.
Output: Signature r,s
Additional Output: P1 := ek*P, r, d*r, d*r+h, kinv.
R0 <-- p
R1 <-- 0
R2 <-- h
R3 <-- d
R4 <-- a
R5 <-- b
R6 <-- xP
R7 <-- yP
R8 <-- ek
R9 <-- 0
R10<-- 0
R11<-- 0
R12<-- 0
R13<-- 0
R14<-- 0
R15<-- 0
// ek*P:
PKC_CONFG <-- 0x000C0806 // A=6, B=8, C=12
PKC_CMD   <-- 0x80000422 // CALC2 | 256 bit | SMULT
[PKC_CMD  <-- 0x800004A2 // CALC2 | 256 bit | SMULT for field=1]
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error
```

```
// load R0 with value n
R0 <-- n
// r:= xP mod n
PKC_CONFG <-- 0x00090C00 // B=12, C=9
PKC_CMD   <-- 0x80000404 // CALC2 | 256 bit | RED
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error
// d*r mod n:
PKC_CONFG <-- 0x000B0903 // A=3, B=9, C=11
PKC_CMD   <-- 0x00000403 // 256 bit | MULTN
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error
// (h + ek*d) mod n:
PKC_CONFG <-- 0x000E0B02 // A=2, B=11, C=14
PKC_CMD   <-- 0x00000401 // 256 bit | ADDN
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error
// kinv:
PKC_CONFG <-- 0x00010800 // B=8, C=1
PKC_CMD   <-- 0x00000406/9 // 256 bit | INV/INV2 if ek even
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error
// s:
PKC_CONFG <-- 0x000A0E01 // A=1, B=14, C=10
PKC_CMD   <-- 0x00000403 // 256 bit | MULTN
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error
R9 = r
R10 = s
```

6.5.2.12 ECDSA Signature Verification ECDSAVER

Attention: This command may not be included in the delivery. Whether it is included, this can be seen in the list of PKC_CMD.TYPE in "Control and Status Registers" on Page 144.

The command **ECDSAVER** realizes the ECDSA signature generation of a message,

Status := ECDSAVER(h,r,s,Q,n,P,a,b,p)

on an elliptic curve E(a,b,p). See **Chapter 6.2.2.5** and **Chapter 6.2.6.5** for notation and more details.

Input values:

- p: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R0**.

- n: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R1**.
- a: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R4**.
- b: The integer or binary polynomial (depending on **PKC_CMD.FIELD**) is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R5**.
- r: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R10**.
- s: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R11**.
- h: The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R12**.
- P=(x_P,y_P):
  x_P: The x-coordinate of P_A is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R2**.
  y_P: The y-coordinate of P_A is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R3**.
- Q=(x_Q,y_Q):
  x_Q: The x-coordinate of P_Q is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R8**.
  y_Q: The y-coordinate of P_Q is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R9**.

Further Input conditions:

- **PKC_CMD.CALCR2** = 1_B, if (R^2mod N) was not computed yet, i.e., this is the first modular operation using the module p.
- **PKC_CMD.SIZE** = 4.
- - If **PKC_CMD.FIELD**=0_B then 0 ≤ x,y,a,b < p < 2^(64*__PKC_CMD.SIZE__), p has to be odd.
  - If **PKC_CMD.FIELD**=1_B then deg(x),deg(y),deg(a),deg(b) < deg(p) < (64*__PKC_CMD.SIZE__), p_0=1_B.
- a, b, p are valid parameters such that an elliptic curve E(a,b,p) is defined.
- P lies on the curve E(a,b,p).
- n is the order of the Point P.
- Q lies on the curve E(a,b,p).
- h is a hash value of the message that has to be signed, generated according to the signature standard which is going to be used. h has to be interpreted as an integer. 0 ≤ h < n.

Output: (After operation finished)

- **PKC_CMD.CALCR2** = 0_B.
- **PKC_CMD.SIGNA** = 0_B, **PKC_CMD.SIGNB** = 0_B, **PKC_SCM_R4** = a, **PKC_SCM_R5** = b.
- **PKC_STATUS.XYNV** = 1_B, if the signature is not valid: This happens, if:
  - x_Q, y_Q are not valid
  - s ≥ n (In this case **PKC_STATUS.FLADD** = A_H),
  - r ≥ n (In this case **PKC_STATUS.FLADD** = A_H).

- **PKC_STATUS.INFTY** = 1_B, if the signature is not valid: This happens, if:
  - $P_1 = 0$. (In this case **PKC_STATUS.FLADD** = 8_H).

Additional Output:

- w := INV(s,n) : The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R13**.
- $P_1 = (x_1,y_1) := P_2+P_3$:
  x_1: The x-coordinate of P_1 is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R14**.
  y_1: The y-coordinate of P_1 is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R15**.
- $P_2 = (x_2,y_2) :=$ SMULT(u_2,P,a,b,p):
  x_2: The x-coordinate of P_2 is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R16**.
  y_2: The y-coordinate of P_2 is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R17**.
- $P_3 = (x_3,y_3) :=$ SMULT(u_3,Q,a,b,p):
  x_3: The x-coordinate of P_3 is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R18**.
  y_3: The y-coordinate of P_3 is an integer or binary polynomial (depending on **PKC_CMD.FIELD**) which is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R19**.
- u_2 := MULTN(h,w,n) : The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R20**.
- u_3 := MULTN(r,w,n) : The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R21**.
- v.= RED(x_1,n): The integer is represented (according to the coding described in **Chapter 6.4.2**) by the value in **PKC_SCM_R22**.

Programming example, for ECDSA Signature Verification without this command:

If this command is not included in the delivery then it still can be implemented by using the following code: This is illustrated for the settings with **PKC_CMD.SIZE** = 4 and **PKC_CMD.SIGNA** = **PKC_CMD.SIGNB** = 0_B. If (**SIGNA**,**SIGNB**) is not (0_B, 0_B) then the bits have to be taken into account for the **first**SMULT-operation.

```
Input: p, n, a, b, r, s, h, P=(xP,yP), Q=(xQ,yQ), as defined above.
Output: accept/not accept
Additional Output: w=s^-1 mod n, u2=h*w mod n, u3=r*w mod n, P2=u2*P, P3=u3Q,P1:=P2+P3,
v=x1 mod n.
R0 <-- n
R1 <-- 0
R2 <-- r
R3 <-- s
R4 <-- a
R5 <-- b
R6 <-- xP
R7 <-- yP
R8 <-- xQ
R9 <-- yQ
```

```
R10<-- h
R11<-- 0
R12<-- 0
R13<-- 0
R14<-- 0
R15<-- 0
// Pre-check: 0<r,s<n => if not, not accept: omitted
...
// w:
PKC_CONFG <-- 0x00010300 // B=3, C=1
PKC_CMD   <-- 0x80000406 // CALC2 | 256 bit | INV
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error
// u2,u3:
PKC_CONFG <-- 0x000C0A01 // A=1, B=10, C=12
PKC_CMD   <-- 0x00000403 // 256 bit | MULTN
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error
PKC_CONFG <-- 0x000D0201 // A=1, B=2, C=13
PKC_CMD   <-- 0x00000403 // 256 bit | MULTN
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error
// load R0 with value p:
R0 <-- p
// P2=u2*P:
PKC_CONFG <-- 0x000E0C06 // A=6, B=12, C=14
PKC_CMD   <-- 0x80000422 // CALC2 | 256 bit | SMULT
[PKC_CMD  <-- 0x800004A2 // CALC2 | 256 bit | SMULT for field=1]
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error
// move u3(in R13) to R12
R12 <-- R13
// P3=u3*Q:
PKC_CONFG <-- 0x00060C08 // A=8, B=12, C=6
PKC_CMD   <-- 0x00000422 // 256 bit | SMULT
[PKC_CMD  <-- 0x000004A2 // 256 bit | SMULT for field=1]
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error
// P1=P2+P3
PKC_CONFG <-- 0x00080E06 // A=6, B=14, C=8
PKC_CMD   <-- 0x00000421 // 256 bit | PADD
[PKC_CMD  <-- 0x000004A1 // 256 bit | PADD for field=1]
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error
// load R0 with value n:
```

```
R0 <-- n
// v:= x1 mod n
PKC_CONFG <-- 0x000A0800 // B=8, C=10
PKC_CMD   <-- 0x80000404 // CALC2 | 256 bit | RED
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error
// r-v mod n:
PKC_CONFG <-- 0x0001020A // A=10, B=2, C=1
PKC_CMD   <-- 0x00000402 // 256 bit | SUBN
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error
//now: R1 = r-v mod n
Check R1 for being = 0. // => if yes, accept. If not, not accept:
```

6.6 Pseudo Coding Example with Test Data

```
In the following, the abbreviation:
PKC_SCM_Rj <-- 0x...
stands for loading the right hand side integer (or binary polynomial, formatted as
integer)into Long Integer Register Rj.
E.g.:
R5 <-- 0x123456789ABCDEF00112233445566778899AABBCCDDEEFF
stands for
PKC_SCM_R5[0] <-- 0xCCDDEEFF
PKC_SCM_R5[1] <-- 0x8899AABB
PKC_SCM_R5[2] <-- 0x44556677
PKC_SCM_R5[3] <-- 0x00112233
PKC_SCM_R5[4] <-- 0x89ABCDEF
PKC_SCM_R5[5] <-- 0x01234567
PKC_SCM_R5[6] <-- 0x00000000
PKC_SCM_R5[7] <-- 0x00000000
Here
PKC_SCM_Rj[i] <-- wd
stands for writing a 32-bit value wd into the SCM with address offset
j*0x20+i*0x04One can view Rj as an array Rj[8] of (32-bit) words of length 8


c := ADDN(a,b,n)
=============================================================================
INPUT:
a = 0x0000000000000000000000000000000207339025D6814EDCB4940AA16FF7F1
b = 0x00000000000000000000000000000014B6E5FF79F4F46183EF63536609DD2
n = 0x00000000000000000000000000000049F77F1F1D2EEBAA9E01F0F0000C3CD
in Z
size   = 2
OPPTRA = 7
OPPTRB = 11
OPPTRC = 2
OUTPUT:
c = 0x0000000000000000000000000000000352A1F01D75D094F4F38A3FD7D095C3
=============================================================================

R0 <-- 0x49F77F1F1D2EEBAA9E01F0F0000C3CD
R7 <-- 0x207339025D6814EDCB4940AA16FF7F1
R11<-- 0x14B6E5FF79F4F46183EF63536609DD2
PKC_CONFG <-- 0x00020B07
PKC_CMD   <-- 0x00000201
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0

Now, the result is:

PKC_SCM_R2[0] = 0xD7D095C3
PKC_SCM_R2[1] = 0xF4F38A3F
```

```
PKC_SCM_R2[2] = 0x1D75D094
PKC_SCM_R2[3] = 0x0352A1F0
PKC_SCM_R2[4] = 0x00000000
PKC_SCM_R2[5] = 0x00000000
PKC_SCM_R2[6] = 0x00000000
PKC_SCM_R2[7] = 0x00000000


c := MULTN(a,b,n)
=========================================================================
INPUT:
a = 0x00000083141AE94B361C9779CE5FC74C529B6005A0E755662FFB52733649ABCC
b = 0x000000BD332F16ED5E7B5B4A2DD1C0A0AC6A7BEBF78409DBEFE2A44BF1EDE8AE
n = 0x000002C352DC107EB45AC407A196F9C8889F77F1F1D2EEBAA9E01F0F0000C3CD
in Z
size   = 4
OPPTRA = 13
OPPTRB = 11
OPPTRC = 9
OUTPUT:
c = 0x0000018A570FAD7E8C669A34591FBBE5C24C6EF2DFAC11258D521902A3548D79
=========================================================================

R0 <-- 0x2C352DC107EB45AC407A196F9C8889F77F1F1D2EEBAA9E01F0F0000C3CD
R13<-- 0x83141AE94B361C9779CE5FC74C529B6005A0E755662FFB52733649ABCC
R11<-- 0xBD332F16ED5E7B5B4A2DD1C0A0AC6A7BEBF78409DBEFE2A44BF1EDE8AE
PKC_CONFG <-- 0x00090B0D
PKC_CMD   <-- 0x80000403
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0

Now, the result is:

PKC_SCM_R9[0] = 0xA3548D79
PKC_SCM_R9[1] = 0x8D521902
PKC_SCM_R9[2] = 0xDFAC1125
PKC_SCM_R9[3] = 0xC24C6EF2
PKC_SCM_R9[4] = 0x591FBBE5
PKC_SCM_R9[5] = 0x8C669A34
PKC_SCM_R9[6] = 0x570FAD7E
PKC_SCM_R9[7] = 0x0000018A

c := MULTN(a,b,n)
=========================================================================
INPUT:
a = 0x00000083141AE94B361C9779CE5FC74C529B6005A0E755662FFB52733649ABCC
b = 0x000000BD332F16ED5E7B5B4A2DD1C0A0AC6A7BEBF78409DBEFE2A44BF1EDE8AE
n = 0x000002C352DC107EB45AC407A196F9C8889F77F1F1D2EEBAA9E01F0F0000C3CD
in F_2[X]
size   = 4
OPPTRA = 13
```

```
OPPTRB = 11
OPPTRC = 9
OUTPUT:
c = 0x000001C0F8520023C2B232CD9B99BF289B0D80AA4333D3A894242B3C0289D0B0
=========================================================================

R0 <-- 0x2C352DC107EB45AC407A196F9C8889F77F1F1D2EEBAA9E01F0F0000C3CD
R13<-- 0x83141AE94B361C9779CE5FC74C529B6005A0E755662FFB52733649ABCC
R11<-- 0xBD332F16ED5E7B5B4A2DD1C0A0AC6A7BEBF78409DBEFE2A44BF1EDE8AE
PKC_CONFG <-- 0x00090B0D
PKC_CMD   <-- 0x80000483
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0

Now, the result is:

PKC_SCM_R9[0] = 0x0289D0B0
PKC_SCM_R9[1] = 0x94242B3C
PKC_SCM_R9[2] = 0x4333D3A8
PKC_SCM_R9[3] = 0x9B0D80AA
PKC_SCM_R9[4] = 0x9B99BF28
PKC_SCM_R9[5] = 0xC2B232CD
PKC_SCM_R9[6] = 0xF8520023
PKC_SCM_R9[7] = 0x000001C0


c := RED(b,n)
=========================================================================
INPUT:
b = 0x0000000000000000000000000000052DC107EB45AC407A196F9C8889F77F1
n = 0x0000000000000000000000000000000000000000000001A9E01F0F0000C3CD
in Z
size   = 2
OPPTRB = 9
OPPTRC = 11
OUTPUT:
c = 0x0000000000000000000000000000000000000000000000AEF08CBBD7A2EF3C
=========================================================================

R0 <-- 0x1A9E01F0F0000C3CD
R9 <-- 0x52DC107EB45AC407A196F9C8889F77F1
PKC_CONFG <-- 0x000B0900
PKC_CMD   <-- 0x80000204
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0

Now, the result is:

PKC_SCM_R11[0] = 0xD7A2EF3C
PKC_SCM_R11[1] = 0xAEF08CBB
PKC_SCM_R11[2] = 0x00000000
```

```
PKC_SCM_R11[3] = 0x00000000
PKC_SCM_R11[4] = 0x00000000
PKC_SCM_R11[5] = 0x00000000
PKC_SCM_R11[6] = 0x00000000
PKC_SCM_R11[7] = 0x00000000

c := INV(b,n)
=========================================================================
INPUT:
b = 0x00000000000000FA71DA0CDF81F0BA965616072F676A5D6053EA55CEED0A972A
n = 0x00000000000000FEB45AC407A196F9C8889F77F1F1D2EEBAA9E01F0F0000C3CD
in Z
size   = 4
OPPTRB = 2
OPPTRC = 11
OUTPUT:
c = 0x000000000000003D9612BAFFFD408B30B92E5DAD8979F474223CBA86530DDD23
=========================================================================

R0 <-- 0xFEB45AC407A196F9C8889F77F1F1D2EEBAA9E01F0F0000C3CD
R2 <-- 0xFA71DA0CDF81F0BA965616072F676A5D6053EA55CEED0A972A
PKC_CONFG <-- 0x000B0200
PKC_CMD   <-- 0x00000406
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS.NOTINV = 0
Now, the result is:

PKC_SCM_R11[0] = 0x530DDD23
PKC_SCM_R11[1] = 0x223CBA86
PKC_SCM_R11[2] = 0x8979F474
PKC_SCM_R11[3] = 0xB92E5DAD
PKC_SCM_R11[4] = 0xFD408B30
PKC_SCM_R11[5] = 0x9612BAFF
PKC_SCM_R11[6] = 0x0000003D
PKC_SCM_R11[7] = 0x00000000


c := MULT(a,b)
=========================================================================
INPUT:
a = 0x0000000000000000000000000000000889F77F1F1D2EEBAA9E01F0F0000C3CD
b = 0x0000000000000000000000000000000366907A1EDCC18C352DC107EB45AC407
in Z
size   = 4
OPPTRA = 11
OPPTRB = 2
OPPTRC = 7
OUTPUT:
c = 0x1D09B0C802B948FAE730BED80DA36975E35A4F0C490D95B4AEF80E4090004E9B
=========================================================================
```

```
R11<-- 0x889F77F1F1D2EEBAA9E01F0F0000C3CD
R2 <-- 0x366907A1EDCC18C352DC107EB45AC407
PKC_CONFG <-- 0x0007020B
PKC_CMD   <-- 0x80000408
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0

Now, the result is:

PKC_SCM_R7[0] = 0x90004E9B
PKC_SCM_R7[1] = 0xAEF80E40
PKC_SCM_R7[2] = 0x490D95B4
PKC_SCM_R7[3] = 0xE35A4F0C
PKC_SCM_R7[4] = 0x0DA36975
PKC_SCM_R7[5] = 0xE730BED8
PKC_SCM_R7[6] = 0x02B948FA
PKC_SCM_R7[7] = 0x1D09B0C8

C := EXP(a,b,n)
================================================================================
INPUT:
a = 0x10C8D86C99DD29BB68602EAFC52762A52D818782C423C65D008EA89D4B2EE325
b = 0x1C7597833A1217DD5C5B56AA0028C22A56CCB6647EBE39E9DB93A9E370DD3751
n = 0x4626DA66D582819E9FEB7E476424B8E20E6B394698FDD3C0A9E0DFBE0000C3CF
in Z
size   = 4
OPPTRA = 10
OPPTRB = 8
OPPTRC = 6
OUTPUT:
c = 0x2746F12DE72DA5A141327E2AAB3ABF8B3D1E8B408DC4511B3C1CA47CB4CF0B55
================================================================================
R0 <-- 0x4626DA66D582819E9FEB7E476424B8E20E6B394698FDD3C0A9E0DFBE0000C3CF
R10<-- 0x10C8D86C99DD29BB68602EAFC52762A52D818782C423C65D008EA89D4B2EE325
R8 <-- 0x1C7597833A1217DD5C5B56AA0028C22A56CCB6647EBE39E9DB93A9E370DD3751
PKC_CONFG <-- 0x0006080A
PKC_CMD   <-- 0x80000410
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0

Now, the result is:

PKC_SCM_R6[0] = 0xB4CF0B55
PKC_SCM_R6[1] = 0x3C1CA47C
PKC_SCM_R6[2] = 0x8DC4511B
PKC_SCM_R6[3] = 0x3D1E8B40
PKC_SCM_R6[4] = 0xAB3ABF8B
PKC_SCM_R6[5] = 0x41327E2A
PKC_SCM_R6[6] = 0xE72DA5A1
PKC_SCM_R6[7] = 0x2746F12D
```

```
PC := PDBL(PA,a,b,p)
================================================================================
INPUT:
p = 0x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0000000000000000000000001
a = 0x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFE
b = 0x00000000B4050A850C04B3ABF54132565044B0B7D7BFD8BA270B39432355FFB4
PAx=0x00000000E6AC2312EDB4AF4ED6B18ACC9DEE076E1D3BB08BFD019F3792CF2A67
PAy=0x0000000011810D5F32F0431E71EFB5EF9D9892C2DBE4F6845CBAE1A58AD6C092
in Z
size   = 4
OPPTRA = 14
OPPTRC = 8
OUTPUT:
PCx=0x00000000373CA1D91A78CDEB41E04CFBC195A7E87DDDBBF4D626B79306F75462
PCy=0x0000000043432DE379DC3DB12C5F4CA0AA9EA96FDCE835CB9C337A4ACF703502
================================================================================

R0 <-- 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0000000000000000000000001
R4 <-- 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFE
R5 <-- 0xB4050A850C04B3ABF54132565044B0B7D7BFD8BA270B39432355FFB4
R14<-- 0xE6AC2312EDB4AF4ED6B18ACC9DEE076E1D3BB08BFD019F3792CF2A67
R15<-- 0x11810D5F32F0431E71EFB5EF9D9892C2DBE4F6845CBAE1A58AD6C092
PKC_CONFG <-- 0x0008060E
PKC_CMD   <-- 0x80000420 // CALC2 | 256 bit | PDBL
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error

Now, the result is:

PKC_SCM_R8[0] = 0x06F75462
PKC_SCM_R8[1] = 0xD626B793
PKC_SCM_R8[2] = 0x7DDDBBF4
PKC_SCM_R8[3] = 0xC195A7E8
PKC_SCM_R8[4] = 0x41E04CFB
PKC_SCM_R8[5] = 0x1A78CDEB
PKC_SCM_R8[6] = 0x373CA1D9
PKC_SCM_R8[7] = 0x00000000
PKC_SCM_R9[0] = 0xCF703502
PKC_SCM_R9[1] = 0x9C337A4A
PKC_SCM_R9[2] = 0xDCE835CB
PKC_SCM_R9[3] = 0xAA9EA96F
PKC_SCM_R9[4] = 0x2C5F4CA0
PKC_SCM_R9[5] = 0x79DC3DB1
PKC_SCM_R9[6] = 0x43432DE3
PKC_SCM_R9[7] = 0x00000000


PC := PADD(PA,PB,a,b,p)
```

```
========================================================================
INPUT:
p = 0x0000000000000000C302F41D932A36CDA7A3463093D18DB78FCE476DE1A86297
a = 0x0000000000000000006A91174076B1E0E19C39C031FE8685C1CAE040E5C69A28EF
b = 0x0000000000000000469A28EF7C28CCA3DC721D044F4496BCCA7EF4146FBF25C9
PAx=0x0000000000000000001782F6C35ED8587FBAE52B9AC587FFDE88D7B544CDE565D3
PAy=0x0000000000000000AB462CFE960A968DCF51E09B81E9619D96A7DC4BF71EB6DD
PBx=0x0000000000000000026EF57395D3A9D7CBC6AC1E7A4EC8B0F856DDB02971AD4F4
PBy=0x0000000000000000027005FEF3CA950CF6D482C19C786AA19D8467EC5C6FC0242
in Z
size   = 4
OPPTRA = 6
OPPTRB = 14
OPPTRC = 12
OUTPUT:
PCx=0x0000000000000000071B39A4787AEA06FEF991B92DCE9B4AC1826BD258057C72A
PCy=0x00000000000000009016290096AA2A791A598D7CC0C753ED962AFDA2084DDCF9
========================================================================

R0 <-- 0xC302F41D932A36CDA7A3463093D18DB78FCE476DE1A86297
R4 <-- 0x6A91174076B1E0E19C39C031FE8685C1CAE040E5C69A28EF
R5 <-- 0x469A28EF7C28CCA3DC721D044F4496BCCA7EF4146FBF25C9
R6 <-- 0x1782F6C35ED8587FBAE52B9AC587FFDE88D7B544CDE565D3
R7 <-- 0xAB462CFE960A968DCF51E09B81E9619D96A7DC4BF71EB6DD
R14<-- 0x26EF57395D3A9D7CBC6AC1E7A4EC8B0F856DDB02971AD4F4
R15<-- 0x27005FEF3CA950CF6D482C19C786AA19D8467EC5C6FC0242
PKC_CONFG <-- 0x000C0E06
PKC_CMD   <-- 0x80000421 // CALC2 | 256 bit | PADD
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error

Now, the result is:

PKC_SCM_R12[0] = 0x8057C72A
PKC_SCM_R12[1] = 0x1826BD25
PKC_SCM_R12[2] = 0xDCE9B4AC
PKC_SCM_R12[3] = 0xEF991B92
PKC_SCM_R12[4] = 0x87AEA06F
PKC_SCM_R12[5] = 0x71B39A47
PKC_SCM_R12[6] = 0x00000000
PKC_SCM_R12[7] = 0x00000000
PKC_SCM_R13[0] = 0x084DDCF9
PKC_SCM_R13[1] = 0x962AFDA2
PKC_SCM_R13[2] = 0xC0C753ED
PKC_SCM_R13[3] = 0x1A598D7C
PKC_SCM_R13[4] = 0x96AA2A79
PKC_SCM_R13[5] = 0x90162900
PKC_SCM_R13[6] = 0x00000000
PKC_SCM_R13[7] = 0x00000000
```

```
PC := SMULT(k,PB,a,b,p)
=========================================================================
INPUT:
p = 0x8000020000000000000000000000000000000000000000400000000000000001
a = 0x8000000000000000000000000000000000000000000000000000000000000001
b = 0x00000066647EDE6C332C7F8C0923BB58213B333B20E9CE4281FE115F7D8F90AD
k = 0x00000156F064D44A640E081838BE18C7153C227078C2A38BDA85EAE0F2028F28
PAx=0x000001CB59598D9618A5FCB2AF8DC86A342595C55BAF914B64E04BDA366907A1
PAy=0x000001C6F1A6622A33C36FA3821292447344C829B4B53881A7875DA3C16B1628
in F_2[X]
size   = 4
OPPTRA = 6
OPPTRB = 14
OPPTRC = 12
OUTPUT:
PCx=0x000000F6DBADAEF6E671906A3D4398D93F5901A58C12E10B389773188790ABB3
PCy=0x0000018945A17B4902832E1F603A30AE246849FBC1D957043C67CBF711B6675C
=========================================================================

R0 <-- 0x2000000000000000000000000000000000000040000000000000000001
R4 <-- 0x1
R5 <-- 0x66647EDE6C332C7F8C0923BB58213B333B20E9CE4281FE115F7D8F90AD
R6 <-- 0x1CB59598D9618A5FCB2AF8DC86A342595C55BAF914B64E04BDA366907A1
R7 <-- 0x1C6F1A6622A33C36FA3821292447344C829B4B53881A7875DA3C16B1628
R14<-- 0x156F064D44A640E081838BE18C7153C227078C2A38BDA85EAE0F2028F28
PKC_CONFG <-- 0x000C0E06
PKC_CMD   <-- 0x800004A2 // CALC2 | 256 bit | SMULT
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error

Now, the result is:

PKC_SCM_R12[0] = 0x8790ABB3
PKC_SCM_R12[1] = 0x38977318
PKC_SCM_R12[2] = 0x8C12E10B
PKC_SCM_R12[3] = 0x3F5901A5
PKC_SCM_R12[4] = 0x3D4398D9
PKC_SCM_R12[5] = 0xE671906A
PKC_SCM_R12[6] = 0xDBADAEF6
PKC_SCM_R12[7] = 0x000000F6
PKC_SCM_R13[0] = 0x11B6675C
PKC_SCM_R13[1] = 0x3C67CBF7
PKC_SCM_R13[2] = 0xC1D95704
PKC_SCM_R13[3] = 0x246849FB
PKC_SCM_R13[4] = 0x603A30AE
PKC_SCM_R13[5] = 0x02832E1F
PKC_SCM_R13[6] = 0x45A17B49
PKC_SCM_R13[7] = 0x00000189
```

```
PC := SMULTcurve25519(k,PB,a,d,p)
ecc25519
========================================================================
INPUT:
p = 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFED
a = 0x00000000000000000000000000000000000000000000000000000000000076D06
(a-2)/4 = 0x00000000000000000000000000000000000000000000000000000000001DB41
b = 0x0000000000000000000000000000000000000000000000000000000000000001
k = 0x777EF57398780459FEAA59BFDA5F5496B3F27C2E0C98C2E9253210F45E94A6E8
PAx=0x0000000000000000000000000000000000000000000000000000000000000009
PAy=0x5F51E65E475F794B1FE122D388B72EB36DC2B28192839E4DD6163A5D81312C14
OPPTRA = 4
OPPTRB = 8
OPPTRC = 14
OUTPUT:
PCx=0x32575AB9FAFFCB1783D722A4B9556A4152736E45E34E05B581A9CB7C0CFFEDB9
PCy=0x2C6D405D488A347C73C28860D270BFACA6C891CF86DD7FF60381A8B4D88D2CBC
========================================================================

R0 <-- 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFED
R4 <-- 0x9
R5 <-- 0x1DB41
R8 <-- 0x777EF57398780459FEAA59BFDA5F5496B3F27C2E0C98C2E9253210F45E94A6E8
PKC_CONFG <-- 0x000E0804
PKC_CMD   <-- 0x80000428 // CALC2 | xxx bit | SMULTCurve25519
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error

Now, the result is:

PKC_SCM_R14[0] = 0x0CFFEDB9
PKC_SCM_R14[1] = 0x81A9CB7C
PKC_SCM_R14[2] = 0xE34E05B5
PKC_SCM_R14[3] = 0x52736E45
PKC_SCM_R14[4] = 0xB9556A41
PKC_SCM_R14[5] = 0x83D722A4
PKC_SCM_R14[6] = 0xFAFFCB17
PKC_SCM_R14[7] = 0x32575AB9

x := Xrecovered25519(y,a,d,p)
ed25519
========================================================================
INPUT:
p = 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFED
a = 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEC
d = 0x52036CEE2B6FFE738CC740797779E89800700A4D4141D8AB75EB4DCA135978A3
D2= 0x0000000000000000000000000000000000000000000000000000000000000000
(p-5)/8= 0x0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD
I = 0x2B8324804FC1DF0B2B4D00993DFBD7A72F431806AD2FE478C4EE1B274A0EA0B0
PAy=0x6666666666666666666666666666666666666666666666666666666666666658
```

```
OUTPUT:
PCx=0x216936D3CD6E53FEC0A4E231FDD6DC5C692CC7609525A7B2C9562D608F25D51A
PCy=0x6666666666666666666666666666666666666666666666666666666666666658
========================================================================

R0 <-- 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFED
R4 <-- 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD
R8 <-- 0x6666666666666666666666666666666666666666666666666666666666666658
R10<-- 0x52036CEE2B6FFE738CC740797779E89800700A4D4141D8AB75EB4DCA135978A3
R11<-- 0x2B8324804FC1DF0B2B4D00993DFBD7A72F431806AD2FE478C4EE1B274A0EA0B0
PKC_CONFG <-- 0x00000000
PKC_CMD   <-- 0x80000429 // CALC2 | xxx bit | XRecoverED25519
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error

Now, the result is:

PKC_SCM_R6[0] = 0x8F25D51A
PKC_SCM_R6[1] = 0xC9562D60
PKC_SCM_R6[2] = 0x9525A7B2
PKC_SCM_R6[3] = 0x692CC760
PKC_SCM_R6[4] = 0xFDD6DC5C
PKC_SCM_R6[5] = 0xC0A4E231
PKC_SCM_R6[6] = 0xCD6E53FE
PKC_SCM_R6[7] = 0x216936D3
OR:
PKC_SCM_R6[0] = 0x70DA2AD3
PKC_SCM_R6[1] = 0x36A9D29F
PKC_SCM_R6[2] = 0x6ADA584D
PKC_SCM_R6[3] = 0x96D3389F
PKC_SCM_R6[4] = 0x022923A3
PKC_SCM_R6[5] = 0x3F5B1DCE
PKC_SCM_R6[6] = 0x3291AC01
PKC_SCM_R6[7] = 0x5E96C92C

PC := SMULTed25519(k,PB,a,d,p)
ed25519
========================================================================
INPUT:
p = 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFED
a = 0x-0000000000000000000000000000000000000000000000000000000000000001
d = 0x52036CEE2B6FFE738CC740797779E89800700A4D4141D8AB75EB4DCA135978A3
D2= 0x590456B4E53F8A4DCB27240F78310D2021430EEF5F8C52E701DB17FDBE8FD3F4
k = 0x633D67A11A64AE974822A985F1958D4B58C333568F1D419EFE3147D74B312EA5
PAx=0x216936D3CD6E53FEC0A4E231FDD6DC5C692CC7609525A7B2C9562D608F25D51A
PAy=0x6666666666666666666666666666666666666666666666666666666666666658
OPPTRA = 2
OPPTRB = 6
OPPTRC = 12
OUTPUT:
```

```
PCx=0x262A841805591EAFA9AF969F3550BF33282A4EC6091C110FCBCF7B286FDDCDBA
PCy=0x0F2888207764F26EDD8E98C43CDAF43BCC5E1BA75114BDB49135B1CD303C5E0F
========================================================================

R0 <-- 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFED
R1 <-- 0x590456B4E53F8A4DCB27240F78310D2021430EEF5F8C52E701DB17FDBE8FD3F4
R2 <-- 0x216936D3CD6E53FEC0A4E231FDD6DC5C692CC7609525A7B2C9562D608F25D51A
R3 <-- 0x6666666666666666666666666666666666666666666666666666666666666658
R6 <-- 0x633D67A11A64AE974822A985F1958D4B58C333568F1D419EFE3147D74B312EA5
PKC_CONFG <-- 0x000C0602
PKC_CMD   <-- 0x8000042A // CALC2 | xxx bit | SMULTED25519
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error

Now, the result is:

PKC_SCM_R12[0] = 0x6FDDCDBA
PKC_SCM_R12[1] = 0xCBCF7B28
PKC_SCM_R12[2] = 0x091C110F
PKC_SCM_R12[3] = 0x282A4EC6
PKC_SCM_R12[4] = 0x3550BF33
PKC_SCM_R12[5] = 0xA9AF969F
PKC_SCM_R12[6] = 0x05591EAF
PKC_SCM_R12[7] = 0x262A8418
PKC_SCM_R13[0] = 0x303C5E0F
PKC_SCM_R13[1] = 0x9135B1CD
PKC_SCM_R13[2] = 0x5114BDB4
PKC_SCM_R13[3] = 0xCC5E1BA7
PKC_SCM_R13[4] = 0x3CDAF43B
PKC_SCM_R13[5] = 0xDD8E98C4
PKC_SCM_R13[6] = 0x7764F26E
PKC_SCM_R13[7] = 0x0F288820

Ed25519CheckValid(...)
ed25519
========================================================================
INPUT:
p = 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFED
a = 0x-0000000000000000000000000000000000000000000000000000000000000001
d = 0x52036CEE2B6FFE738CC740797779E89800700A4D4141D8AB75EB4DCA135978A3
D2= 0x590456B4E53F8A4DCB27240F78310D2021430EEF5F8C52E701DB17FDBE8FD3F4
Ax=0x24257A24D1A9664345F24E9668E9C83E11EC6B9BC9E44C64F1F3CB39540FC550
Ay=0x70286A0D4646B3E5FA1616FFE3BBDB72E079B8492464A3287AACA4ADDB67FF68
Bx=0x216936D3CD6E53FEC0A4E231FDD6DC5C692CC7609525A7B2C9562D608F25D51A
By=0x6666666666666666666666666666666666666666666666666666666666666658
Rx=0x129FCC11297110E87C11F0B26AB909BB122B5FD96F348C90F8045A38B6990B98
Ry=0x069F0246816B8CB596DCD3C1B422E8971CD5839A50CDB4796ACADB43CF6EE94C
S = 0x0CA7EAF57C70635EBD00C41EE52621AA76A905E66D30F341360F5CAD27177FA3
h = 0x029C71EA0F94997BFC46D4F1644D7B0BE7FE147345EE60A475063703431B0392
OUTPUT:
```

```
P1x=0x4CFE7900DE559A72A6FE6BB2649215113B13E6895F3AF9EDD76CDB2CD5469FEC
P1y=0x2168FA7908509972A32C4EE46BEEFDCA5C132EB69B6FB7102690E796BB88AB2B
P2x=0x176DC8053F6F41DA2C3CFD1D40774A8588BD9CCA424CAD7223A8C8490703080A
P2y=0x1F3D9DA2A6CB92DFCFB31787E1181D6F1646703A910BD170DDDAD58F3177FCA9
P3x=0x4CFE7900DE559A72A6FE6BB2649215113B13E6895F3AF9EDD76CDB2CD5469FEC
P3y=0x2168FA7908509972A32C4EE46BEEFDCA5C132EB69B6FB7102690E796BB88AB2B
=========================================================================

R0 <-- 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFED
R1 <-- 0x590456B4E53F8A4DCB27240F78310D2021430EEF5F8C52E701DB17FDBE8FD3F4
R2 <-- 0x216936D3CD6E53FEC0A4E231FDD6DC5C692CC7609525A7B2C9562D608F25D51A
R3 <-- 0x6666666666666666666666666666666666666666666666666666666666666658
R4 <-- 0xCA7EAF57C70635EBD00C41EE52621AA76A905E66D30F341360F5CAD27177FA3
R5 <-- 0x29C71EA0F94997BFC46D4F1644D7B0BE7FE147345EE60A475063703431B0392
R6 <-- 0x24257A24D1A9664345F24E9668E9C83E11EC6B9BC9E44C64F1F3CB39540FC550
R7 <-- 0x70286A0D4646B3E5FA1616FFE3BBDB72E079B8492464A3287AACA4ADDB67FF68
R8 <-- 0x129FCC11297110E87C11F0B26AB909BB122B5FD96F348C90F8045A38B6990B98
R9 <-- 0x69F0246816B8CB596DCD3C1B422E8971CD5839A50CDB4796ACADB43CF6EE94C
PKC_CMD   <-- 0x8000042B // CALC2 | xxx bit | CHECKVALIDED25519
PKC_CTRL  <-- 0x00000001
Wait until PKC_STATUS.BUSY = 0
Check that PKC_STATUS = 0 // No error

Now, the result is:

PKC_SCM_R10[0] = 0xD5469FEC
PKC_SCM_R10[1] = 0xD76CDB2C
PKC_SCM_R10[2] = 0x5F3AF9ED
PKC_SCM_R10[3] = 0x3B13E689
PKC_SCM_R10[4] = 0x64921511
PKC_SCM_R10[5] = 0xA6FE6BB2
PKC_SCM_R10[6] = 0xDE559A72
PKC_SCM_R10[7] = 0x4CFE7900
PKC_SCM_R11[0] = 0xBB88AB2B
PKC_SCM_R11[1] = 0x2690E796
PKC_SCM_R11[2] = 0x9B6FB710
PKC_SCM_R11[3] = 0x5C132EB6
PKC_SCM_R11[4] = 0x6BEEFDCA
PKC_SCM_R11[5] = 0xA32C4EE4
PKC_SCM_R11[6] = 0x08509972
PKC_SCM_R11[7] = 0x2168FA79
PKC_SCM_R12[0] = 0x0703080A
PKC_SCM_R12[1] = 0x23A8C849
PKC_SCM_R12[2] = 0x424CAD72
PKC_SCM_R12[3] = 0x88BD9CCA
PKC_SCM_R12[4] = 0x40774A85
PKC_SCM_R12[5] = 0x2C3CFD1D
PKC_SCM_R12[6] = 0x3F6F41DA
PKC_SCM_R12[7] = 0x176DC805
PKC_SCM_R13[0] = 0x3177FCA9
PKC_SCM_R13[1] = 0xDDDAD58F
```

```
PKC_SCM_R13[2] = 0x910BD170
PKC_SCM_R13[3] = 0x1646703A
PKC_SCM_R13[4] = 0xE1181D6F
PKC_SCM_R13[5] = 0xCFB31787
PKC_SCM_R13[6] = 0xA6CB92DF
PKC_SCM_R13[7] = 0x1F3D9DA2
PKC_SCM_R14[0] = 0xD5469FEC
PKC_SCM_R14[1] = 0xD76CDB2C
PKC_SCM_R14[2] = 0x5F3AF9ED
PKC_SCM_R14[3] = 0x3B13E689
PKC_SCM_R14[4] = 0x64921511
PKC_SCM_R14[5] = 0xA6FE6BB2
PKC_SCM_R14[6] = 0xDE559A72
PKC_SCM_R14[7] = 0x4CFE7900
PKC_SCM_R15[0] = 0xBB88AB2B
PKC_SCM_R15[1] = 0x2690E796
PKC_SCM_R15[2] = 0x9B6FB710
PKC_SCM_R15[3] = 0x5C132EB6
PKC_SCM_R15[4] = 0x6BEEFDCA
PKC_SCM_R15[5] = 0xA32C4EE4
PKC_SCM_R15[6] = 0x08509972
PKC_SCM_R15[7] = 0x2168FA79
If any of the input variables is changed (by e.g. flipping one bit)
then PKC_STATUS != 0  // Signature not valid
i.e. PKC_STATUS.9 = 1 // Signature not valid
```

7 Timer Module

7.1 Introduction

The timer module contains two independent 16-bit general purpose timers. The two timers are denoted timer 0 and timer 1.

7.2 Overview

The timer module offers the following set of features:

Feature List

- Two independent 16-bit up counting timers.
- Four selectable clock sources per timer.
- Individually configurable prescaler.
- Reload capability on overflow.
- Interrupt capability.

7.3 Functional Description

Figure 7-1 shows an overview block diagram of the two timer implementations.

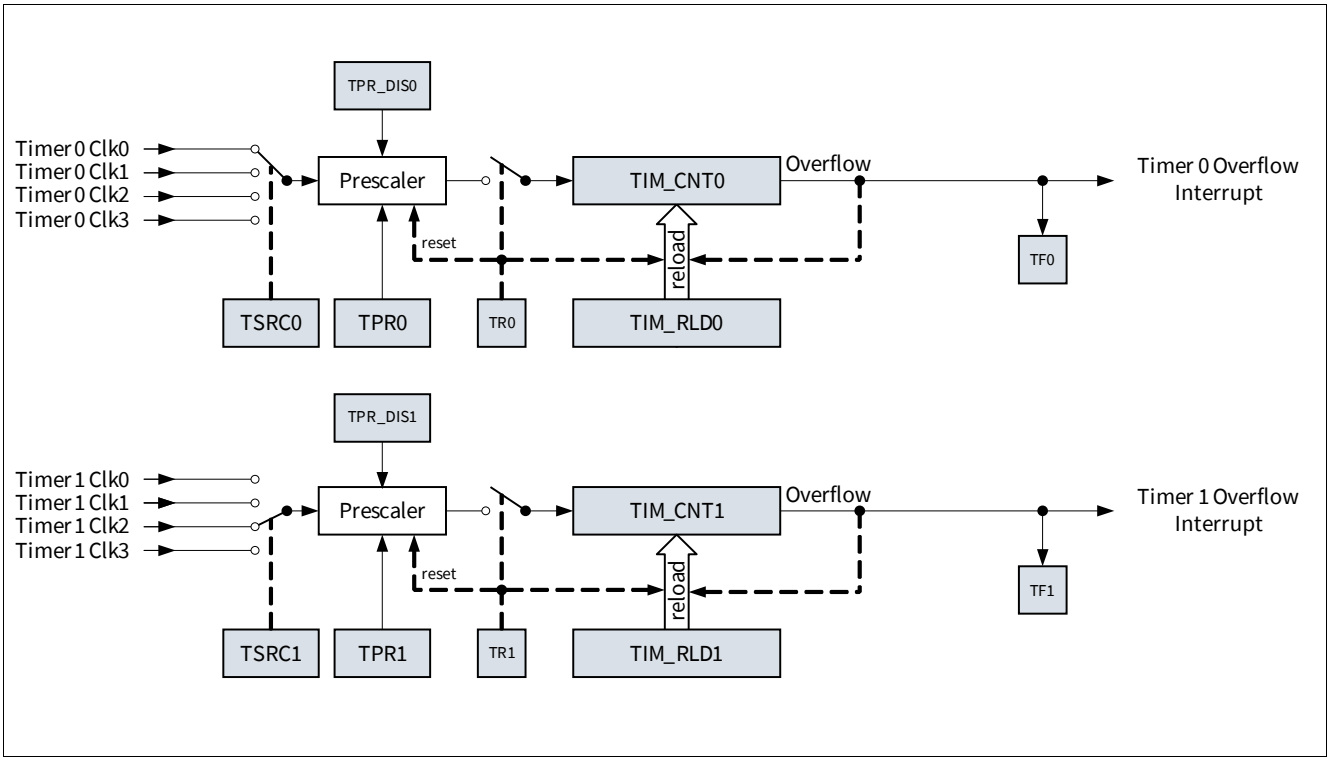


Figure 7-1 Timer Overview

7.3.1 Timer Clock Selection

Each timer provides up to four clock inputs. The required clock source can be selected independently for each timer by setting the TSRC bits in the timer control register **TIM_CTRL** appropriately. The respective timer must be stopped before the clock source is changed (otherwise, the result may be undefined), which is done by clearing **TIM_CTRL**.TR0 or TR1.

Table 7-1 Clock Sources for Timer 0[1)]

Source Input	Name and Description	Available in Sleep Mode	TSRC0 Value
Timer Clk0	HSM System clock (100 / 50 / 25 MHz[2)])	Yes	00_B
Timer Clk1	HSM System clock (100 / 50 / 25 MHz[2)])	No	01_B
Timer Clk2	-	-	10_B
Timer Clk3	-	-	11_B

1) Empty fields indicate “no clock source connected”

2) The frequency depends on the setting of the clock divider **CLKCTRL.CLKDIV**

Table 7-2 Clock Sources for Timer 1[1)]

Source Input	Name and Description	Available in Sleep Mode	TSRC1 Value
Timer Clk0	HSM System clock (100 / 50 / 25 MHz[2)])	Yes	00_B
Timer Clk1	HSM System clock (100 / 50 / 25 MHz[2)])	No	01_B
Timer Clk2	Overflow of Timer 0. This clock source is used to construct a 32-bit timer. Note, that the prescaler of timer 1 should be disabled in this case.	Yes	10_B
Timer Clk3	-	-	11_B

1) Empty fields indicate “no clock source connected”

2) The frequency depends on the setting of the clock divider **CLKCTRL.CLKDIV**

7.3.1.1 Timer Clocks in Sleep and Halt Mode

Each timer gets two clock source for selection, one running during sleep mode and another one being stopped during sleep mode. If the clock source running during sleep is selected, a timer interrupt can be used to wake-up the system.

In debug or halt mode the timers will be stopped regardless of the selected clock source.

7.3.2 Prescaler

The frequency of the selected input clock can be divided by a programmable prescaler factor. The TPR bits in the timer configuration register **TIM_CFG** determine whether the input clock will be divided by 4, 16, 64 or 512. A prescaler factor of ’n’ means that the counter register will be incremented by every n^{th} input clock cycle. If no prescaler is needed, it may be disabled using the TPR_DIS bit (which effectively gives a prescaler factor of 1).

The timer must be stopped before modifying the prescaler factor. Attempts to alter the prescaler factor while the timer is running will be ignored – the TPR value will be changed, but the current prescaler factor will be

retained (which implies that the newly set prescaler value will be read back even though the timer is still running on the old factor). The prescaler is reset when the timer is started. This means that a change of the prescaler factor while the timer is running will become effective when that timer is stopped and started again.

7.3.3 Timer Reload

The key elements of the timer module are the 16-bit counter registers **TIM_CNT0** and **TIM_CNT1** and the 16-bit timer reload registers **TIM_RLD0** and **TIM_RLD1** containing the initial value for the timer counter registers. When a timer is started by setting bit TR in the timer control register **TIM_CTRL**, i.e. with the transition “timer stopped” to “timer start”, the content of the timer reload register will be copied into the counter register. The timer then starts counting upwards using the prescaled clock source. When the counter value overflows from $FFFF_H$ to 0000_H, an overflow event will be generated. This event sets the overflow flag TF and may cause an interrupt. Furthermore, the counter register will be initialized again with the value of the timer reload register **TIM_RLD0** or **TIM_RLD1** and the timer continues counting.

It is recommended to change the reload value only when the timer is stopped. However, it is possible to write a new reload value at any time, but for proper functionality it must be ensured that the reload value remains stable at the point in time when the reload takes place. Therefore modifications of the reload register content have to be done a sufficient period of time before or after the timer overflow event. If the reload value is changed at the time of the reload the behavior will be undefined. A newly written reload value always becomes effective for the timer counter at the next overflow condition.

7.3.4 Timer Overflow and Interrupts

The overflow flag **TIM_CTRL**.TF is set by hardware when an overflow event occurs and stays set until it is cleared by software. To modify the status of the overflow flag by software, bit **TIM_CTRL**.UNLOCK_TF has to be set during the write access.

Independently of the overflow flag the timer generates an interrupt request each time an overflow event occurs.

7.4 Timer Register Description

A number of special function registers is implemented to control the timer operation, as listed in **Table 7-4**.

Table 7-3 Register Address Space

Module	Base Address	End Address	Note
TIM	EC00 0000_H	EC00 $00FF_H$	Timer module

Table 7-4 Register Overview

Register Short Name	Register Long Name	Offset Address	Reset Value
Timer Register Description, Timer Registers			
TIM_CTRL	Timer Control Register	00_H	0000_H
TIM_CFG	Timer Configuration Register	04_H	0000_H
TIM_CNT0	Timer Counter 0 Register	10_H	$XXXX_H$
TIM_CNT1	Timer Counter 1 Register	14_H	$XXXX_H$
TIM_RLD0	Timer Reload 0 Register	20_H	$XXXX_H$
TIM_RLD1	Timer Reload 1 Register	24_H	$XXXX_H$

The registers are addressed wordwise.

7.4.1 Timer Registers

These special function registers control the timer module's functions and present status.

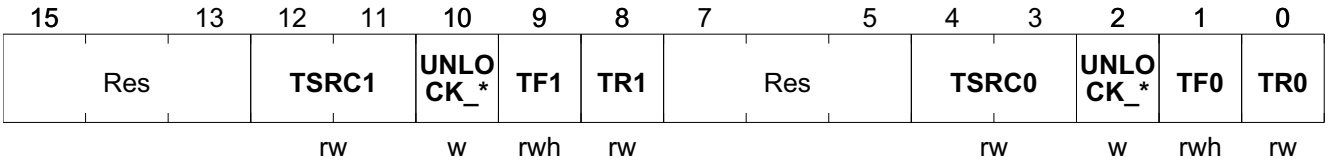
Timer Control Register

This register controls the clock sources, overflow flag and provides status for two timers.

Notes

1. *Start and stop requests require that the timer is provided with a running clock. If this is not ensured in a particular situation, the TR can be read to obtain the actual status.*
2. *If a TR bit is cleared and an overflow event from the corresponding timer is already in the process of synchronization from the timer to the system clock domain it is possible that the TF flag is set even a short time after the timer has been requested to stop.*

TIM_CTRL	Offset	Reset Value
Timer Control Register	00_H	0000_H



Field	Bits	Type	Description
TSRC1	12:11	rw	**Timer 1 Clock Source** Selects the clock source of timer 1 00_B **T1CLK0**, Timer 1 clock source 0. 01_B **T1CLK1**, Timer 1 clock source 1. 10_B **T1CLK2**, Timer 1 clock source 2. 11_B **T1CLK3**, Timer 1 clock source 3.
UNLOCK_TF1	10	w	**Unlock TF1** On a write access this bit controls the access to the TF1 bit. Reading returns always '0'. 0_B **T1NC**, write protection for TF1 is enabled 1_B **T1WE**, write enable for TF1 is set
TF1	9	rwh	**Timer 1 Overflow Flag** This bit can be set by software and hardware. Software can only write this bit if **UNLOCK_TF1** is set as well. 0_B **T1NO**, No overflow occurred / clear overflow 1_B **T1OV**, Overflow occurred / set overflow In case of simultaneous changes, the priority is software over hardware.
TR1	8	rw	**Timer 1 Run** 0_B **T1S**, Timer is off / stop timer 1_B **T1R**, Timer is running / start timer

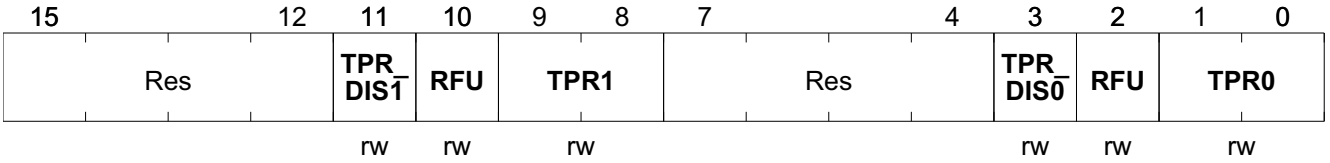
Field	Bits	Type	Description
TSRC0	4:3	rw	**Timer 0 Clock Source** Selects the clock source of timer 0 00_B **T0CLK0**, Timer 0 clock source 0. 01_B **T0CLK1**, Timer 0 clock source 1. 10_B **T0CLK2**, Timer 0 clock source 2. 11_B **T0CLK3**, Timer 0 clock source 3.
UNLOCK_TF0	2	w	**Unlock TF0** On a write access this bit controls the access to the TF0 bit. Reading returns always '0'. 0_B **T0NC**, write protection for TF0 is enabled 1_B **T0WE**, write enable for TF0 is set
TF0	1	rwh	**Timer 0 Overflow Flag** This bit can be set by software and hardware. Software can only write this bit if **UNLOCK_TF0** is set as well. 0_B **T0NO**, No overflow occurred / clear overflow 1_B **T0OV**, Overflow occurred / set overflow In case of simultaneous changes, the priority is software over hardware.
TR0	0	rw	**Timer 0 Run** 0_B **T0S**, Timer is off / stop timer 1_B **T0R**, Timer is running / start timer

Timer Configuration Register

This register controls the external reload and prescaler behavior.

Note: The prescaler factors can only be changed if the corresponding timer is stopped.

TIM_CFG	Offset	Reset Value
Timer Configuration Register	04_H	0000_H



Field	Bits	Type	Description
TPR_DIS1	11	rw	**Timer 1 Prescaler Disable** 0_B **T1PENB**, prescaler for timer 1 is enabled 1_B **T1PDIS**, prescaler for timer 1 is disabled
RFU	10	rw	**Reserved for Future Use** The bit shall be written as “0”
TPR1	9:8	rw	**Timer 1 Prescaler Adjust** The prescaler factor definition can be found in **Table 7-5**. If bit TPR_DIS1 is set, these bits are don’t care.
TPR_DIS0	3	rw	**Timer 0 Prescaler Disable** 0_B **T0PENB**, prescaler for timer 0 is enabled 1_B **T0PDIS**, prescaler for timer 0 is disabled
RFU	2	rw	**Reserved for Future Use** The bit shall be written as “0”
TPR0	1:0	rw	**Timer 0 Prescaler Adjust** The prescaler factor definition can be found in **Table 7-5**. If bit TPR_DIS0 is set, these bits are don’t care.

Prescaler Adjustment

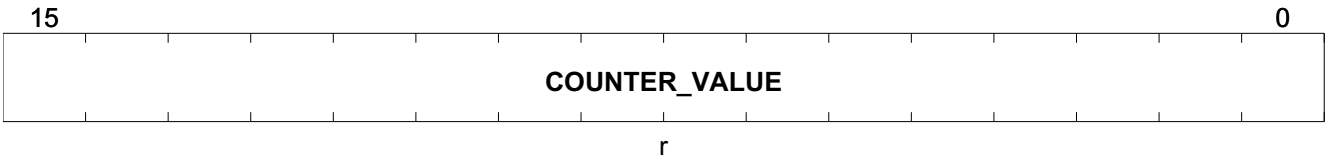
Table 7-5 Prescaler Adjustment

Prescaler Division Factor	TPR_DISx	TPRx
1 (prescaler is disabled)	1_B	xx_B
4	0_B	00_B
16	0_B	01_B
64	0_B	10_B
512	0_B	11_B

Timer Counter 0 Register

This register holds the current counter value of timer 0. The counter value can be modified by software through **TIM_RLD0**, see **Section 7.3.3**.

TIM_CNT0	Offset	Reset Value
Timer Counter 0 Register	10_H	$XXXX_H$

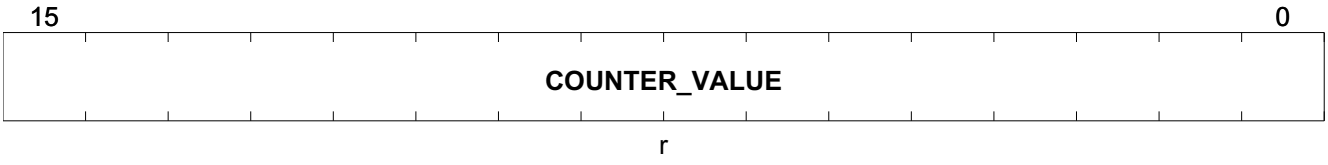


Field	Bits	Type	Description
COUNTER_VALUE	15:0	r	**Timer 0 Counter Value** Current content of counter register for timer 0.

Timer Counter 1 Register

This register holds the current counter value of timer 1. The counter value can be modified by software through **TIM_RLD1**, see **Section 7.3.3**.

TIM_CNT1	Offset	Reset Value
Timer Counter 1 Register	14_H	$XXXX_H$

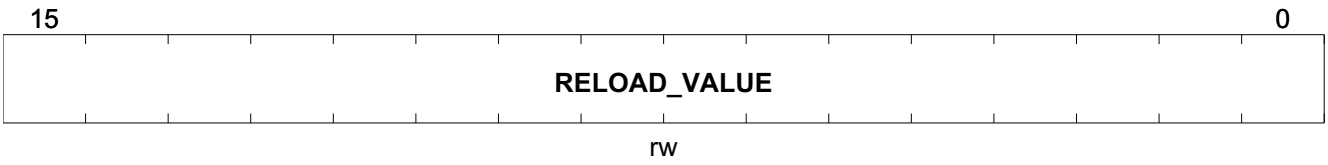


Field	Bits	Type	Description
COUNTER_VALUE	15:0	r	**Timer 1 Counter Value** Current content of counter register for timer 1.

Timer Reload 0 Register

Updating the reload register does not affect the counter register immediately. The counter register is automatically loaded with the contents of the reload register when one of the conditions described in **Section 7.3.3** occurs.

TIM_RLD0	Offset	Reset Value
Timer Reload 0 Register	20_H	$XXXX_H$

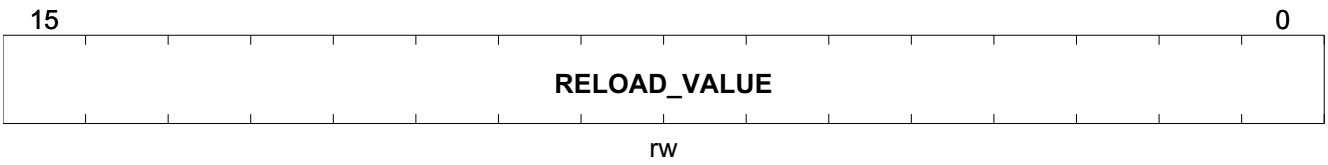


Field	Bits	Type	Description
RELOAD_VALUE	15:0	rw	**Timer 0 Reload Value** Reload Value for timer 0.

Timer Reload 1 Register

Updating the reload register does not affect the counter register immediately. The counter register is automatically loaded with the contents of the reload register when one of the conditions described in **Section 7.3.3** occurs.

TIM_RLD1	Offset	Reset Value
Timer Reload 1 Register	24_H	$XXXX_H$



Field	Bits	Type	Description
RELOAD_VALUE	15:0	rw	**Timer 1 Reload Value** Reload Value for timer 1.

7.5 Programming Model

The timers can be used for various applications:

- single shot event triggered by software
- periodic multi-shot events
- software triggered watchdog

For all of the above applications the configuration of the timer resolution and period is done in an identical way.

7.5.1 Timer Resolution and Period

The resolution of a timer in clock cycles of the input clock matches the prescaler factor $n_{prescaler}$. The prescaler factor can be set to one with bit **TIM_CFG.TPR_DIS0** or **TIM_CFG.TPR_DIS1** respectively (this effectively disables the prescaler). Note that the prescaler value as well as the prescaler disable bits should only be written when the respective timer is not running.

Assuming a stable frequency of the input clock, the resolution of the timer is:

$$t_{res}[\mu s] = \frac{n_{prescaler}}{f[MHz]} \tag{7.1}$$

The timer period in clock cycles, i.e. the number of cycles of the selected input clock between two overflow events when the timer is running without external reloads is:

$$n_{period} = n_{prescaler} \cdot (65536 - RELOAD\_VALUE) \tag{7.2}$$

With a stable input clock frequency the time between two overflow events when the timer is running without external reloads is:

$$t_{period}[\mu s] = t_{res}[\mu s] \cdot (65536 - RELOAD\_VALUE) = \frac{n_{prescaler} \cdot (65536 - RELOAD\_VALUE)}{f[MHz]} \tag{7.3}$$

Note: For some timer periods the values of the prescaler factor and the reload value are not well defined, i.e. different settings of $n_{prescaler}$ and RELOAD_VALUE produce the same timer period.

7.5.2 Polling vs. Interrupt Handling

A timer overflow event can be handled by polling the timer overflow flag or by an interrupt service routine.

Polling mode

In polling mode the timer overflow flag TF has to be checked periodically by software. If the flag is set a timer overflow event has occurred since the last time the flag was cleared (by software). To detect a further overflow event TF has to be cleared first.

To clear an overflow flag the value 0_B has to be written to TF while 1_B is written to the corresponding unlock bit UNLOCK_TF in the same write access.

Interrupt mode

In interrupt mode the timer overflow event triggers an interrupt request. If the corresponding interrupt is configured and enabled an interrupt service routine is executed in case of an overflow event.

The timer itself is prepared for interrupt mode and needs no additional configuration.

Note that the timer overflow flag and timer overflow interrupt generation are independent, so there is no need to check or clear the timer overflow flag by the interrupt service routine. However, the overflow flag is only a valid indication that the timer has generated an interrupt if it has been cleared before (see **Polling mode** above).

It is recommended to configure and enable the interrupt before starting the timer.

7.5.3 Single Shot Event Triggered by Software

A timer is configured for a single overflow event by the following steps:

- Stop the timer by clearing the timer run bit TR if it is not already stopped. This is necessary since the prescaler factor can be changed only if the timer is not running. Also the reload value is loaded into the counter register only if the timer run bit is changed from 0_B to 1_B (or if an overflow occurs).
- Select the appropriate clock source by configuring TSRC.
- Configure the prescaler factor and reload value for the desired time according to **Section 7.5.1**.
- For interrupt mode only: configure and enable the timer overflow interrupt. An interrupt service routine has to be available.
- Clear the overflow flag TF if not already cleared, see **Section 7.5.2**. This is mandatory for polling mode and optional for interrupt mode.
- Start the timer by setting TR: the reload value is loaded into the counter register automatically.
- After detection of the overflow event by polling or by the interrupt service routine the timer has to be stopped by clearing TR.

7.5.4 Periodic Multi-shot Events

The configuration of a timer for periodic timer overflow events is almost identical to the software triggered single shot event configuration, see **Section 7.5.3**. Periodic timer overflow events are normally handled in interrupt mode.

- Stop the timer by clearing the timer run bit TR if it is not already stopped. This is necessary since the prescaler factor can be changed only if the timer is not running.
- Select the appropriate clock source by configuring TSRC.
- Configure the prescaler factor and reload value for the desired timer period according to **Section 7.5.1**.
- For interrupt mode only: configure and enable the timer overflow interrupt. An interrupt service routine has to be available.
- Clear the overflow flag TF if not already cleared, see **Section 7.5.2**. This is mandatory for polling mode and optional for interrupt mode.
- Start the timer by setting TR: the reload value is loaded into the counter register.

In contrast to the software triggered single shot event case the timer is not stopped after the first overflow event.

To avoid missing an overflow event the timer period must be longer than the worst case overflow handling time. This includes the time for entering the interrupt service routine, execution of the service routine, return from the service routine and runtime of all interrupts and exceptions with higher priority.

7.5.5 Software Triggered Watchdog

To use a timer as watchdog the same timer configuration as for a periodic multi-shot events can be used, see **Section 7.5.4**. The difference is that the timer counter value is loaded with a new value by software without generation of any overflow events. As the counter registers **TIM_CNT0** and **TIM_CNT1** are read only the counter value has to be changed via the reload register **TIM_RLD0** or **TIM_RLD1**:

- Stop the timer by clearing the timer run bit TR. This is necessary since the reload value is loaded into the counter register only in case of an overflow event or if the timer run bit is changed from 0_B to 1_B.
- Write the new counter value in **TIM_RLD0** or **TIM_RLD1** respectively. For calculation of the timer period, see **Section 7.5.1**.
- Start the timer again by setting TR, the reload value will be loaded into the counter register.

7.5.6 Cascading of Timers

Two timers can be cascaded to construct a 32-bit timer, whereas timer 0 represents the lower 16 bits and timer 1 represents the upper 16 bits. The following configuration must be set to construct a proper 32-bit timer:

- The overflow of timer 0 must be fed to the clock input of timer 1.
  Select clock source 2 for timer 1 **TIM_CTRL.TSRC1** = 10_B
- The prescaler for timer 1 must be disabled (**TIM_CFG.TPR_DIS1** = 1_B).
- The reload value for timer 0 must be set to 0 (**TIM_RLD0.RELOAD_VALUE** = 0000_H).
- The overflow interrupt for timer 0 must be disabled in the NVIC.

8 Watchdog Timer

8.1 Introduction

The timer module features a watchdog timer to monitor system operation for possible time-outs and to check for the correct order of operations.

8.2 Overview

The watchdog timer module offers the following set of features:

Feature List

- One 16-bit upcounting watchdog timer.
- Two selectable clock sources.
- Fixed prescaler.
- Checkpoint functionality.
- Separate watchdog time-out and checkpoint mismatch events.

8.3 Functional Description

Figure 8-1 shows an overview block diagram of the watchdog timer implementation. The grey boxes are registers or bit fields which can be read or written by software. The watchdog time-out and checkpoint mismatch outputs trigger an interrupt when asserted.

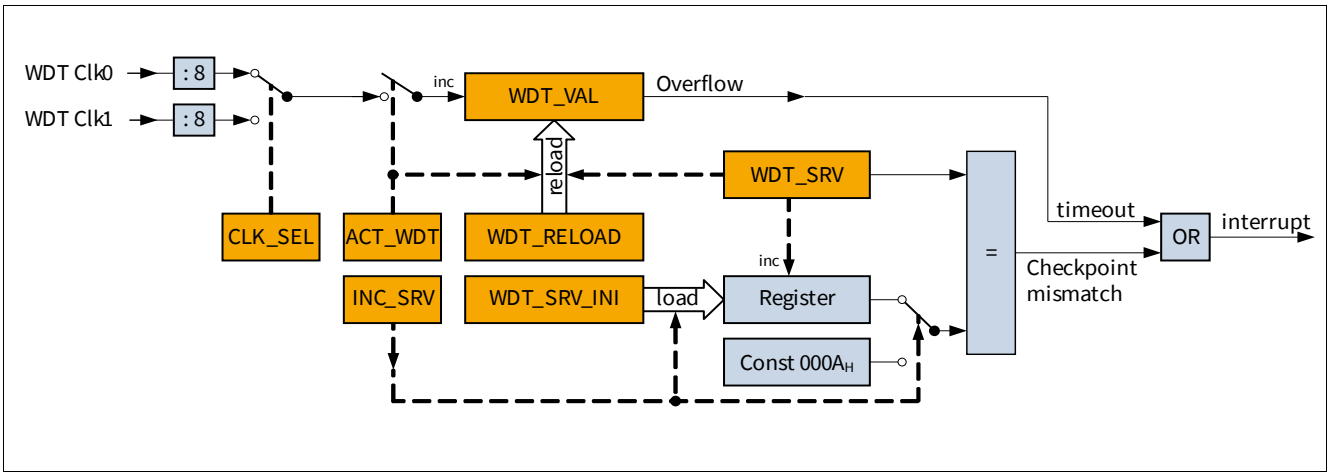


Figure 8-1 Watchdog timer overview

8.3.1 Clock Selection

The watchdog timer provides two clock inputs. The required clock source can be selected by setting the **CLK_SEL** bits in the watchdog timer control register **WDT_CTRL** appropriately.

Table 8-1 Clock Sources for Watchdog Timer Module

Name and Description	Available during Sleep Mode	CLK_SEL Value
HSM System clock (100 / 50 / 25 MHz[1]) divided by 128	Yes	0_B
HSM System clock (100 / 50 / 25 MHz[1]) divided by 128	No	1_B

1) The frequency depends on the setting of the clock divider **CLKCTRL.CLKDIV**

The frequency of the selected input clock is divided by another fixed factor of “8” (see also **Figure 8-1**). Effectively, the HSM system clock is divided by a factor of 1024 (128 * 8). Assuming a system frequency of 100 MHz the watchdog timer can realize time out periods up to ~ 670 ms.

8.3.1.1 Timer Clocks in Sleep and Halt Mode

The watchdog timer gets two clock source for selection, one running during sleep mode and another one being stopped during sleep mode. If the clock source running during sleep is selected, the watchdog timer can expire during sleep mode and can generate a non-maskable interrupt, which then wakes up the system.

In debug or halt mode the watchdog timer will be stopped regardless of the selected clock source.

8.3.2 Watchdog Timer Operation

The key elements of the watchdog timer are a counter register **WDT_VAL** and a watchdog timer reload register **WDT_RELOAD** containing the initial value for the counter register. When the watchdog timer is started by setting bit **WDT_CTRL**.ACT_WDT, i.e. by the transition from cleared to set, the content of the reload register will be copied to the counter register and the watchdog timer starts counting upwards using the prescaled clock source. The counter register **WDT_VAL** is also reset to the value of **WDT_RELOAD** every time the watchdog is “serviced” by writing to the service register **WDT_SRV**.

When the counter value switches from $FFFF_H$ to 0000_H an overflow occurs. This time-out of the watchdog causes an interrupt.

8.3.3 Watchdog Service

The watchdog timer is serviced by writing to the service register **WDT_SRV**. Two modes can be distinguished depending on the setting of control bit **WDT_CTRL**.INC_SRV:

- If bit INC_SRV is cleared, the watchdog timer has to be serviced by writing the constant value $000A_H$ to register **WDT_SRV**. Any other value results in a checkpoint mismatch event.
- If bit INC_SRV is set, the value which will be written to the service count register **WDT_SRV** must be incremented by one each time the register is written, otherwise an interrupt is triggered. The initial value of the service counter (contained in **WDT_SRV_INI**) has to be set by software before operation of the watchdog is started. For example, if the value of **WDT_SRV_INI** is $000A_H$, the first value to be written to **WDT_SRV** for serving the watchdog timer must be $000B_H$. The next value must be $000C_H$, and so on.

8.3.4 Service Counter Period

When the control bit **WDT_CTRL**.INC_SRV is set the value which should be written to the service count register **WDT_SRV** must be incremented each time the service register is accessed. The period respectively the length of the service counter can be configured by the control bits **WDT_CTRL**.PERIOD. With these control bits the period can be restricted to 2^{PERIOD}. **WDT_CTRL**.PERIOD should only be changed while control bit **WDT_CTRL**.INC_SRV is cleared. **WDT_SRV_INI** is not affected by **WDT_CTRL**.PERIOD. However, the value of **WDT_SRV_INI** must be smaller than 2^{PERIOD}. This has to be considered when changing the period value.

8.4 Watchdog Timer Register Description

A set of special function registers is implemented to control the watchdog timer operation, as listed in Table 8-3.

Table 8-2 Register Address Space

Module	Base Address	End Address	Note
WDT	EC00 0100_H	EC00 $01FF_H$	Watchdog module

Table 8-3 Register Overview

Register Short Name	Register Long Name	Offset Address	Reset Value
Watchdog Timer Register Description, Watchdog Timer Registers			
WDT_CTRL	Watchdog Timer Control Register	00_H	$C000_H$
WDT_VAL	Watchdog Timer Value Register	10_H	$XXXX_H$
WDT_RELOAD	Watchdog Timer Reload Register	14_H	$XXXX_H$
WDT_SRV	Watchdog Service Register	18_H	$000A_H$
WDT_SRV_INI	Watchdog Service Initialization Register	$1C_H$	$XXXX_H$
WDT_ACC_CTRL	Watchdog Timer Access Control Register	40_H	0003_H

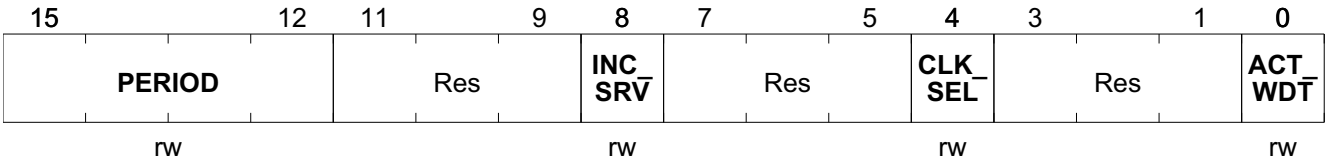
The registers are addressed wordwise.

8.4.1 Watchdog Timer Registers

These special function registers control the watchdog timer functions and present status.

Watchdog Timer Control Register

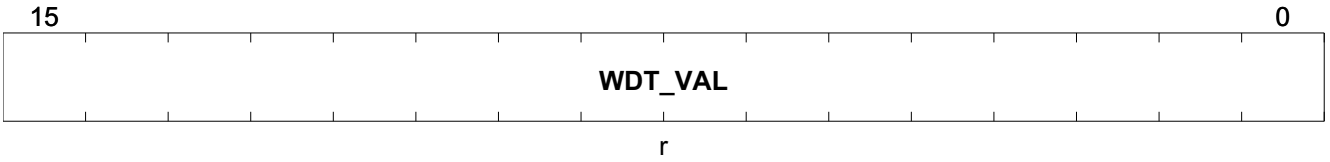
WDT_CTRL	Offset	Reset Value
Watchdog Timer Control Register	00_H	$C000_H$



Field	Bits	Type	Description
PERIOD	15:12	rw	**Watchdog Period** These bits define the period (= length of the service counter register).
INC_SRV	8	rw	**Increment Service** Increment service counter flag that determines how the watchdog timer is serviced. Writing an incorrect value will cause a checkpoint mismatch event. 0_B **ISN**, The value $000A_H$ must be used to service the watchdog. 1_B **IS1**, The value to service the watchdog must be incremented by one with each access.
CLK_SEL	4	rw	**Clock Select** Selects the clock source of the watchdog timer. Note, the watchdog timer must be stopped before changing the clock source. 0_B **WDTCLK0**, Watchdog timer clock source 0. 1_B **WDTCLK1**, Watchdog timer clock source 1.
ACT_WDT	0	rw	**Active Watchdog Timer** Whenever ACT_WDT is changed from 0 to 1, the value in **WDT_RELOAD** is loaded into **WDT_VAL**. 0_B **WTS**, Watchdog timer is stopped. 1_B **WTR**, Watchdog timer is running.

Watchdog Timer Value Register

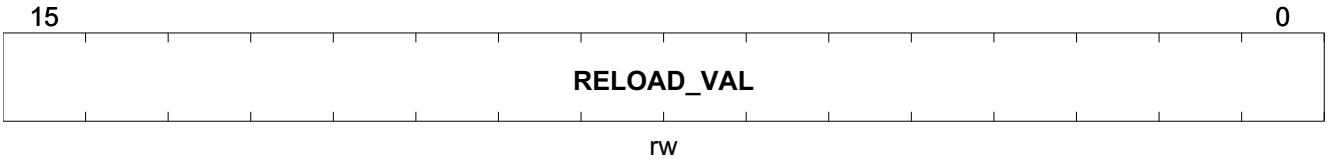
WDT_VAL	Offset	Reset Value
Watchdog Timer Value Register	10_H	$XXXX_H$



Field	Bits	Type	Description
WDT_VAL	15:0	r	**Watchdog Timer Value** Current value of the watchdog timer. The initial value is obtained from the reload register **WDT_RELOAD** each time the timer is serviced (provided that the watchdog timer is active with **WDT_CTRL**.ACT_WDT set).

Watchdog Timer Reload Register

WDT_RELOAD	Offset	Reset Value
Watchdog Timer Reload Register	14_H	$XXXX_H$



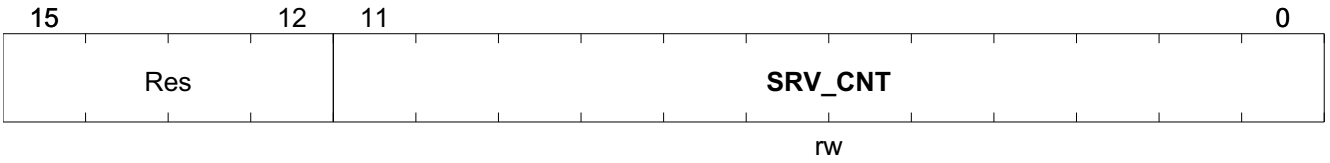
Field	Bits	Type	Description
RELOAD_VAL	15:0	rw	**Watchdog Timer Reload Value** Value to be loaded into the watchdog timer value register (**WDT_VAL**) each time the timer is (re)started.

Watchdog Service Register

This register is used to “service” the watchdog timer. An interrupt due to a checkpoint mismatch will be triggered if a wrong value is written to the **WDT_SRV** register.

Note: This register shall not be updated via read-increment-write operations, as this compromises the checkpoint functionality for which this register is intended.

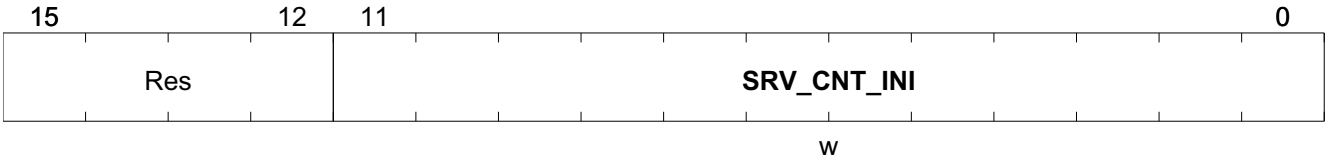
WDT_SRV	Offset	Reset Value
Watchdog Service Register	**18_H**	**$000A_H$**



Field	Bits	Type	Description
SRV_CNT	11:0	rw	**Service Counter** Watchdog service counter that needs to be written with an ascending value to reload the watchdog timer. For a detailed description see **“Watchdog Service” on Page 206**.

Watchdog Service Initialization Register

WDT_SRV_INI	Offset	Reset Value
Watchdog Service Initialization Register	$1C_H$	$XXXX_H$



Field	Bits	Type	Description
SRV_CNT_INI	11:0	w	**Service Counter Initialization Value** The initial value for the **WDT_SRV** register is defined by this SFR. If the **WDT_CTRL**.INC_SRV bit is '0', writing to this register changes the initial value of **WDT_SRV**, which will be used as soon as **WDT_CTRL**.INC_SRV is set. If **WDT_CTRL**.INC_SRV is '0', the service value is always $000A_H$. *Note: The initial value has to be written before enabling WDT_CTRL.INC_SRV.*

Watchdog Timer Access Control Register

The **WDT_ACC_CTRL** register determines the access rights to the watchdog timer registers. These are separated into two groups: control and data registers.

This register can only be written at privilege level. User level can only read **WDT_ACC_CTRL**.

The reset value of the register allows the user level to access the watchdog timer registers. Privilege access rights to watchdog data registers and to watchdog control registers can be assigned separately.

For instance, setting **WDT_ACC_CTRL** = 0002_H disables user level access to the control registers of the watchdog while the service register **WDT_SRV** can still be accessed.

WDT_ACC_CTRL	Offset	Reset Value
Watchdog Timer Access Control Register	**40_H**	**0003_H**

Field	Bits	Type	Description
DATA_ACC	1	rw	**Data Access** Privilege level required to get access to the data registers of the module (**WDT_SRV**): 0_B **PRIV**, Privilege level. 1_B **USER**, User level.
CTRL_ACC	0	rw	**Control Access** Privilege level required to get access to the control registers of the module (**WDT_CTRL**, **WDT_VAL**, **WDT_RELOAD**, **WDT_SRV_INI**): 0_B **PRIV**, Privilege level. 1_B **USER**, User level.

8.5 Programming Model

The watchdog timer supports two main applications:

- watchdog operation and
- checkpoint operation.

As the watchdog and checkpoint operations are independent of each other, different modes of operation are possible as listed in **Table 8-4**.

Table 8-4 Watchdog Checkpoint Operations

WDT_CTRL		Effect
ACT_WDT	INC_SRV	
0	0	Watchdog timer is not running, the value $000A_H$ must be used whenever writing to **WDT_SRV**.
1	0	Watchdog timer operation: the value $000A_H$ must be used whenever writing to **WDT_SRV**.
0	1	Checkpoint operation: the value written to **WDT_SRV** must be incremented by one each time.
1	1	Combined Watchdog and Checkpoint operation: the watchdog timer is running, the value written to **WDT_SRV** must be incremented by one each time.

8.5.1 Watchdog Timer Resolution and Period

As the prescaler of the watchdog is fixed the resolution of a watchdog timer is always 8 clock cycles of the selected input clock.

Assuming a stable frequency of the input clock the resolution of the timer is:

$$t_{res}[\mu s] = \frac{128 \cdot 8}{f[MHz]} \tag{8.1}$$

The time-out of the watchdog in clock cycles, i.e. the maximum number of cycles of the selected input clock between a watchdog service and watchdog timer overflow is:

$$n_{period} = 128 \cdot 8 \cdot (65536 - RELOAD\_VALUE) \tag{8.2}$$

or:

$$\begin{aligned} t_{period}[\mu s] &= t_{res}[\mu s] \cdot (65536 - RELOAD\_VALUE) \\ &= \frac{128 \cdot 8 \cdot (65536 - RELOAD\_VALUE)}{f[MHz]} \end{aligned} \tag{8.3}$$

8.5.2 Watchdog Operation

The basic application of a watchdog timer is to “service” it periodically such that a time-out of the watchdog will be avoided. The watchdog timer is serviced by writing to the service register **WDT_SRV** causing a restart

of the watchdog timer with the value of **WDT_RELOAD**. Serving the watchdog during execution of critical code monitors that the code is executed in a certain time.

The watchdog is configured for this operation by the following steps:

- Disable checkpoint operation by clearing bit INC_SRV.
- Stop the watchdog by clearing the timer run bit ACT_WDT if it is not already stopped. This is necessary since the reload value is loaded into the counter register only if the watchdog activation bit is changed from 0 to 1.
- Select the appropriate clock source by configuring CLK_SEL.
- Configure the reload value for the desired time according to **Section 8.5.1**.
- Start the watchdog operation by setting ACT_WDT.

The watchdog is serviced by writing the constant value $000A_H$ to register **WDT_SRV** before watchdog time-out.

8.5.3 Checkpoint Operation

For checkpoint operation the watchdog is serviced by writing ascending values into the service counter register **WDT_SRV**. This approach is used to check that the service points are handled in the correct order allowing software to make use of a hardware supported flow control.

- Disable watchdog operation by clearing bit ACT_WDT.
- Clear bit INC_SRV. This is necessary since the initial value for the service counter is only loaded from **WDT_SRV_INI** if INC_SRV changes from 0 to 1.
- Configure the initial value for the service counter by writing to **WDT_SRV_INI**.
- Enable checkpoint operation by setting INC_SRV: the first valid service value is the value of **WDT_SRV_INI** + 1.

The watchdog is serviced in checkpoint operation mode by writing ascending values to **WDT_SRV** starting with the value of **WDT_SRV_INI** + 1.

9 Bridge Module

This chapter describes the HSM's bridge module. It contains the following sections:

- **Introduction**.
- **Functional Description**.
- **Bridge Registers**.

9.1 Introduction

The main purpose of the bridge module is to connect the Hardware Security Module (HSM) subsystem to the host system and to enable communication between the two systems. All interaction between HSM and host runs through the bridge module.

The bridge module includes some Special Function Registers (SFRs) which allow synchronization of the communication between the host and HSM. This includes interrupt generation for both sides, see **Section 9.2.2**.

The HSM has principally full access to the host system, i.e. to memories and peripherals.

The host system has only restricted access to the HSM resources. While not in HSM debug mode or memory testing, access to all internal HSM memories and peripherals is not possible. Some of the HSM bridge registers are accessible for the host system, namely the communication registers (**Section 9.3.1**).

The bridge module allows control of the clock frequency for the HSM independent of the host system (**Section 9.2.3**) and provides a means to recognize and handle specific system errors like bus faults, access violations and ECC errors, see **Section 9.2.4**.

For debugging purposes the HSM internal resources can be made visible to a debugging module in the host system through a memory window, see **Section 9.2.5**. Both debugging of the host system and debugging of the HSM has to be authenticated by the HSM.

The bridge module maps up to 32 external interrupts to one interrupt node of the NVIC, see **Section 9.2.6**.

After system reset the local RAM and cache can be tested by the host system. After testing, this mode is irreversibly locked and is not available during normal operation, see **Section 9.2.7**.

Figure 9-1 on the next page gives an illustration of the bridge module architecture. Special function registers are colored in green, memory windows in light blue. Units that are accessible for the HSM only are colored yellow, the units accessible by the host only grey and units that both sides can access in orange.

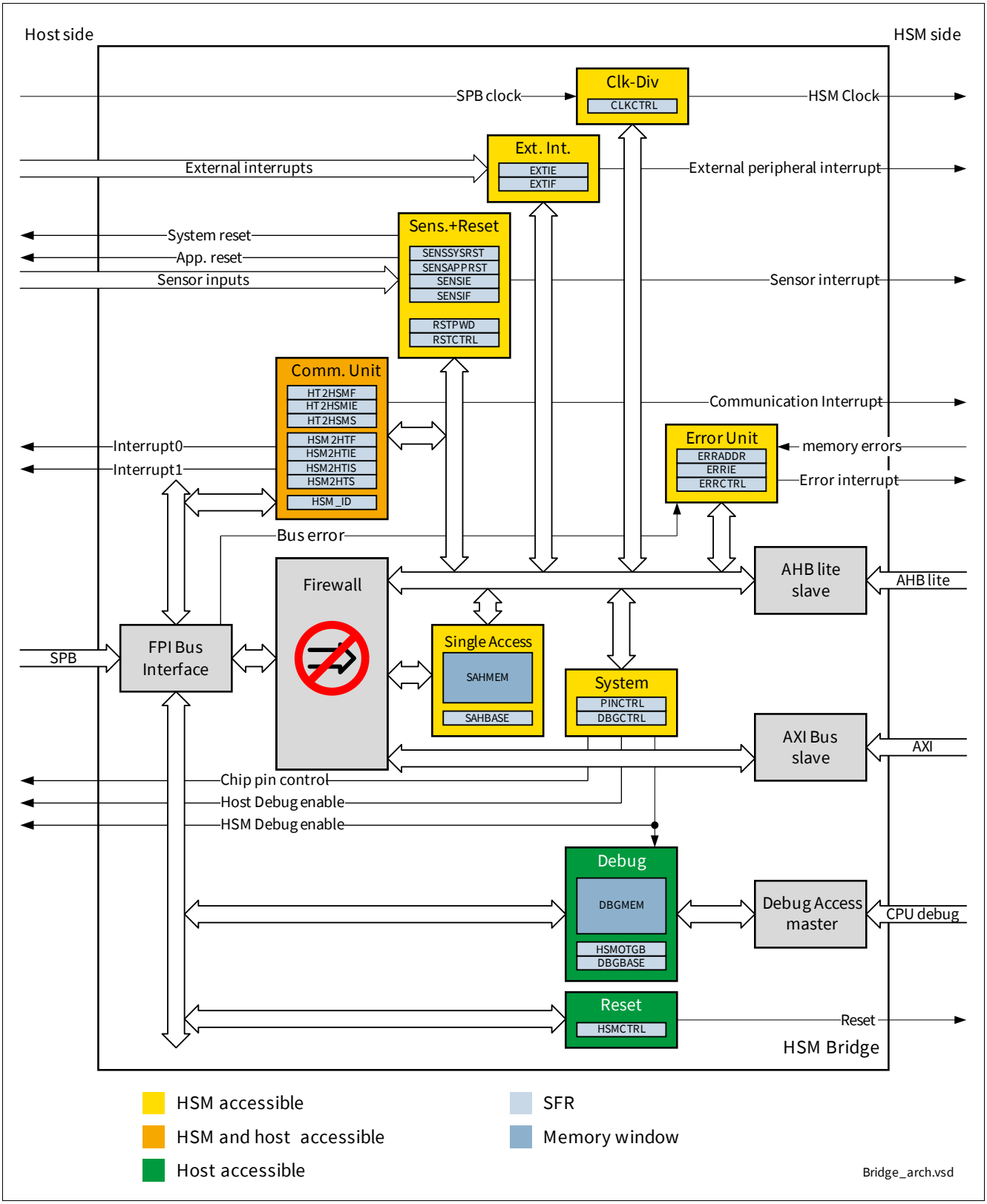


Figure 9-1 HSM bridge overview

9.2 Functional Description

This section defines the various bridging tasks of this module.

9.2.1 Bridging Functionality

One of the main purposes of the bridge module is to allow the HSM to access resources in the host system. Accesses in the other direction, i.e. host access to the HSM internal resources, are possible only under special conditions – in debug mode (see **Section 9.2.5**) or for testing after reset (see **Section 9.2.7**). However, host access to some of the bridge registers (partly read-only) is possible.

The bridge supports read and write operation in single access and burst mode access (4x32-bit bursts). Atomic read-modify-write operations are not supported. In order not to overload the host's System Peripheral Bus (SPB) and allow other host resources to use it, the bridge releases the bus request for at least one wait cycle after each burst transaction.

The address range 0000 0000_H - DFFF $FFFF_H$ is accessed through the cache in 16 byte granularity, see also **Section 2.1**.

The bridge allows also single byte, halfword and word access to this address range through the base register and the corresponding 64 KB memory window **SAHMEM** without involving the cache.

The address for an access through **SAHMEM** has two components: the upper 16 bits are defined by the contents of the address base register **SAHBASE**. The lower 16 bits are defined by the positive offset of the access address in the HSM address space to the base of the memory window (address of **SAHMEM**):

```
Hostaddress = [SAHBASE]+(HSMadress-SAHMEM)& 0xFFFF
```

The HSM address has to be in the memory window, i.e. in the range of SAHMEM and SAHMEM+0000 $FFFF_H$ to generate an access through the window.

Note: The addressing scheme of the single access to host memory window is equivalent to the debug access, see Section 9.2.5.

The single access to host memory window can be disabled by setting the upper 4 bits of **SAHBASE**.

Accessing the same host resource through the cache on the one hand and through **SAHMEM** on the other hand can lead to inconsistency.

Special care has to be taken to protect accesses through **SAHMEM**: the Core MPU can protect accesses according to HSM addresses. The translation mechanism of the single access to host memory window is not visible by the Core MPU. Therefore **SAHBASE** and **SAHMEM** have to be protected additionally, e.g. by setting a MPU descriptor into the host memory window.

9.2.2 Communication Unit

The resources of the host system can be accessed by the host CPUs and the HSM. Possible use cases are:

- Information exchange between host and HSM.
- Joint usage of host peripherals.

Information exchange can be done directly by the bridge registers in the communication unit and – for larger amounts of data – by a shared memory region in the host system. Beyond that the HSM can technically use all peripherals of the host system.

In any case, access to shared resources has to be synchronized. As there is no atomic read-modify-write support locks (e.g. semaphores and mutexes) cannot be implemented by memories, therefore the communication registers have to be used for this task.

With the exception of SFR **HSM2HTIS**, the registers for host to HSM communication and HSM to host communication are symmetrical. The status registers (**HSM2HTS** and **HT2HSMS**) can be used to transfer 32-bit of information from one side to the other. These registers are readable and writable from one side and read-only for the other side. A use case for these registers is that one side is writing status or event information to the corresponding register (e.g. new data in a shared memory region available, size of data) and the other side polls the register, waits for a change of the status and reacts in case of an event (read data from shared memory).

A more efficient method than polling is interrupt driven synchronization. The bridge supports one software-controlled interrupt for the HSM and two interrupts for the host system.

For simplicity the following description is for host interrupt generation: Register **HSM2HTF** provides 32 flags which can be set from the HSM and reset by the host. Setting/resetting is done by writing a '1' to the corresponding bit, writing '0' to a bit doesn't change the state of a flag. These flags can be used as interrupt request flags for the host. Each flag has a corresponding interrupt enable bit in register **HSM2HTIE**. If corresponding bits in **HSM2HTF** and **HSM2HTIE** are set (corresponding means same bit position in both registers) a host interrupt can be triggered. Note that **HSM2HTIE** is readable for both sides, but writable by the host only, i.e. the host controls which HSM interrupts are enabled. Each of the 32 bridge interrupts for the host can be individually mapped to one of two host interrupt lines by the corresponding bit in register **HSM2HTIS**. Again **HSM2HTIS** is controlled by the host. A cleared bit in **HSM2HTIS** means that the corresponding interrupt is mapped to host interrupt 0, a set bit maps it to host interrupt 1. See the host documentation **[10]** for host interrupt handling and the Service Request Nodes of the two HSM interrupts.

The functionality for HSM interrupt generation is quite similar with two exceptions:

- There is only one interrupt from the Communication Unit. Therefore an analog register to **HSM2HTIS** does not exist.
- Register **HT2HSMIE** is not readable by the host.

For details on interrupt handling on the HSM, see the ARM TRM **[2]**.

Both host and HSM use pulse triggered interrupts. For handling of interrupts first the interrupt request in the interrupt controller of the host or HSM has to be acknowledged (by clearing the interrupt pending flag) and afterwards the request flag in the bridge can be reset. For the HSM the pending flag is automatically cleared when entering the interrupt handler, see **[2]**.

9.2.3 Clock Control

The clock control unit contains only one SFR bit field that controls the clock frequency of the HSM system. After a reset, the bit field **CLKCTRL.CLKDIV** is cleared and the HSM runs with SPB clock frequency.

To reduce power consumption the HSM software can set this bits and reduce the HSM clock frequency to half or a quarter of the SPB clock frequency. The clocks of the host system are not affected. If more performance is needed the HSM software can reset **CLKCTRL.CLKDIV** to return to the maximum HSM clock frequency.

Register **CLKCTRL** is not accessible from the host system. If the host system needs to know the current HSM clock frequency the information has to be passed on by software communication.

9.2.4 Error and Event Handling

The bridge module contains two SFRs to allow handling of the following errors:

- Bus errors on the host system triggered by an HSM access.
- Correctable and uncorrectable bit errors while reading from HSM memories.

In contrast to bus errors on the host system, bus errors on the HSM internal busses may trigger a BusFault exception. Also access violations of the Core MPU a MemManage exception, see [2].

Each error source has a corresponding flag in register **ERRCTRL** and an interrupt enable bit in register **ERRIE**. If one or more of the flags are set an HSM bridge error interrupt may be triggered for the HSM CPU. The bridge error interrupt can be enabled/disabled in the NVIC by writing to the SETENA and/or CLRENA registers. Please refer to the ARM Cortex M3 Technical Reference Manual for details about interrupt and NVIC programming [2]. Additionally each of the error sources can be individually enabled/disabled to trigger an error interrupt by setting/clearing the corresponding bits in register **ERRIE**.

If a specific error flag is set, an error of the corresponding type has occurred since the flag was reset the last time. After handling of the error (by interrupt or polling) the flag of the handled source has to be reset by software. For the order of resetting interrupt flags the same rules as for the communication interrupts apply, see **Chapter 9.2.2**.

For bus errors the address causing the fault is stored in register **ERRADDR**. Note that the address of only the last occurring fault is stored, i.e. in case of multiple errors the addresses of previous faults are overwritten. Therefore it is recommended to omit bus errors in the handler code as these faults would overwrite the address of the fault that triggered the handler.

Note that a bus error may additionally trigger a BusFault exception. In this case the error can be handled by the BusFault handler (by clearing the corresponding flag in **ERRCTRL** and writing to CLEARPEND register to clear the pending flag in the NVIC).

For testing purposes it may be necessary to count the single and double bit errors in the boot ROM and local RAM. This can be done by either polling **ERRCTRL** after a RAM or ROM access or by activating the interrupt. Note that the number of errors may be too high if the counting routine is executed from a faulty memory region.

For the cache memories it is more difficult to determine the exact number of errors as it is probably impossible to access each memory address once by software.

Register **ERRCTRL** also contains the state of the external debugger. This can be used for compatibility to *SHE*, see [1]. Changing of the external debugger state does not trigger an interrupt.

9.2.5 Debug Access

This section describes how the ARM processor debug interface is made accessible to the host system's OCDS. See **Chapter 12** for details about the HSM debugging concept.

The HSM address space is mapped through a 64 KB window in the host system address space. A debugging module can access (read and write) the HSM internal resources through this window. These resources include

- CPU internal registers, (e.g. the debug registers)
- HSM internal peripherals
- HSM boot ROM
- Local RAM
- Bridge module registers.

The address for an HSM internal access has two components: the upper 16 bits are defined by the contents of the debug address base register **DBGBASE**. The lower 16 bits are defined by the positive offset of the access address in the host address space to the base of the memory window (address of **DBGMEM**):

```
HSMaddress = [DBGBASE]+(hostadress-DBGMEM)& 0xFFFF
```

Note that the host address has to be in the memory window, i.e. in the range of DBGMEM and DBGMEM+0000 $FFFF_H$ to generate an HSM debug access.

Debug access is only available if HSM debugging is enabled. This can be done from the HSM software only by setting **DBGCTRL.HSM**. In this case the debugging module also gains access to the HSM specific memory areas in the code and data flash and tracing of HSM data transfers on the host system busses is possible.

For security reasons the HSM debug access shall be protected properly and enabled after strong authentication only.

Additionally the HSM controls the debug access for the host system by register bit **DBGCTRL.HOST**.

The customer software is responsible for secure control of debug access for both host and HSM.

Technically the following scenarios are possible, but lead to a bus conflict:

- The host debug module accesses a host resource (memory or peripheral) through the HSM address space (including the single access to host memory window).
- The host debug module accesses an HSM internal memory through the cache. As a side effect this may lead to a writeback of a dirty cache block to a host memory.

The behavior in these scenarios is undefined and shall therefore be avoided by the debugger. For details, see **Section 12.3.6**.

9.2.6 External Interrupts

The bridge module has 32 inputs for external peripheral interrupts and the registers **EXTIF** and **EXTIE** to control interrupt generation.

The programming model is similar to **HT2HSMF** and **HT2HSMIE** in the communication unit, see **Chapter 9.2.2**.

The external interrupt request flags in register **EXTIF** are set by an interrupt request of the corresponding interrupt inputs. The flags can be cleared individually by software by writing a ‘1’ to the flag. Writing a ‘0’ does not change the state of a flag.

An interrupt request for the NVIC is triggered if one or more of the flags and the corresponding interrupt enable bits in register **EXTIE** are set. An interrupt will be performed if the external peripheral interrupt is enabled in the NVIC.

Table 9-1 below shows the mapping of external interrupt sources. The number corresponds to the bit number in **EXTIF** and **EXTIE**.

Table 9-1 External Interrupt mapping

Number	Source	SRN equivalent
0	GTM_TIM0_IRQ4	SRC_GTMTIM04
1	GTM_TIM0_IRQ5	SRC_GTMTIM05
2	GTM_TIM0_IRQ6	SRC_GTMTIM06
3	GTM_TIM0_IRQ7	SRC_GTMTIM07
4	GTM_ATOM0_IRQ2	SRC_GTMATOM02
5	GTM_ATOM0_IRQ3	SRC_GTMATOM03
6	GTM_MCS0_IRQ0	SRC_GTMMCS00
7	GTM_MCS0_IRQ1	SRC_GTMMCS01
8	DMA Error Service Request	SRC_DMAERR

Table 9-1 External Interrupt mapping (cont'd)

Number	Source	SRN equivalent
9	RCU	No related SRN, activated when Shutdown Trap request is sent by the RCU to the CPU(s).
10	CCU60 Service Request 0	SRC_CCU60SR0
11	CCU61 Service Request 0	SRC_CCU61SR0
12	PSI5	SRC_PSI50
13	MCMCAN0 Service Request 0	SRC_CAN0INT0
14	MCMCAN0 Service Request 1	SRC_CAN0INT1
15	MCMCAN0 Service Request 2	SRC_CAN0INT2
16	MCMCAN0 Service Request 3	SRC_CAN0INT3
17	MCMCAN0 Service Request 4	SRC_CAN0INT4
18	MCMCAN0 Service Request 5	SRC_CAN0INT5
19	VADC Common Group 0 SR 0	SRC_VADCCG0SR0
20	VADC Common Group 0 SR1	SRC_VADCCG0SR1
21	VADC Common Group 1 SR 0	SRC_VADCCG1SR0
22	VADC Common Group 1 SR 1	SRC_VADCCG1SR1
23	DS ADC SRM3 Service Request	SRC_DSADCSRM3
24	DAM0 Ready	SRC_DAMRI0
25	DAM0 Limit	SRC_DAMLI0
26	DAM1 Ready	SRC_DAMRI1
27	DAM1 Limit	SRC_DAMLI1
28	Ethernet	SRC_ETH
29	PMU0 End of busy (CPU interface)	SRC_PMUHOST
30	PMU0 End of busy (HSM interface)	no related SRN, direct connection to PMU0
31	CPU	SRC_GPSR00 (only triggering via INT_SRB0.0 is signaled to HSM)

9.2.7 HSM Reset

After chip reset, the RAMs of the HSM can be tested and configured by the host system. During this test the HSM system is held in a reset state, i.e. the HSM CPU is not running.

After testing and configuring the HSM RAMs the HSM is released by the host startup firmware by setting **HSMCTRL.EOMBT** if the HSM feature is activated. After that the HSM system leaves the reset state and starts running.

For more details about memory testing see the corresponding chapters of **[10]**.

9.2.8 Control of Pin Outputs

The HSM is able to override the normal function of two pins and control the output of these pins. For this functionality register **PINCTRL** provides two bits for each pin.

While PINCTRL.OENx (x=0..1) is cleared the corresponding pin has its normal behavior. If OENx is set the pin is configured as output and the value of the corresponding bit PINCTRL.VALx is driven on the pin. To read the pin inputs the corresponding registers in the host system have to be accessed.

After HSM reset **PINCTRL** can be modified arbitrarily until the register is locked by setting bit **LOCK**: when LOCK is set the register is write-protected, any further write access is ignored until the next reset. The register holds the value of the last successful write access.

Note: Control of the Pin outputs by the above feature can be disabled by host configuration, see [10].

9.2.9 Sensor Interrupts and Chip Resets

The HSM provides 10 inputs of sensors located in the host system.

A flag in register **SENSIF** is set if the corresponding sensors signals that the HSM is operated outside the range specified by the sensor. The flags can be cleared individually by software by writing a '1' to the flag. Writing a '0' does not change the state of a flag.

Note: The temperature sensor sets the corresponding bits in SENSIF only if a 0 to 1 transition of bit DTSCON.LLU or DTDCON.UOF occurs, see [10].

Each set flag in **SENSIF** can trigger one of the following actions:

- Sensor interrupt: if the corresponding enable bit in **SENSIE** is set, a set sensor flag can trigger a sensor interrupt. All sensor interrupts share the same interrupt node of the NVIC, see **Section 2.3.1**. For handling of interrupts first the interrupt request in the NVIC has to be acknowledged (by clearing the request flag) and afterwards the request flag in the bridge can be reset.
- Application reset: if the corresponding enable bit in **SENSAPPRST** is set, a set sensor flag can trigger an application reset of the chip, see, *AURIX™ Target Specification* **[10]**.
- System reset: if the corresponding enable bit in **SENSSYSRST** is set, a set sensor flag can trigger a system reset of the chip, see, *AURIX™ Target Specification* **[10]**.

Note: The system reset trigger has the highest priority, followed by the application reset trigger and sensor interrupt in that order. In the case that several actions are triggered at the same time, the action with the highest priority will be executed while the lower priority actions will be masked – e.g., if a sensor interrupt and an application reset are triggered simultaneously, the interrupt will not be detected.

Table 9-2 below shows the mapping of external interrupt sources. The number corresponds to the bit number in **SENSIF**, **SENSIE**, **SENSAPPRST**and **SENSSYSRST**.

Table 9-2 Sensor mapping

Number	Source
0	5.0 V undervoltage
1	3.3 V undervoltage
2	1.2 V undervoltage
3	5.0 V overvoltage
4	3.3 V overvoltage
5	1.2 V overvoltage
6	DTS over-temperature
7	DTS under-temperature

Table 9-2 Sensor mapping (cont'd)

Number	Source
8	SPB (fast) over-clocking detection configured with SCU_CCUCONSM
9	SPB over- and under-clocking detection configured with SCU_CCUCON4

Additionally the HSM allows triggering of application and system reset by software: first the correct password has to be written to register **RSTPWD** (prevents unintentional changes of **RSTCTRL**). After that **RSTCTRL** can be written. for details see **Section 9.3.7**.

The source of a chip reset is stored in the SCU register SCU_RSTSTAT:

- if bit 26 is set the last reset was a system reset triggered by the HSM.
- if bit 27 is set the last reset was an application reset triggered by the HSM.

Note: Trigger of application and system reset by the HSM (either via a sensor or by software) can be disabled by host configuration, see [10].

9.3 Bridge Registers

Table 9-3 Register Address Space

Module	Base Address	End Address	Note
Bridge	F004 0000$_H$	F005 FFFF$_H$	

Table 9-4 Register Overview

Register Short Name	Register Long Name	Offset Address	Reset Value
Bridge Registers, Communication Unit			
HSM_ID	Module Identifier Register	008$_H$	00BE C002$_H$
HT2HSMF	Host to HSM Flag Register	020$_H$	0000 0000$_H$
HT2HSMIE	Host to HSM Interrupt Enable	024$_H$	0000 0000$_H$
HSM2HTF	HSM to Host Flag Register	028$_H$	0000 0000$_H$
HSM2HTIE	HSM to Host Interrupt Enable	02C$_H$	0000 0000$_H$
HSM2HTIS	HSM to Host Interrupt Select	030$_H$	0000 0000$_H$
HSM2HTS	HSM to Host Status	034$_H$	0000 0000$_H$
HT2HSMS	Host to HSM Status	038$_H$	0000 0000$_H$
Bridge Registers, Clock Divider			
CLKCTRL	Clock Control Register	040$_H$	0000 0000$_H$
Bridge Registers, System Control			
DBGCTRL	Debug Control Register	060$_H$	0000 0XXX$_H$
PINCTRL	Pin Control Register	064$_H$	0000 0000$_H$
Bridge Registers, Error Unit			
ERRCTRL	Error Control Register	080$_H$	0000 0000$_H$
ERRIE	Error Interrupt Enable Register	084$_H$	0000 0000$_H$
ERRADDR	Error Address Register	088$_H$	0000 0000$_H$
Bridge Registers, External Interrupts			
EXTIF	External Interrupt Flag Register	0A0$_H$	0000 0000$_H$
EXTIE	External Interrupt Enable	0A4$_H$	0000 0000$_H$
Bridge Registers, Single Access to Host Memories			
SAHBASE	Single Access to Host Base Address Register	0C0$_H$	F000 0000$_H$
SAHMEM	Single Access to Host Memory Window	10000$_H$ to 1FFFF$_H$	Undefined$_H$
Bridge Registers, Sensor Interrupts and Reset			
RSTCTRL	Reset Control Register	0E0$_H$	0000 0000$_H$
RSTPWD	Reset Password Register	0E4$_H$	0000 0000$_H$
SENSIF	Sensor Interrupt Flag Register	0F0$_H$	0000 0000$_H$
SENSIE	Sensor Interrupt Enable Register	0F4$_H$	0000 0000$_H$

Table 9-4 Register Overview (cont'd)

Register Short Name	Register Long Name	Offset Address	Reset Value
SENSAPPRST	Sensor Application Reset Enable Register	$0F8_H$	$0000\ 0000_H$
SENSSYSRST	Sensor System Reset Enable Register	$0FC_H$	$0000\ 0000_H$

The registers are addressed wordwise.

Table 9-5 Register Access Types

Type	Symbol	Description
no access	-	Register is not accessible, an exception is triggered.
write only	w	Register is write-only, the register reads as '0'.
read only	ro	Register is read-only, a write access is ignored
read/write	rw	Register is read and writable
read/reset	rr	Register is readable. Writing '1' to register bits reset them, writing '0' do not modify the corresponding bits.
read/set	rs	Register is readable. Writing '1' to register bits sets them, writing '0' do not modify the corresponding bits
read/lockable	rl	Register is readable. Register is also writable after reset until it is locked. After locking register is read only until reset if not otherwise noted.

Parts of the bridge registers can be accessed from both the HSM and the host. For this reason there are two independent access types for each register. The first symbol in the registers type description defines the HSM access rights, the second the host access rights. Access to unused addresses results in a bus error.

Registers and debug memory window can only be accessed 32-bit wise and 32-bit aligned from both sides. Using 8-bit, 16-bit or un-aligned accesses lead to a bus error.

9.3.1 Communication Unit

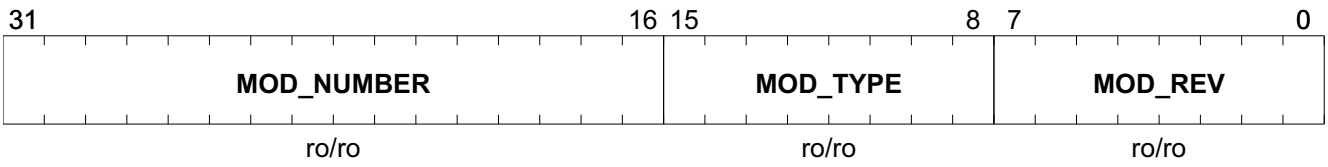
The following registers are accessible from both host and HSM system. The first symbol in the registers type description defines the HSM access rights, the second the host access rights.

Besides the Module Identifier Register the registers in this section are used to synchronize the communication between HSM and host system.

HSM_ID

The HSM Module Identifier register contains an identification number and revision number not only for the bridge, but for the whole HSM. Its main purpose is for identification in the host system, nevertheless it is also readable from the HSM.

HSM_ID	Offset	Reset Value
Module Identifier Register	008_H	$00BE\ C002_H$

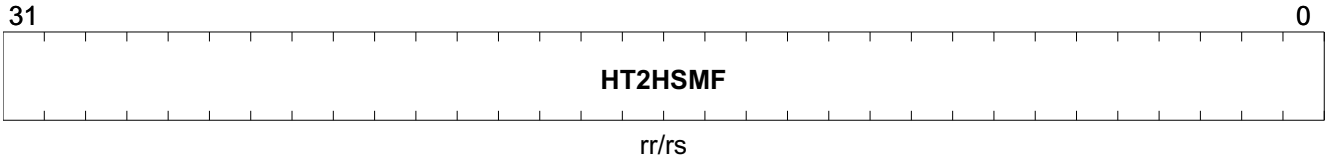


Field	Bits	Type	Description
MOD_NUMBER	31:16	ro/ro	**Module Number Value** This bit field defines a module identification number. The value for the HSM module is $00BE_H$.
MOD_TYPE	15:8	ro/ro	**Module Type** The bit field is set to $C0_H$ which defines the module as a 32-bit module.
MOD_REV	7:0	ro/ro	**Module Revision Number** This bit field defines the module revision number.

HT2HSMF

This register holds flags allowing the host to request an interrupt for the HSM.

HT2HSMF	Offset	Reset Value
Host to HSM Flag Register	020_H	$0000\ 0000_H$

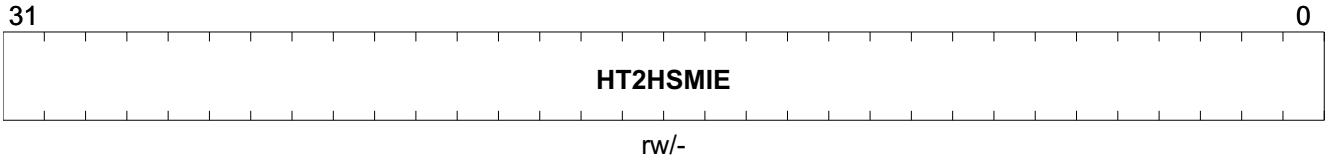


Field	Bits	Type	Description
HT2HSMF	31:0	rr/rs	**Host to HSM Flags** Each flag can be individually set by the host to generate an HSM interrupt request if the corresponding interrupt enable flag is set. The HSM can reset the flags individually. If a simultaneous set and reset occurs, the value of the flag is ‘1’.

HT2HSMIE

This register controls enabling HSM interrupts for each flag of **HT2HSMF**.

HT2HSMIE	Offset	Reset Value
Host to HSM Interrupt Enable	024_H	$0000\ 0000_H$

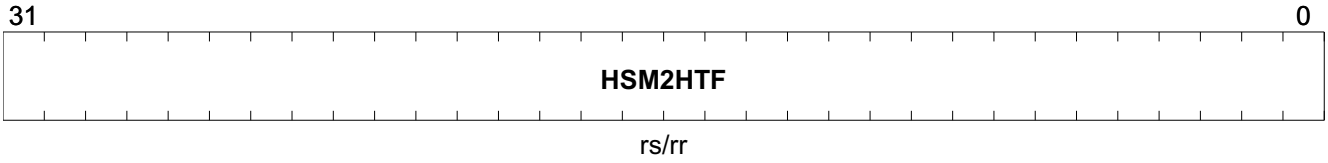


Field	Bits	Type	Description
HT2HSMIE	31:0	rw/-	**Host to HSM Interrupt Enable** Each set bit enables the interrupt for the corresponding flags in register **HT2HSMF**. If cleared bit disables the interrupt.

HSM2HTF

This register holds flags allowing the HSM to request an interrupt for the host.

HSM2HTF	Offset	Reset Value
HSM to Host Flag Register	028_H	$0000\ 0000_H$

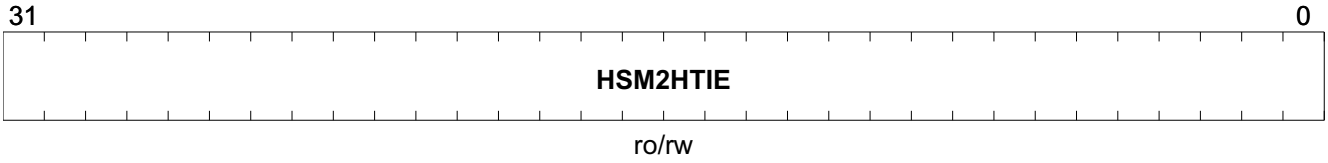


Field	Bits	Type	Description
HSM2HTF	31:0	rs/rr	**HSM to Host Flags** Each flag can be individually set by the HSM to generate an host interrupt request if the corresponding interrupt enable flag is set. The host can reset the flags individually. If a simultaneous set and reset occurs, the value of the flag is ‘1’.

HSM2HTIE

This register controls enabling host interrupts for each flag of **HSM2HTF**.

HSM2HTIE	Offset	Reset Value
HSM to Host Interrupt Enable	$02C_H$	$0000\ 0000_H$

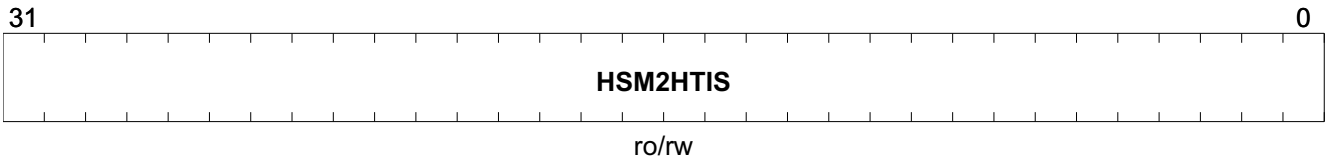


Field	Bits	Type	Description
HSM2HTIE	31:0	ro/rw	**HSM to Host Interrupt Enable** Each set bit enables the interrupt for the corresponding flags in register **HSM2HTF**. If cleared bit disables the interrupt.

HSM2HTIS

This register maps each individual interrupt flag/enable in registers **HSM2HTF** and **HSM2HTIE** to one of two host interrupts.

HSM2HTIS	Offset	Reset Value
HSM to Host Interrupt Select	030_H	$0000\ 0000_H$

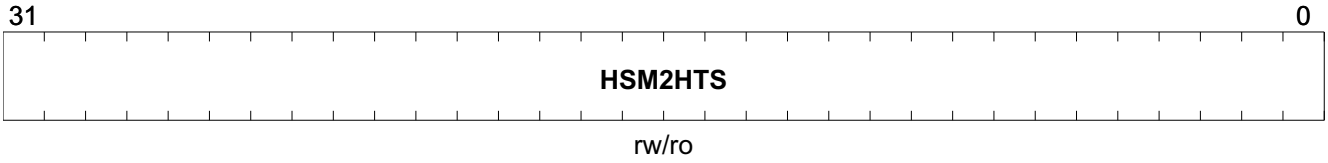


Field	Bits	Type	Description
HSM2HTIS	31:0	ro/rw	**HSM to Host Interrupt Select** If a bit is set the corresponding interrupt controlled by registers **HSM2HTF** and **HSM2HTIE** is mapped to host interrupt 1, otherwise to host interrupt 0.

HSM2HTS

This register holds information to signal the HSM internal status to the host.

HSM2HTS	Offset	Reset Value
HSM to Host Status	034_H	$0000\ 0000_H$

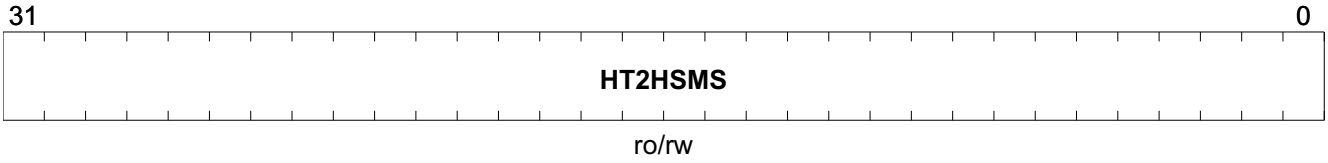


Field	Bits	Type	Description
HSM2HTS	31:0	rw/ro	**HSM to Host Status** 32 bits of information to signal HSM internal status to the host.

HT2HSMS

This register holds information to signal the host software status to the HSM .

HT2HSMS	Offset	Reset Value
Host to HSM Status	038_H	$0000\ 0000_H$



Field	Bits	Type	Description
HT2HSMS	31:0	ro/rw	**Host to HSM Status** 32 bits of information to signal the host software status to the HSM .

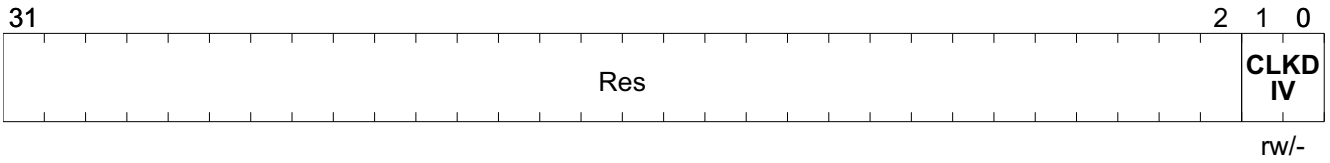
9.3.2 Clock Divider

The register in this section is accessible by the HSM only. Access by the host results in a bus error. Access types are valid for HSM access.

CLKCTRL

This register contains only one bitfield to control the clock frequency of the HSM subsystem.

CLKCTRL	Offset	Reset Value
Clock Control Register	040_H	$0000\ 0000_H$



Field	Bits	Type	Description
CLKDIV	1:0	rw/-	**Clock Divider** Controls the clock divider for the HSM subsystem 00_B **DIV1**, HSM runs with maximum possible clock frequency. 01_B **DIV2**, HSM runs with half of the maximum possible clock frequency. 10_B **DIV4**, HSM runs with a quarter of the maximum possible clock frequency. 11_B **RFU**, Reserved for other clock division factors. HSM runs with maximum possible clock frequency.

Note: In case a data transfer between the HSM and the host is ongoing while the setting of the clock divider is being changed, the change of the clock frequency becomes effective only after the data transfer has been finished.

9.3.3 System Control

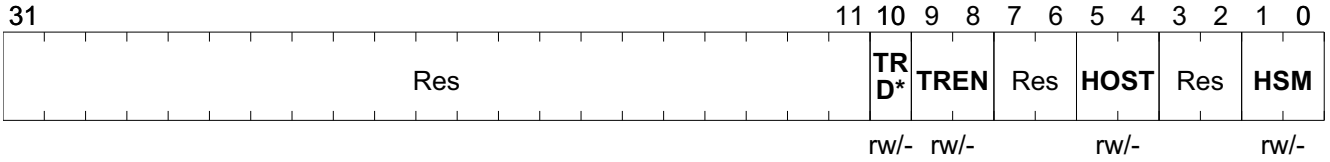
This section collects registers to control host resources like pins and debug access.

The registers in this section are accessible by the HSM only. Access by the host results in a bus error. Access types are valid for HSM access.

DBGCTRL

This register controls the debug and trace accesses for both the host system and the HSM (including debug memory window). For security reasons the debug access bits are encoded redundantly.

DBGCTRL	Offset	Reset Value
Debug Control Register	060_H	$0000\ 0XXX_H$

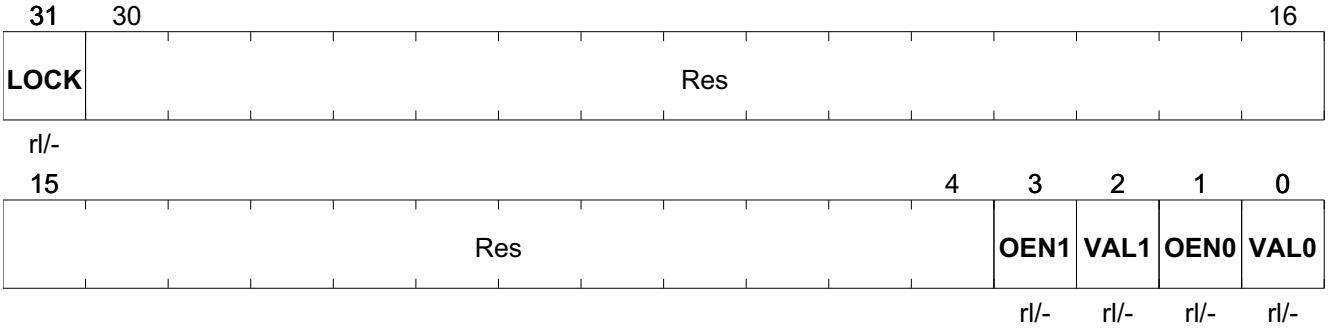


Field	Bits	Type	Description
TRDATA	10	rw/-	**Type of trace information** This information is only valid if tracing is enabled via **TREN**. 0_B **TRADDR**, Trace addresses only. 1_B **TRDATA**, Trace addresses and data.
TREN	9:8	rw/-	**Enables host side tracing of HSM accesses on SPB bus** 00_B **TRACEDIS0**, Tracing disabled. 01_B **TRACEDIS1**, Tracing disabled. 10_B **TRACEDIS2**, Tracing disabled. 11_B **TRACEEN**, Tracing enabled.
HOST	5:4	rw/-	**Debug control for host system** *Note: To enable host debugging, these bits must be set as described below and, in addition, the debug lock bit must be cleared, as explained in the Cerberus module description. See [10].* 00_B **HTDDIS0**, Host debug disabled. 01_B **HTDDIS1**, Host debug disabled. 10_B **HTDDIS2**, Host debug disabled. 11_B **HTDEN**, Host debug enabled.
HSM	1:0	rw/-	**Debug control for HSM subsystem and debug memory window** 00_B **HSMDDIS0**, HSM debug disabled. 01_B **HSMDDIS1**, HSM debug disabled. 10_B **HSMDDIS2**, HSM debug disabled. 11_B **HSMDEN**, HSM debug enabled.

PINCTRL

This register allows direct control of output pins of the chip. To read the level of the pins the corresponding registers in the host system have to be accessed. The register is writable after reset until it is locked by setting **PINCTRL.LOCK**.

PINCTRL	Offset	Reset Value
Pin Control Register	064_H	$0000\ 0000_H$



Field	Bits	Type	Description
LOCK	31	rl/-	**Pin control register lock** 0_B **UNLOCKED**, **PINCTRL** is not locked. all register bits can be written. 1_B **LOCKED**, Register is readonly until next reset. A write access will be ignored.
OEN1	3	rl/-	**Pin 1 output enable** 0_B **P1DIS**, Pin 1 output of HSM disabled, pin has normal functionality. 1_B **P1EN**, Pin 1 output of HSM enabled, the value of **VAL1** is driven.
VAL1	2	rl/-	**Pin 1 output value** The value is only driven if **OEN1** is set. 0_B **P1L**, low level. 1_B **P1H**, high level.
OEN0	1	rl/-	**Pin 0 output enable** 0_B **P0DIS**, Pin 0 output of HSM disabled, pin has normal functionality. 1_B **P0EN**, Pin 0 output of HSM enabled, the value of **VAL0** is driven.
VAL0	0	rl/-	**Pin 0 output value** The value is only driven if **OEN0** is set. 0_B **P0L**, low level. 1_B **P0H**, high level.

Note: The bits VAL0 and OEN0 control port pin HSM1, VAL1 and OEN1 control port pin HSM2. Direct control of these pins by the HSM can be disabled by host configuration.

9.3.4 Error Unit

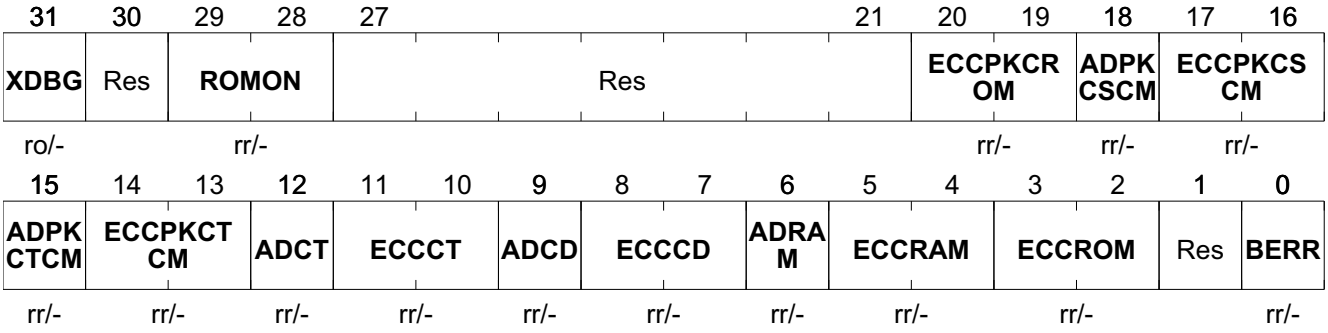
The registers in this section are accessible by the HSM only. Access by the host results in a bus error. Access types are valid for HSM access.

ERRCTRL

Control register for error handling. The flags are set by error conditions and have to be reset by software by writing a '1' to the corresponding bit. The register also holds the external debugger state and allows switching off the ROM after leaving the boot software. The ROM switch bits can only be cleared. For reasons of robustness the ROM switch bits are encoded redundantly.

Note: The HSM startup software must clear the error bits before use as they may be set during initialization of the memories.

ERRCTRL	Offset	Reset Value
Error Control Register	080_H	$0000\ 0000_H$



Field	Bits	Type	Description
XDBG	31	ro/-	**External Debugger** Shows the state of an external debugger. for compatibility to SHE standard. Cleared by application reset, stays set after activation. This status bit is not a source of the error interrupt. 0_B **XDDIS**, External debugger inactive. 1_B **XDEN**, External debugger active.
ROMON	29:28	rr/-	**ROM switch** 00_B **ROFF**, ROM is switched off. 01_B **RON1**, ROM is switched on. 10_B **RON2**, ROM is switched on. 11_B **RON3**, ROM is switched on.
ECCPKCROM	20:19	rr/-	**ECC errors of PKC ROM** 00_B **PKCROMNE**, No ECC error detected in PKC ROM. 01_B **PKCROMCE**, Correctable error detected in PKC ROM. 10_B **PKCROMUE**, Uncorrectable error detected in PKC ROM. 11_B **PKCROMCU**, Correctable and uncorrectable error detected in PKC ROM.

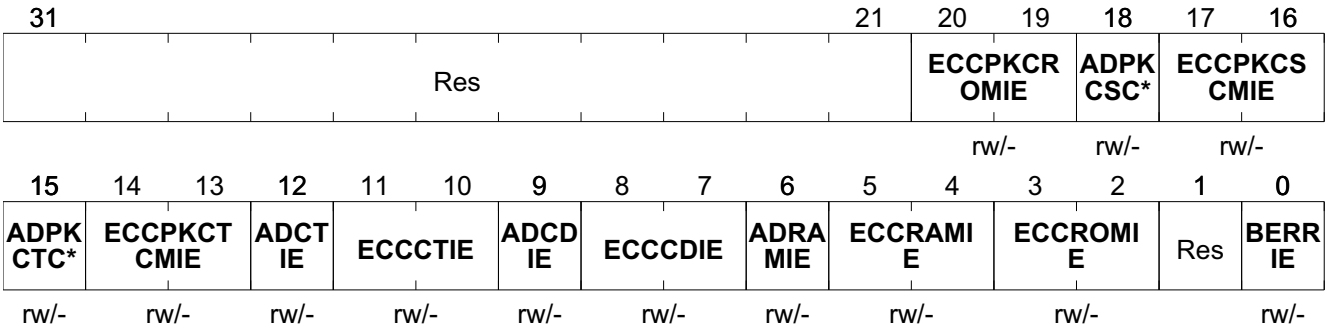
Field	Bits	Type	Description
ADPKCSCM	18	rr/-	**Address Error of PKC SCM** If set an address error has occured while the PKC engine accessed the SCM memory.
ECCPKCSCM	17:16	rr/-	**ECC errors of PKC SCM** 00_B **PKCSCMNE**, No ECC error detected in PKC SCM. 01_B **PKCSCMCE**, Correctable error detected in PKC SCM. 10_B **PKCSCMUE**, Uncorrectable error detected in PKC SCM. 11_B **PKCSCMCU**, Correctable and uncorrectable error detected in PKC SCM.
ADPKCTCM	15	rr/-	**Address Error of PKC TCM** If set an address error has occured while the PKC engine accessed the TCM memory.
ECCPKCTCM	14:13	rr/-	**ECC errors of PKC TCM** 00_B **PKCTCMNE**, No ECC error detected in PKC TCM. 01_B **PKCTCMCE**, Correctable error detected in PKC TCM. 10_B **PKCTCMUE**, Uncorrectable error detected in PKC TCM. 11_B **PKCTCMCU**, Correctable and uncorrectable error detected in PKC TCM.
ADCT	12	rr/-	**Address Error of cache tag field** If set an address error while accessing cache tag fields has occurred.
ECCCT	11:10	rr/-	**ECC errors of cache tag field** 00_B **CTNE**, No ECC error detected in cache tag field. 01_B **CTCE**, Correctable error detected in cache tag field. 10_B **CTUE**, Uncorrectable error detected in cache tag field. 11_B **CTCU**, Correctable and uncorrectable error detected in cache tag field.
ADCD	9	rr/-	**Address Error of cache data** If set an address error while accessing cache data has occurred.
ECCCD	8:7	rr/-	**ECC errors of cache data** 00_B **CDNE**, No ECC error detected in cache data. 01_B **CDCE**, Correctable error detected in cache data. 10_B **CDUE**, Uncorrectable error detected in cache data. 11_B **CDCU**, Correctable and uncorrectable error detected in cache data.
ADRAM	6	rr/-	**Address Error of Local RAM** If set an address error while accessing the local RAM has occurred.
ECCRAM	5:4	rr/-	**ECC errors of Local RAM** 00_B **RAMNE**, No ECC error detected in RAM. 01_B **RAMCE**, Correctable error detected in RAM. 10_B **RAMUE**, Uncorrectable error detected in RAM. 11_B **RAMCU**, Correctable and uncorrectable error detected in RAM.

Field	Bits	Type	Description
ECCROM	3:2	rr/-	**ECC errors of HSM boot ROM** 00_B **ROMNE**, No ECC error detected in ROM. 01_B **ROMCE**, Correctable error detected in ROM. 10_B **ROMUE**, Uncorrectable error detected in ROM. 11_B **ROMCU**, Correctable and uncorrectable error detected in ROM.
BERR	0	rr/-	**Bus Error** If set a host system bus error occurred due to an HSM access. In case of concurrent accesses setting BERR has priority over the attempt to clear the bit by writing a “1”. It is therefore strongly recommended to read BERR first, and clear BERR only if the bit was set.

ERRIE

Interrupt enable register for error handling. The enable bits control error interrupt generation for the corresponding error condition.

ERRIE	Offset	Reset Value
Error Interrupt Enable Register	084_H	$0000\ 0000_H$



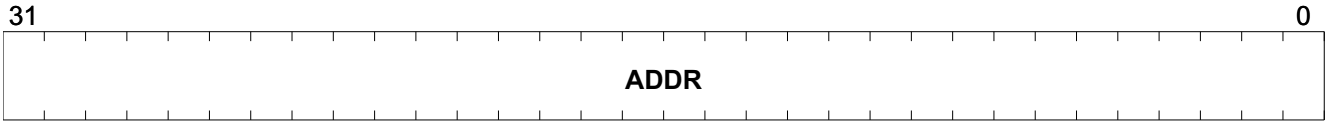
Field	Bits	Type	Description
ECCPKCROMIE	20:19	rw/-	**ECC errors of PKC ROM Interrupt Enable** 00_B **PKCROMECCID**, PKC ROM ECC error interrupts disabled. 01_B **PKCROMCEIE**, Correctable error interrupt enabled. 10_B **PKCROMUEIE**, Uncorrectable error interrupt enabled. 11_B **PKCROMCUIE**, Correctable and uncorrectable error interrupt enabled.
ADPKCSCMIE	18	rw/-	**Address Error of PKC SCM Interrupt Enable** 0_B **PKCSCMADID**, Address error of PKC SCM interrupt disabled. 1_B **PKCSCMADIE**, Address error of PKC SCM interrupt enabled.
ECCPKCSCMIE	17:16	rw/-	**ECC errors of PKC SCM Interrupt Enable** 00_B **PKCSCMECCID**, PKC SCM ECC error interrupts disabled. 01_B **PKCSCMCEIE**, Correctable error interrupt enabled. 10_B **PKCSCMUEIE**, Uncorrectable error interrupt enabled. 11_B **PKCSCMCUIE**, Correctable and uncorrectable error interrupt enabled.
ADPKCTCMIE	15	rw/-	**Address Error of PKC TCM Interrupt Enable** 0_B **PKCTCMADID**, Address error of PKC TCM interrupt disabled. 1_B **PKCTCMADIE**, Address error of PKC TCM interrupt enabled.
ECCPKCTCMIE	14:13	rw/-	**ECC errors of PKC TCM Interrupt Enable** 00_B **PKCTCMECCID**, PKC TCM ECC error interrupts disabled. 01_B **PKCTCMCEIE**, Correctable error interrupt enabled. 10_B **PKCTCMUEIE**, Uncorrectable error interrupt enabled. 11_B **PKCTCMCUIE**, Correctable and uncorrectable error interrupt enabled.

Field	Bits	Type	Description
ADCTIE	12	rw/-	**Address Error of cache tag field Interrupt Enable** 0_B **CTADID**, Address error of cache tag field interrupt disabled. 1_B **CTADIE**, Address error of cache tag field interrupt enabled.
ECCCTIE	11:10	rw/-	**ECC errors of cache tag field Interrupt Enable** 00_B **CTEDDID**, Cache tag ECC error interrupts disabled. 01_B **CTCEIE**, Correctable error interrupt enabled. 10_B **CTUEIE**, Uncorrectable error interrupt enabled. 11_B **CTCUIE**, Correctable and uncorrectable error interrupt enabled.
ADCDIE	9	rw/-	**Address Error of cache data Interrupt Enable** 0_B **CDADID**, Address error of cache data interrupt disabled. 1_B **CDADIE**, Address error of cache data interrupt enabled.
ECCCDIE	8:7	rw/-	**ECC errors of cache data Interrupt Enable** 00_B **CDEDDID**, Cache data ECC error interrupts disabled. 01_B **CDCEIE**, Correctable error interrupt enabled. 10_B **CDUEIE**, Uncorrectable error interrupt enabled. 11_B **CDCUIE**, Correctable and uncorrectable error interrupt enabled.
ADRAMIE	6	rw/-	**Address Error of Local RAM Interrupt Enable** 0_B **RAMADID**, Address error of local RAM interrupt disabled. 1_B **RAMADIE**, Address error of local RAM interrupt enabled.
ECCRAMIE	5:4	rw/-	**ECC errors of Local RAM Interrupt Enable** 00_B **RAMECCID**, Local RAM ECC error interrupts disabled. 01_B **RAMCEIE**, Correctable error interrupt enabled. 10_B **RAMUEIE**, Uncorrectable error interrupt enabled. 11_B **RAMCUIE**, Correctable and uncorrectable error interrupt enabled.
ECCROMIE	3:2	rw/-	**ECC errors of HSM boot ROM Interrupt Enable** 00_B **ROMECCID**, ROM ECC error interrupts disabled. 01_B **ROMCEIE**, Correctable error interrupt enabled. 10_B **ROMUEIE**, Uncorrectable error interrupt enabled. 11_B **ROMCUIE**, Correctable and uncorrectable error interrupt enabled.
BERRIE	0	rw/-	**Bus Error Interrupt Enable** 0_B **BERRID**, Bus error interrupt disabled. 1_B **BERRIE**, Bus error interrupt enabled.

ERRADDR

Holds the address of the last bus error. In case of any ECC error **ERRADDR** is not updated. **ERRADDR** saves no history, i.e. in case of multiple errors the address of the last error is shown. Address information about previous errors is lost.

ERRADDR	Offset	Reset Value
Error Address Register	088_H	$0000\ 0000_H$



Field	Bits	Type	Description
ADDR	31:0	ro/-	**Error Address** Holds the address of the last bus error.

Note: If two bus errors occur nearly at the same time (e.g. due to simultaneous cache (AXI) and register (AHB) accesses to the host system) and the clock divider (CLKCTRL.CLKDIV) is set to a value larger than 1 , it could happen that ERRADDR does not show the last error but the previous error address.

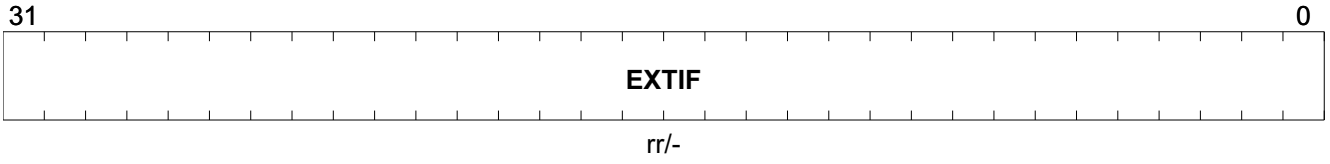
9.3.5 External Interrupts

The registers in this section are accessible by the HSM only. Access by the host results in a bus error. Access types are valid for HSM access.

EXTIF

This register holds the interrupt request flags of the external interrupt sources. An interrupt request to the NVIC is only triggered if the corresponding interrupt enable bit is set.

EXTIF	Offset	Reset Value
External Interrupt Flag Register	$0A0_H$	$0000\ 0000_H$

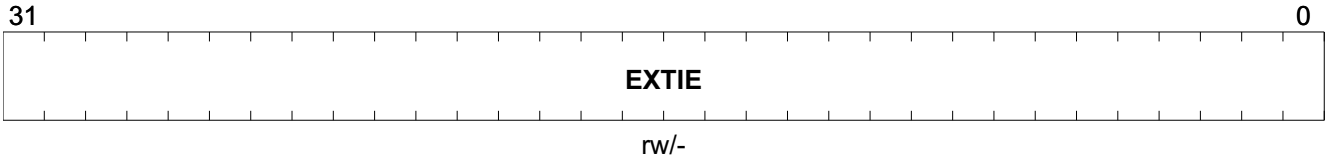


Field	Bits	Type	Description
EXTIF	31:0	rr/-	**External Flags** Each flag can be individually set by the external interrupt signal to generate an interrupt request if the corresponding interrupt enable flag is set. The HSM can reset the flags individually. If a simultaneous set and reset occurs, the value of the flag is '1'

EXTIE

This register controls enabling of the HSM external peripheral interrupt for each flag of **EXTIF**.

EXTIE	Offset	Reset Value
External Interrupt Enable	$0A4_H$	$0000\ 0000_H$



Field	Bits	Type	Description
EXTIE	31:0	rw/-	**External Interrupt Enable** Each set bit enables the interrupt for the corresponding flags in register **EXTIF**. If cleared bit disables the interrupt.

9.3.6 Single Access to Host Memories

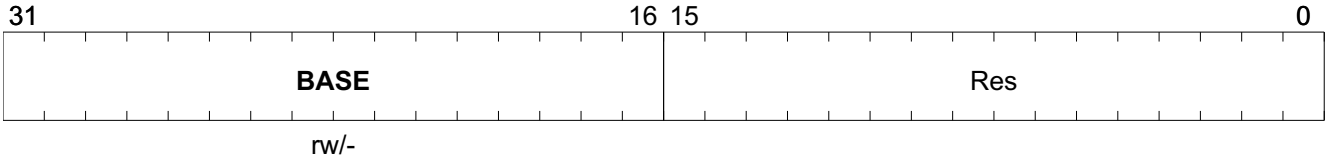
SAHBASE is accessible by the HSM only. Access by the host results in a bus error. Access types are valid for HSM access.

SAHBASE

This register controls the base address of the single access to host memory window.

The single access to host memory window can be switched off by setting the upper 4 bits of **SAHBASE**, i.e. by writing Fxxx $xxxx_H$

SAHBASE	Offset	Reset Value
Single Access to Host Base Address Register	$0C0_H$	$F000\ 0000_H$



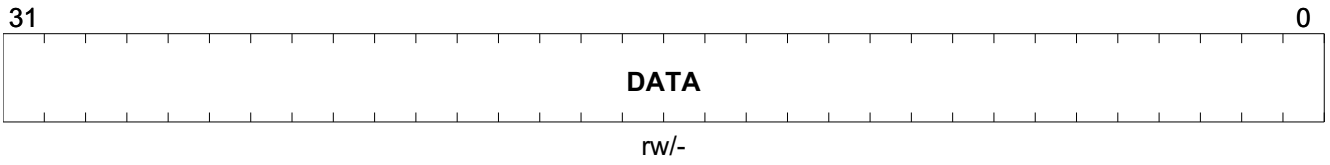
Field	Bits	Type	Description
BASE	31:16	rw/-	**Base Address.** Base address for single access to host memory window. The effective address of a single access to the host address space is **SAHBASE**.**BASE**+offset of access address to base of the memory window. $Fxxx_H$**SAHOFF**, Single access to host memory window is switched off.

SAHMEM

This is the memory window for single access to the host address space from the HSM. The window is only available if it is enabled in register **SAHBASE**.

Access types in the register descriptions are valid for HSM access. It is not visible to the host.

SAHMEM	Offset	Reset Value
Single Access to Host Memory Window	10000_H to $1FFFF_H$	Undefined$_H$



Field	Bits	Type	Description
DATA	31:0	rw/-	**Data for single access to host address space.** Byte, halfword and word access are possible (if supported by the target in the host system).

9.3.7 Sensor Interrupts and Reset

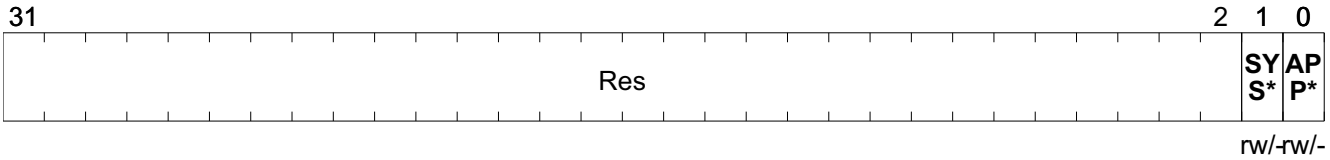
This section covers the registers to control sensor interrupts, trigger of application and system reset, and trigger sources for the temperature sensor .

The registers in this section are accessible by the HSM only. Access by the host results in a bus error. Access types are valid for HSM access.

RSTCTRL

Allows to trigger either a system or application reset for the chip by the HSM software. This register is write-protected unless **RSTPWD** holds the correct password value. Otherwise it is readonly and a write access is ignored.

RSTCTRL	Offset	Reset Value
Reset Control Register	$0E0_H$	$0000\ 0000_H$



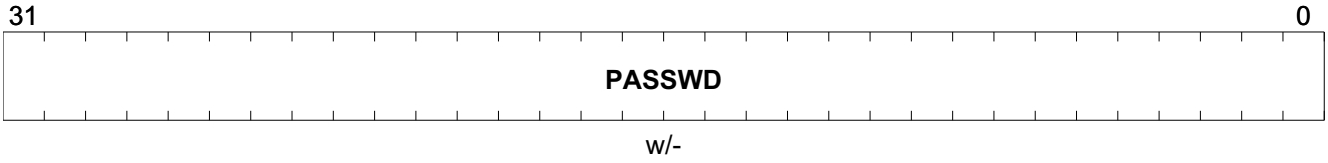
Field	Bits	Type	Description
SYSRST	1	rw/-	**System reset** If set by software a system reset is triggered.
APPRST	0	rw/-	**Application reset** If set by software an application reset is triggered.

Note: If both bits are set simultaneously a system reset is triggered.

RSTPWD

Password register to allow write access to **RSTCTRL**.

RSTPWD	Offset	Reset Value
Reset Password Register	$0E4_H$	$0000\ 0000_H$

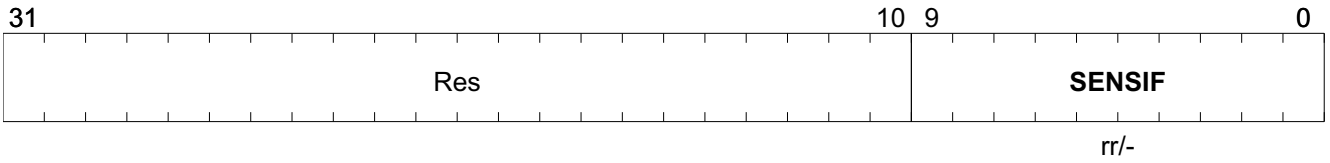


Field	Bits	Type	Description
PASSWD	31:0	w/-	**Password value for RSTCTRL.** If RSTPWD contains the constant 55AA $00FF_H$, **RSTCTRL** is writable. Reading this field always returns $0000\ 0000_H$.

SENSIF

Interrupt flags of the sensor signals. Can be used to trigger either an interrupt, application or system reset. For the mapping of sensor signals, see **Table 9-2**.

SENSIF	Offset	Reset Value
Sensor Interrupt Flag Register	$0F0_H$	$0000\ 0000_H$

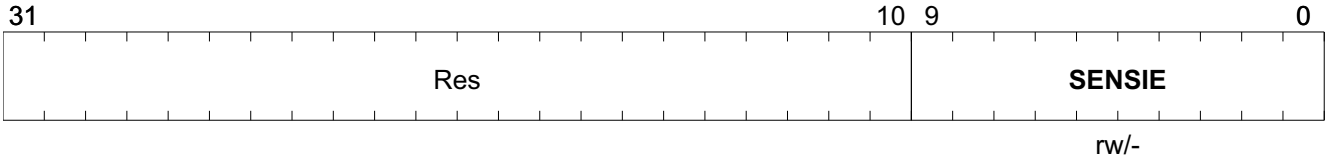


Field	Bits	Type	Description
SENSIF	9:0	rr/-	**Sensor flags.** Each flag can be individually set by a sensor signal to generate an interrupt request, application reset or system reset if the corresponding enable flag is set. The HSM can reset the flags individually. If a simultaneous set and reset occurs, the value of the flag is ‘1’.

SENSIE

Interrupt enable for the sensor signals. For the mapping of sensor signals, see **Table 9-2**.

SENSIE	Offset	Reset Value
Sensor Interrupt Enable Register	$0F4_H$	$0000\ 0000_H$

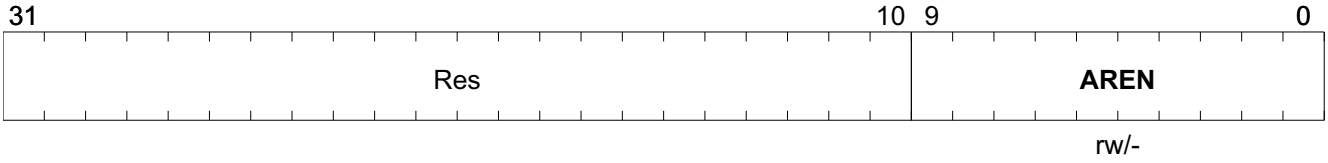


Field	Bits	Type	Description
SENSIE	9:0	rw/-	**Sensor Interrupt Enable.** Each set bit enables the interrupt for the corresponding flags in register **SENSIF**. If cleared the interrupt is disabled.

SENSAPPRST

Application reset trigger enable for the sensor signals. For the mapping of sensor signals, see **Table 9-2**.

SENSAPPRST	Offset	Reset Value
Sensor Application Reset Enable Register	**$0F8_H$**	**$0000\ 0000_H$**

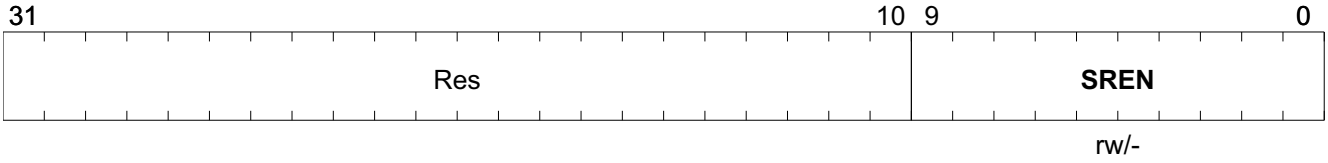


Field	Bits	Type	Description
AREN	9:0	rw/-	**Application Reset Enable.** Each set bit enables triggering of an application reset for the corresponding flags in register **SENSIF**. If cleared no application reset is triggered by the corresponding sensor source.

SENSSYSRST

System reset trigger enable for the sensor signals. For the mapping of sensor signals, see **Table 9-2**.

SENSSYSRST	Offset	Reset Value
Sensor System Reset Enable Register	$0FC_H$	$0000\ 0000_H$



Field	Bits	Type	Description
SREN	9:0	rw/-	**System Reset Enable.** Each set bit enables triggering of an system reset for the corresponding flags in register **SENSIF**. If cleared no system reset is triggered by the corresponding sensor source.

9.4 Bridge Registers (Host side access only)

Table 9-6 Register Address Space

Module	Base Address	End Address	Note
Bridge_EXT	F004 0000$_H$	F005 FFFF$_H$	

Table 9-7 Register Overview

Register Short Name	Register Long Name	Offset Address	Reset Value
Bridge Registers (Host side access only), HSM Reset			
HSMCTRL	HSM Control Register	1000$_H$	0000 0003$_H$
Bridge Registers (Host side access only), Debug Registers			
DBGBASE	Debug Base Address Register	1010$_H$	0000 0000$_H$
HSMOTGB	OCDS Suspend and Trigger Bus Control	1020$_H$	0000 0000$_H$
DBGMEM	Debug Memory Window	10000$_H$ to 1FFFF$_H$	Undefined$_H$

The registers are addressed wordwise.

Table 9-8 Register Access Types

Type	Symbol	Description
no access	-	Register is not accessible, an exception is triggered.
write only	w	Register is write-only, the register reads as ‘0’.
read only	ro	Register is read-only, a write access is ignored
read/write	rw	Register is read and writable
read/reset	rr	Register is readable. Writing ‘1’ to register bits reset them, writing ‘0’ do not modify the corresponding bits.
read/set	rs	Register is readable. Writing ‘1’ to register bits sets them, writing ‘0’ do not modify the corresponding bits
read/lockable	rl	Register is readable. Register is also writable after reset until it is locked. After locking register is read only until reset if not otherwise noted.

Parts of the bridge registers can be accessed from both the HSM and the host. For this reason there are two independent access types for each register. The first symbol in the registers type description defines the HSM access rights, the second the host access rights. Access to unused addresses results in a bus error.

Registers and debug memory window can only be accessed 32-bit wise and 32-bit aligned from both sides. Using 8-bit, 16-bit or un-aligned accesses lead to a bus error.

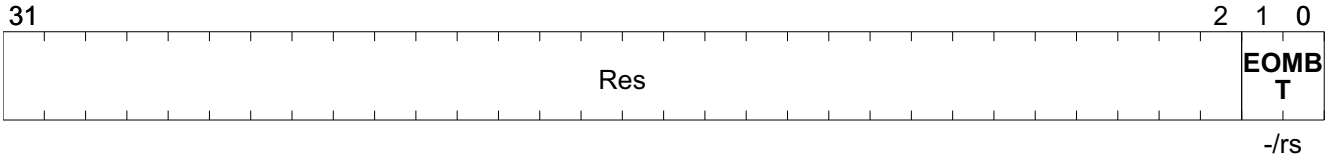
9.4.1 HSM Reset

The register in this section is accessible by the host only. Access by the HSM results in a bus error. Access types are valid for host access.

HSMCTRL

This register controls the reset of the HSM subsystem. The HSM must only be started when MBIST mode is not active.

HSMCTRL	Offset	Reset Value
HSM Control Register	1000_H	$0000\ 0003_H$



Field	Bits	Type	Description
EOMBT	1:0	-/rs	**End of Memory BIST test.** For security reasons this functionality is encoded redundantly. 00_B **MON**, MBIST mode activated. 01_B **MOFF1**, MBIST mode deactivated. See below. 10_B **MOFF2**, MBIST mode deactivated. See below. 11_B **MOFF3**, MBIST mode deactivated. Start of HSM subsystem. Cannot be reset by software

9.4.2 Debug Registers

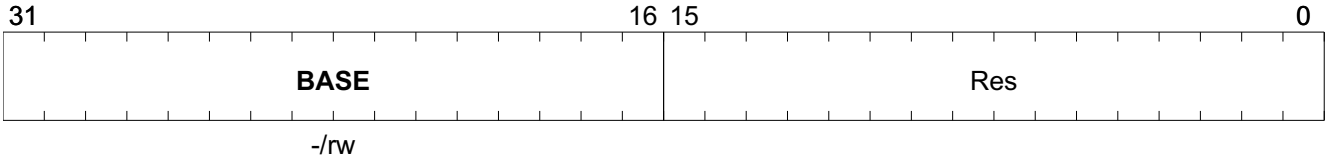
This section describes the interface for debugging. This address region is available in the host system only and only if HSM debugging is enabled (**DBGCTRL**.**HSM** set).

If HSM debugging is disabled access by the host results in a bus error. Access to the registers by the HSM results in a bus error. Access types in the register descriptions are valid for host access.

DBGBASE

This register controls the base address of the debug memory window.

DBGBASE	Offset	Reset Value
Debug Base Address Register	1010_H	$0000\ 0000_H$



Field	Bits	Type	Description
BASE	31:16	-/rw	**Base Address.** Base address for debug memory window. The effective address of a debug access is **DBGBASE**.**BASE**+offset of access address to base of the memory window.

HSMOTGB

The HSMOTGB control register is cleared by the application reset.

HSMOTGB	Offset	Reset Value
OCDS Suspend and Trigger Bus Control	1020_H	$0000\ 0000_H$

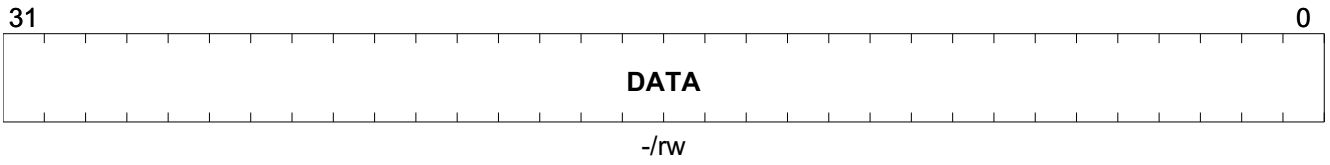
Field	Bits	Type	Description
TG_P	3	-/w	**TGS, TGB Write Protection** TGS and TGB are only overwritten if TG_P is set at the same access, otherwise unchanged. Reads as 0. 0_B **NC**, No change. 1_B **WE**, Write enable for the current access.
TGB	2	-/rw	**OTGB0/1 Bus Select** 0_B **BUS0**, Trigger Set is output on OTGB0. 1_B **BUS1**, Trigger Set is output on OTGB1.
TGS	1:0	-/rw	**Trigger Set for OTGB0/1** This module supports only one Trigger Set. The following values are defined: 00_B **NOTS**, No Trigger Set output. 01_B **TS1**, Trigger Set 1. 10_B **TS2**, No Trigger Set output. 11_B **TS3**, Trigger Set 1.

DBGMEM

This is the memory window for debug access from the host to the HSM. The window is only available if HSM debug is enabled.

Access types in the register descriptions are valid for host access. It is not visible to the HSM.

DBGMEM	Offset	Reset Value
Debug Memory Window	10000_H to $1FFFF_H$	Undefined$_H$



Field	Bits	Type	Description
DATA	31:0	-/rw	**Data for debug access.** Only word access to word-aligned addresses are supported.

10 Memories

The HSM utilizes three types of physical memories:

- **ROM**.
- **RAM**.
- Shared Flash (please refer to the AURIX™ Target Specification **[10]**).

10.1 ROM

The Hardware Security Module contains a Boot ROM for the CPU and a µ-code ROM for the PKC engine.

10.1.1 Boot ROM

The Boot ROM contains code and read-only data that is necessary for the start up of the HSM subsystem after reset. The ROM can be switched off when the boot software is finished via **DBGCTRL**. This enhances the security protection and reduces the power consumption.

The ROM data is protected with a single bit error correction, double bit error detection code. ECC errors in ROM are signaled to the Bridge module.

The size of the boot ROM is 4 KB.

The ROM is tested by a checksum.

10.1.2 PKC µ-Code ROM

The µ-Code ROM of the PKC module contains µ-code for the PKC engine. The code is not accessible by the CPU.

The µ-Code ROM data is protected with a single bit error correction, double bit error detection code. ECC errors in ROM are signaled to the Bridge module.

The size of the µ-Code ROM is 1 K x 18 bits (excluding ECC).

10.2 RAM

The Hardware Security Module contains a local RAM for the CPU and two RAM modules for the PKC engine.

10.2.1 Local RAM

The RAM data is protected with a single bit error correction, double bit error detection code. ECC errors in RAM are signaled to the Bridge module.

The size of the local RAM (excluding ECC) is 96 KB.

The RAM is tested via MBIST.

The contents of the local RAM is not preserved over all types of reset:

- it is completely cleared in case of a cold “power-on reset”. This is done by the MBISTs triggered by the startup software running on the host before the HSM is released. See also **[10]**.
- in case of a system reset or a warm “power-on” reset the contents of the local RAM can be preserved. This depends on the settings for the boot process.
- Parts of the local RAM are cleared for any reset type by the HSM startup software, for details see **Section 11.2.3.2**.

10.2.2 PKC RAMs

The PKC module uses two RAM modules, a Shared Crypto Memory (SCM) which can be accessed by the CPU and by the PKC engine, and a Tightly Coupled Memory (TCM) which is used for data processing during operation of commands. The TCM is not accessible by the CPU.

Both RAMs use a single bit error correction, double bit error detection code. ECC errors are signaled to the Bridge module.

The size of the SCM is 1 KB (excluding ECC) logically organized as 256 x 32 bits.

The size of the TCM is 1 KB (excluding ECC) logically organized as 256 x 32 bits.

The RAMs are tested via MBIST.

11 Firmware Architecture

This chapter describes the HSM low-level firmware architecture:

- **HSM Firmware Overview**
- **Boot System**
- **Information Exchange between the Host and BOS**

11.1 HSM Firmware Overview

A boot system (BOS) is provided for HSM. All mandatory BOS functions for internal testing, production usage and startup behavior are stored in the boot ROM of the HSM. No patch functionality is available for this code. The BOS is the only HSM software component provided by Infineon.

Certain configuration information, such as module-specific configurations, security keys for AES, etc. is stored in the HSM Config Area that is located in an HSM-exclusive 1 KB sector of the Flash memory. This configuration information is encrypted with a mask-individual key stored in the BOS-ROM. Additionally, the configuration data is protected by a cryptographic checksum. For further details refer to **Figure 11-1**.

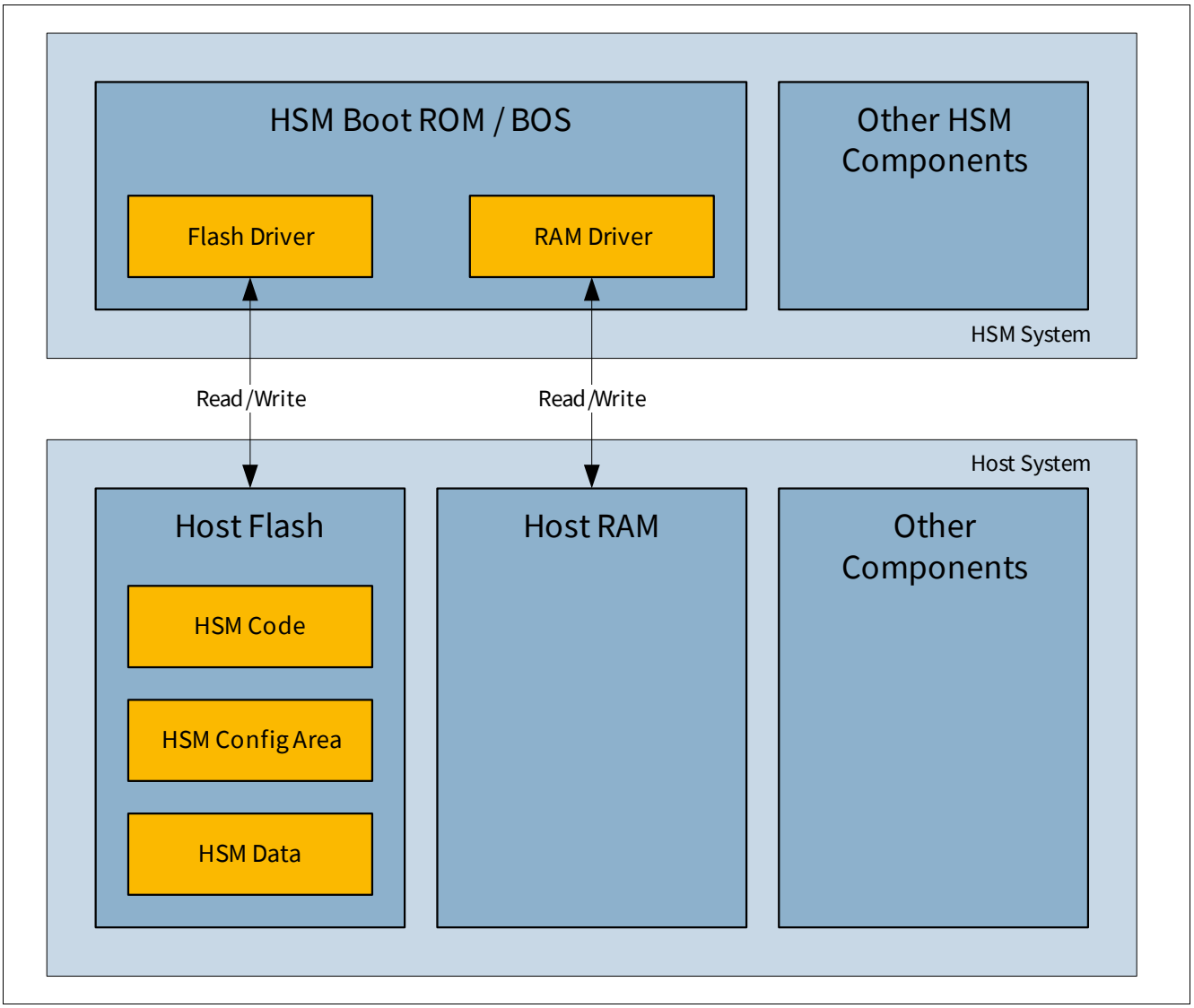


Figure 11-1 HSM-embedded software overview

11.2 Boot System

This chapter describes the boot system of the HSM.

11.2.1 Introduction

The secure boot system (BOS) has to fulfill a number of tasks on the HSM, as listed below:

- Full hardware access of HSM to the HSM Config Area for testing and configuration during chip production
- Life cycle management after reset
- Lock of debug functionality (if enabled)
- Update of security settings (e.g. AES keys, ...)
- Consistency check for used BOS configuration data
- Initialization of SFRs for certain HSM modules during startup
- Evaluation of host RAM and execution of the appropriate Testmode command, if available and requested
- Configuration and start of TRNG
- Fill local RAM regions used by the BOS during startup with logical 00_H before entering Usermode[1)]

Following additional tasks are performed for security reasons:

- All security-critical information (e.g. in cache, register file, ...) used during BOS startup is invalidated before entering user code.
- Security critical bits are re-checked after change (e.g. BOS bits, locking of HSM Config Area)
- Secure coding measures are implemented for flow and data control.
- Stack overflow protection via MPU
- The following 2 lock mechanisms are provided for the HSM Config Area:
  - read lock: always enabled during Usermode startup at an early point
  - write lock: enabled after finally programming the HSM Config Area which also blocks the Testmode!

For each kind of HSM reset, the HSM hardware performs the following steps in order to properly start the BOS:

- reset all registers (according to the applicable cold or warm reset condition)
- read the BOS vector table located in the BOS-ROM at offset 0. Based on that information, the HSM hardware initializes the stack pointer and jumps to the reset vector location.

Thus the BOS is executed first after every HSM reset. Therefore the BOS is responsible for a correct startup sequence of the HSM. First a basic **Boot Initialization** is performed. If the content of **HT2HSMS** is valid, the BOS reads **HT2HSMS**.REQ_MODE (refer to **Chapter 11.3** for details) in order to distinguish between following two different startup modes:

- Testmode and
- Usermode

If the content of **HT2HSMS** is not valid, the BOS always jumps to the Usermode path. If the Testmode is requested but blocked via UCB (because of enabled write lock for the HSM Config Area) an error is issued after locking the HSM-Config Area and the Sleep Mode is entered. **Figure 11-2** provides a rough overview of this flow.

1) Only in cases of a power-on reset the whole local RAM is filled with 00_H during the MBIST before the BOS starts running.

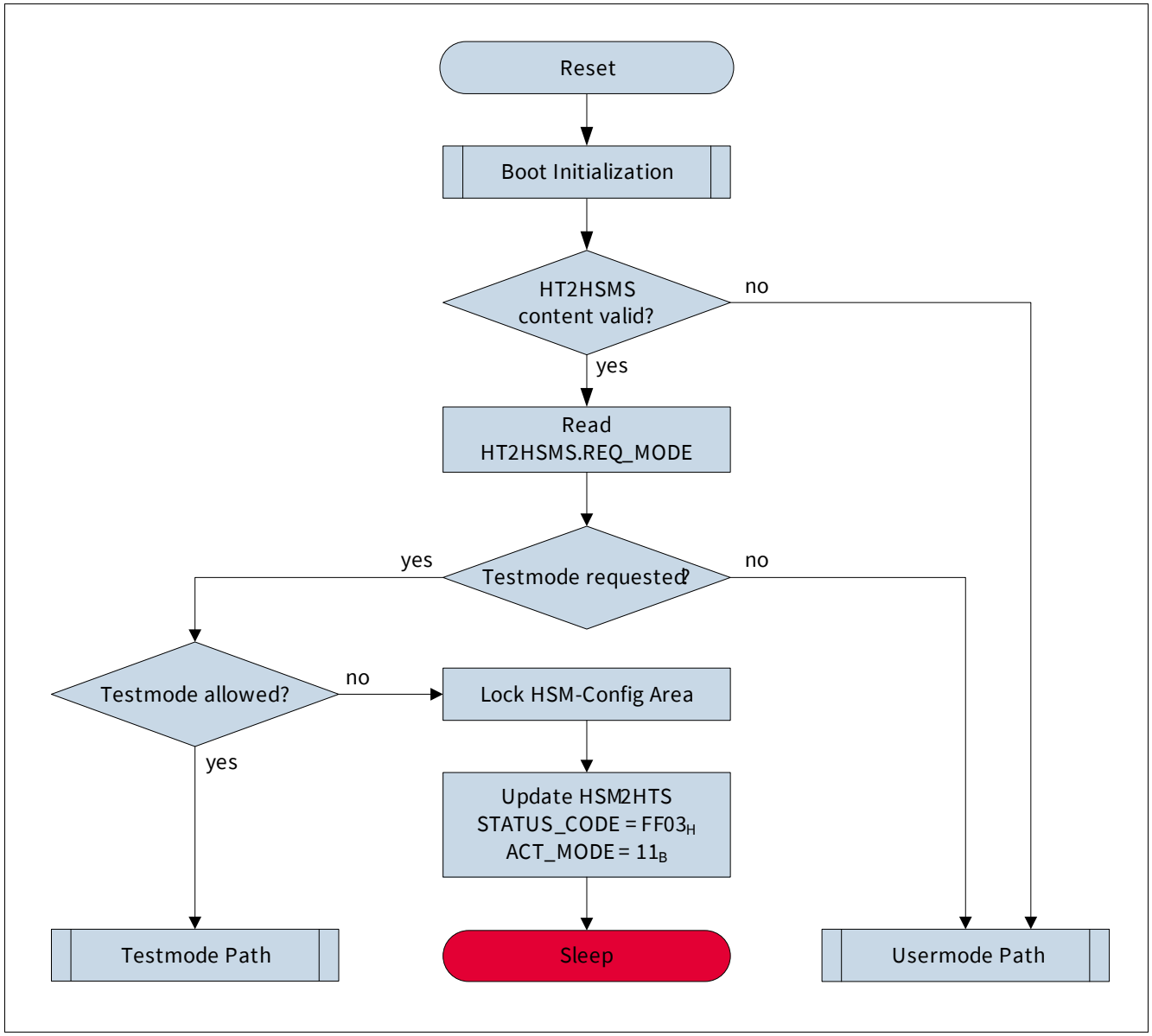


Figure 11-2 BOS startup overview

11.2.2 Boot Initialization

During boot initialization (see **Figure 11-3**) the basic preparation is done by the BOS which is needed for both the Testmode and Usermode paths. First the interrupt and exception handling is configured according to **Table 11-1**. Then the BOS checks the correctness of the cryptographic checksum of the HSM Config Area. To this end, a 16-byte hash value is calculated using the AES module for the used encrypted HSM configuration data, starting from the offset address 010_H. The AES is used in CBC mode where the key and the initial vector are taken from the BOS-ROM.

If the HSM Config Area hash value is invalid, the TRNG is started with the hardware reset values and the appropriate status code is set in **HSM2HTS** register. Before exiting the boot initialization in this case the debug locking is configured.

If the HSM Config Area is valid, all used data bytes are decrypted with the AES module in ECB mode with the mask-individual key stored in the BOS-ROM. Then the TRNG configuration is loaded and the TRNG is started.

Afterwards the first two AES keys (*AesKey0* and *AesKey1*) are copied from the plain HSM Config Area into the AES hardware module and the AES is locked. The implementation shall be in a way that the AES keys do not have to be stored in local RAM temporarily. Then the HSM Config Area is locked against further read accesses. Before exiting boot initialization in this case, debug locking is also configured. In this regard, the BOS evaluates bits HSMDBGDIS and DBGIFLCK of register DMU_SP_PROCONHSM and respectively updates the HSM and HOST bits in the debug control register **DBGCTRL**.

Table 11-1 Configuration of Interrupts and Exception Handling

Exception Type	Pos.	Interrupt	Handler
-	0	-	
Reset	1	-	BOS reset vector
Non-maskable interrupt	2	-	Generic Exception Handler
Hard Fault	3	-	Generic Exception Handler
Memory Management	4	-	Generic Exception Handler
Bus Fault	5	-	Special Exception Handler
Usage Fault	6	-	Generic Exception Handler
-	7-10	-	Generic Exception Handler
SVCall	11	disabled	Generic Exception Handler
Debug Monitor	12	disabled	Generic Exception Handler
-	13	-	Generic Exception Handler
PendSV	14	disabled	Generic Exception Handler
SysTick	15	disabled	Generic Exception Handler
Timer 0	16	disabled	Generic Exception Handler
Timer 1	17	disabled	Generic Exception Handler
TRNG	18	disabled	Generic Exception Handler
Bridge Service	19	disabled	Generic Exception Handler
Bridge Error	20	enabled	see details below for position 20
Bridge Error (ECCPKCROM)	20	ECCPKCROMIE = 10_B	Special Exception Handler
Bridge Error (ADPKCSCM)	20	ADPKCSCMIE = 1_B	Special Exception Handler
Bridge Error (ECCPKCSCM)	20	ECCPKCSCMIE = 10_B	Special Exception Handler
Bridge Error (ADPKCTCM)	20	ADPKCTCMIE = 1_B	Special Exception Handler
Bridge Error (ECCPKCTCM)	20	ECCPKCTCMIE = 10_B	Special Exception Handler
Bridge Error (ADCT)	20	ADCTIE = 1_B	Special Exception Handler
Bridge Error (ECCCT)	20	ECCCTIE = 10_B	Special Exception Handler
Bridge Error (ADCD)	20	ADCDIE = 1_B	Special Exception Handler
Bridge Error (ECCCD)	20	ECCCDIE = 10_B	Special Exception Handler
Bridge Error (ADRAM)	20	ADRAMIE = 1_B	Special Exception Handler
Bridge Error (ECCRAM)	20	ECCRAMIE = 10_B	Special Exception Handler
Bridge Error (ECCROM)	20	ECCROMIE = 10_B	Special Exception Handler

Table 11-1 Configuration of Interrupts and Exception Handling (cont'd)

Exception Type	Pos.	Interrupt	Handler
Bridge Error (BERR)	20	BERRIE = 1_B	Special Exception Handler
Sensor Interrupt	21	disabled	Generic Exception Handler
External Interrupt	22	disabled	Generic Exception Handler
-	23	-	Generic Exception Handler
PKC Ready Interrupt	24	disabled	Generic Exception Handler
PKC Error Interrupt	25	disabled	Generic Exception Handler

Generic Exception Handler:

- This exception handler shall lock the read access to the HSM Config Area and indicate to the host system the following information via the register **HSM2HTS** (for details refer to **Table 11-4**)
  - proper 16-bit status information in **STATUS_CODE** in following format $EExx_H$, where xx defines the interrupt that occurred
  - set **ACT_MODE** to 11_B and enter Sleep Mode

Special Exception Handler:

The behavior of this exception handler depends on the "CHECK_OS_VEC_TABLE" marker, which is to be set by the BOS only during evaluation of the valid user boot segment.

If this marker is set and the cause of the exception is a bus error, then:

- ignore interrupt and clear interrupt request
- return from interrupt

If this marker is NOT set:

- lock the read access to the HSM Config Area and indicate to the host system the following information via the register **HSM2HTS**[1] (for details refer to **Table 11-4**):
  - proper 16-bit status information in **STATUS_CODE**. In the case of a bus fault $EF00_H$ is used; in the case of a bridge error the following format is used: 100xxxxx $xxxxxxxx_B$, where the last 13 bits indicate the error case taken from register **ERRCTRL**.
  - set **ACT_MODE** to 11_B and enter Sleep Mode

1) Interrupts are configured in a way that they only enter the service function in case that more than a 1-bit error occurred for ECCCT, ECCCD, ECCRAM, ECCROM or ADCT, ADCD, ADRAM.

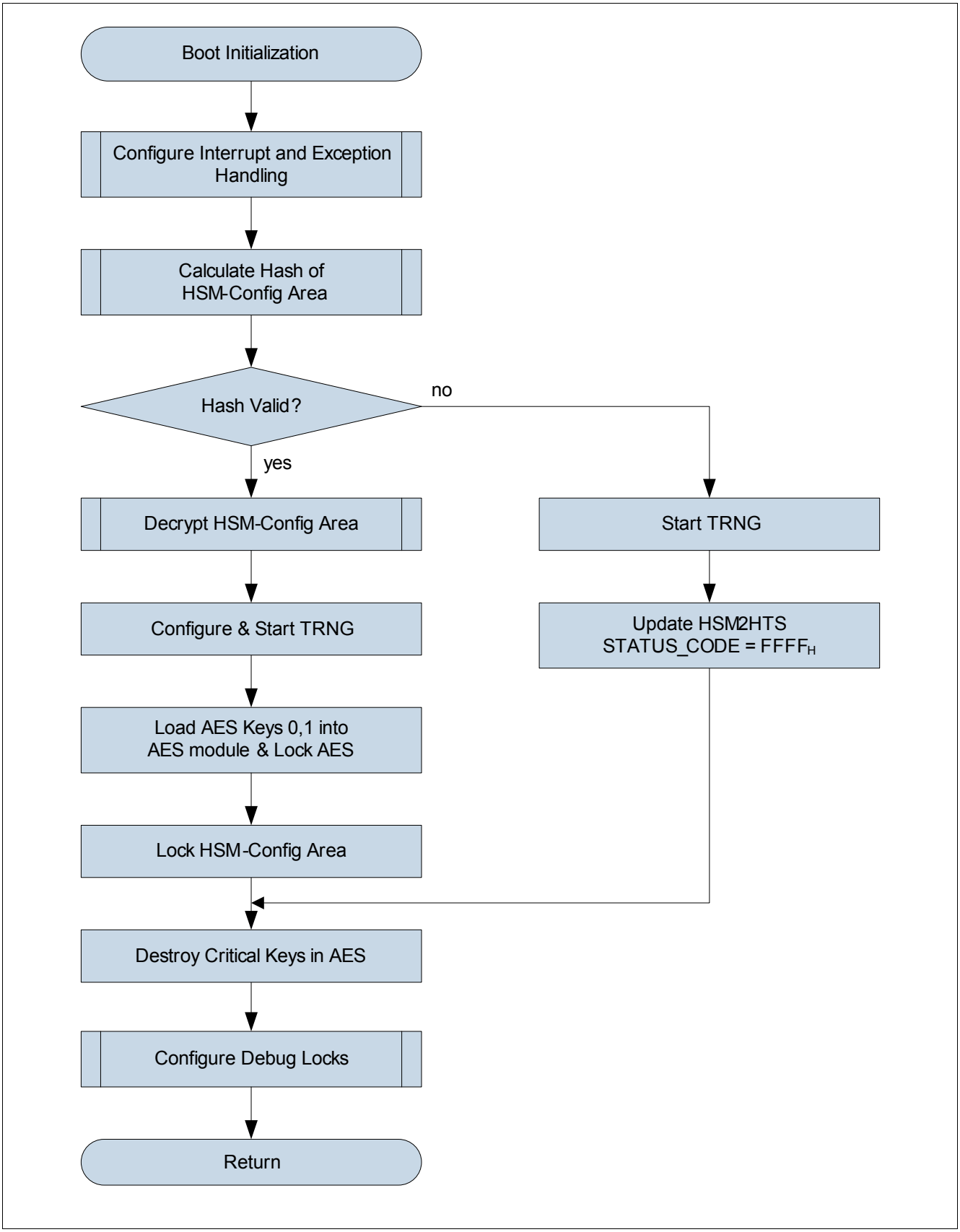


Figure 11-3 Boot initialization

11.2.3 User Mode

If the Usermode path is selected, the BOS first checks the correctness of the cryptographic checksum of the HSM Config Area by evaluating the status code in **HSM2HTS**. If the hash value is wrong, the BOS locks read access to the HSM Config Area (for security reasons) and stops execution after updating **HSM2HTS**, otherwise the Usermode path is continued to be followed. In this case the BOS checks whether the two AES keys are properly locked in the AES module. All security-relevant data in local RAM is overwritten at this point.

Afterwards the BOS looks for the proper User code segment in order to determine the stack pointer value, the reset vector and the VTOR address. For details about the mechanism, please refer to **Chapter 11.2.3.1**. If no valid User-Code is found, the BOS ends up in an error, else the BOS sets the Usermode active flag in **HSM2HTS** to signal to the host firmware that the BOS is prepared for entering Usermode. For synchronization with the host the BOS waits until **HT2HSMS.OS_ENTRY_ALLOWED** is set.

Then the BOS checks whether the “HALT After Reset” (HAR) feature is requested by checking if **HT2HSMS.HAR_ENABLE** (refer to **Chapter 11.3** for details) is set. Additionally the HSM debug mode has to be activated for HAR mode. If these conditions are fulfilled, the BOS configures via debug registers that a BREAK is triggered by hardware when executing the first User-OS instruction (boot vector). If HSM debug mode is not enabled, the HAR request will be ignored. For further details also refer to **Chapter 12.3.4**. Then the RAM content used by the BOS during startup is filled with a logical 00_H pattern for security reasons in order to avoid that it can be read by the User-OS later. Afterwards bridge interrupts are disabled, the User-OS stack pointer is loaded and the vector table offset address (VTOR) is set in order to point to the valid User-OS vector table, followed by the transition to the User-OS which is prepared in the cache and executed from there. This also includes disabling of the HSM ROM via **ERRCTRL**.ROMON.

In the case of an error during BOS startup, the detailed error information is updated in the register **HSM2HTS** (refer to **Chapter 11.3** for details) and the HSM is set to sleep mode. In case of no error **HSM2HTS** is cleared before entering the user code.

The basic initialization steps performed by the BOS are also illustrated in **Figure 11-4** and **Figure 11-6**.

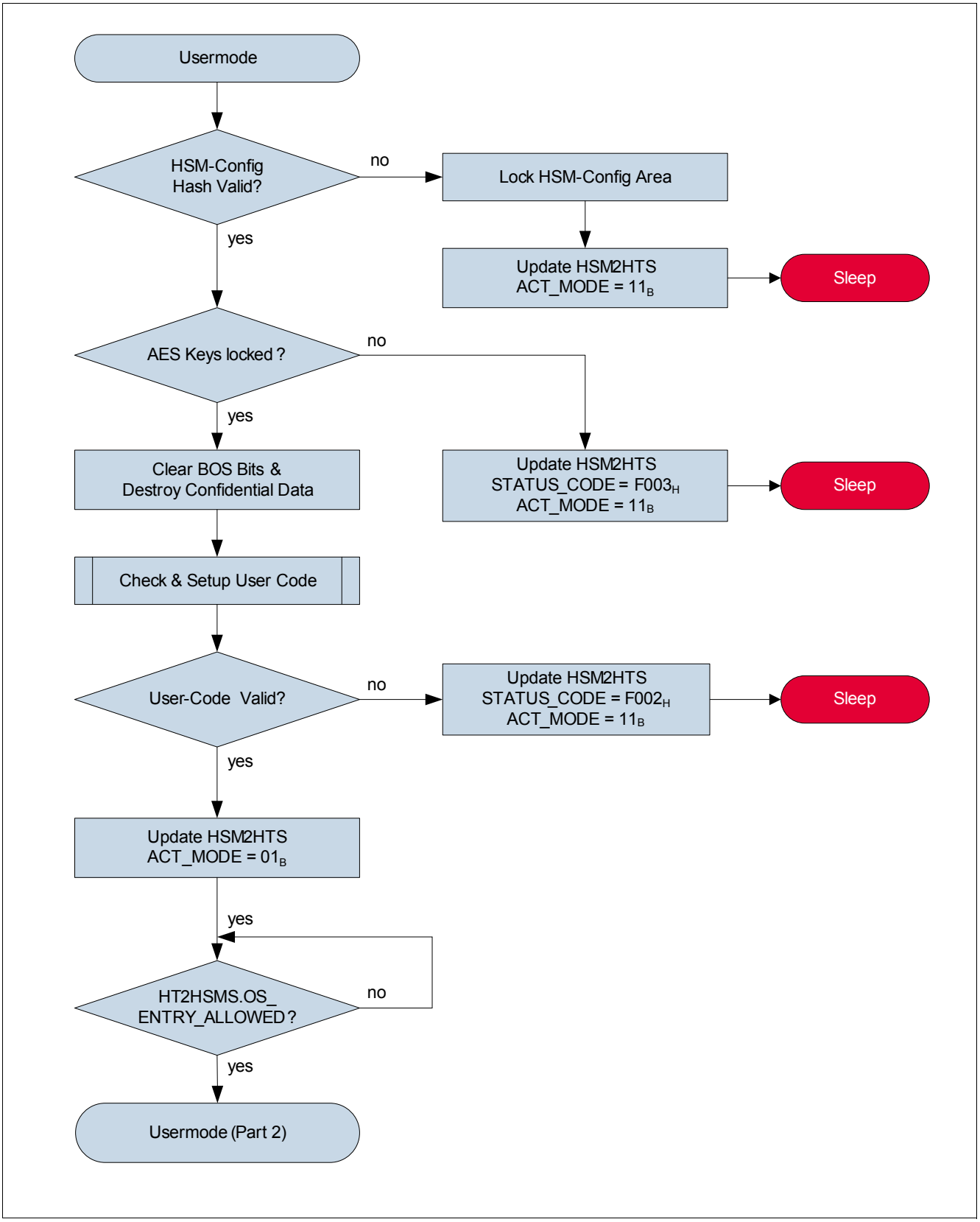


Figure 11-4 BOS Usermode flow (part 1)

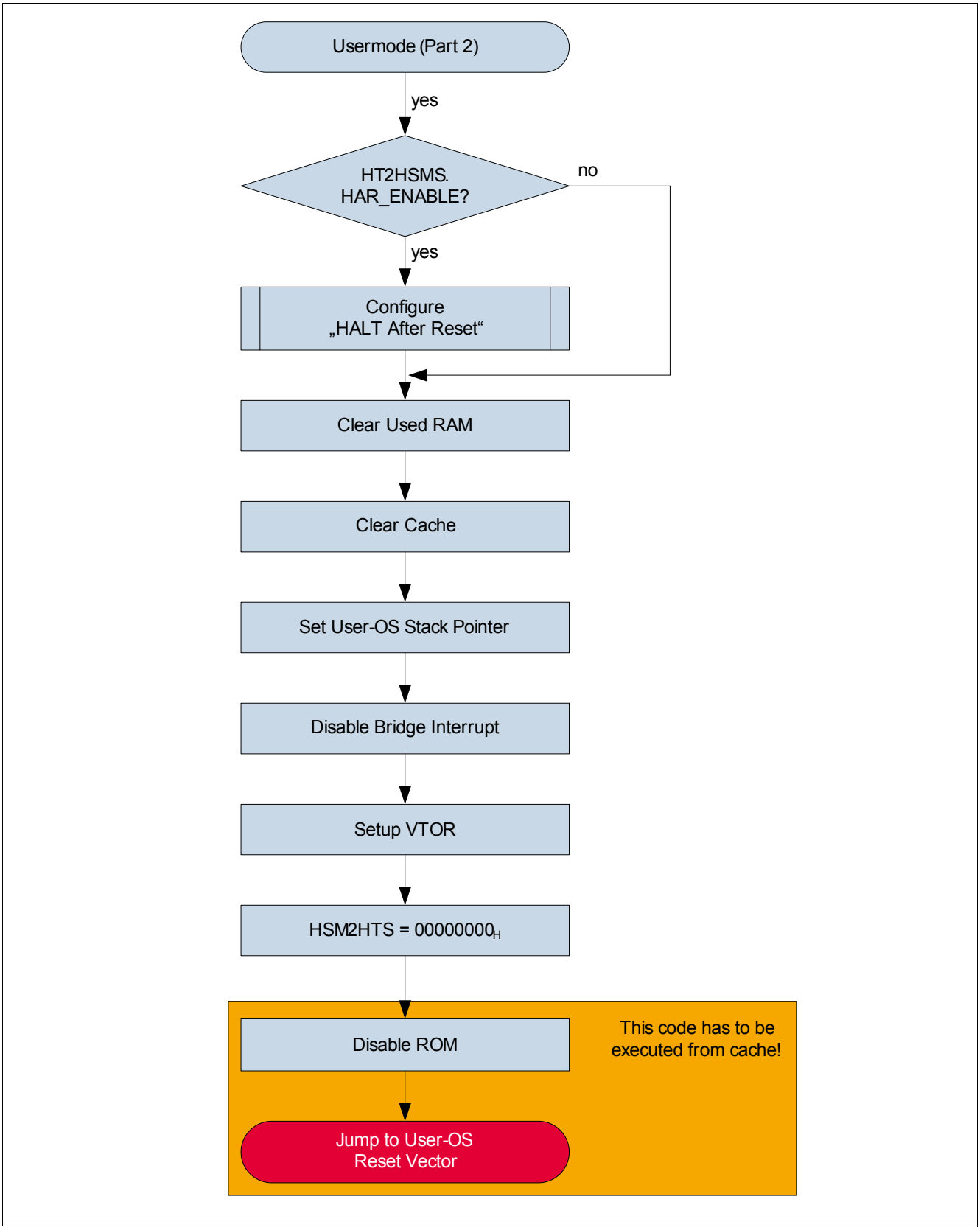


Figure 11-5 BOS Usermode flow (part 2)

11.2.3.1 User-OS Code Check

The user code is located in the lower 640 KB of the Program Flash 0 (see memory map, **Table 2-1**). It consists of 40 sectors of 16 KB each. As protection against resets during e.g. HSM software updates, the BOS has implemented a 4-step approach to select the valid User-OS boot segment. The host register DMU_SP_PROCONHSMCBS defines four bit fields BOOTSEL0 .. BOOTSEL3 which can point to any of the potential HSM exclusive PFlash sectors. The coding for BOOTSELx is shown in **Table 11-2**.

Table 11-2 Boot Sector Selection

BOOTSELx (x = 0-3)	Description	Address
00_H	Sector HSM0X is searched	8000 0000_H
01_H	Sector HSM1X is searched.	8000 4000_H
02_H	Sector HSM2X is searched.	8000 8000_H
:	:	
25_H	Sector HSM37X is searched.	8009 4000_H
26_H	Sector HSM38X is searched.	8009 8000_H
27_H	Sector HSM39X is searched.	8009 $C000_H$

The 4-step approach to find a valid User-OS boot segment as shown in **Figure 11-6** is performed as follows:

- Step 0:
  Set marker “CHECK_OS_VEC_TABLE” indicating that ECC errors are allowed and need a special handling
- Step 1:
  Determine address of potential boot sector based on DMU_SP_PROCONHSMCBS.BOOTSEL0 and check whether it is a valid sector (BOOTSEL0 values larger than 27_H are not allowed). If sector is not valid continue with step 2.
  Search for boot code: Read vector table at the beginning of the sector and perform the actions in following order:
  - check if ECC error occurred[1)]. Jump to Step 2 in case of an error
  - check that stack pointer points into internal RAM. Jump to Step 2 in case of an error
  - check that reset vector points into a valid HSM PFlash sector (sector HSM0X to HSM39X). Jump to Step 2 in case of an error
  - Jump to Step “PASS”
- Step 2:
  Determine address of potential boot sector based on DMU_SP_PROCONHSMCBS.BOOTSEL1 and check whether it is a valid sector (BOOTSEL1 values larger than 27_H are not allowed). If sector is not valid continue with step 3.
  Search for boot code: Read vector table at the beginning of the sector and perform the actions in following order:
  - perform same three checks as in Step 1. Jump to Step 3 in case of an error
  - Jump to Step “PASS”

1) An ECC error in the flash that it signalled via an FPI bus error can lead to both, a bus error interrupt and a bus Fault exception!

- Step 3:
  Determine address of potential boot sector based on DMU_SP_PROCONHSMCBS.BOOTSEL2 and check whether it is a valid sector (BOOTSEL2 values larger than 27_H are not allowed). If sector is not valid continue with step 4.
  Search for boot code: Read vector table at the beginning of the sector and perform the actions in following order:
  - perform same three checks as in Step 1. Jump to Step 4 in case of an error
  - Jump to Step “PASS”
- Step 4:
  Determine address of potential boot sector based on DMU_SP_PROCONHSMCBS.BOOTSEL3 and check whether it is a valid sector (BOOTSEL3 values larger than 27_H are not allowed). If sector is not valid exit function with “FAIL”.
  Search for boot code: Read vector table at the beginning of the sector and perform the actions in following order:
  - perform same three checks as in Step 1. Exit function with “FAIL” in case of an error
  - Jump to Step “PASS”
- Step “PASS”:
  A valid User-OS boot segment has been found. The BOS stores the stack pointer, the User-Code start address as well as the vector table base address in order to be able to enter the User-OS later on. Finally the marker “CHECK_OS_VEC_TABLE” is cleared and the function exits with a “PASS”.

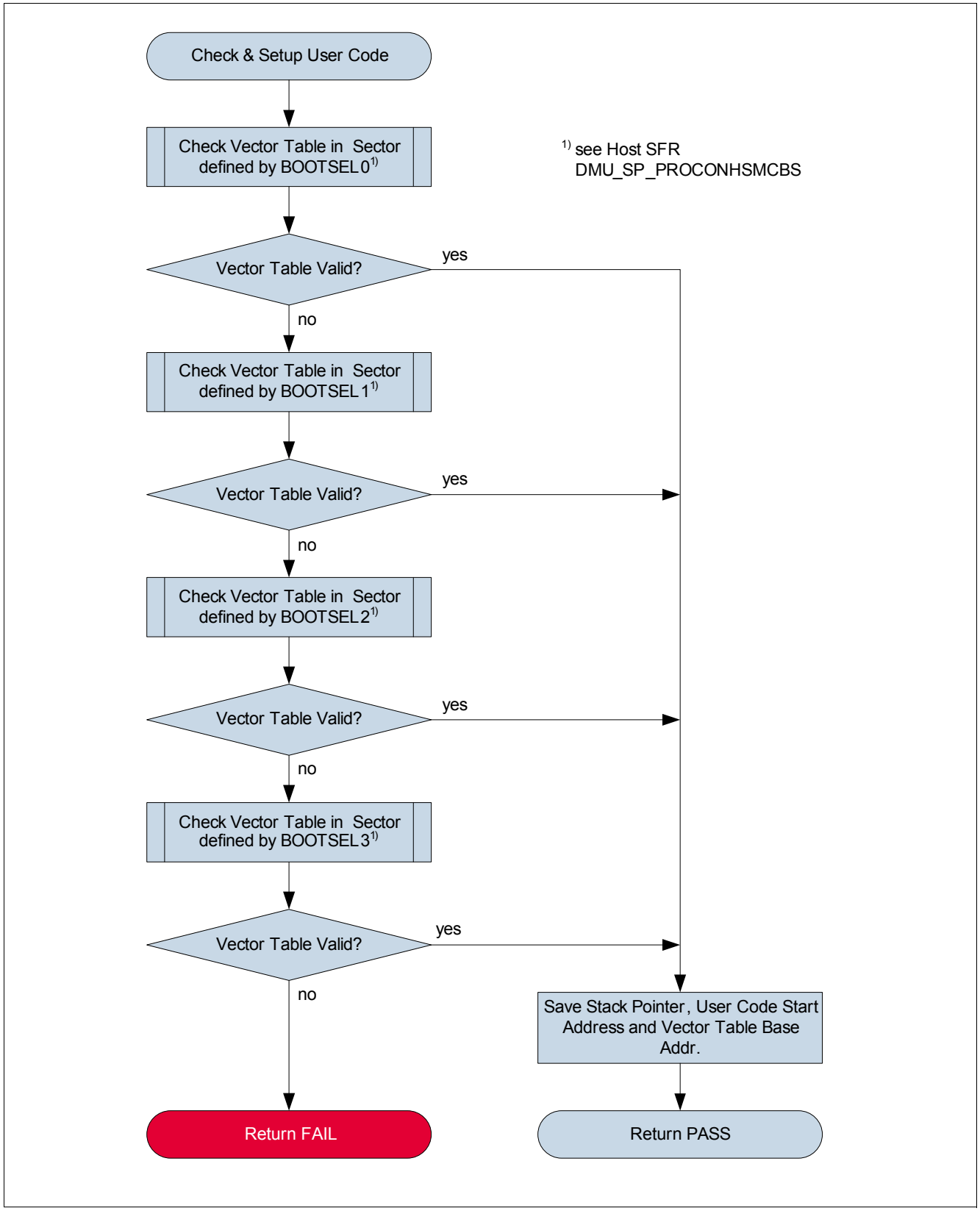


Figure 11-6 BOS User-OS code check

11.2.3.2 RAM Memory Layout in User Mode

At each HSM startup in user mode the BOS needs a certain amount of RAM for stack and variables. Therefore the first 384 bytes of the physical local RAM are reserved for the BOS. Before entering the user-OS, the RAM used by the BOS is filled with logical 00_H for security reasons. The rest of the local RAM is not touched by the BOS at all. For further details please refer to **Figure 11-7**.

Note: In User Mode the whole physical local RAM can be used by the User-OS

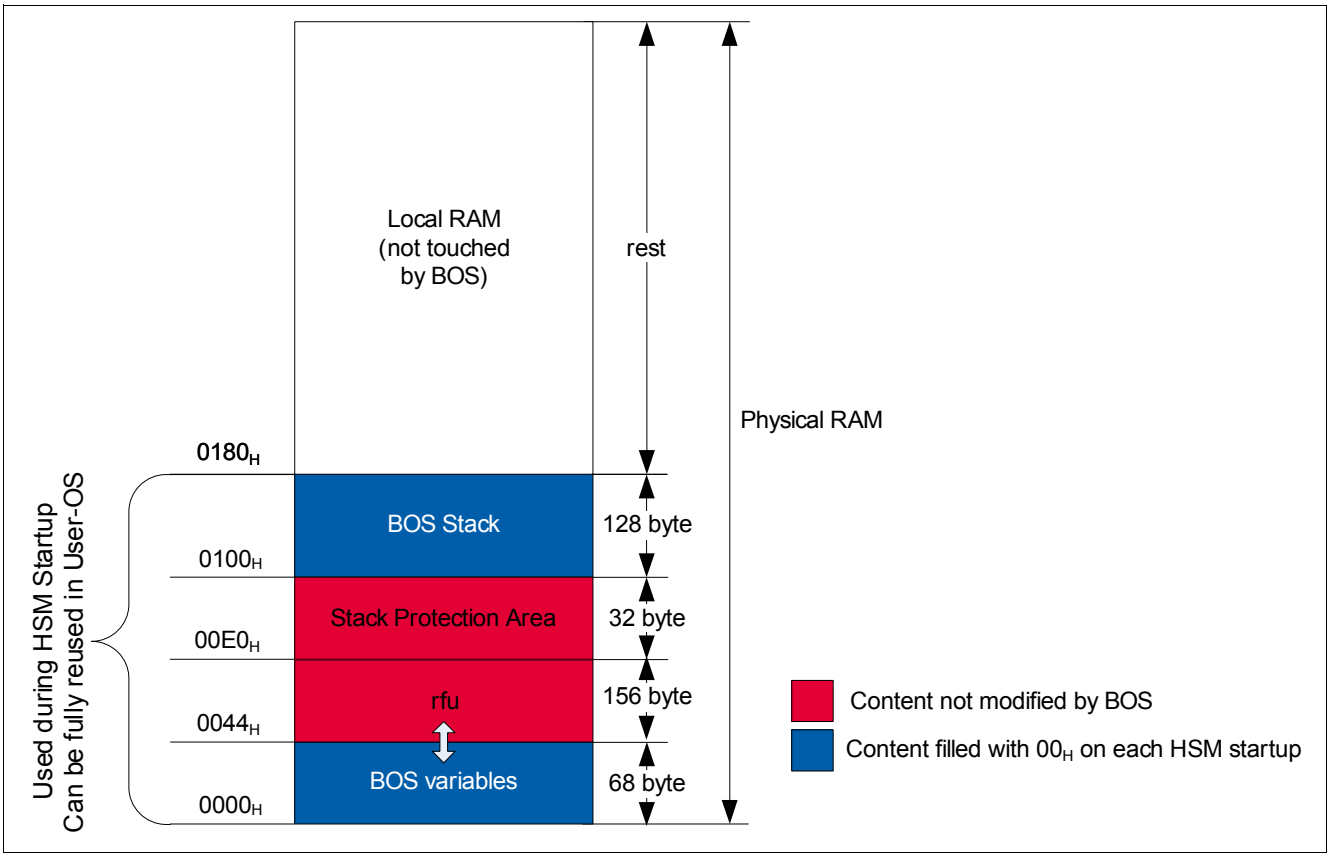


Figure 11-7 RAM Memory Layout (User Mode)

11.2.4 Test Mode

At the beginning of its life cycle the chip including HSM will be tested after wafer production. During this test phase all logical units and memory areas will be checked. In addition to structural tests (scan, MBIST) the HSM offers a Testmode for functional testing and analysis.

For security reasons this mode can only be entered when the HSM-Config Area is not write protected and the host system indicates the mode via **HT2HSMS** and in addition provides the correct mask-individual signature in the host RAM after reset. This signature is a ROM mask-individual 128-bit pattern located in the BOS-ROM. The Testmode functionality is available only if all 16 bytes match the values provided via the host RAM.

After successful verification of the signature the BOS checks the requested command by evaluating the command identifier. Depending on the result, either the execution of a Downloadable Test (DLT) or one single test command is requested. After execution the result is stored again in the host RAM; **HSM2HTS** is updated in order to indicate to the host system that the operation has finished, and finally the HSM enters sleep mode.

The BOS uses the host RAM of CPU0 for Testmode commands as well as DLTs. The address range is 7000 0000_H to 7001 $FFFF_H$.

11.3 Information Exchange between the Host and BOS

As the BOS on the one hand has to evaluate conditions like the reset type provided by the host system and, on the other hand, has to provide the host system with certain information like the current mode and some status information, the BOS uses the 32-bit registers **HT2HSMS** and **HSM2HTS** as communication channels between the host system and BOS.

Table 11-3 describes the format of **HT2HSMS** set by the host system before booting the HSM.

Table 11-3 Information Exchange Format from the Host to the BOS (via HT2HSMS)

Field	Bits	Description
HAR_ENABLE	31:28	"Halt After Reset" (HAR) mode configuration. Only bit combination 0101_B enables HAR, in all other cases HAR is disabled which is also the default value if CONT_VALID = 0. HAR can only be enabled if HSM debug mode is activated, otherwise the HAR request will be ignored.
REQ_MODE	27:24	Requested BOS mode (Testmode/Usermode) by host system Only bit combination 0101_B activates Testmode, in all other cases Usermode is active which is also the default value if CONT_VALID = 0
RFU	23:5	Reserved for future use Bits are not evaluated by the BOS
OS_ENTRY_ALLOWED	4	Synchronization mechanism between host and HSM to allow the host to configure when the User-OS entry is allowed 0_B , User-OS entry not allowed (default if CONT_VALID = 0) 1_B , User-OS entry allowed
RFU	3:2	Reserved for future use Bits are not evaluated by the BOS
RST_TYPE	1	Information about type of reset[1] that has occurred 0_B , Power-on reset (default if CONT_VALID = 0) 1_B , Application reset
CONT_VALID	0	Indication as to whether the **HT2HSMS** register contents are valid. Only if this bit is 1 can the other bits in this register be evaluated by the BOS. 0_B , Register contents are not valid 1_B , Register contents are valid

1) In the current concept there is no difference anymore between power-on or application reset from the BOS's point of view!

Table 11-4 describes the format of **HSM2HTS** set by the BOS during operation. This information can be evaluated by the host system during and after booting and by the User-OS after booting.

Table 11-4 Information Exchange Format from the BOS to the Host (via HSM2HTS)

Field	Bits	Description
RFU	31:25	Reserved for future use All bits must be set to 0
TM_CMD_FINISHED	24	Indication for host as to whether the BOS Testmode command execution has finished (and the response is available in the host RAM). 0_B , Command execution not finished 1_B , Command execution finished
RFU	23:18	Reserved for future use All bits must be set to 0
ACT_MODE	17:16	Information about active mode 00_B , Boot process ongoing 01_B , Usermode Entry prepared 10_B , Testmode active 11_B , Error occurred during BOS operation
STATUS_CODE	15:0	16-bit status code 0000_H, No error occurred so far $F001_H$, Security attack detected $F002_H$, User OS reset vector or stack pointer is invalid (Usermode only) $F003_H$, AES key locking failed (Usermode only) $F004_H$, HSM Config Area locking failed (Usermode only) $F005_H$, CACHE CONFIG BOS bit locking failed (Usermode only) $F006_H$, PAU BOS bit locking failed (Usermode only) $FF01_H$, Testmode signature is invalid (Testmode only) $FF02_H$, DLT execution permission check failed (Testmode only) $FF03_H$, Testmode functionality blocked $FFFF_H$, HSM Config Area hash is invalid $C000_H$, Special Exception - Bus Fault C000H | $(bridgeStatus)_H$, Special Exception - Bridge Error EE00H | $(isrNumber)_H$, Generic Exception - Active ISR number

Notes regarding error handling

The error handling required in the event that any kind of sanity check fails (due to errors, attacks, ...) during startup is a very sensitive point for the automotive market as the host system must be able to analyze the problem and select a mechanism to recover from that state. For this reason, the BOS writes detailed error information into the bits 15:0 of the **HSM2HTS** register in the case of an error and also sets **HSM2HTS**.ACT_MODE to 11_B. The BOS then enters the sleep mode.

11.4 Information Exchange between Host SSW and HSM User Code

If secure boot is enabled for the device (DMU_SP_PROCONHSMCFG.SSWWAIT is set) and no halt after reset is requested, the SSW delays execution of host user code and waits for an acknowledge by the HSM.

SSW flow and respectively the host behavior in this start-up phase is completely handled by the HSM user code.

Depending on whether a foreground or background secure boot is done the user code

- sets **HSM2HTF**.0 immediately (background secure boot). The SSW will jump to the host user code and the secure boot checks are done in parallel to the host user code execution.
- performs the secure boot checks and sets **HSM2HTF**.0 after the checks and only in case no security violations are detected (foreground secure boot). The host user code is delayed until the checks are finished. In case of a detected security violation the host user code is not started at all.

In case of foreground secure boot the HSM user code can optionally write a seed value to **HSM2HTS**. This seed value is used by the SSW to generate some noise in order to obscure the power profile. The noise generation ends once **HSM2HTF**.0 is set.

12 System Debug

This chapter covers the debug system of the HSM. The HSM debug system consists of:

- **Flash Patch & Breakpoint unit (FPB)** providing hardware breakpoints
- **Data Watchpoint and Trace Unit (DWT)** providing watch points
- **ROM Table** providing information about the available debug modules

The modules provide together with the core debug functionality the following debugging features:

- HSM memories
  - read of HSM boot ROM (only before jump to NVM hasn't been done and ROM hasn't been switched off)
  - read/write of HSM local RAM
- read/write of HSM core registers
- read/write of bridge module registers
- read/write of HSM internal (memory mapped) peripherals
- 6x (fast) hardware breakpoints
- 4x trigger serving as
  - watch points
  - one value compare (data breakpoint has certain value)

The debug system does not contain the units of the Cortex™ M3 known as Instrumentation Trace Macrocell (ITM), Embedded Trace MacroCell (ETM) and the Trace Port Interface Unit (TPIU). That means that the HSM has no trace capabilities.

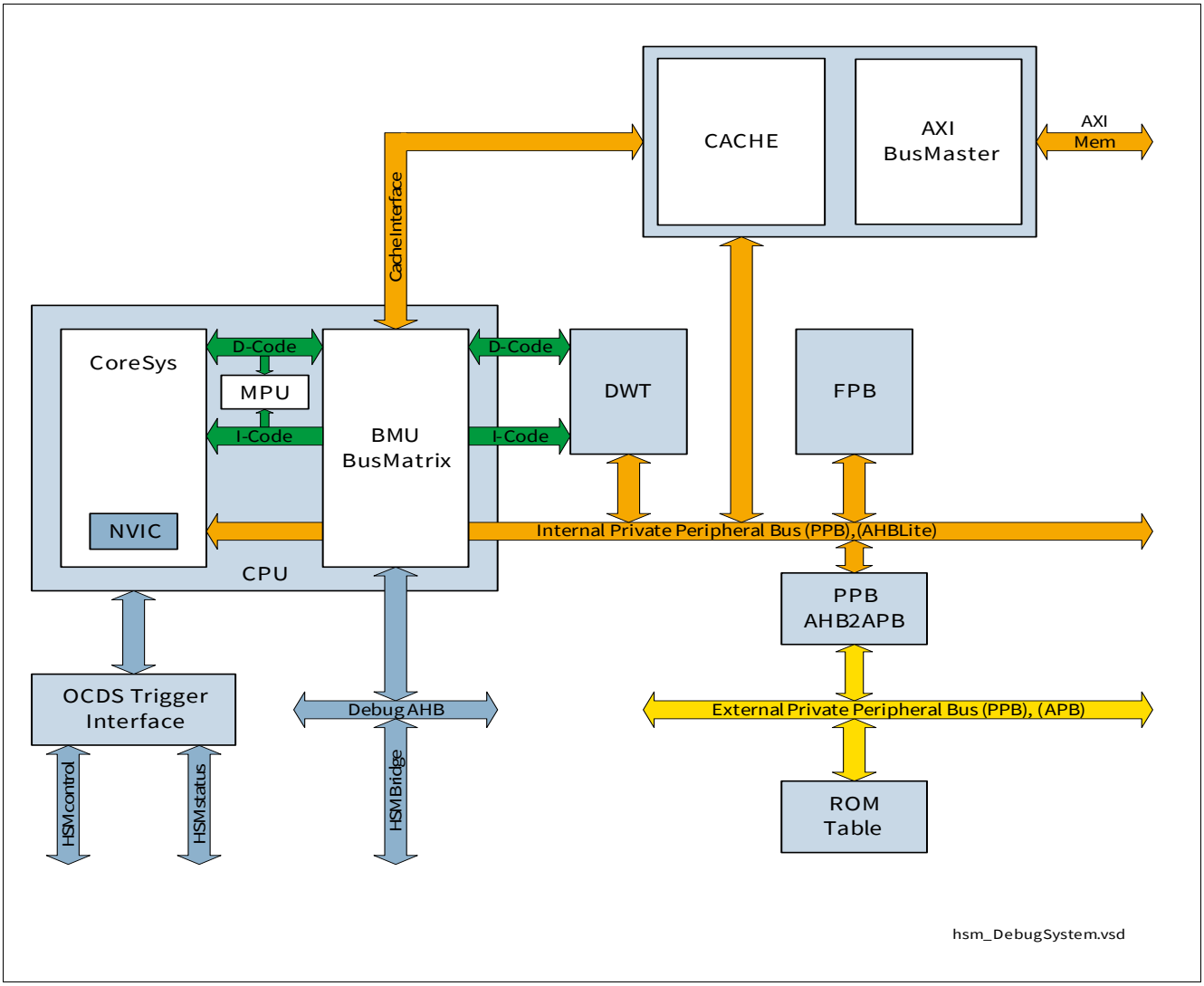


Figure 12-1 HSM debug block diagram

To have at least a minimum trace support the OCDS trigger interface provides some status information of the HSM sub system to the host OCDS.

Figure 12-1 shows a block diagram of the debug relevant part. The FPB and DWT are connected to the internal private peripheral bus (PPB) which is connected via the BMU bus matrix to the core. The ROM table is connected to the external PPB.

As mentioned in **Section 9.2.5** the HSM address space is mapped through a 64 KB window in the host system address space. A debugger can read and write the HSM internal resources via the host's OCDS through this window. The debugger has to create the correct addresses to access the DWT, FPB, core debug register or boot ROM and local RAM addresses.

12.1 Debug Modules

This sub chapter describes the involved debug modules in more detail.

12.1.1 Flash Patch & Breakpoint unit (FPB)

The Flash Patch & Breakpoint (FPB) unit provides six hardware breakpoints. This is done by 6 comparator registers FP_COMPx (x=0..5). The comparators can be configured to serve either as breakpoint or as patch address. For the HSM the flash patch mechanism is disabled and therefore the flash patch functionality and the literal load comparators can't be used. Beside this the Cortex™ M3 FPB module is unchanged for the HSM.

The Flash Patch Control Register FP_CTRL is configured with NUM_CODE2 = 000_B, NUM_LIT = 0010_B and NUM_CODE1 = 0110_B. Note that there is no flag to indicate that the flash patch mechanism is not available and the number of literal loads is two although no literal load comparators are supported. The compare address is extended to 32 bit to support breakpoints for HSM Code in the Pflash memory (see **Chapter 12.4.1**) above 512 MB addresses.

A match of a HW breakpoint will cause a debug event before the instruction is executed and the system will enter the debug state or Debug Monitor exception depending on the configuration of the Debug Halting Control and Status Register (DHCSR). Note that for the HSM the debug monitor mode shall not be used by the debugger.

The FPB is described in more detail in ARM Cortex M3 Technical Reference Manual **[2]**, chapter 11.4 "FPB".

12.1.2 Data Watchpoint and Trace Unit (DWT)

The Data Watchpoint and Trace unit (DWT) provides four comparators that can be configured either:

- as a hardware watchpoint (read, write or read/write)
- the first comparator against the clock cycle count register
- the second comparator together with another one as data comparator

The other configurations only make sense in combination with the ETM and are not mentioned here. In addition the DWT contains counter for:

- clock cycles CYCCNT
- folded instructions
- Load-Store-Unit operations
- sleep cycles
- cycles per instructions CPI (all instruction cycles except for the first cycle)
- interrupt overhead

In case a DWT debug event occurs, this is indicated by setting the Debug Fault Status Register bit DFSR.DWTTRAP.

The DWT module is taken from the Cortex™ M3 without any changes.

The DWT is described in more detail in ARM Cortex M3 Technical Reference Manual **[2]**, chapter 11.5 "DWT".

12.1.3 ROM Table

The ROM Table is used to provide information about debug components, e.g. the addresses of the debug components connected to the interface and whether a certain module is available or not.

The ROM table will only contain the entries for NVIC, DWT and FPB and all other modules marked as not present. For the content of the ROM table see ARM Cortex M3 Technical Reference Manual **[2]**, chapter 4.3

“ROM memory table”, Tab 4-3 and ARM Debug Interface v5 Architecture Specification [11], chapter 14 “ROM Tables”.

12.2 OCDS Interface

The OCDS Trigger Switch (OTGS) of the host allows to route trigger sources to trigger targets. The following trigger signals are defined:

- break request from host system (BRKIN)
- suspend request from host system (SUSIN)
- HSM is in halt mode (HALT)
- break request from HSM to host system (BRKOUT)

The OCDS Trigger Mux (OTGM) of the host routes Trigger Sets and interrupt requests from peripherals to the OCDS Trigger Switch (OTGS) and the MCDS. Trigger Sets are a collection of signals, which support a specific debug use case for a certain peripheral. For the HSM up to 16 core status bits may be used as on Trigger Set. The following signals are defined for the HSM Trigger Set:

- current interrupt priority (CURRPR)
- interrupt status (INTSTAT)
- HSM is in sleep mode (SLEEPING)
- HSM is in halt mode (HALTED)

To enable the trigger signal INTSTAT for the HSM interrupt status it is necessary to set the Trace Enable bit in the Debug Exception and Monitor Control Register (DEMCR.TRCENA).

12.3 Debug System

This sub chapter summarizes the debugger relevant topics for the HSM debug system.

12.3.1 Reset Handling

The bridge module generates the Reset signal for the HSM (Application \ Class3 reset). This HSM Reset resets the entire HSM subsystem including the debug components (similar to the Power-on reset of the standard Cortex™ M3).

Note that there is no possibility to reset only the debug part of the HSM. If the debugger is connected either the HSM software or the debugger has to reset/clear the debug register. No dedicated HSM debug control over the reset generation will be provided.

12.3.2 Clock and Sleep Mode Handling

The clock for HSM subsystem is also provided by the bridge module, the clock frequency for the HSM is derived from the host system. The HSM runs with the SPB clock frequency or with a reduced frequency. This HSM clock corresponds to the Cortex™ M3 free-running clock FCLK. The HSM will generate a “gated” main processor clock similar to the Cortex™ HCLK clock which is always synchronous to the HSM clock.

The debug modules and the debug portion of NVIC will always be provided with the free running clock which ensures that the debugger can always access the debug modules even if the HSM is sleeping.

Note that the sleepdeep functionality is not supported for the HSM. The sleepdeep has the same functionality as the normal sleep mode and has no additional effect or advantage.

In case the user code enters the sleep mode, e.g. via Wait For Interrupt (WFI) or the Wait For Event (WFE) instructions the SLEEPING signal is asserted indicating a Sleep-now or Sleep-on-exit mode. This indicates that the clock to the processor HCLK can be stopped. On receipt of a new interrupt or event in the case of WFE, the NVIC resets the SLEEPING signal, releasing the core from sleep. Halting the core by the debugger by setting bit C_HALT in the Debug Halting Control and Status Register (DHCSR) also causes the sleep mode to be exited because the debug portion is clocked with the free running clock FCLK.

Note that the debug system can’t be set back in the sleep mode and user code continues to be execute although for the software no wake-up event is visible. The user has to take care that the HSM is put again into sleep if no wake-up condition was met, e.g. by a while() loop.

12.3.3 Debug Access / Authentication

Debug access to the host and HSM subsystem is controlled via the debug control register DBGCTRL. During and after reset the debugger has no access to the HSM, host debug is permitted by the HSM. The reset behavior can be changed by the BOS. It’s up to the HSM customer software to provide an authentication mechanism to the debugger supplier and to enable after a successful authentication the debug access via DBGCTRL.HOST DBGCTRL.HSM enable bits.

12.3.4 Halt-After-Reset Implementation

At the end of the startup software routine it must checked whether the OSTATE.HARR bit in OCDS is set. If halt after reset (HAR) is requested and the flash is not protected, the FW must enable DEBUG (set (DHCSR.C_DEBUGEN), enable the FPB module (set FP_CTRL.ENABLE, set FP_CTRL.KEY) and then set a break point at the jump address to the user software via one of the Flash Patch Comparator registers. The HSM will stop and enter the HALT mode at the defined address. The HSM debug mode must be enabled to allow the HAR request, otherwise the HAR request must be ignored.

12.3.5 Peripheral Behavior in HALT Mode

The timer module, the watchdog timer, and the TRNG as well as the SysTick counter are suspended in HALT mode. The AES module is not suspended in HALT mode.

12.3.6 SPB Bus Conflict Triggered by HSM Debugging

The debugger accesses HSM internal resources through the 64 KB debug window, see **Section 9.2.5**. The access is done via the SPB bus on the host side. During a single debug access through the window the SPB bus is blocked and cannot be used for other accesses. As a consequence an HSM access to host resources during (i.e. triggered by) a debug access leads to a bus conflict on the SPB.

The following scenarios may cause a bus conflict:

- explicitly: the debugger accesses a host resource through the HSM address space. This can be a memory access (RAM, NVM) through the cache or single access window or a peripheral resource (e.g. SFR). This situation has to be avoided: the debugger has to address host resources always through host addresses and never through the HSM.
- implicitly: the debugger accesses HSM internal memories always through the cache inside the HSM. If the data is not already available in the cache it has to be loaded into the cache. As a side effect a used cache block may be replaced by the new block. If the used block is dirty (i.e. contains data that differs from the source memory) and the source memory is a host memory the HSM will issue a write access to the host memory while the SPB is still blocked by the debug access.

For the second scenario the following solutions are possible:

- Before accessing any HSM internal memories the debugger shall halt the HSM CPU via **DHCSR.C_HALT** and shall clean the cache by using the cache set clean register (64 writes to CACHE_SC with different set address). After each write access to CACHE_SC the debugger has to wait until the set is written back to the memory. This procedure also provides consistency of data in the HSM cache and host memories.
- The software running on the HSM avoids dirty host memory cache blocks by writing host memories via the single access to host memory window only.

Note: Cache clean by the debugger is not reasonable while the HSM CPU is still running. Therefore periodic window update shall be disabled for debug memory windows pointing to HSM RAM addresses.

12.4 Debug Modules Register Extension

This chapter describes the register extensions for the HSM debug modules compared to the standard Cortex™ M3.

Table 12-1 Register Address Space

Module	Base Address	End Address	Note
Debug Extension	E000 2000$_H$	E000 EFFF$_H$	

Table 12-2 Register Overview

Register Short Name	Register Long Name	Offset Address	Reset Value
Debug Modules Register Extension, Flash Patch Registers			
FP_COMPH0	Flash Patch Comparator High Register 0	0028$_H$	0000 0000$_H$
FP_COMPH1	Flash Patch Comparator High Register 1	002C$_H$	0000 0000$_H$
FP_COMPH2	Flash Patch Comparator High Register 2	0030$_H$	0000 0000$_H$
FP_COMPH3	Flash Patch Comparator High Register 3	0034$_H$	0000 0000$_H$
FP_COMPH4	Flash Patch Comparator High Register 4	0038$_H$	0000 0000$_H$
FP_COMPH5	Flash Patch Comparator High Register 5	003C$_H$	0000 0000$_H$
FP_COMPH6	Flash Patch Comparator High Register 6	0040$_H$	0000 0000$_H$
FP_COMPH7	Flash Patch Comparator High Register 7	0044$_H$	0000 0000$_H$
PID4	Peripheral IDentification Register 4	0FD0$_H$	0000 0000$_H$
PID0	Peripheral IDentification register 0	0FE0$_H$	0000 0001$_H$
PID1	Peripheral IDentification Register 1	0FE4$_H$	0000 0010$_H$
PID2	Peripheral IDentification Register 2	0FE8$_H$	0000 001C$_H$
PID3	Peripheral IDentification Register 3	0FEC$_H$	0000 0001$_H$
Debug Modules Register Extension, Debug Halting Control and Status Register			
DHCSR	Debug Halting Control and Status Register	CDF0$_H$	0X0X 0000$_H$

The registers are addressed wordwise.

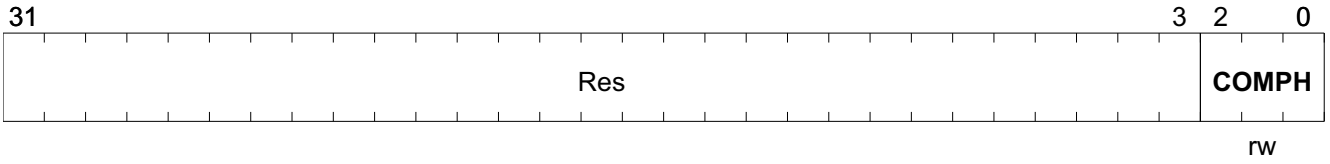
12.4.1 Flash Patch Registers

Description of the FPB register extensions.

FP_COMPH0

Defines the upper most three bits of the comparison address. The 32 bit comparator address is combined of the FP_COMP0.COMP field (and the FP_COMP0.COMPH field. This an extension of the standard Cortex™ M3 to be able to use a full 32 bit comparator address instead of only 28 bit to address especially the HSM Code in the Pflash memory.

FP_COMPH0	Offset	Reset Value
Flash Patch Comparator High Register 0	0028_H	$0000\ 0000_H$



Field	Bits	Type	Description
COMPH	2:0	rw	**High bits of Comparison address.** This field contains bit [31:29] of the comparison address. The lower [28:2] bits of the comparison address are taken from the corresponding FP_COMPx register

Flash Patch Comparator High Registers

For each of the eight comparators a corresponding Flash Patch Comparator Register High exists similar to the Flash Patch Comparator Register FPB_COMP0 to FPB_COMP7. They have the same layout as **FP_COMPH0**. Their names and offset addresses are listed below:

Note: Only FPB_COMP0 to FPB_COMP5 can be used, FPB_COMP6 to FPB_COMP7 are intended for literal loads which are not supported for the HSM.

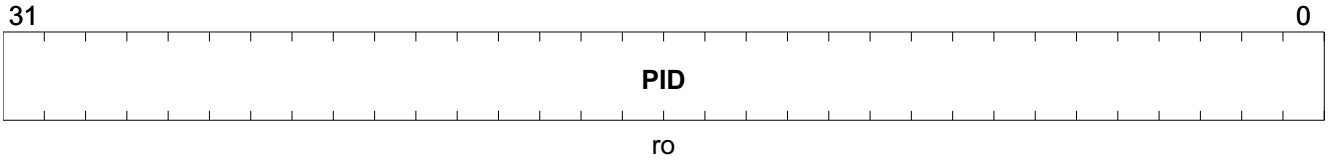
Table 12-3 Flash Patch Comparator High Registers

Register Short Name	Register Long Name	Offset Address	Reset Value
FP_COMPH1	Flash Patch Comparator High Register 1	$002C_H$	$0000\ 0000_H$
FP_COMPH2	Flash Patch Comparator High Register 2	0030_H	$0000\ 0000_H$
FP_COMPH3	Flash Patch Comparator High Register 3	0034_H	$0000\ 0000_H$
FP_COMPH4	Flash Patch Comparator High Register 4	0038_H	$0000\ 0000_H$
FP_COMPH5	Flash Patch Comparator High Register 5	$003C_H$	$0000\ 0000_H$
FP_COMPH6	Flash Patch Comparator High Register 6	0040_H	$0000\ 0000_H$
FP_COMPH7	Flash Patch Comparator High Register 7	0044_H	$0000\ 0000_H$

PID0

The Peripheral IDentification (PID) registers define the configuration of the FPB.

PID0	Offset	Reset Value
Peripheral IDentification register 0	$0FE0_H$	$0000\ 0001_H$



Field	Bits	Type	Description
PID	31:0	ro	**PID value.** This field contains the PID value

Peripheral IDentification Registers

The reset values of the registers PID0..PID5 are changed compared to the standard Cortex™ M3. The remaining PID5..PID7 and CID registers are unchanged. The addresses and reset values of the changed PID registers listed below. They have the same layout as **PID0**.

Table 12-4 Peripheral IDentification Registers

Register Short Name	Register Long Name	Offset Address	Reset Value
PID1	Peripheral IDentification Register 1	$0FE4_H$	$0000\ 0010_H$
PID2	Peripheral IDentification Register 2	$0FE8_H$	$0000\ 001C_H$
PID3	Peripheral IDentification Register 3	$0FEC_H$	$0000\ 0001_H$
PID4	Peripheral IDentification Register 4	$0FD0_H$	$0000\ 0000_H$

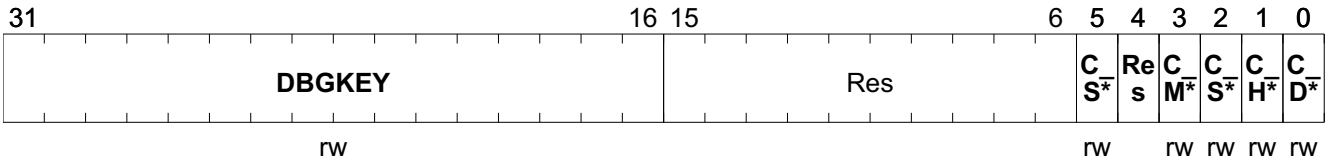
12.4.2 Debug Halting Control and Status Register

Description of the core debug core register extension.

DHCSR

The access right of the bit DHCSR.C_DEBUGEN is changed compared to the standard Cortex™ M3.

DHCSR	Offset	Reset Value
Debug Halting Control and Status Register	$CDF0_H$	$0X0X\ 0000_H$



Field	Bits	Type	Description
DBGKEY	31:16	rw	**Debug Key** $A05F_H$ must be written whenever this register is written. If not written as Key, the write operation is ignored and no bits are written into the register. Reads back as status bits [25:16]. For details, see [2].
C_SNAPSTALL	5	rw	See [2] for details.
C_MASKINTS	3	rw	**Mask interrupts** See [2] for details.
C_STEP	2	rw	**Steps the core in halted mode** See [2] for details.
C_HALT	1	rw	**Halts the core.** See [2] for details.
C_DEBUGEN	0	rw	**DEBUG enable.** Enables debug. This bit can only be written by the debugger and by the HSM SW only in case of HSM debugging is enabled.

12.5 SW Tool Extensions

The HSM subsystem is from the SW tools point of view a normal Cortex™ M3-based product. Therefore the SW development flow is the same as for the Cortex™ M3. There are no extensions in the ARM standard tools needed and no specific post processing tools required. It is assumed that the standard ARM development tools MDK-ARM 5.x under µVision4 from Keil™ are used but any other Software Tool Development Kit for the Cortex™ M3 may be used alternatively.

Based on the MDK-ARM Infineon Technologies will provide an Install-Shield containing:

- CMSIS compliant include files for the HSM device
- a CMSIS compliant startup code
- a uVision Device Database for the HSM device
- an example project

References

[1] HIS AK Security: SHE – Secure Hardware Extension Functional Specification; Version 1.1 Proposal for Standardization Revision 439; 1st April 2009

[2] ARM Cortex M3 Technical Reference Manual, Revision r2p0, **http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337g/DDI0337G_cortex_m3_r2p0_trm.pdf**

[3] ARM Cortex-M3 / Cortex-M3 with ETM (AT420/AT425) Errata Notice, Document Revision 2.0, **http://infocenter.arm.com/help/topic/com.arm.doc.eat0420c/Cortex-M3-Errata-r2p0-v2.pdf**

[4] ARMv7M Architecture Reference Manual (ARM DDI0403B), available at **http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.cortex/index.html**

[5] NIST/FIPS 197: Announcing the Advanced Encryption Standard (AES); November 26, 2001; **http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf**

[6] NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques, Morris Dworkin, December 2001, **http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf**

[7] NIST Special Publication 800-38B: Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication, Morris Dworkin, May 2005, **http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf**

[8] NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, Morris Dworkin, November 2007, **http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf**

[9] IEEE Standard 1619-2007: Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices, December 2007

[10] AURIX™ TC3xx 32-Bit Single-Chip Microcontroller, Target Specification, V1.0.0, 2014

[11] ARM Debug Interface v5 Architecture Specification, 8 February 2006, **http://infocenter.arm.com/help/topic/com.arm.doc.ihi0031a/index.html**

BSI

[12] Recommendations for RNG certification with AIS 31 (http://www.bsi.de)

Terminology

AES	Advanced Encryption Standard
AHB	Advanced High-performance Bus
APB	Advanced Peripheral Bus
AXI	Advanced eXtensible Interface
BIST	Built-In Self-Test
BOS	BOot System
CBC	Cipher Block Chaining
CFB	Cipher FeedBack
C-MPU	Core MPU
CMSIS	Cortex Microcontroller Software Interface Standard
CPU	Central Processing Unit
CTR	CounTeR
CTS	Cipher Text Stealing
DTS	Die Temperature Sensor
DWT	Data Watchpoint and Trace unit
ECB	Electronic CodeBook
ECC	Error Correction Code
ECDSA	Elliptic Curve Digital Signature Algorithm
EEPROM	Electrically Erasable Programmable Read Only Memory
EVR	Embedded Voltage Regulator
FPB	Flash Patch and Breakpoint unit
FPI	Flexible Peripheral Interconnect
GCM	Galois Counter Mode
HSM	Hardware Security Module
HW	HardWare
IV	Initial Value
LRU	Least Recently Used
NVIC	Nested Vectored Interrupt Controller
MBIST	Memory Built-In Self-Test
MPC	Miyaguchi-Preneel Compression
MPU	Memory Protection Unit
NMI	Non Maskable Interrupt
OCDS	On Chip Debug Support
OFB	Output Feedback Mode
OTP	One-Time Programmable
PKC	Public Key Cryptography

PRNG	Pseudo Random Number Generator
RAM	Random Access Memory
rfu	reserved for future use
ROM	Read-Only Memory
SCM	Shared Crypto Memory
SFR	Special Function Register
SHE	Secure Hardware Extension
SPB	System Peripheral Bus
SRI	Shared Ressource Interconnect
SW	SoftWare
TCB	Tweaked CodeBook
TCM	Tightly Coupled Memory
TRNG	True Random Number Generator
XEX	Xor-Encrypt-Xor
XN	eXecute Never
XTS	XEX-TCB-CTS

www.infineon.com

Edition 2016-03-04

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
Doc_Number

Legal Disclaimer

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics. With respect to any examples or hints given herein, any typical values stated herein and/or any information regarding the application of the device, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation, warranties of non-infringement of intellectual property rights of any third party.

Information

For further information on technology, delivery terms and conditions and prices, please contact the nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements, components may contain dangerous substances. For information on the types in question, please contact the nearest Infineon Technologies Office. Infineon Technologies components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.