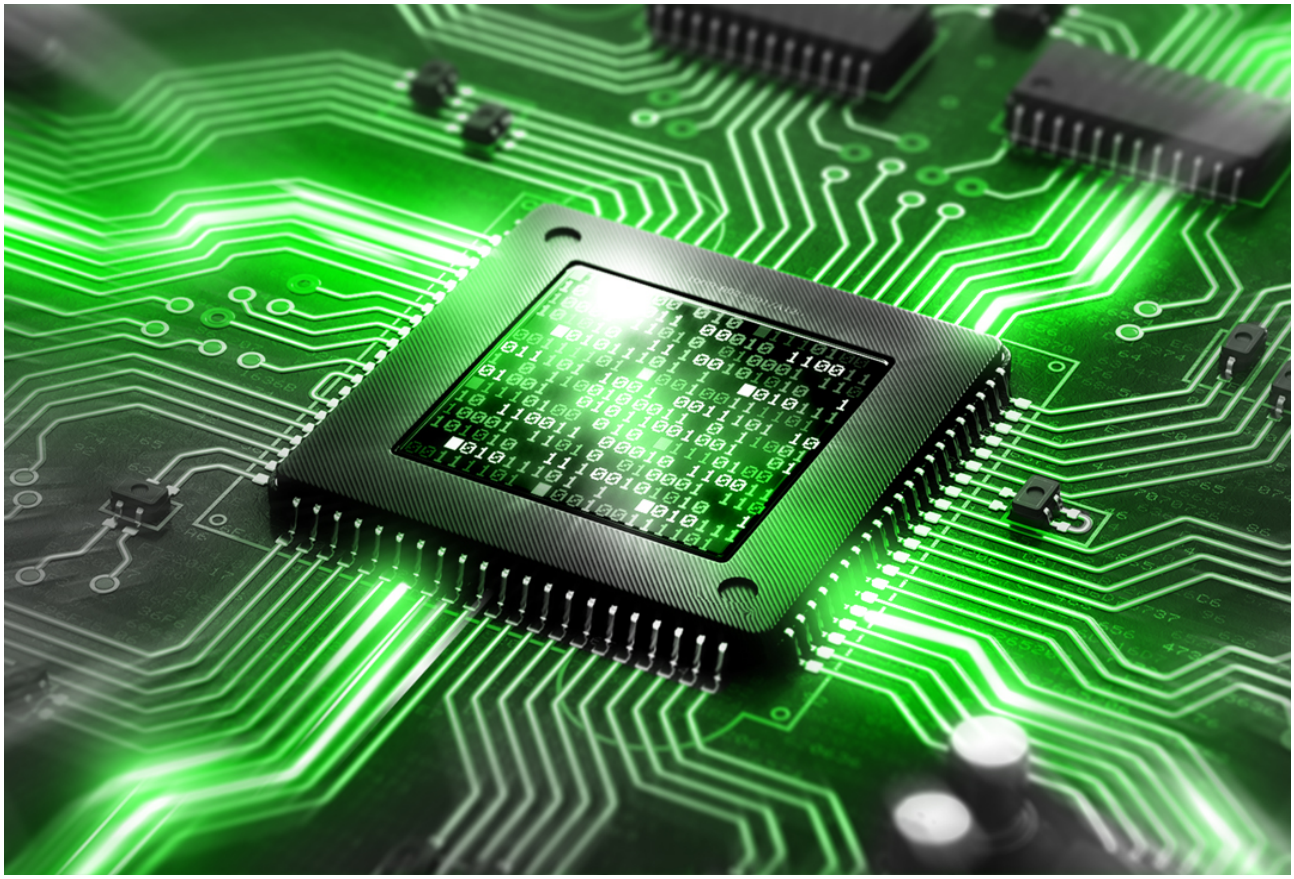
Elektrobit

EB zentur HSM Firmware

User Documentation

Release version 1.17.52 - December 07 2021



Elektrobit Automotive GmbH
Am Wolfsmantel 46
91058 Erlangen, Germany
Phone: +49 9131 7701 0
Fax: +49 9131 7701 6333
Email: info.automotive@elektrobit.com

Technical support

Europe	Japan	USA
Phone: +49 9131 7701 6060	Phone: +81 3 5577 6110	Phone: +1 888 346 3813

Support URL

https://www.elektrobit.com/support

Legal notice

Table of Contents

1. Overview 7
1.1. Abbreviations 7
2. Release notes 10
2.1. Version history 10
2.2. New features 13
2.3. Limitations 15
2.4. Supported architectures 16
2.5. Compiler 16
2.5.1. Tested compilers 17
2.5.2. Compiler options 17
3. Supported features 20
3.1. Feature overview 20
3.2. EB zentur HSM Firmware features 20
3.3. Cryptographic features 22
3.4. SHE Compatibility 23
3.4.1. SHE+ requirements 23
3.4.2. Key types and supported keys 24
3.4.3. Deviations 24
3.5. Key Generation 25
3.5.1. Key Generation Configuration 25
3.5.2. Key Generation Supported Keys 25
3.6. Secure boot 25
3.6.1. Configuration of the secure boot process 26
3.6.2. Secure boot procedure in normal operation mode 27
3.6.3. Secure boot extension 27
3.6.4. Secure boot extension add 28
3.6.5. Secure boot extension verify 28
3.6.6. Secure boot flow charts 29
3.6.7. Settings on the host side 31
3.7. Certificate Management 32
3.7.1. Certificate configuration 32
3.7.2. Certificate formats and algorithms 32
3.7.3. Certificate fields 32
3.7.4. Certificate loading 33
3.8. Concurrent PFlash Operations 34
3.8.1. Prerequisites 35
3.8.2. Stopping HSM execution from PFlash 36
3.8.3. Resuming HSM execution from PFlash 36
3.9. Firmware update 36

3.9.1. Overview 37
3.9.2. FWU interface 37
3.9.3. FWU data profile 38
3.9.4. FWU procedure 41
3.9.5. FWU rollback 42
3.9.6. FWU get bootloader version 42
3.9.7. FWU get version 43
3.9.8. FWU protection 44
3.9.8.1. FWU sanity check 44
3.9.8.2. FWU downgrade protection 44
3.10. Challenge-response protocol 44
3.11. Life Cycle Management 46
3.11.1. State Init 47
3.11.2. State Secure Level 1 48
3.12. Diagnostic interface 48
4. User's guide 49
4.1. Overview 49
4.2. HSM Firmware 49
4.2.1. Basic terminology 49
4.2.1.1. SHE 49
4.2.1.2. Host 49
4.2.1.3. HSM 50
4.2.1.4. Job 50
4.2.1.5. Channel 50
4.2.2. HSM Firmware context view 50
4.2.3. HSM core Firmware 52
4.2.4. Integrating EB zentur HSM Firmware 53
4.2.5. Flashing EB zentur HSM Firmware 56
4.2.6. Enabling secure operation 56
4.3. HSM proxy architecture 58
4.3.1. Logical view 58
4.3.2. HSM bridge 59
4.3.2.1. HSM bridge access 59
4.3.2.2. HSM bridge registers 60
4.3.2.2.1. Host to HSM registers 61
4.3.2.2.2. HSM to host registers 63
4.3.3. Job operation 64
4.3.3.1. Asynchronous job processing 65
4.3.3.2. Synchronous job processing 65
4.4. Use cases 65
4.4.1. HSM Setup 65
4.4.2. Locking/Unlocking HSM 66

4.4.3. Initializing EB zentur HSM proxy and Firmware 67
4.4.4. Dispatching a job 70
5. Module references 73
5.1. Application programming interface (API) 73
5.1.1. Type definitions 73
5.1.1.1. eb_hsm_callout_GetCounterValue 73
5.1.1.2. eb_hsm_callout_GetElapsedValue 73
5.1.1.3. eb_hsm_callout_IntDisable 73
5.1.1.4. eb_hsm_callout_IntRestore 73
5.1.1.5. eb_hsm_callout_Lock 73
5.1.1.6. eb_hsm_callout_Ticks2Ms 74
5.1.1.7. eb_hsm_callout_Unlock 74
5.1.1.8. eb_hsm_callouts_t 74
5.1.2. Macro constants 74
5.1.2.1. EB_HSM_TO_CMD_MS 74
5.1.2.2. EB_HSM_TO_DISPATCH_MS 75
5.1.2.3. EB_HSM_TO_INIT_MS 75
5.1.3. Objects 75
5.1.3.1. eb_hsm_callouts 75
5.1.4. Functions 75
5.1.4.1. eb_hsm_advance_life_cycle 75
5.1.4.2. eb_hsm_aes 76
5.1.4.3. eb_hsm_alive 77
5.1.4.4. eb_hsm_boot_failure 78
5.1.4.5. eb_hsm_boot_ok 78
5.1.4.6. eb_hsm_cancel 79
5.1.4.7. eb_hsm_debug_activation 80
5.1.4.8. eb_hsm_export_pub_key 80
5.1.4.9. eb_hsm_export_ram_key 81
5.1.4.10. eb_hsm_fw_rollback 82
5.1.4.11. eb_hsm_fw_update_data 82
5.1.4.12. eb_hsm_fw_update_finish 83
5.1.4.13. eb_hsm_fw_update_start 83
5.1.4.14. eb_hsm_gen_rnd 85
5.1.4.15. eb_hsm_get_bl_version 85
5.1.4.16. eb_hsm_get_challenge 86
5.1.4.17. eb_hsm_get_fw_version 86
5.1.4.18. eb_hsm_get_id 87
5.1.4.19. eb_hsm_get_life_cycle 88
5.1.4.20. eb_hsm_get_load_key_id_and_range 88
5.1.4.21. eb_hsm_get_result_channel 89
5.1.4.22. eb_hsm_get_she_status 89

5.1.4.23. eb_hsm_hash 90
5.1.4.24. eb_hsm_init 90
5.1.4.25. eb_hsm_integ_CalloutsGetRef 91
5.1.4.26. eb_hsm_integ_CryptoJobDataGetRef 91
5.1.4.27. eb_hsm_integ_GetCounterValue 91
5.1.4.28. eb_hsm_integ_GetElapsedValue 92
5.1.4.29. eb_hsm_integ_IntDisable 92
5.1.4.30. eb_hsm_integ_IntRestore 92
5.1.4.31. eb_hsm_integ_Lock 93
5.1.4.32. eb_hsm_integ_TICKS2MS 93
5.1.4.33. eb_hsm_integ_Unlock 93
5.1.4.34. eb_hsm_key_gen 94
5.1.4.35. eb_hsm_load_asym_priv_key 95
5.1.4.36. eb_hsm_load_asym_pub_key 95
5.1.4.37. eb_hsm_load_key 96
5.1.4.38. eb_hsm_load_plain_key 97
5.1.4.39. eb_hsm_mac 98
5.1.4.40. eb_hsm_mem_block_add 99
5.1.4.41. eb_hsm_mem_block_update 100
5.1.4.42. eb_hsm_mem_block_update_init 101
5.1.4.43. eb_hsm_mem_block_verify 102
5.1.4.44. eb_hsm_poll_channel 102
5.1.4.45. eb_hsm_resumeHsmExecution_fromFlash 103
5.1.4.46. eb_hsm_rnd_extend_seed 103
5.1.4.47. eb_hsm_rnd_init 104
5.1.4.48. eb_hsm_secure_boot 104
5.1.4.49. eb_hsm_setup 105
5.1.4.50. eb_hsm_sign 106
5.1.4.51. eb_hsm_stopHsmExecution_fromFlash 106
6. Appendix 108
6.1. Restrictions 108
6.2. XORShift128 algorithm 109
6.2.1. Parameters involved in the XORShift128 algorithm 109
6.2.2. Overview of steps involved in the XORShift128 algorithm 110
6.2.3. Test vectors for the XORShift128 algorithm 111
Bibliography 113

1. Overview

Welcome to the EB zentur HSM Firmware product documentation.

This document provides:

- Chapter 2, “Release notes“

  This chapter shows the release notes for the EB zentur HSM Firmware.

- Chapter 3, “Supported features“

  This chapter lists all cryptographic services and features supported by the EB zentur HSM Firmware.

- Chapter 4, “User's guide“

  This chapter contains background information and instructions for usage of the EB zentur HSM Firmware.

- Chapter 5, “Module references“

  This chapter gives information about configuration parameters and the application programming interface of the EB zentur HSM Firmware.

1.1. Abbreviations

The following table describes abbreviations that are used in this document:

Abbreviation	Description
ACG	AutoCore Generic
AES	Advanced Encryption Standard (standardized encryption algorithm)
API	Application Programming Interface
BMHD	Boot Mode HeaDer
CBC	Cipher Block Chaining
CMAC	Cipher based Message Authentication Code
CrySHE	Cryptographic library module for SHE
DER	Distinguished Encoding Rules
DBGCTRL	system control specific DeBuG ConTRol bridge register
ECB	Electronic Code Book
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography

Abbreviation	Description
Ecc	Error correction code
ECDSA	Elliptic Curve Digital Signature Algorithm
EdDSA	Edwards-curve Digital Signature Algorithm
EEPROM	Electrically Erasable Programmable Read Only Memory
FCON	Flash CONfiguration register
FEE	Flash EEPROM Emulation
FWU	FirmWare Update
GCM	Galois/Counter Mode
HMAC	Hash-based Message Authentication Code
HSM	Hardware Security Module
HSM2HT	HSM to Host
HT2HSM	Host to HSM
HW	Hardware
ITU-T	International Telecommunication Union-Telecommunication
ITU-T X.690	ITU-T X.690 is a standard defining the Abstract Syntax Notation One encoding rules
KDF	Key Derivation Function
MAC	Message Authentication Code
MCAL	MicroController Abstraction Layer
NIST	National Institute of Standards and Technology
NIST P-256	Defines a NIST P-256 curve over prime fields
NVM	Non Volatile Memory
OTP	One Time Programmable
PFlash	Program Flash
PKC	Public Key Cryptography
PKCS	Public-Key Cryptography Standards
PRNG	Pseudo Random Number Generator
RSA	Rivest–Shamir–Adleman public-key cryptosystem
RSA-PSS	RSA Probabilistic Signature Scheme
RSA-PKCS1	Signature Scheme standardized in version 1.5 of PKCS#1
SHA	Secure Hash Algorithm

Abbreviation	Description
SHE / SHE+	Secure Hardware Extension as specified by the *Herstellerinitiative Software (HIS)*, an association of German automobile manufacturers
SSW	Start-up Software
TRNG	True Random Number Generator
UCB	User Configuration Block
UID	Unique IDentification
VKMS	Vehicle Key Management System
X.509	X.509 is a standard defining the format of public key certificates

Table 1.1. Abbreviations

2. Release notes

2.1. Version history

Module version	Change date	Description
Release v1.17.67	2021-12-22	DD release for TRICORE TC38XQ. Support for reduced size HSM (less than 64 kB).
Release v1.17.47	2021-09-03	RFM release for TRICORE TC38XQ. ECDH and HKDF functionality to support ECIES. Key range parameter added to MAC. New VKMS key list.
Release v1.17.42	2021-07-30	DD release for TRICORE TC38XQ. Support for ECIES.
Release v1.17.40	2021-07-16	RFM release for TRICORE TC38XQ.
Release v1.17.39	2021-07-02	DD release for TRICORE TC38XQ.
Release v1.17.34	2021-06-25	RFM release for TRICORE TC39XX.
Release v1.17.33	2021-06-15	DD release for TRICORE TC39XX. Support for secure boot update using asymmetric cryptography and signature verification.
Release v1.17.32	2021-06-10	RFM release for TRICORE TC39XX.
Release v1.17.30	2021-06-01	RFM release for TRICORE TC39XX. Support for secure boot update using asymmetric cryptography and signature verification.
Release v1.17.28 (TC39XX)	2021-06-04	DD release for TRICORE TC39XX.
Release v1.17.27	2021-05-21	RFM release for TRICORE TC23X.
Release v1.17.26	2021-06-16	RFM release for TRICORE TC36XD.
Release v1.17.25	2021-05-21	DD release for TRICORE TC36XD. New VKMS key list.
Release v1.17.24	2021-04-23	RFM release for TRICORE TC39XX.
Release v1.17.17	2021-04-15	DevDrop for TRICORE TC39XX. Support for SOTA dual banking and DMKM extra keys.
Release v1.17.16	2021-04-16	RFM release for TRICORE TC38XQ.
Release v1.17.13	2021-04-27	RFM release for TRICORE TC37XT. Add VKMS 2.2 support.
Release v1.17.12	2021-03-26	RFM release for TRICORE TC39XX. Support for SOTA dual banking.
Release v1.17.11	2021-03-31	DevDrop release for TRICORE TC38XQ. Support the derivation of TLS session keys based on an available PSK.

Module version	Change date	Description
Release v1.17.7	2020-12-09	RFM release for TRICORE TC33XL. Add TC33XL support.
Release v1.16.8	2020-11-27	RFM release for TRICORE TC38XQ. New VKMS key list.
Release v1.16.7	2020-11-20	DevDrop release for TRICORE TC38XQ. New VKMS key list.
Release v1.17.5	2020-11-13	RFM release for TRICORE TC23X. Support for High tech compiler.
Release v1.17.4	2020-10-27	HMAC-SHA256 support for TRICORE TC3XX added.
Release v1.17.3	2020-10-23	DevDrop release for TRICORE TC23X. Support for High tech compiler.
Release v1.7.10	2020-10-21	RFM release for TRICORE TC277. Corrections into following apis: eb_-hsm_aes, eb_hsm_cmac, eb_hsm_stopHsmExecution_fromFlash.
Release v1.16.6	2020-10-16	RFM release for TRICORE TC37XT. Fix for returning version information of initial HSM FW application image. Fix for firmware update process when the encrypted FW image is sent in a single block.
Release v1.17.1	2020-10-13	RFM release for TRICORE TC36XD. Same feature set as 1.17.0.
Release v1.16.5	2020-10-01	RFM release for TRICORE TC37XT. Support for bootloader and firmware version information retrieval. Update for VKMS key IDs. Fix for firmware update process when the encrypted FW image is sent in a single block.
Release v1.17.2	2020-09-30	DevDrop release for TRICORE TC23X. Secure event logging, diagnostic interface, and AES-GCM added.
Release v1.17.0	2020-09-23	Added support for TC36XD. Development Drop.
Release v1.16.4	2020-07-31	RFM release for TRICORE TC37XT. New API for retrieving bootloader version information.
Release v1.16.3	2020-07-03	RFM release for TRICORE TC38XQ.
Release v1.16.2	2020-06-04	RFM release for TRICORE TC37XT. Missing signature generation algorithms added.
Release v1.16.1	2020-05-29	DevDrop for SPC58XC
Release v1.16.0	2020-06-05	Port for the TC38XQ version. Memory map updated. Development Drop.
Release v1.16.0	2020-05-08	New features Vehicle Key Management System (VKMS) and SipHash24. Development Drop.
Release v1.14.1	2020-04-22	RFM release of the HSM port to TC375
Release v1.12.2	2020-04-03	New fix for "reading from erased PFLASH" issue, using Infineon's marker concept. Development Drop.
Release v1.12.1	2020-03-28	First fix solution for "reading from erased PFLASH" issue. Development Drop.

Module version	Change date	Description
Release v1.10.3	2020-03-25	RFM release for TRICORE TC29XT.
Release v1.12.0	2020-02-06	New feature CVC certificate, and up to 50 symmetric keys support. Development drop.
Release v1.10.2	2019-12-19	RFM release of Secure Boot Extension.
Release v1.10.1	2019-11-29	RFM release of Firmware Update.
Release v1.8.1	2019-09-30	New feature Secure Boot Extension. Development drop.
Release v1.8.0	2019-09-27	Firmware Update. Development drop for TRICORE TC38X.
Release v1.7.8	2019-12-13	RFM release. Stop-Resume HSM execution (from PFlash) for concurrent PFlash operations, using the XORShift128 algorithm beforehand in order to activate the feature.
Release v1.7.7	2019-11-29	DevDrop release. Bug fix for getting incorrect cipherTextLength on Cry_She_SymEncryptUpdate() call.
Release v1.7.6	2019-11-11	Stop-Resume HSM execution (from PFlash) for concurrent PFlash operations, using the XORShift128 algorithm beforehand in order to activate the feature. For this feature, the challenge-response protocol (released as part of v1.7.4) is now replaced with the XORShift128 algorithm. Development drop for TRICORE TC277.
Release v1.7.5	2019-08-09	Quality improvements for previous release. Development drop.
Release v1.7.4	2019-07-31	Stop-Resume HSM execution (from PFlash) for concurrent PFlash operations, using the challenge-response protocol beforehand in order to activate the feature.
Release v1.7.3	2019-07-19	Support for TRICORE TC275. Development drop.
Release v1.7.2	2019-07-09	Stop-Resume HSM execution (from PFlash) for concurrent PFlash operations. Development drop for TRICORE TC277.
Release v1.7.1	2019-05-29	HW-accelerated ECC key generation for NIST P-256, but allowing asymmetric key pair generation only once.
Release v1.7.0	2019-05-10	HW-accelerated ECC key generation for NIST P-256. Development drop for TRICORE TC3XX.
Release v1.6.3	2019-04-26	Support for TRICORE TC38X.
Release v1.6.2	2019-04-12	Support for TRICORE TC38X. Development drop.
Release v1.6.1	2019-03-05	Support for TRICORE TC3XX and new features, Secure Boot and Certificate Management.
Release v1.6	2019-02-18	Support for TRICORE TC3XX and new features, Secure Boot and Certificate Management. Development drop.

Module version	Change date	Description
Release v1.6	2019-02-11	Support for TRICORE TC27X and new feature Secure Boot.
Release v1.4	2018-10-22	Basic feature set according to SHE+, symmetric HW-accelerated crypto (AES, CMAC), random number generation, key management.
Release v1.2	2018-07-06	Basic feature set according to SHE+, development drop.
Release v1.0	2018-03-29	Initial version, development drop.

Table 2.1. Change log

2.2. New features

This section lists major changes between releases.

- In release **1.17.47** added
  - Key range parameter added to eb_hsm_mac().
  - New VKMS key list with added key for FOD (Function on Demand).
- In release **1.17.42** added
  - ECDH and HKDF implemented to support elliptic curve integrated encryption scheme (ECIES).
- In release **1.17.33** added
  - Support for secure boot update using asymmetric cryptography and signature verification.
- In release **1.17.30** added
  - Support for secure boot update using asymmetric cryptography and signature verification.
- In release **1.17.17** added
  - Support for DMKM extra keys for TC39XX.
- In release **1.17.13** added
  - Support for VKMS 2.2
- In release **1.17.12** added
  - Support for SOTA dual banking for TC39XX.
- In release **1.17.11** added
  - Support the derivation of TLS session keys based on an available PSK.
- In release **1.17.7** added
  - Support for new hardware: TC33XL
- In release **1.17.3** added

- HMAC-SHA256 with a key length of 128-bit.

- In release **1.17.2** added
  - Secure Event Logging
  - Diagnostic interface
  - AES-GCM
- In release **1.17.0** added
  - Support for new hardware: TC36XD
- In release **1.16.6** added
  - RFM release for TC37XT.
- In release **1.16.5** added
  - Support for bootloader and firmware version information retrieval.
- In release **1.16.4** added
  - Support for retrieving bootloader version information
- In release **1.16.0** added
  - Support for Vehicle Key Management System (VKMS)
  - Support for SipHash24
- In release **1.10.3** added
  - Support for new hardware: TC29XT
- In release **1.8.1** added
  - New feature: Secure boot extension. See details in Section 3.6.3, “Secure boot extension”.
- In release **1.8.0** added
  - Firmware Update.
- In release **1.7.6** added
  - Use of the XORShift128 algorithm instead of the challenge-response protocol (released as part of v1.-7.4) in order to protect accessing the stop HSM API for concurrent PFlash operations
- In release **1.7.5** added
  - Quality improvements of Stop-Resume feature implementation
- In release **1.7.4** added
  - Use of the challenge-response protocol in order to protect accessing the stop HSM API for concurrent PFlash operations
- In release **1.7.3** added
  - Support for new hardware: TC275

- In release **1.7.2** added
  - Stop-Resume HSM execution from PFlash for concurrent PFlash operations
- In release **1.7.1** added
  - Fix for key generation: allow asymmetric key pair generation only once
- In release **1.7.0** added
  - ECC NIST P-256 HW-accelerated key generation
- In release **1.6.3** added
  - Bug fix for eb_hsm_sign() where missing cache flush causes the use of outdated data
- In release **1.6.2** added
  - Support for new hardware: TC38X
- In release **1.6.1** added
  - New feature: Certificate management. See details in Section 3.7, “Certificate Management”.
- In release **1.6** added
  - Support for new hardware: TC27X
  - New feature: Secure boot. See details in Section 3.6, “Secure boot”.

2.3. Limitations

This section lists the limitations for EB zentur HSM Firmware.

- Input buffers, containing the messages to be processed, must be 16-byte aligned. Output buffers, receiving the encrypted/decrypted messages, must be 4-byte aligned.

  Length variable(s) for input/output buffer, like signLengthPtr in eb_hsm_sign(), must be 16-byte aligned and must have a size of at least 16 bytes.
- Number of supported asymmetric keys:
  - 1 ECDSA private key
  - 1 EdDSA private key
  - 9 ECC public keys
  - 1 RSA private key
  - 4 RSA public keys
- RSA key length of 1024 bits is not supported.
- For MAC verification with the key BOOT_MAC_KEY the MAC length must be 128 bits.
- For ECC key generation, only NIST P-256 curve is supported.

- Only one ECC key pair (generated using the hardware accelerator) can be stored.
- Key generation of an asymmetric key pair for a certain key slot (only 1 available currently) can be executed only once to avoid overwriting. If triggered again, the error COMM_KEY_UPDATE_ERROR is thrown.
- The FWU feature was not tested with the signature algorithms CMAC and EDDSA, see Table 3.4, “FWU data profile parameters” (Signature Algo - Tag=0x84).

2.4. Supported architectures

The EB zentur HSM Firmware supports different hardware architectures and derivatives, which are listed in Table 2.2, “Supported architectures and derivatives”.

Architecture	Derivate	Compiler	Feature set
Infineon TriCore 2xx	TC23x	GCC 6.3.1	SHE/SHE+, Secure Boot, Secure Boot extension, Secure Event Logging, Diagnostic Interface
Infineon TriCore 2xx	TC27x	GCC 6.3.1	SHE/SHE+, Secure Boot, Secure Boot extension
Infineon TriCore 2xx	TC29x	GCC 6.3.1	SHE/SHE+, Secure Boot, Secure Boot extension
Infineon TriCore 3xx	TC39x	GCC 6.3.1	SHE/SHE+, Secure Boot, Secure Boot extension, Certificate management, ECC key generation, FWU, SipHash, HMAC-SHA256
Infineon TriCore 3xx	TC38x	GCC 6.3.1	SHE/SHE+, Secure Boot, Secure Boot extension, Certificate management, ECC key generation, FWU, SipHash, HMAC-SHA256, VKMS
Infineon TriCore 3xx	TC37x	GCC 6.3.1	SHE/SHE+, Secure Boot, Secure Boot extension, Certificate management, ECC key generation, FWU, SipHash, HMAC-SHA256, VKMS
Infineon TriCore 3xx	TC36x	GCC 6.3.1	SHE/SHE+, Secure Boot, Secure Boot extension, Certificate management, ECC key generation, FWU, SipHash, HMAC-SHA256
STM Chorus family	SPC58XC	GHS 7.1.4	SHE/SHE+, Secure Boot, Secure Boot extension

Table 2.2. Supported architectures and derivatives

2.5. Compiler

This chapter lists the compilers used in testing the release.

2.5.1. Tested compilers

This delivery was tested with the following compiler:

- GNU tools for ARM 6-2017-Q2-update (6.3.1 20170620)

2.5.2. Compiler options

This summarizes under which conditions the EB zentur HSM Firmware is built. Software is tested using these compiler options.

Target HW	Options
Infineon AURIX TC23X	`-Wdeclaration-after-statement -Werror -W -Wall -Wextra -Wshadow -Wpointer-arith -Wcast-qual -Wmissing-prototypes -Wstrict-prototypes -Wno-pointer-sign -fno-stack-protector -fno-strict-aliasing -fno-strict-overflow -fno-common -ffreestanding -nostdinc -march=armv7-m -mcpu=cortex-m3 -mlittle-endian -mfix-cortex-m3-ldrd -mthumb -O2 -fomit-frame-pointer -DEBHSM_FW -DHAS_64BIT_TYPES -UEBHSM_USE_PRNG -DEBHSM_FLASH_ENABLE -UENABLE_X509 -UENABLE_AES_SW -UENABLE_EC -UENABLE_ED -UENABLE_HASH -UENABLE_RSA -UENABLE_TC23X_FLASH_ECC_TRAPS -DNDEBUG -DDEBUGMODE=DEBUG_NONE -UTESTING -DTOOL=TOOL_GCC -DARCH=ARCH_TC2XX -DARCH_DERIVATE_TC23X`
Infineon AURIX TC27X	`-Wdeclaration-after-statement -Werror -W -Wall -Wextra -Wshadow -Wpointer-arith -Wcast-qual -Wmissing-prototypes -Wstrict-prototypes -Wno-pointer-sign -fno-stack-protector -fno-strict-aliasing -fno-strict-overflow -fno-common -ffreestanding -nostdinc -march=armv7-m -mcpu=cortex-m3 -mlittle-endian -mfix-cortex-m3-ldrd -mthumb -O2 -fomit-frame-pointer -DEBHSM_FW -DHAS_64BIT_TYPES -UEBHSM_USE_PRNG -DEBHSM_FLASH_ENABLE -UENABLE_X509 -UENABLE_AES_SW -UENABLE_EC -UENABLE_ED -UENABLE_HASH -UENABLE_RSA -DENABLE_TC23X_FLASH_ECC_TRAPS -DNDEBUG -DDEBUGMODE=DEBUG_NONE -UTESTING -DTOOL=TOOL_GCC -DARCH=ARCH_TC2XX`
Infineon AURIX TC29XT	`-Wdeclaration-after-statement -Werror -W -Wall -Wextra -Wshadow -Wpointer-arith -Wcast-qual -Wmissing-prototypes -Wstrict-prototypes -Wno-pointer-sign`

	-fno-stack-protector -fno-strict-aliasing -fno-strict-overflow -fno-common -ffreestanding -nostdinc - -march=armv7-m -mcpu=cortex-m3 -mlittle-endian -mfix-cortex-m3-ldrd -mthumb -O2 -fomit-frame-pointer -DEBHSM_FW -DHAS_64BIT_TYPES -DPRNG_KEY_VOLATILE -UEBHSM_USE_PRNG -DEBHSM_FLASH_ENABLE -UENABLE_FW_UPDATE -DBL_VERSION_MATCH -UENABLE_BOOT_LOADER -DSTATIC_FW -UENABLE_KEY_GENERATION -USIGNATURE_SW_ECCNIST -UENABLE_X509 -UENABLE_CVC -UENABLE_DERFWU -UFWU_SYM_ALGOS -UENABLE_AES_SW -UENABLE_EC -UENABLE_ED -UENABLE_RSA -UENABLE_HASH -DENABLE_TC23X_FLASH_ECC_TRAPS -DNDEBUG -DDEBUGMODE=DEBUG_NONE -UTESTING -DTOOL=TOOL_GCC -DARCH=ARCH_TC2XX -DARCH_DERIVATE_TC29X
Infineon AURIX TC3XX	-Wdeclaration-after-statement -Werror -W -Wall -Wextra -Wshadow -Wpointer-arith -Wcast-qual -Wmissing-prototypes -Wstrict-prototypes -Wno-pointer-sign -fno-stack-protector -fno-strict-aliasing -fno-strict-overflow -fno-common -fno-jump-tables -ffreestanding -nostdinc -march=armv7-m -mcpu=cortex-m3 -mlittle-endian -mfix-cortex-m3-ldrd -mthumb -O2 -fomit-frame-pointer -DEBHSM_FW -DHAS_64BIT_TYPES -UEBHSM_USE_PRNG -DEBHSM_FLASH_ENABLE -DENABLE_FW_UPDATE -DBUILD_FWU_IHU -DRELOC_FW -DENABLE_KEY_GENERATION -USIGNATURE_SW_ECCNIST -UENABLE_CVC -UFWU_ASYM_ALGOS -UENABLE_AES_SW -DENABLE_HASH -DENABLE_TC23X_FLASH_ECC_TRAPS -DNDEBUG -DDEBUGMODE=DEBUG_NONE -UTESTING -DTOOL=TOOL_GCC -DARCH=ARCH_TC3XX -DHSM_APP_SIZE=0x20000UL -DBL_START_ADDR=0x80008000UL
STMicroelectronics SPC58XC	--quit_after_warnings --remarks --ghstd=last --diag_suppress=1824 --diag_suppress=1798 --diag_suppress=2021 --diag_suppress=2023 --diag_suppress=32 --diag_suppress=2003 --diag_suppress=1846 --diag_suppress=1890 --diag_suppress=7 --warnings -Wimplicit-int -Wshadow -Wtrigraphs -Wundef -cpu=core_ppce200z0 -vle -sda=all -nostdinc -nostdlib --gnu_asm -O -DEBHSM_FW -UEBHSM_USE_PRNG -DEBHSM_FLASH_ENABLE -UENABLE_SYM_KEY_EXTENSION -UENABLE_FW_UPDATE -DBL_VERSION_MATCH -UENABLE_BOOT_LOADER -DSTATIC_FW -UENABLE_VKMS -UENABLE_KEY_GENERATION -USIGNATURE_SW_ECCNIST -UENABLE_X509 -UENABLE_CVC -

	UENABLE_DERFWU -UFWU_SYM_ALGOS -UENABLE_AES_SW -UENABLE_EC -UENABLE_ED -UENABLE_RSA -UENABLE_SIPHASH -UENABLE_AES_GCM -UENABLE_HASH -DENABLE_TC23X_FLASH_ECC_TRAPS -UENABLE_STOPRESUME_HSM -DSTOPRESUME_HSM_ALGO_TYPE_XORSHIFT -USTOPRESUME_HSM_ALGO_TYPE_CHALLENGE_RESPONSE -DNDEBUG -DDEBUGMODE=DEBUG_NONE -UTESTING -DTOOL=TOOL_GHS -DARCH=ARCH_PA

3. Supported features

3.1. Feature overview

This chapter provides information on EB zentur HSM Firmware features and related itemized cryptographic services. Additionally an overview on the SHE compatibility is given.

You will find the following information in this chapter:

- Section 3.2, “EB zentur HSM Firmware features” lists all features supported by EB zentur HSM Firmware.
- Section 3.3, “Cryptographic features” lists all supported cryptographic features.
- Section 3.4, “SHE Compatibility” explains the compatibility of the EB zentur HSM Firmware with the SHE standard.
- Section 3.5, “Key Generation” explains the key generation functionality of the EB zentur HSM Firmware.
- Section 3.6, “Secure boot” explains the Secure Boot handling of the EB zentur HSM Firmware for verification of host bootloader.
- Section 3.7, “Certificate Management” explains the handling of certificate in the EB zentur HSM Firmware.
- Section 3.8, “Concurrent PFlash Operations” explains the handling of stop-resume HSM operations (from PFlash) in order to facilitate concurrent PFlash operations.
- Section 3.9, “Firmware update” explains how to update a firmware application image and how to roll back to a previous firmware application image.
- Section 3.10, “Challenge-response protocol” explains the challenge-response protocol for authenticating the users of the HSM Firmware.
- Section 3.11, “Life Cycle Management” explains restrictions and actions on each state.
- Section 3.12, “Diagnostic interface” explains the diagnostic interface functionality.

3.2. EB zentur HSM Firmware features

The EB zentur HSM Firmware supports a basic feature set that is aligned to SHE and SHE+. It can be extended according to the hardware possibilities. The EB zentur HSM Firmware supports the following basic feature sets:

- Resource management for hardware resources, e.g. TRNG, SHA, AES or PKC for TC3XX
- Key management with symmetric AES based keys with 128 bits (see Section 3.4.2, “Key types and supported keys”)

  - 1 Secret key slot
  - 1 Master ECU key slot
  - 1 Boot MAC key slot
  - 1 Boot MAC slot
  - 1 RAM key slot
  - TC3XX: 50 NVM Flash key slots

    Other derivates: 20 NVM Flash key slots
  - Ciphered key container load according to SHE specification v1.1
- Key management for asymmetric keys with maximum storage space of 800 bytes (1200 bytes for RSA). See restrictions (see Section 3.11, “Life Cycle Management”)
  - ECDSA and EdDSA private keys
  - ECC public keys
  - RSA private and public key

Depending on the support of the target device, cryptographic algorithms use the available hardware accelerators. Depending on the project scope, additional crypto algorithms can be integrated into the EB zentur HSM Firmware. The following crypto algorithms are supported:

- With use of hardware accelerators:
  - Cipher-based message authentication code (CMAC) generation and verification, AES-128 based
  - Symmetric encryption and decryption using AES with a key length of 128-bits. The following modes are supported: ECB, CBC and GCM. AES interface is a single call interface.
  - Pseudo Random number generation (PRNG) of 128 bits, including generation of a random seed through a TRNG and random seed extension
  - Loading plain key into RAM key slot
  - Loading ciphered keys (in: M1-M3, out: M4-M5) into RAM key slot and non-volatile key slots
  - Exporting RAM key as ciphered key container
  - SHE command CMD_DEBUG with restriction that it can only be used to delete all stored key in the flash memory
  - HMAC-SHA256 generation and verification with a 128-bit key.
  - HASH generation SHA2-256
  - ECC key pair generation using
    - ECDSA NIST curve P-256 (secp256r1)
  - Exporting ECC public key in a raw format

- Digital signature generation and verification using
  - ECDSA NIST curve P-256 (secp256r1)
- With use of software:
  - Digital signature generation using
    - EdDSA Ed25519
    - RSASSA-PSS
    - RSASSA-PKCS1-v1_5
  - Digital signature verification using
    - EdDSA Ed25519
    - RSASSA-PSS
    - RSASSA-PKCS1-v1_5
  - SipHash
    - Supports Siphash-2-4 and SipHash-4-8 families
    - Supports user definable 128-bit key
    - Supports 64-bit message authentication code

3.3. Cryptographic features

Table 3.1, “Cryptographic services” lists all cryptographic services supported by EB zentur HSM Firmware.

Cryptographic service	Description
Digital Signatures	Generate or verify digital signature using asymmetric cipher suite
Export Key	Export the RAM KEY into a format protected by SECRET KEY. The Key can be imported again via a Secure Key Update.
Export Public Key	Export the EC public key into a raw format. The key can only be exported. Since, currently we only support one key-slot for ECC NIST P-256 key pair (generated via hardware accelerator), the key can not be modified after its creation.
Generate Random Number	Generates a vector of 128 random bits
Hashing	SHA-2 256 bits
Key Generation	Generate a NIST ECC P-256 key pair using the hardware accelerator.
MAC Generation	Generate a MAC of a given message with the help of a given key.

Cryptographic service	Description
MAC Verification	Verify a MAC of a given message with the help of a given key against a provided MAC.
Plain Key Update	A given key is loaded (without encryption and verification of the key), so the key is handed over in plain text. The plain key can only be loaded into the RAM KEY slot.
RNG initialization	Initialize the seed and RNG and derive a key for the PRNG
Secure Key Update	Update/Load an internal/protected/ciphered key.
Symmetric Block Decryption, ECB and CBC	Decrypt a given ciphered text with a given key and return a plain text (which can have a multiple length of block width 128bit).
Symmetric Block Encryption, ECB and CBC	Encrypt a given plain text (which can have a multiple length of block width 128bit) with a given key and return a ciphered text.
Symmetric Decryption	Decrypt a given ciphered text with a given key and return a plain text.
Symmetric Encryption	Encrypt a given plain text with a given key and return a ciphered text
Symmetric Stream Ciphering, GCM	GCM is essentially a stream cipher in which an input is combined with a keystream, which allows operation with a variable length input.

Table 3.1. Cryptographic services

3.4. SHE Compatibility

EB zentur HSM Firmware is compatible with the SHE specification v1.1, see [SHE_Spec].

3.4.1. SHE+ requirements

In addition to SHE, EB zentur HSM Firmware implements the following requirements. This is commonly known as SHE+:

- Additional key slots (SHE+ extended keys)

  Additionally to the 10 generic key slots specified by SHE v1.1., EB zentur HSM Firmware implements additional generic key slots: - For TC3XX: 40 key slots, resulting in 50 supported generic key slots total. - For other variants: 10 key slots, resulting in 20 supported generic key slots total.

- Verify-only key property flag

  This additional flag can be specified to any regular SHE key or any of the extended SHE+ keys. If this flag is set, then SHE will restrict usage of the key to verifying MACs only.

3.4.2. Key types and supported keys

This section illustrates different key types supported by EB zentur HSM Firmware. SHE conceptually distinguishes the following key types:

- ROM key

  This is a non-updateable key that is stored in the read-only memory area. SHE specifies one single ROM key, the SHE SECRET_KEY. The ROM key is unique to the chip.

- NVM keys

  These are non-volatile memory (NVM) keys. The stored NVM keys persist through a power cycle.

- RAM key

  This is a volatile key held in secure RAM area. SHE specified one single RAM key slot, the SHE RAM_KEY.

EB zentur HSM Firmware supports the SHE standard keys specified in Table 3.2, “SHE keys”:

Key name	Key type	Key ID	SHE / SHE+
SECRET_KEY	ROM key	0x0	SHE
MASTER_KEY	NVM key	0x1	SHE
BOOT_MAC_KEY used by Secure Boot	NVM key	0x2	SHE
BOOT_MAC used by Secure Boot	NVM key (MAC calculated)	0x3	SHE
KEY_1 - KEY_10	NVM key	0x4 - 0xd	SHE
RAM_KEY	RAM key	0xe	SHE
KEY_11 - KEY_20 (*)	NVM key	0xf - 0x18	SHE+

Table 3.2. SHE keys

(*) KEY_15 is used for the secure logging feature on TC23X if activated

3.4.3. Deviations

The following deviation to the SHE specification 1.1 exists:

- BOOT_MAC is write-protected from loading outside the HSM

  A CMAC of the host bootloader used in secure boot is stored in BOOT_MAC. It can be written with the knowledge of the provided image's signature. This violates the specification chapter 4.2.3, which states that BOOT_MAC can be written with the knowledge of the MASTER_ECU_KEY or BOOT_MAC_KEY and is protected by the common lock mechanisms described in chapters 4.1.1, 4.1.2, 4.1.3, and 4.1.4.

3.5. Key Generation

This feature provides ECC key generation (currently only on NIST P-256 Curve) on HSM. The main purpose is to generate a key pair and store a private key on the device that never leaves HSM. The related public key can be exported to the host.

The complete process of key generation is carried out using two subsequent steps. In the first step, the private key is generated using PRNG that involves TRNG for the seed generation. The generation of the private key is as described in the SHE specification chapter 5.1 [SHE_Spec]. In the second step, the private key along with several ECC domain parameters is fed to the hardware accelerator to perform scalar multiplication in order to generate the related public key. The private key is stored on the device and never exposed to the outside environment. The public key is delivered to the host. It is also stored to HSM as back up but it is never used in any of the crypto operations in HSM. However, the public key can be exported to host on a request.

There is currently only one key slot available to the key pair generated using this feature. The generated key pair is used to sign the data and to verify the signature. Once the key pair is generated, the slot is marked as occupied. The generated private key is used to sign the data (e.g. certificate data). The key generation process can not be executed more than one time to avoid overwriting. If triggered again, an error (COMM_-KEY_UPDATE_ERROR) is thrown.

3.5.1. Key Generation Configuration

ECC key pair is stored to the private key slot. There is only 1 slot available at the moment.

3.5.2. Key Generation Supported Keys

EB zentur HSM Firmware supports NIST P-256, an ECC curve over prime field curve.

3.6. Secure boot

The main purpose of secure boot is to ensure the integrity of the application software that runs on the host. The architecture is based on the SHE specification but adapted to the context of EB tresos and EB zentur HSM Firmware on a Classic AUTOSAR system on chip (SoC).

The feature involves two steps that are described in the following sections:

- Secure boot to validate the host bootloader. For the two phases of this step, see
  - Section 3.6.1, “Configuration of the secure boot process”
  - Section 3.6.2, “Secure boot procedure in normal operation mode”

- Secure boot extension to validate the host software applications. See Section 3.6.3, “Secure boot extension”.

The feature flowchart is shown in the following pictures:

- Secure boot configuration: Figure 3.1, “Secure boot configuration”
-
- Secure boot procedure: Figure 3.2, “Secure boot procedure”
- TC3XX specific secure boot procedure: Figure 3.3, “TC3XX secure boot procedure”

The validation of the EB zentur HSM Firmware image itself is not within the scope of the secure boot functionality of EB zentur HSM Firmware. To preserve the chain of trust, validation needs to happen before the EB zentur HSM Firmware is initialized. The verification of EB zentur HSM Firmware may be provided by an HSM bootloader, a SoC HSM boot ROM, the OTP protection of the EB zentur HSM Firmware flash, or by other mechanisms supported by the hardware.

TC3XX supports the HSM FW image secure boot. Initially the configuration is done when the HSM iHU image is loaded and the MAC is calculated over the HSM image. This is done during the production phase in a safe environment. The verification is then done when the HSM bootloader is started and HSM FW verified. An update of the calculated MAC is executed when the firmware update of the HSM is finished. For more details about the phases, see Section 3.9, “Firmware update”.

For the secure boot procedure, BOOT_MAC_KEY and BOOT_MAC are important. Both are empty after production and protected by the common lock mechanisms:

- BOOT_MAC_KEY
  - The BOOT_MAC_KEY is used for secure booting to verify the integrity and authenticity of the software and to verify a CMAC. The BOOT_MAC_KEY must be configured.
  - The BOOT_MAC_KEY can be loaded/updated when the MASTER_ECU_KEY or the BOOT_MAC_KEY itself is known by the user.
  - When changing the BOOT_MAC_KEY, the BOOT_MAC is automatically recalculated in the EB zentur HSM Firmware.
- BOOT_MAC
  - The BOOT_MAC is treated like any other key and considered to be a secret information.
  - The BOOT_MAC stores the CMAC of the host bootloader in the secure boot procedure.

3.6.1. Configuration of the secure boot process

This procedure must be run if the secure boot is not configured yet. The secure boot process has to be initiated by using the `eb_hsm_secure_boot()` API and providing the memory range of the host bootloader to

check, i.e. start address and data length. The start address of the memory range must be 16-bytes aligned. A BOOT_MAC_KEY must be programmed beforehand. If this key is not programmed, the secure boot process is not configured and the return value is COMM_NO_SECURE_BOOT. During initialization, the EB zentur HSM Firmware calculates and stores the BOOT_MAC. If this procedure completes successfully, COMM_NO_ERROR is returned.

3.6.2. Secure boot procedure in normal operation mode

During normal operation, once configured, the secure boot procedure is executed during HSM start-up. If the verification of the internally stored BOOT_MAC passes, the secure boot process concludes to a preliminary OK. It may be supplemented with further information received from the host side using the APIs `eb_hsm_boot_ok()` or `eb_hsm_boot_failure()`. The first locks the outcome of the secure boot process to OK, whereas the second transforms it to FAIL.

If the verification of the internally stored BOOT_MAC fails, the secure boot process status is set to FAIL. Non-volatile cryptographic keys, for which the security flag `Secure boot failure` (chapter 4.1.2 of SHE specification) is used, are set as disabled and not usable.

For TRICORE derivatives: If the SSWWAIT bit in register PROCONHSMCOTP is set, the TRICORE start-up software (SSW) delays the execution of the host application and waits for an acknowledgment from the HSM. If the secure boot succeeds, the host starts. If the secure boot fails, the EB zentur HSM Firmware disables the communication between host and HSM.

3.6.3. Secure boot extension

The main purpose of the secure boot extension is to provide the validation of host SW applications. First, the range of memory of the host SW application must be provided. This is done initially in production phase as described in Section 3.6.4, “Secure boot extension add”. When the host application is updated, the memory block information must be updated as described in . An existing memory block of the host SW application can be verified as per Section 3.6.5, “Secure boot extension verify”. If the result is PASS, the host SW application can be executed. If the result is FAIL, the host SW application is not allowed to run.

For the secure boot extension procedure, a key for each memory block must be used.

- Memory block key
  - The key is a symmetric key. It is used for secure boot extension to verify the integrity and authenticity of the host software memory block and to verify a CMAC. For TC3XX: Valid keys are COMM_KEY_1 to COMM_KEY_50. For other variants: Valid keys are COMM_KEY_1 to COMM_KEY_20.
  - The key must be generated before a memory block is added. If the memory block is updated, the key is regenerated automatically and the CMAC of the memory block is recalculated. Once generated, the key cannot be separately regenerated.

- Memory block information
  - The memory block information is treated like any other HSM internal information and considered to be a secret information.
  - The memory block information stores the memory start address and the size of the host SW application, and the calculated CMAC of the secure boot procedure.

3.6.4. Secure boot extension add

The first step of the secure boot extension is to add the host SW application memory blocks. This is done in the trusted environment and can be called for each memory block only once. A key used for CMAC calculation must be generated before. This is done by calling `eb_hsm_key_gen()`. Then `eb_hsm_mem_block_add()` is called using the same key ID as a parameter. Other parameters are the start address and length of the memory block, and the memory block ID. The start address must be 16-bytes aligned. By using the key and the memory block information, a CMAC is calculated and saved inside the HSM. There can be maximum 5 of the memory ranges for secure boot extension.

3.6.5. Secure boot extension verify

During the secure boot extension verify phase, the stored CMAC is verified against the memory block that is indicated with the memory block ID. This is done by calling `eb_hsm_mem_block_verify()` with the memory block ID. The result of the verification is returned.

3.6.6. Secure boot flow charts

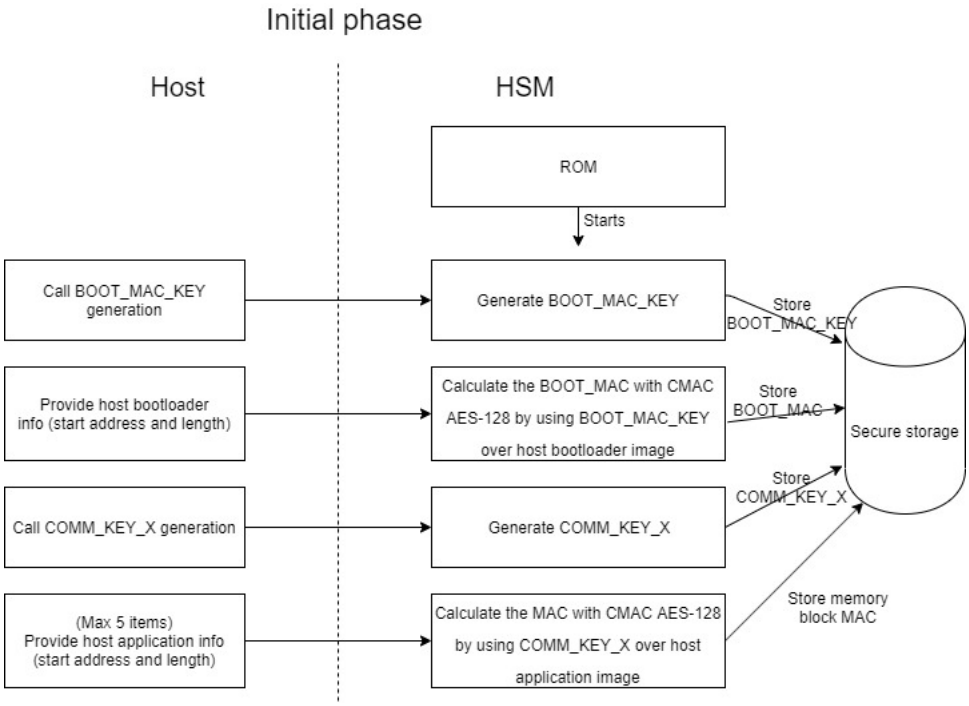


Figure 3.1. Secure boot configuration

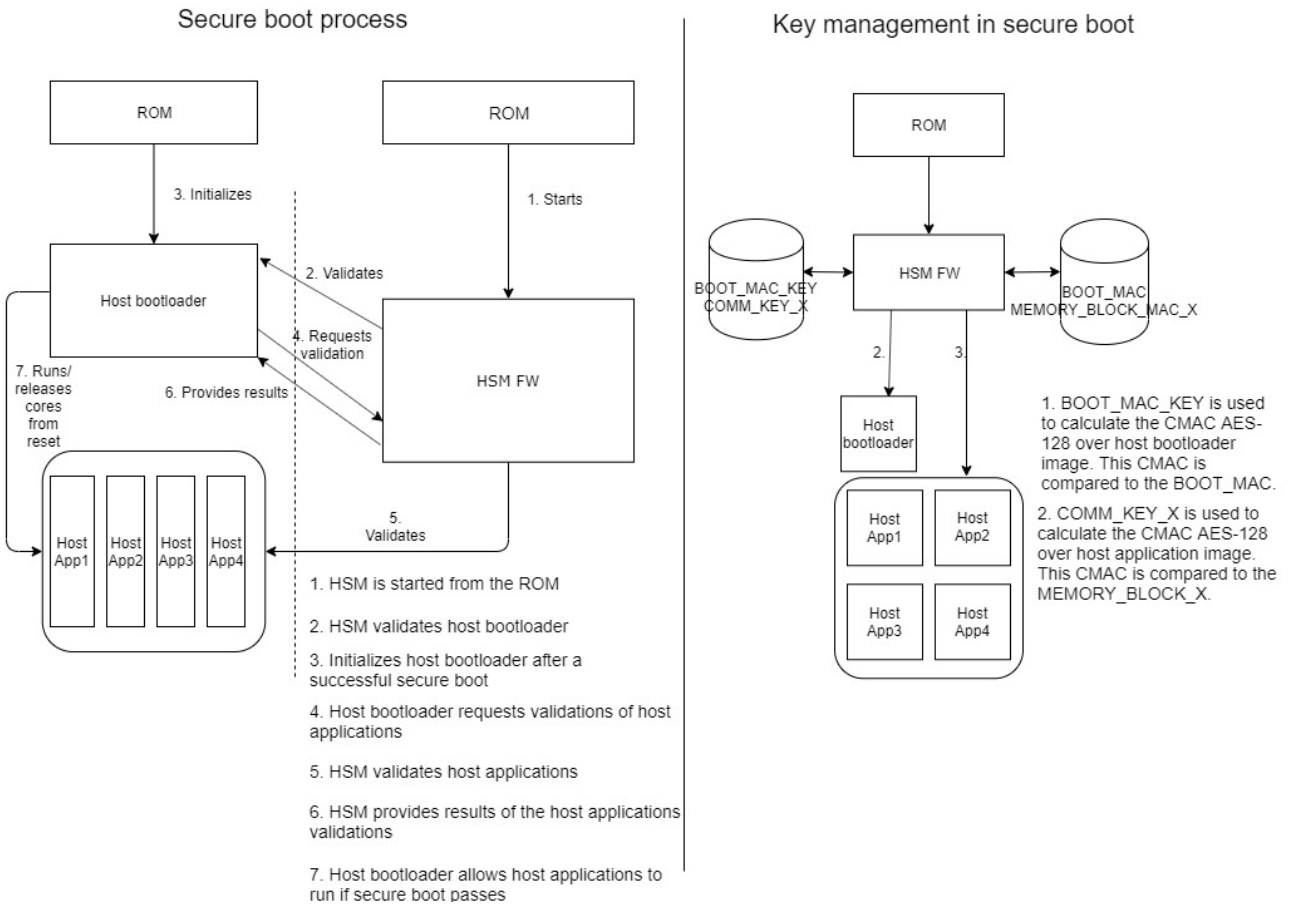


Figure 3.2. Secure boot procedure

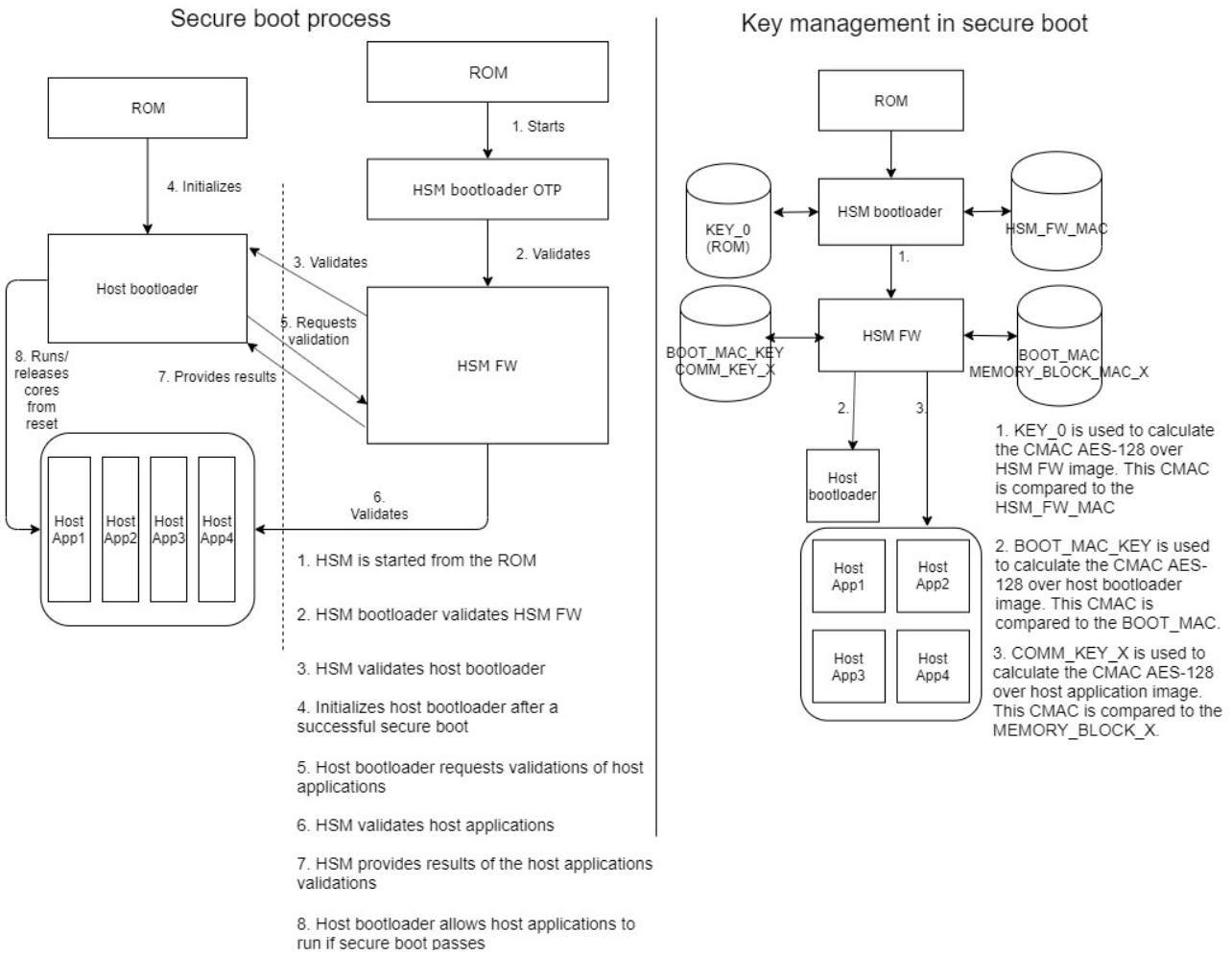


Figure 3.3. TC3XX secure boot procedure

3.6.7. Settings on the host side

The host-side memory regions where the host bootloader and the host applications reside must be initialized. Uninitialized values in the memory regions might cause a secure boot failure due to different results in the MAC calculation.

For TC23XL, both HSM code and host code are in the PFlash. This can cause concurrent access or Ecc bus errors during the execution of the secure boot functions. Errors are prevented by disabling the traps of these errors. In synchronous mode, the traps are enabled at the end of the secure boot function execution. In asynchronous mode, the traps are enabled in a poll function when the HSM returns `COMM_CHANNEL_READY`. In case of errors, traps are enabled at the end of the secure boot function execution.

3.7. Certificate Management

The main purpose of the certificate management is to provide a storage for certificates and a chain of trust in order to verify all public keys used for signature verification. The usage of public keys is invisible to the client. Public keys in certificates are used for signature verification. This means that, when a client requires a signature verification, the key used for a verification is either a raw key or the key is within the certificate.

3.7.1. Certificate configuration

Certificates share the same key slots with asymmetric raw keys. There are 9 slots with the amount of 800 bytes for ECDSA and 4 slots with the amount of 1200 bytes for RSA. It is up to the client to decide the usage of key slots, i.e. how many raw keys and how many certificates are loaded to key slots.

3.7.2. Certificate formats and algorithms

EB zentur HSM Firmware supports X.509v3 ITU-T X.690 Distinguished Encoding Rules (DER) and OTC-CVC-Profile Version 1.0 format certificates. The following algorithms are supported:

- ECDSA NIST P-256
- RSA-PKCS1 v1.5 and RSA-PSS with the key length of 2048 and 3072 bits

3.7.3. Certificate fields

The following X.509 certificate fields are supported:

- Certificate
  - signatureAlgorithm
  - signatureValue
  - TBSCertificate.version == v3(2)
  - TBSCertificate.serialNumber
  - TBSCertificate.signature
  - TBSCertificate.subjectPublicKeyInfo
  - TBSCertificate.extensions
    - AuthorityKeyIdentifier.keyIdentifier
    - SubjectKeyIdentifier.keyIdentifier

- KeyUsage
- BasicConstraints
- ExtendedKeyUsage

The following CVC certificate fields are supported:

- Certificate
  - Profile Identifier
  - Authority Reference
  - Public Key
  - Holder Reference
  - Signature

3.7.4. Certificate loading

The certificates are loaded and updated through `eb_hsm_load_asym_pub_key()`, and the public keys in the certificate are used through `eb_hsm_sign()`. Certificates are verified during their storage to EB zentur HSM Firmware. During an update procedure the certificate is verified and if the validation is successful the previous certificate is overwritten.

Certificates must be loaded hierarchically. This means the Root Certificate must be loaded first in order to validate the following intermediate certificate. When the intermediate certificate is loaded it validates the now following signing certificate. See Figure 3.4, “Certificate chain with intermediate certificate”.

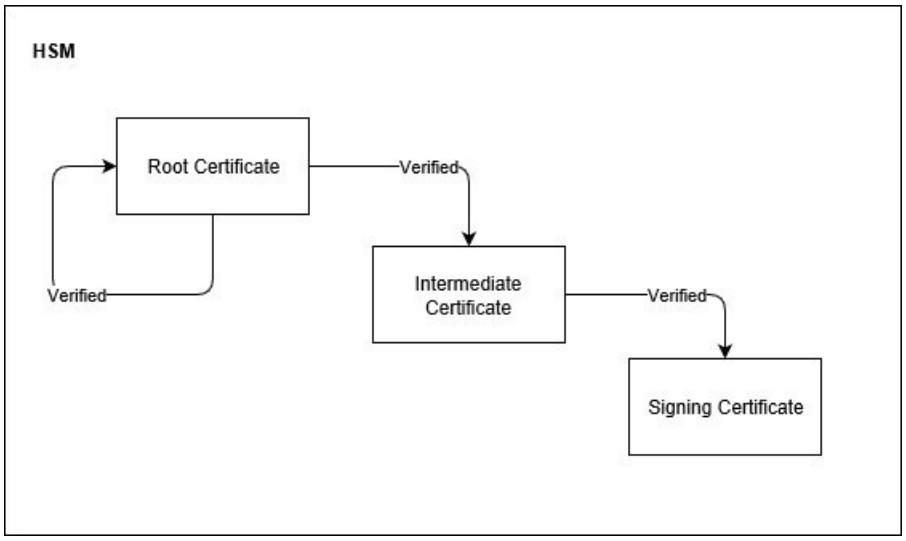


Figure 3.4. Certificate chain with intermediate certificate

If there are only two certificates, the signing certificate is validated by the root certificate directly. See Figure 3.5, “Minimum certificate setup”.

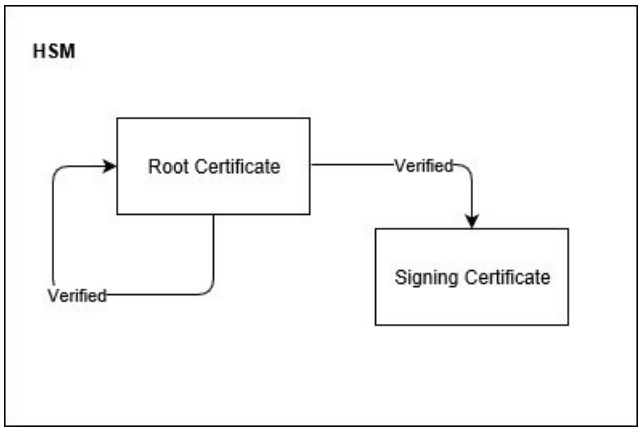


Figure 3.5. Minimum certificate setup

3.8. Concurrent PFlash Operations

If the HSM is running from PFlash and another application, e.g. the host bootloader, tries to do erase/write operations on the same PFlash bank, then the microcontroller generates an exception. To avoid this issue, the HSM needs to be stopped from executing on PFlash. This feature is protected by the XORShift128 algorithm. The host side has to calculate an authorization value based on the XORShift128 algorithm which is verified at HSM side.

The authorization response (required to activate the stop HSM command) is calculated using the XORShift128 algorithm. The calculated response must be passed while requesting to stop execution on PFlash. The intent behind XORShift128 algorithm is to protect from the perspective of safety accidental calls of the stop command due to false code or misconfiguration (any execution error). Only after successful comparison of the two responses (response received from Host and response calculated by HSM), the HSM can be stopped from executing on PFlash so that concurrent PFlash operations are allowed.

This feature includes the implementation of dedicated stop/resume APIs to stop/resume HSM execution from the PFlash. It uses the XORShift128 algorithm in order to activate the stop HSM API. The details regarding the use of XORShift128 algorithm to generate the response and the steps involved in this algorithm are described in Section 6.2, “XORShift128 algorithm”. Only after the successful execution of the XORShift128 algorithm, stop HSM command shall be activated. The overall functional call diagram is shown in Figure 3.6, “Functional call diagram”.

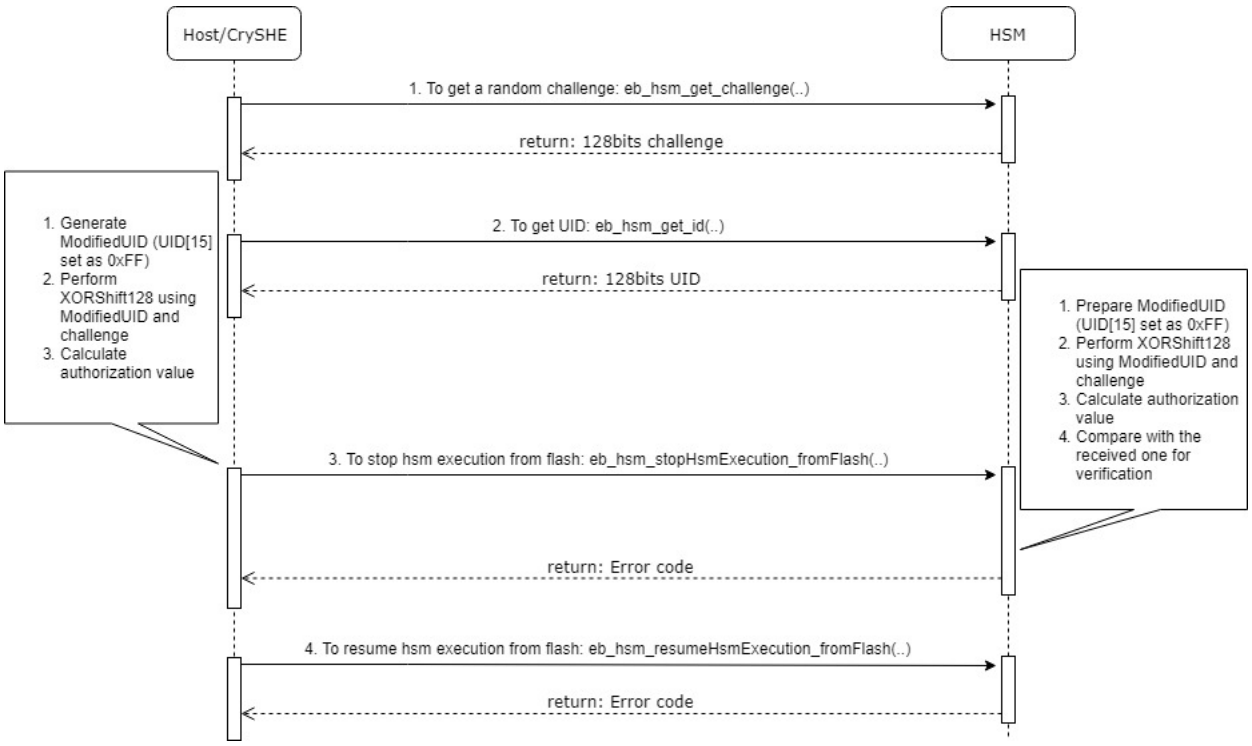


Figure 3.6. Functional call diagram

3.8.1. Prerequisites

First, as prerequisites, two initial steps shall be performed by host before requesting the challenge:

- The host shall initialize the EB HSM proxy and FW by calling proxy API `eb_hsm_init()`. The host shall check that the return value of the API is `COMM_NO_ERROR`.
- The host shall initialize the RNG by calling proxy API `eb_hsm_rnd_init()`. The host shall check that the return value of the API is `COMM_NO_ERROR`.

Once those two steps mentioned above are performed successfully, the host can perform following steps accordingly:

- Requesting the challenge:
  The first step is to get the challenge from the HSM. The host can request a challenge by calling the proxy API `eb_hsm_get_challenge()`. In order to get a challenge for the activation of the stop HSM functionality, the mode `COMM_CHALLENGE_MODE_STOPHSM` must be used as a parameter when calling the proxy API `eb_hsm_get_challenge()`. The HSM will generate and return a random challenge for the specific mode.
- Requesting the UID:
  The second step is to get the UID from the HSM. The host can request the UID by calling the proxy API `eb_hsm_get_uid()`. The HSM shall return the 16 bytes UID. The host can request the UID by calling the

proxy API `eb_hsm_get_uid()`. The HSM shall return 16 bytes of data which includes UID of 15 bytes and sreg (the value of Status Register) of 1 byte (MSB). The UID in the EB HSM FW is implemented according to the section 4.4.2 of the SHE Functional Specification v1.1.

Only after the successful execution of the XORShift128 algorithm, stop HSM command shall be activated.

3.8.2. Stopping HSM execution from PFlash

Stopping the HSM execution from the PFlash has to be done by calling the proxy API `eb_hsm_stopHsmExecution_fromFlash()`. The host must generate a response using the XORShift128 algorithm as described in the section Section 6.2, “XORShift128 algorithm”. When calling the proxy API `eb_hsm_stopHsmExecution_fromFlash()`, the correct response must be passed with the call. The response shall be the word state `a` (4 bytes output of the last iteration of the XORShift128 algorithm).

The first step in the stop HSM command is to re-calculate the response from the previously generated challenge and compare it with the received response. The EB HSM FW shall calculate the response using the XORShift128 algorithm (as described in Section Section 6.2, “XORShift128 algorithm”). Both responses must match in order to activate the stop HSM command. If both responses do not match, the HSM shall return `COMM_INVALID_AUTHORIZATION` error code. The stop HSM command runs with the highest priority. You must ensure that no other jobs are running on the HSM. Otherwise, the stop HSM command fails with the `COMM_BUSY` error code. If there are no ongoing HSM operations, the HSM switches to RAM execution. However, in this mode, no further jobs can be processed. Hence such requests are rejected with the error cause `COMM_IGNORE_UPCOMING_REQUESTS`. The error code can be retrieved by calling `eb_hsm_get_error()`. The host can now perform any erase/write operations on the same PFlash bank.

3.8.3. Resuming HSM execution from PFlash

In order to resume HSM execution from PFlash and thus to continue the normal HSM operation mode, the proxy API `eb_hsm_resumeHsmExecution_fromFlash()` needs to be called. This API shall only be called after the stop HSM command was successfully executed. Otherwise, the error cause `COMM_GENERAL_ERROR` is returned. When the resume HSM command was successfully executed, the HSM can handle any request and can use the PFlash as before.

3.9. Firmware update

The EB zentur HSM Firmware can be updated with the firmware update (FWU) feature. Both the current application and the update application are located at PF0 PFlash.

WARNING **Limitation on memory usage**

Do not write or fill in any way the memory gaps of the HSM application. This applies to both initial and production HSM application images.

Filling up the memory gaps poses the risk that the partition metadata that is important for the operation of the firmware is not written correctly. This is caused by the limitations of the hardware.

3.9.1. Overview

- In order to establish the FWU, certain APIs are provided to directly interface with the FWU feature, see Section 3.9.2, “FWU interface”.
- The creation of the FWU data profile, which you must provide as DER encoded binaries, is described in Section 3.9.3, “FWU data profile”.
- For the steps required to run a FWU procedure, see Section 3.9.4, “FWU procedure”.
- Rolling back to a previous firmware application image is described in Section 3.9.5, “FWU rollback”.
- For security aspects and counter measures that are arranged to protect the FWU against manipulation, corruption or wrong execution, see Section 3.9.8, “FWU protection”.

For the flashing of the EB zentur HSM Firmware image and bootloader, see Section 4.2.5, “Flashing EB zentur HSM Firmware”.

3.9.2. FWU interface

EB zentur HSM Firmware provides certain APIs to directly control the FWU feature. For details about input parameters and the signature of these FWU APIs, see Chapter 5, “Module references“. An overview of these FWU APIs is given in the following Table 3.3, “Overview of FWU APIs”:

FWU API	Description
`eb_hsm_fw_update_start()`	▶ Starts the FWU process by providing a pointer to the FWU data profile and the corresponding byte length. ▶ The FWU data profile must be ASN.1 DER encoded and protected by a signature. ▶ For a detailed description of the FWU data profile including content and creation, see Section 3.9.3, “FWU data profile”.
`eb_hsm_fw_update_data()`	▶ Sends the EB zentur HSM Firmware application image to the HSM by providing a pointer to the EB zentur HSM Firmware application image and the corresponding byte length.

FWU API	Description
	▶ Pointer to the HSM Firmware application image must be 16-byte aligned. ▶ Can be called in blocks multiple times if the whole EB zentur HSM Firmware application image cannot be sent via a single call.
`eb_hsm_fw_update_finish()`	▶ Completes the FWU process by running several sanity checks and finally stores the FW image management data. ▶ Checks whether the full EB zentur HSM Firmware application image was flashed, whether the signature of the EB zentur HSM Firmware application image matches, and whether the version of the EB zentur HSM Firmware application image is correct.
`eb_hsm_fw_rollback()`	▶ Executes a rollback of the recent EB zentur HSM Firmware application image by providing a pointer to the FWU data profile and the corresponding byte length. ▶ The FWU data profile must be ASN.1 DER encoded and protected by a signature. ▶ For a detailed description of the FWU data profile including content and creation, see Section 3.9.3, “FWU data profile”. ▶ For details about the rollback procedure, see Section 3.9.5, “FWU rollback”.

Table 3.3. Overview of FWU APIs

3.9.3. FWU data profile

As a prerequisite for updating the EB zentur HSM Firmware application image, you have to create the FWU data profile [FWU_data_profile]. Here the Distinguished Encoding Rules (DER) are used, which are a subset of the Basic Encoding Rules. They give exactly one way to represent any ASN.1 value as an octet string. The DER defines rules to code or decode values where:

- **Tag** defines the type of content and is encoded in one or two octets
  - e.g., cipher info is encoded in one octet (0x82), the FWU data body is encoded in two octets (0x7F, 0x4E)
- **Length** defines the length of content octets
  - If the **Value** contains fewer than 128 octets, the **Length** field requires only one octet to specify the content length.
  - If the **Value** contains more than 127 octets, bit 7 of the **Length** field is set to 1, and bits 6 through 0 specify the number of additional **Length** octets to be used to identify the content length.

- **Value** represents the content that is encoded in zero or more octets.

The content of the FWU data profile is shown in Table 3.4, “FWU data profile parameters”, which serves as input for the API `eb_hsm_fw_update_start()`.

Data objects	Tag	Length	Value
FWU data profile	0x7F7E	0x8v	▶ The **Length** of the FWU data profile represents its overall **Length**. ▶ The bit 7 of the first **Length** octet is set. This indicates that the FWU Data profile consists of more than 127 octets. ▶ v specifies the number of additional **Length** octets, e.g.: FWU data profile consists of 552 octets (=0x0228). This means v=2 because two additional **Length** octets are needed to indicate the length (0x02, 0x28) of the FWU data profile. That again means the **Length** is expressed with 3 octets in total: 0x82, 0x02, 0x28. ▶ There are no **Value** octets for parameter FWU data profile.
Cipher information	0x82	0x02	▶ octet 1 = cipher algorithm used to encrypt the FWU data body and data body signature, as well as the EB zentur HSM Firmware application image. This version of the EB zentur HSM Firmware does not support encryption of FWU data body. The EB zentur HSM Firmware application image can be either plain or encrypted. ▶ 0x00 = no encryption ▶ 0x01 = AES ECB 128 encryption ▶ octet 2 = key identifier indicating AES key used for encryption
FWU data body	0x7F4E	0x8v	▶ The **Length** of the FWU data body represents its overall **Length**. ▶ The bit 7 of the first **Length** octet is set. This indicates that the FWU data body consists of more than 127 octets. ▶ v specifies the number of additional **Length** octets, e.g.: FWU data body consists of 282 octets (=0x011A). This means v=2 because two additional **Length** octets are needed to indicate the length (0x01, 0x1A) of the FWU data body. That again means the **Length** is expressed with 3 octets in total: 0x82, 0x01, 0x1A. ▶ There are no **Value** octets for parameter FWU data body.

Data objects	Tag	Length	Value
FWU data profile identifier	0x5F29	0x01	▶ octet 1 = 0x00 (version of the FWU data profile)
Firmware version information	0x81	0x04	▶ octet 1 = major firmware version info ▶ octet 2 = minor firmware version info ▶ octet 3 = patch firmware version info ▶ octet 4 = 0x0 (not used)
Key identifier	0x83	0x01	▶ octet 1 = signature key identifier
Signature algo	0x84	0x01	▶ octet 1 = used signature algorithm ▶ 0x01 = CMAC AES-128 ▶ 0x02 = RSA SSA PSS SHA-256 ▶ 0x03 = RSA SSA PKCS#1 v1.5 SHA-256 ▶ 0x04 = ECDSA SECP SHA-256 ▶ 0x05 = EDDSA Ed25519ph
Firmware image length	0x85	0x04	▶ octet 1 = firmware image length (MSB) ▶ octet 2 = firmware image length ▶ octet 3 = firmware image length ▶ octet 4 = firmware image length (LSB)
Firmware image signature	0x86	0x8v	▶ The firmware image signature has a variable content length without limitation. ▶ The signature is calculated over the plain EB zentur HSM Firmware. ▶ The bit 7 of the first **Length** octet is set which indicates that the firmware image signature consists of more than 127 octets. ▶ v specifies the number of additional **Length** octets, e.g.: The firmware image signature consists of 256 octets (=0x0100). This means v=2 because two additional **Length** octets are needed to indicate the content length (0x01, 0x00). That again means the **Length** is expressed with 3 octets in total: 0x82, 0x01, 0x00. ▶ The real 256 octets of the firmware image signature are attached to the 3 **Length** octets.
FWU data body signature	0x5F37	0x8v	▶ The FWU data body signature has a variable content length without limitation.

Data objects	Tag	Length	Value
			▶ The bit 7 of the first **Length** octet is set, which indicates that the FWU Data Body signature consists of more than 127 octets. ▶ v specifies the number of additional **Length** octets, e.g.: The FWU data body signature consists of 256 octets (=0x0100). This means v=2 because two additional **Length** octets are needed to indicate the content length (0x01, 0x00). That again means the **Length** is expressed with 3 octets in total: 0x82, 0x01, 0x00. ▶ The real 256 octets of the FWU data body signature are attached to the 3 **Length** octets.

Table 3.4. FWU data profile parameters

3.9.4. FWU procedure

The basic procedure to execute a FWU is as follows:

- Install the system services via the API eb_hsm_setup().
- Initialize the HSM via the API eb_hsm_init().
- You must select the algorithm for the FWU and the firmware image signature (CMAC, RSA, ECDSA, EDDSA). Furthermore, you must create a key pair that is used for the signature generation and verification. You provide the key to be stored at the HSM depending on the selected algorithm:
  - In case of RSA/ECDSA/EDDDSA, the API eb_hsm_load_asym_pub_key() must be called to load the asymmetric key that is used to verify the EB zentur HSM Firmware image and FWU data signatures.
  - In case of CMAC-128, the APIs eb_hsm_get_load_key_id_and_range() and eb_hsm_load_key() must be called.
- You must generate the signature of the EB zentur HSM Firmware image and FWU data, followed by the assembling of the different parameters of the FWU data profile (Section 3.9.3, “FWU data profile”). Finally, the FWU data profile needs to be assembled and signed to get it ready for starting the FWU procedure.
- To start the FWU procedure, the API eb_hsm_fw_update_start() must be called. The API gets a pointer to the FWU data profile and its length, see Section 3.9.3, “FWU data profile”.
- It is possible to program the EB zentur HSM Firmware image as a whole with one single call to the API eb_hsm_fw_update_data(). Also, there is an option to program the EB zentur HSM Firmware image in blocks with multiple calls to eb_hsm_fw_update_data(). In both cases, the pointer to the EB zentur HSM Firmware image must be 16-byte aligned. In the latter case, the length of the EB zentur HSM Firmware image block must be a multiple of 32 bytes except for the last block.
- The FWU procedure is finished when the API eb_hsm_fw_update_finish() is called, followed by a restart of the system.

3.9.5. FWU rollback

You can do a rollback to the previous version of the EB zentur HSM Firmware application if the recent version of the EB zentur HSM Firmware application image has issues. The rollback procedure consists of several steps and certain API calls to be executed in a specific order, which is explained below.

- Install the system services via the API eb_hsm_setup().
- Initialize the HSM via the API eb_hsm_init().
- For security reasons, an authentication is needed to successfully execute a rollback. For this, the signed FWU data profile is used. See section Section 3.9.3, “FWU data profile”. You must select the algorithm for the rollback (CMAC, RSA, ECDSA, EDDSA). Furthermore, you must create a key pair that is used for the signature generation and verification. You provide the key to be stored at the HSM depending on the selected algorithm:
  - In case of RSA/ECDSA/EDDDSA, the API eb_hsm_load_asym_pub_key() must be called to load the asymmetric key that is used to verify the EB zentur HSM Firmware image and FWU data signatures.
  - In case of CMAC-128, the APIs eb_hsm_get_load_key_id_and_range() and eb_hsm_load_key() must be called.
- The data object **Firmware version information** (Tag=0x81) of the FWU data profile must contain the version of the EB zentur HSM Firmware application that has to be rolled back. The rollback is only executed if the version matches the version of the recent EB zentur HSM Firmware application.
- To start the rollback procedure, the API eb_hsm_fw_rollback() must be called. The API gets a pointer to the FWU data profile and its length, see Section 3.9.3, “FWU data profile”. The FWU data profile that was used to install a specific version of the EB zentur HSM Firmware application image can be reused for the rollback of that version.
- In order to activate the previous EB zentur HSM Firmware application image, a system restart needs to be executed afterwards.

3.9.6. FWU get bootloader version

With the function eb_hsm_get_bl_version(), it is possible to retrieve the version information for the EB zentur HSM bootloader. The function returns status and version number for the bootloader. The function also provides information about whether the initial or a production bootloader is flashed to the board.

For storing the status and version number of the bootloader that is currently flashed to the board, the function eb_hsm_get_bl_version() needs the parameter `blVersionInfo`. This parameter has to be a **comm_BL_-VersionInfoType** type pointer, and a buffer must be provided for the status and version information.

The bootloader status can have the following values:

- COMM_BL_IBL_VERSION: The bootloader is the initial version.

- COMM_BL_PBL_VERSION: The bootloader is the production version.
- COMM_BL_INVALID_VERSION: The bootloader is invalid.

The bootloader version number consists of major, minor, and patch version number.

3.9.7. FWU get version

There are two memory slots for EB zentur HSM Firmware applications. One memory slot is active and the other is passive. It is possible to get version numbers of the EB zentur HSM Firmware applications via eb_hsm_get_fw_version(). The function returns status and version numbers for active and passive EB zentur HSM Firmware applications.

- The get version function needs the two parameters `activeFirmwareInfo` and `passiveFirmwareInfo`. Active information contains status and version number of the EB zentur HSM Firmware application that is currently running. Passive information contains the status and version number of the EB zentur HSM Firmware application that is currently not running.
- These two parameters have to be **comm_FW_VersionInfoType** type pointers, and buffer must be provided for the status and version information. If the passive information is not needed, a NULL pointer can be used.
- The firmware status can have the following values:
  - COMM_FW_NEWER_VERSION: The firmware is the newer version of the two firmwares, and it is a valid EB zentur HSM Firmware application.
  - COMM_FW_OLDER_VERSION: The firmware is the older version of the two firmwares, and it is a valid EB zentur HSM Firmware application.
  - COMM_FW_INVALID: The firmware slot is empty, corrupted, or the firmware was rolled back.

For status examples, see Table 3.5, “Status of EB zentur HSM Firmware in different situations”. Note that COMM_FW_NEWER_VERSION always gives the slot from where the HSM will boot if the HSM is rebooted.

Order of events	Status of active slot (currently running)	Status of passive slot (currently not running)
After first boot	COMM_FW_NEWER_VERSION	COMM_FW_INVALID
After firmware update (no reboot)	COMM_FW_OLDER_VERSION	COMM_FW_NEWER_VERSION
After firmware update reboot	COMM_FW_NEWER_VERSION	COMM_FW_OLDER_VERSION
After rollback (no reboot)	COMM_FW_INVALID	COMM_FW_NEWER_VERSION
After rollback reboot	COMM_FW_NEWER_VERSION	COMM_FW_INVALID

Table 3.5. Status of EB zentur HSM Firmware in different situations

3.9.8. FWU protection

EB zentur HSM Firmware applies counter measures to protect against accidental or malicious HSM manipulation and corruption.

3.9.8.1. FWU sanity check

After providing the encoded FWU data profile via `eb_hsm_fw_update_start()`, the EB zentur HSM Firmware checks the validity of the given FWU data profile and verifies the FWU data profile signature. Then the EB zentur HSM Firmware analyzes if an FWU can be installed. Therefore the encoded FWU data profile (Section 3.9.3, “FWU data profile”) is decoded, and the different FWU data profile parameters are checked for correct length and content. The partition is selected (not the active one!) and erased.

3.9.8.2. FWU downgrade protection

The provided FWU version is compared to the already installed firmware version. Only if the FWU version is newer or equal compared to the installed version, the FWU is going to be installed. This approach prevents the installation of an older and thus authenticated firmware version that may pose a security risk.

3.10. Challenge-response protocol

Challenge-response protocol is an authorization protocol to validate the user of the HSM firmware. The protocol will use MASTER_ECU_KEY `MEK` as the common secret for the authentication. The protocol is required in following features:

- Debugger activation
- Stop HSM

Below is described the steps to calculate the authorization:

- The MASTER_ECU_KEY `MEK` which is provided by the vehicle manufacturer must be loaded and known for later derivation of the `DERIVED_KEY`.
- The first step in challenge-response protocol is that the user request a 16 byte `CHALLENGE` via API **eb_hsm_get_challenge()** with parameters channel ID 0 and the corresponding challenge mode. The challenge modes are:
  - Debug activation: COMM_CHALLENGE_MODE_DEBUG
  - Stop HSM: COMM_CHALLENGE_MODE_STOPHSM

The EB zentur HSM Firmware will generate a 16 byte random number which serves as the CHALLENGE for the user. The CHALLENGE shall be 16 byte memory aligned.

- Furthermore the user must fetch the unique identification item UID from EB zentur HSM Firmware. It is a 16 byte serial number which is inserted during chip fabrication by the semiconductor manufacturer. The UID is directly accessible via API **eb_hsm_get_id()** with channel ID 0 and synchronous processing. Note: In this case the parameters CHALLENGE, SREG and MAC are not taken into consideration, so these parameters can be set to zero for this specific API call.
- Next a key DERIVED_KEY must be derived from the MASTER_ECU_KEY MEK and a function specific key constant. The derivation is done by a KDF (Key Derive Function) using the Miyaguchi-Preneel compression algorithm based on the NIST special publication 800-108, which is a recommendation for key derivation using pseudorandom functions. The function specific key constants are following:
  - Debug activation: 0x0103 0x5348 0x4500 0x8000 0x0000 0x0000 0x0000 0x00b0.
  - Stop HSM: 0x0106 0x5348 0x4500 0x8000 0x0000 0x0000 0x0000 0x00b0.
- Subsequently the user must calculate the response AUTHORIZATION, which has a width of 32 bits. The AUTHORIZATION shall be 16 byte memory aligned. Based on the derived key the AUTHORIZATION is composed of a CMAC over the retrieved parameters CHALLENGE from **eb_hsm_get_challenge()**, and UID from **eb_hsm_get_id()**.
- Corresponding calculation formula:
  - DERIVED_KEY = KDF (MEK , KEY_CONSTANT)
  - AUTHORIZATION = $\mathrm{CMAC}_{\mathrm{DERIVED\_KEY}}$ (CHALLENGE | UID)
- Finally the calculated AUTHORIZATION must be given to the EB zentur HSM Firmware. The EB zentur HSM Firmware recalculates the AUTHORIZATION from the previously generated CHALLENGE and compares it with the received AUTHORIZATION. Both responses must match in order to successfully validate the user. If these doesn't match, the action must be stopped and COMM_INVALID_AUTHORIZATION error code returned.

3.11. Life Cycle Management

The purpose of the life cycle management is to define allowed operations on each state. Currently 2 states are defined, an Init and Secure Level 1 states. A life cycle is in an init state after flashing HSM-firmware for the first time or executing eb_hsm_debug_activation(). A state transition from the Init state to the Secure Level 1 is executed with the proxy call eb_hsm_advance_life_cycle(). The current life cycle state can be checked with the eb_hsm_get_life_cycle().

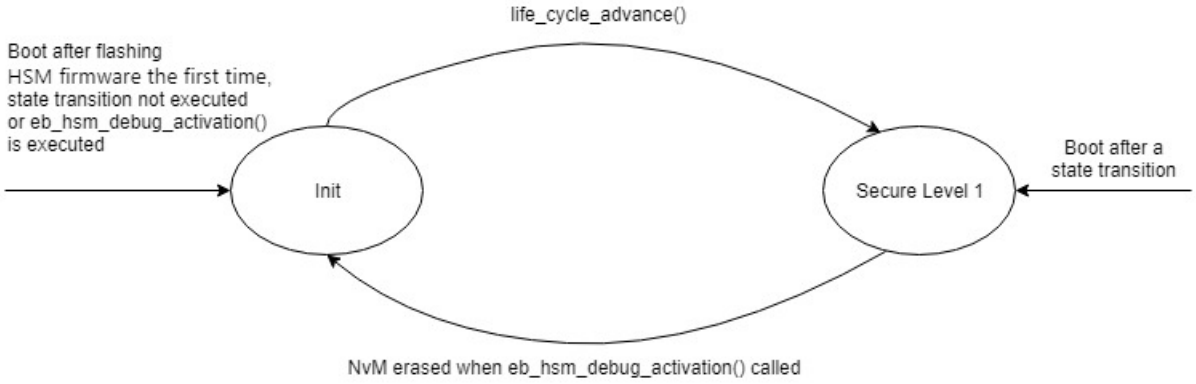


Figure 3.7. Life Cycle States

Life cycle states and restrictions listed in Table 3.6, “Life cycle states and restrictions”:

State	Restrictions
Init	All services allowed.
Secure Level 1	Root certificate loading not allowed. Asymmetric raw key loading not allowed.

Table 3.6. Life cycle states and restrictions

3.11.1. State Init

The following diagram outlines a certificate and raw asymmetric key loading in the init state.

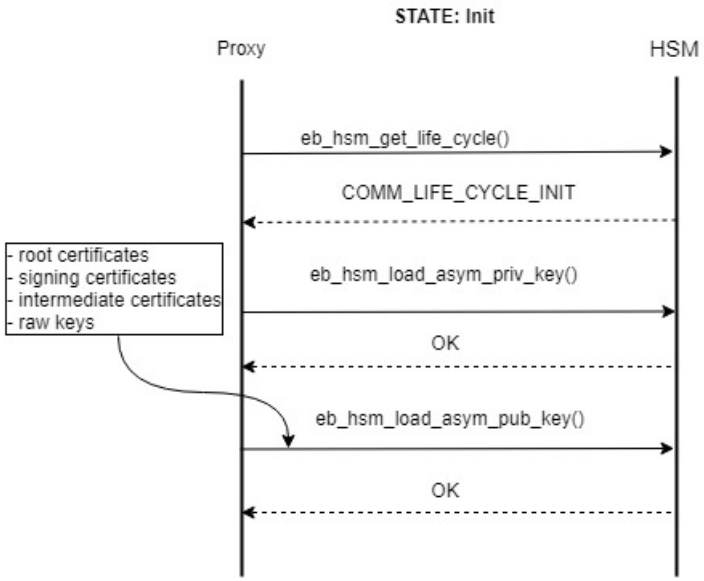


Figure 3.8. Life Cycle - Init State

3.11.2. State Secure Level 1

The following diagram outlines a certificate and raw asymmetric key loading in the Secure Level 1 state.

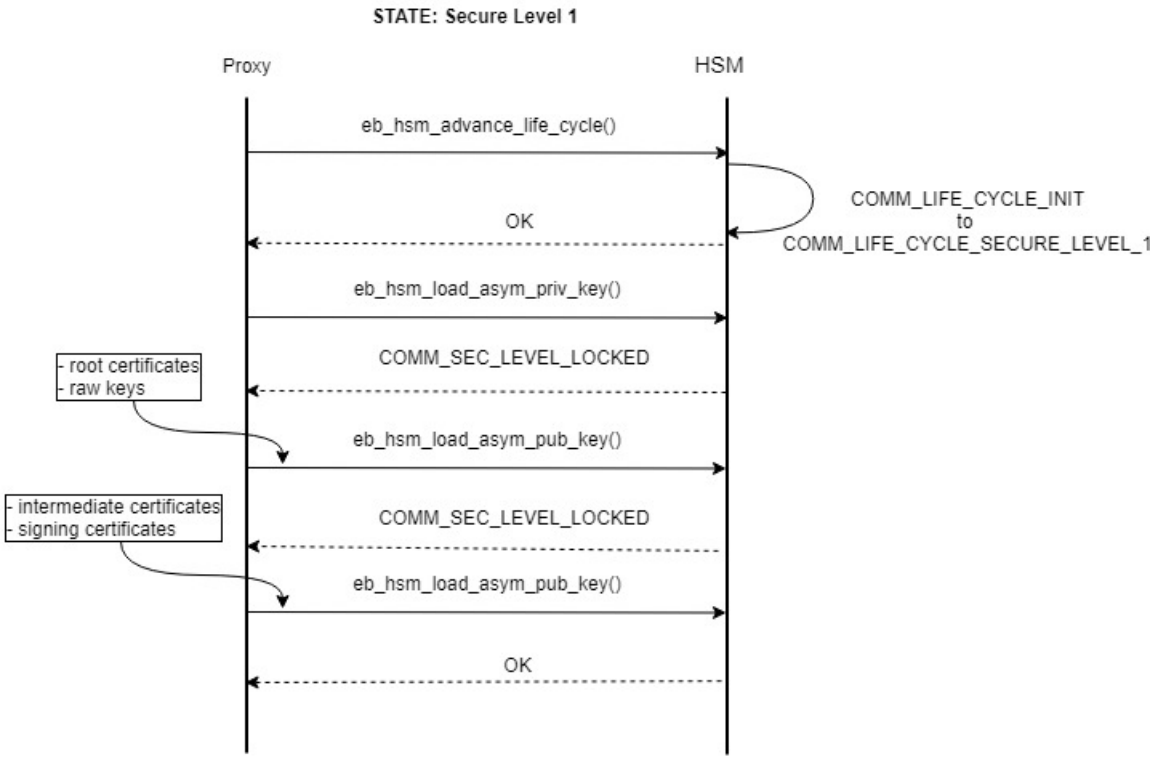


Figure 3.9. Life Cycle - Secure Level 1 State

3.12. Diagnostic interface

The purpose of the diagnostic interface is to provide information when a readout is made from a trust store and to provide the following status information of each symmetric key stored in the HSM:

- key corruption
- public key readout (only possible for a RAM key)
- key change

The diagnostic interface also provides an API to reset the status information for each symmetric key.

4. User's guide

4.1. Overview

Section 4.2, “HSM Firmware” provides a general overview of important terminology, definitions and abbreviations. It shows how the EB zentur HSM Firmware is embedded in the AUTOSAR context and the integration and flashing procedure is explained. Finally, an overview on the supported hardware derivatives is given.

Section 4.3, “HSM proxy architecture” provides an overview of the EB zentur HSM Firmware proxy layer. The communication basics of the HSM bridge module is explained and the job operation modes are interpreted.

Section 4.4, “Use cases” lists basic uses cases for usage of the EB zentur HSM Firmware. It shows the interaction between host, HSM bridge module and EB zentur HSM Firmware and illustrates the dynamic behavior of the software.

4.2. HSM Firmware

4.2.1. Basic terminology

In the following section, the most important terminologies used in this document are listed.

4.2.1.1. SHE

The Secure Hardware Extension (SHE) is an on-chip extension to any given microcontroller. It is intended to move the control over cryptographic keys from the software domain into the hardware domain and protect those keys from software attacks. The concept behind SHE is to have a secured zone and to prevent user access to security functions, other than those give by logic. The main SHE security objectives are:

- Protect cryptographic key from software attacks and provide authentic software environment
- Let the security only depend on the strength of the underlying algorithm and the confidentiality of the keys
- Allow for distributed key ownerships

4.2.1.2. Host

Host can be any target on which EB zentur HSM Firmware is integrated.

4.2.1.3. HSM

The HSM (Hardware Security Module) is a peripheral hardware module and contains all necessary elements to allow software to implement the Secure Hardware Extension (SHE).

4.2.1.4. Job

A job refers to a cryptographic operation running on the EB zentur HSM Firmware. A job uses a channel for the communication with the EB zentur HSM Firmware. For jobs that make use of the same crypto hardware, if running in parallel a lower priority job will run to completion even if a job with a higher priority is started afterward. This is a design choice and may change for hardware that supports multiple instances of the same crypto hardware.

4.2.1.5. Channel

A channel handles only one job per time. In total eight channels are supported. A channel implicates a job priority, the higher the value the higher the priority.

- Queuing is not supported when EB zentur HSM Firmware is integrated based on AUTOSAR 4.0.3 products.
- Queuing is supported when EB zentur HSM Firmware is integrated based on AUTOSAR 4.3 products.

4.2.2. HSM Firmware context view

The EB zentur HSM Firmware is a performance and resource optimized Firmware solution. It provides key management with secure access to cryptographic hardware accelerators and software algorithms. The communication between the host and the EB zentur HSM Firmware is encapsulated by a CrySHE module or by a Crypto Driver hardware module. The EB zentur HSM Firmware can be integrated into the following products:

- EB tresos AutoCore Generic 7 products (ACG7 AUTOSAR 4.0/4.1/4.2) accomplished by using a CrySHE module, see Figure 4.1, “ACG 7 integration”
- EB tresos AutoCore Generic 8 products (ACG8 AUTOSAR 4.3) accomplished by using a Crypto Driver hardware module, see Figure 4.2, “ACG 8 integration”

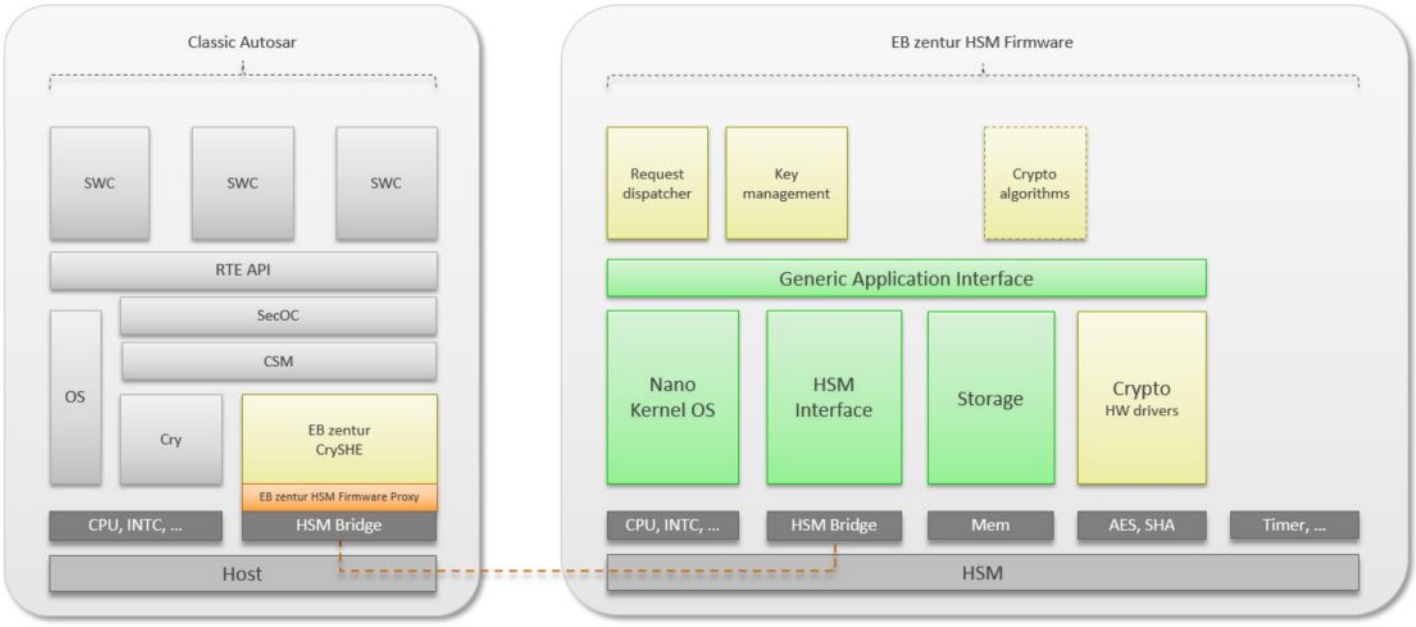


Figure 4.1. ACG 7 integration

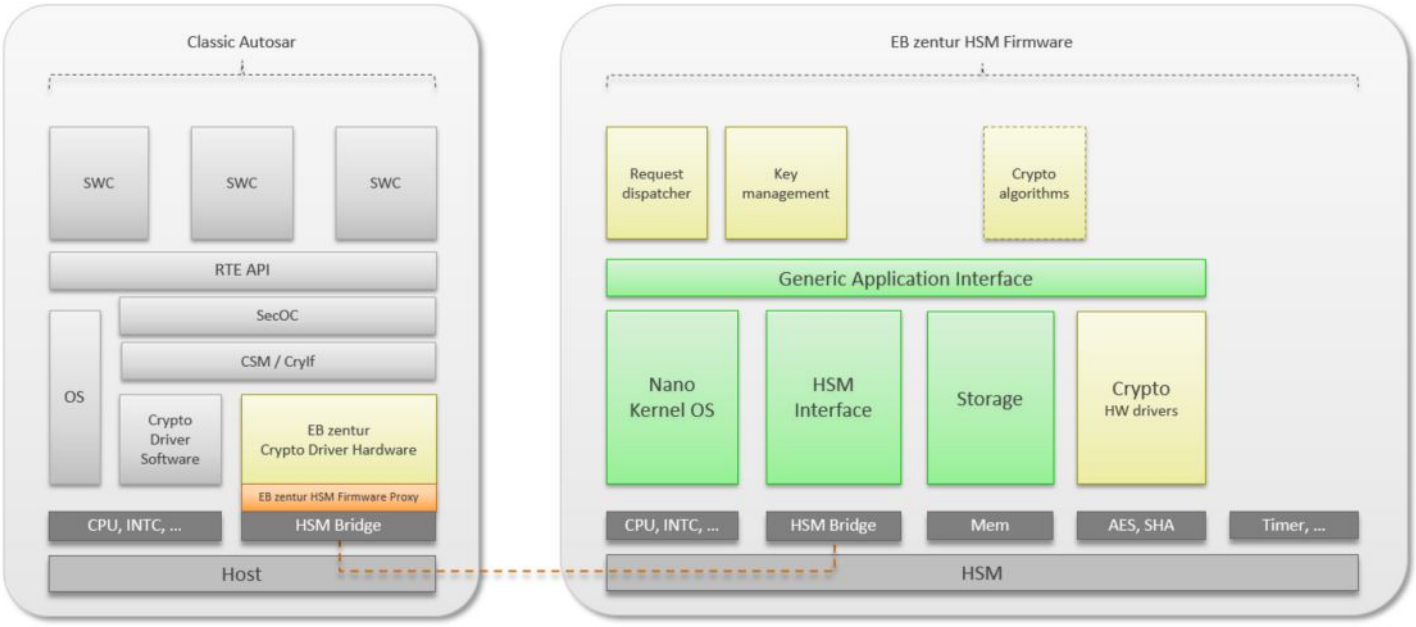


Figure 4.2. ACG 8 integration

4.2.3. HSM core Firmware

The architecture of the EB zentur HSM Firmware is not the main focus of this document, so only the main characteristics of the EB zentur HSM Firmware have been included. Figure 4.3, “EB zentur HSM Firmware modules” depicts the EB zentur HSM Firmware core components.

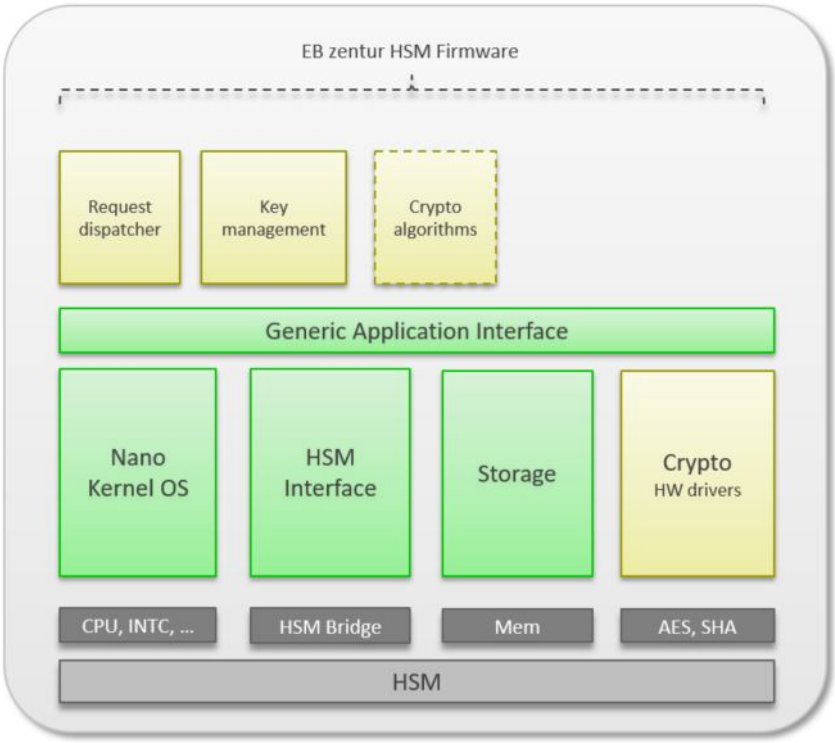


Figure 4.3. EB zentur HSM Firmware modules

The EB zentur HSM Firmware relies on an operating system described by OSEK. The nano-kernel operating system is limited to basic tasks only whereas each task can have different priority. Each task terminates by returning from its top level function. The main responsibilities of the nano operating system kernel are the task configuration activation, handling and activation. In addition, the interrupt service routine handling is another important aspect, with the limitation that nested interrupts are not supported. That means interrupts are either locked or unlocked. Scheduling decisions are done while interrupts are locked, so the reentrancy in the nano operating system kernel is very limited.

The EB zentur HSM Firmware controls the whole key management and is responsible for storing keys and MACs. The information is stored in a a non-volatile memory (NVM) that is available after power cycles and resets of the microcontroller. Temporary information is stored in a volatile memory. The EB zentur HSM Firmware handles certain key management security mechanism with the help of key property flags as specified in the SHE specification.

The EB zentur HSM Firmware controls the storage by reading and writing data to NVM media like Flash or EEPROM. Below that, the FEE module abstracts from the device a specific addressing scheme and segmen-

tation. This provides the upper layers with a virtual addressing scheme, segmentation and a *virtually* unlimited number of erase cycles. Below the storage top layer NVM and storage middle layer FEE, the storage lowest layer the Flash Driver initializes and reads/writes to Flash memory. There are different flash sectors used for different derivatives. The Data Flash is exclusive for HSM for TC27X, TC29X and TC3XX, but the Program Flash needs to be used for TC23X.

The EB zentur HSM Firmware is responsible for executing all cryptographic services. The EB zentur HSM Firmware provides the abstraction layers and execution functions for the following services:

- AES algorithm, single call
- HMAC-SHA256, single call
- CMAC algorithm, single call
- TRNG handling
- PRNG handling
- HASH algorithms, single call
- PKC handling

4.2.4. Integrating EB zentur HSM Firmware

This section provides guidance on how to integrate and install the proxy layer onto your project in order to make use of the EB zentur HSM Firmware. In addition the content of the product delivery package:

- The proxy layer acts as the the main communication interface to the EB zentur HSM Firmware.
- The proxy layer does time supervision of the operations it dispatches to the EB zentur HSM Firmware.
- The proxy layer guarantees a controlled access to the HSM critical resources e.g. ciphering devices, random number generator, hardware registers and flash memory.
- To keep the proxy layer generic for usage on different microcontrollers and to avoid dependencies to the applied host OS, the application has to provide this functionality.
- The memory buffer that is used for the data exchange between the proxy layer and the EB zentur HSM Firmware has to be provided by the application. This is to avoid dependencies to the compiler and hardware.
- The common source file subfolders contain the framework for the services. The framework ships with an sample implementation. The integrator has to adapt the implementation to the real hardware and the available operating system.

The proxy layer of the EB zentur HSM Firmware is delivered within the plugin of the CrySHE driver or CryptoDriver hardware in the following directories:

- `lib_include` directory with target specific header files and definitions for the host/HSM communication, common data types, wrapper functions for integration and the header files for the APIs

- `lib_src` directory including the implementation details of the proxy layer

The content of the delivery is listed in Table 4.1, “EB zentur HSM Firmware delivery contents”:

Item	Content
eb_hsm_comm.h	Primitive Services of the communications library
eb_hsm_comm_types.h	Data types of the communications library
eb_hsm_integ.h/.c	Wrapper functions for target system services to generic service required by the proxy layer
eb_hsm_proxy.h/.c	API functions of the proxy layer for the communication to the EB zentur HSM Firmware
eb_hsm_bridge.h	Register definitions for the bridge module (target specific)
eb_hsm_cache.h	Cache specific definitions (target specific)
eb_hsm_proxy_arch.h	Architecture dependent proxy layer functions (target specific)
eb_hsm_MemMap.h	Proxy layer memory mapping definitions (target specific, generated)

Table 4.1. EB zentur HSM Firmware delivery contents

In the following, more details are given on the delivered files that are integrated on the host side:

- Primitive services of the communications library

  The file `eb_hsm_comm.h` lists all available primitive services, for example, COMM_CMD_AES_CIPHER is used to request an AES ciphering or COMM_CMD_CMAC is used to request a CMAC generation or CMAC verification. It serves as input for the EB zentur HSM Firmware for the sake of executing the right command. It also lists basic definitions like the maximum number of supported channels, bit masks for the bridge registers, and more.

- Data types of the communications library

  The file `eb_hsm_comm_types.h` lists all common data types shared between the host and EB zentur HSM Firmware.

- Wrapper functions for target system services

  The files `eb_hsm_integ.h` and `eb_hsm_integ.c` contain wrapper functions for target system services e.g. especially timer services that are required by the proxy layer. These functions must be registered during the setup of EB zentur HSM Firmware, see Section 4.4.1, “HSM Setup”. Note: The setup procedure of the EB zentur HSM Firmware must be done in advance of the initialization procedure of the EB zentur HSM Firmware and is needed to register the system services. The wrapper functions are needed to:

  - read current timer ticks
  - get the number of ticks between the current tick value and a previously read tick value

- convert the elapsed timer ticks to real time in milliseconds
- provide resource locking and/or resource unlocking to synchronize mutual exclusive access to HSM
- disable hardware interrupts and/or restores the interrupt status

A more detailed description of the wrapper APIs for target system services is given in Chapter 5, “Module references“.

- EB zentur HSM Firmware proxy APIs

  The file `eb_hsm_proxy.h` imports the APIs for the EB zentur HSM Firmware, which is provided via a proxy layer. A more detailed description of the EB zentur HSM Firmware APIs is given in Chapter 5, “Module references“.

- Bridge module definitions

  The file `eb_hsm_bridge.h` lists all definitions for the registers of the bridge module. It serves as communication foundation between the host and the HSM. A more detailed description of the target dependent bridge registers is given in Section 4.3.2, “HSM bridge”.

- Cache module definitions

  The file `eb_hsm_cache.h` lists target dependent definitions for the registers of the cache module.

- EB zentur HSM Firmware architecture dependent functions

  The file `eb_hsm_proxy_arch.h` consists of architecture dependent proxy layer functions. They serve also as communication access functions for the SFRs of the bridge module.

- Proxy layer memory mapping definitions

  The file `eb_hsm_MemMap.h` consists of definitions requiring mapping to correct memory regions during software integration.

Memory section	**Description**
EB_HSM_<START/STOP>_SEC_CODE	Begins/ends a code section (mapped to application block, boot block, external flash etc.).
EB_HSM_<START/STOP>_SEC_VAR_-INIT_8	Begins/ends a section of 8-bit aligned variables and constants that are initialized with values after every reset (including similarly aligned composite data types).
EB_HSM_<START/STOP>_SEC_VAR_-INIT_32	Begins/ends a section of 32-bit aligned variables and constants that are initialized with values after every reset (including similarly aligned composite data types).
EB_HSM_<START/STOP>_SEC_VAR_-INIT_UNSPECIFIED	Begins/ends a section of unaligned variables and constants that are initialized with values after every reset.

Memory section	Description
EB_HSM_<START/STOP>_SEC_VAR_-CLEARED_8	Begins/ends a section of 8-bit aligned variables and constants that are cleared to zero after every reset (including similarly aligned composite data types).
EB_HSM_<START/STOP>_SEC_VAR_-CLEARED_32	Begins/ends a section of 32-bit aligned variables and constants that are cleared to zero after every reset (including similarly aligned composite data types).
EB_HSM_<START/STOP>_SEC_VAR_-CLEARED_128	Begins/ends a section of 128-bit aligned variables and constants that are cleared to zero after every reset (including similarly aligned composite data types).
EB_HSM_<START/STOP>_SEC_VAR_-CLEARED_UNSPECIFIED	Begins/ends a section of unaligned variables and constants that are cleared to zero after every reset.

Table 4.2. Proxy layer memory mapping definitions

4.2.5. Flashing EB zentur HSM Firmware

4.2.6. Enabling secure operation

This section describes how to enable secure operation for TC2XX. An unauthorized read or write access to the flash memory can be prevented by a security protection mechanism. With this mechanism, protected flash sectors can only be read when booting from internal flash and can only be changed after authentication. The following flash sectors have separate read protectors:

- DFlash EEPROM logical sectors that are commonly used for EEPROM emulation
- DFlash UCB logical sectors that are used for protection installation
- PFlash HSM code sectors

It is possible to control the read and write access to the relevant sectors, particularly the UCB sectors and the PFlash HSM code sectors S6 and S17. The write (erase/program) protection and the read protection to the PFlash HSM code sectors depend on the setting of the bit `HSMsX` of the HSM code flash OTP-related protection configuration register `PROCONHSMCOTP`. This register defines the *HSM_exclusive* attribute of the HSM code sectors.

The *HSM_exclusive* flag reflects a PFlash feature that enables PFlash sectors to be configured as a *HSM code sector*, and PFlash HSM code sectors are exclusively indicated by setting the *HSM_exclusive* flag. The PFlash logical sectors S6 and S17 are used as HSM code sectors. The UCB `UCB_HSMCOTP` configures the *HSM_-exclusive* protection for the HSM code sectors. The write and read protection to the DFlash UCBs is handled in a similar way.

The UCBs are part of the DFlash and are used for protection settings. The protection is determined by the content of the protection configuration registers `PROCONx` that are loaded during start-up. Each UCB has its own access control and can be erased and programmed by using the *HSM command interface*:

- a UCB is erased by the command `Erase Logical Sector Range`
- a UCB is programmed by the command `Write Page`

In the context of secure operation, the following UCBs are of interest:

- UCB_PFlash
  The PFlash UCB includes the read protection for the whole PFlash and the write protection for the logical sectors of PFlash. This is expressed by the PFlash protection configuration register `PROCONP`.
  - It is possible to activate PFlash read protection for the whole PFlash via the single bit `RPRO`.
  - Each single PFlash sector can be protected against write access via the bit `SxL`.
  - Note: The HSM code specific sectors S6 `S6L` and S17 `S17L` cannot be locked (i.e. write-protected) if the corresponding *HSM_exclusive* flags are set.
- UCB_DFlash
  The DFlash UCB includes the read/write protection for the DFlash EEPROM sectors. This is expressed by the DFlash protection configuration register `PROCOND`.
  - Bit `L` can set the DFlash sectors EEPROMx to write-protected.
  - Bit `RPRO` can set the DFlash sectors EEPROMx to read-protected.
- UCB_HSMCOTP
  The HMSCOTP UCB configures the OTP and *HSM_exclusive* protection for the HSM code sectors S6 and S17. This is expressed by the HSM code Flash OTP protection configuration register `PROCONHSMCOTP`.
  - Bit `HSMDX` indicates whether the HSM data sector HSMx is configured as *HSM_exclusive*.
  - Bit `HSD6X` indicates whether the HSM code sector S6 is configured as *HSM_exclusive*.
  - Bit `HSD17X` indicates whether the HSM code sector S17 is configured as *HSM_exclusive*.
- UCB_OTP
  The OTP UCB configures the OTP protection. This is expressed by a one-time programmable protection configuration register `PROCONOTP`.
  - Each PFlash sector can be protected (i.e. locked indefinitely) with a read-only functionality via a single bit `SxROM`.
  - The HSM code specific sectors `S6ROM` and `S17ROM` cannot be locked indefinitely. These HSM code specific sectors are OTP protected by the HSM code Flash OTP protection configuration register `PROCONHSMCOTP`.
- UCB_HSM
  The HSM UCB reflects the HSM interface configuration. This is expressed by the HSM interface protection configuration register `PROCONHSM`.

- Bit `HSMDBGDIS` indicates whether the HSM debug is configured as disabled or enabled.
- Bit `DBGIFLCK` indicates whether the chip debug is configured as unlocked or locked.

4.3. HSM proxy architecture

4.3.1. Logical view

The EB zentur HSM Firmware complies to the Security Hardware Extension (SHE). Its architecture is composed of:

- HSM Firmware
- HSM Proxy

The HSM Proxy is integrated on the host side and interacts with the CrySHE drivers for the AUTOSAR Crypto stack

- ACG 7 (AUTOSAR 4.0 / 4.1 / 4.2)
- ACG 8 (AUTOSAR 4.3)

The CrySHE module, as a lower interface to the CSM module, works as a communication handler with the coexisting HSM Firmware. In case of utilizing EB zentur HSM Firmware, the CrySHE module encapsulates the communication via a HSM Bridge module, which uses the HSM registers and provides functionalities to the CSM module. Figure 4.4, “Logical view ” depicts a logical overview of involved modules:

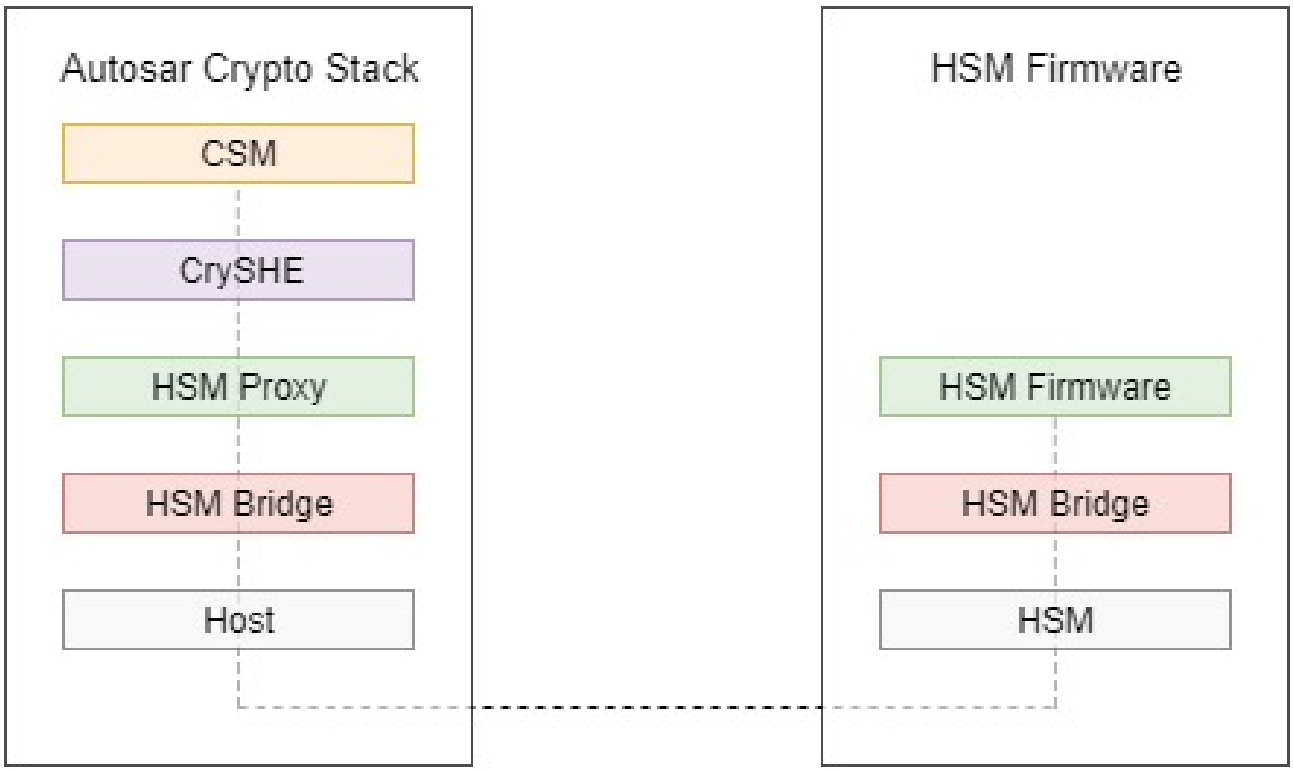


Figure 4.4. Logical view

4.3.2. HSM bridge

4.3.2.1. HSM bridge access

The bridge module is located on the Hardware Security Module. It connects the HSM subsystem to the host subsystem and enables communication between the two as follows:

- The bridge module acts as a firewall to protect HSM subsystem internals from outside accesses.
- The bridge module is responsible for all interaction between the HSM and the host. Therefore, the bridge module includes several 32bit wide Special Function Registers (SFR) for:
  - communication synchronization between the HSM and host
  - interrupt generation from the HSM to the host and vice versa
  - exchange of data and status information between the HSM and host
- The HSM has full access to the host via the bridge module.
- The host has only restricted access to the HSM via the bridge Module.

- Exchange of status information between the host and HSM subsystems is handled via the HSM bridge SFRs, see Section 4.3.2.2, “HSM bridge registers”, located in the HSM bridge communication unit. For larger amounts of data a shared memory is used, located on the host.
- Debugging the host and HSM subsystems must be authenticated by the HSM. For debugging purposes, the HSM address space can be made visible to a debugging module in the host address space through a 64KB memory window; a debugging module can read/write from/to the HSM internal resources, e.g. bridge registers, RAM, HSM boot ROM, CPU debug registers, through this window.
- The EB zentur HSM Firmware controls the debug access for the host and the HSM subsystem via a system control specific debug control Bridge register DBGCTRL (Figure 4.5, “DBGCTRL bridge register”). Access to this register via the host would end up in a bus error. To avoid a SPB bus conflict triggered by HSM debugging, HSM debugging can be disabled by setting the compiler switch *HSMDBGDIS*. You can disable HSM debug mode by compiling the software with *HSMDEBUG* option, by doing so, the HSM debug mode is disabled whenever a command is received/started.

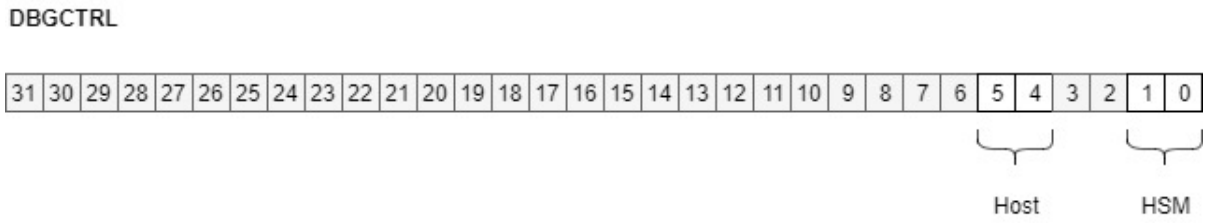


Figure 4.5. DBGCTRL bridge register

4.3.2.2. HSM bridge registers

The software related bridge module is reflected in `eb_hsm_bridge.h`, which is part of the delivery package, see Table 4.1, “EB zentur HSM Firmware delivery contents”. It contains all register definitions of the Bridge Module, in detail the registers for:

- Communication Unit including the SFRs
- Clock divider
- System control
- Error unit
- External interrupts
- Sensor interrupts
- Single access registers to host memory window and to host base address registers

The focus for host to HSM communication via the HSM proxy layer lies on the communication unit, including the SFRs, see Figure 4.6, “HSM bridge registers”. Host to HSM registers and HSM to host registers are explained in the following section.

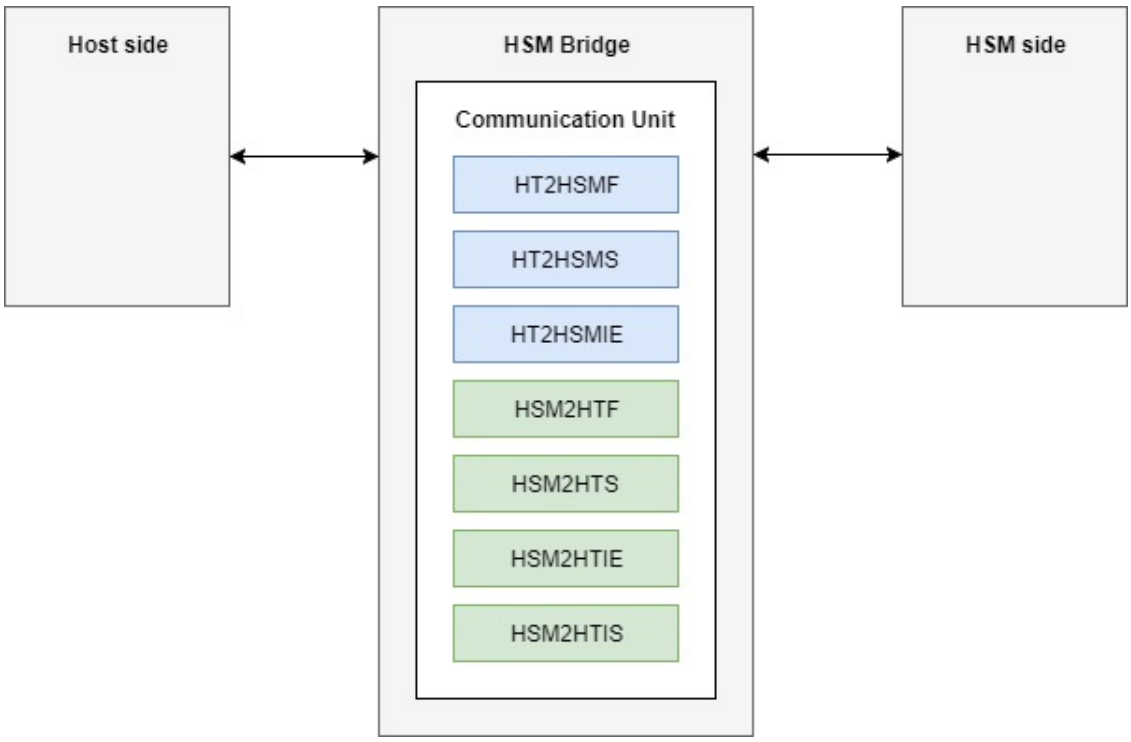


Figure 4.6. HSM bridge registers

4.3.2.2.1. Host to HSM registers

For the communication from the host to the HSM subsystem, the flag register, the status register, and the interrupt enable register are important. The status register serves as a parameter for the corresponding command provided in the flag register. In other words, the flag register and the status register are handled in a paired way, and the status register provides an argument for the command set in the flag register.

- Host to HSM flag register `HT2HSMF`

  The `HT2HSMF` is mainly served in two ways:

  - To exchange status information during initialization phase
  - To trigger the job command for all supported primitives

  It is the host that sets the `HT2HSMF` initialization bit 3 and the `HT2HSMF` interrupt bit 0 to inform the HSM subsystem to handle the initialization procedure on the HSM side. As a consequence, the HSM subsystem reads `HT2HSMF` within the bridge interrupt service routine and resets the `HT2HSMF` interrupt bit 0. Afterwards, the HSM subsystem handles initialization tasks and resets the `HT2HSMF` initialization bit 3. As long as the HSM subsystem is not finished with its initialization tasks and reseting the mentioned status bits in `HT2HSMF`, the host waits for a maximum of 3000ms. For this, a timeout counter is running on the host and the initialization bit in `HT2HSMF` is polled.

During the exchange of status information in the initialization phase, the second task for HT2HSMF bridge register is the dispatch handling job commands for all HSM supported primitives. For this task, the host sets the interrupt bit 0 and the job command bit 1. As a consequence, the HSM subsystem reads HT2HSMF within the bridge interrupt service routine and resets the HT2HSMF interrupt bit 0. Afterwards, the HSM subsystem executes its job and – when done – resets the job command bit 1 in HT2HSMF. As long as the HSM subsystem is not finished with its job tasks and reseting the mentioned status bits in HT2HSMF, the host waits for a maximum of 1000ms. For this, a timeout counter is running on the host and the job command bit 1 in HT2HSMF is polled.

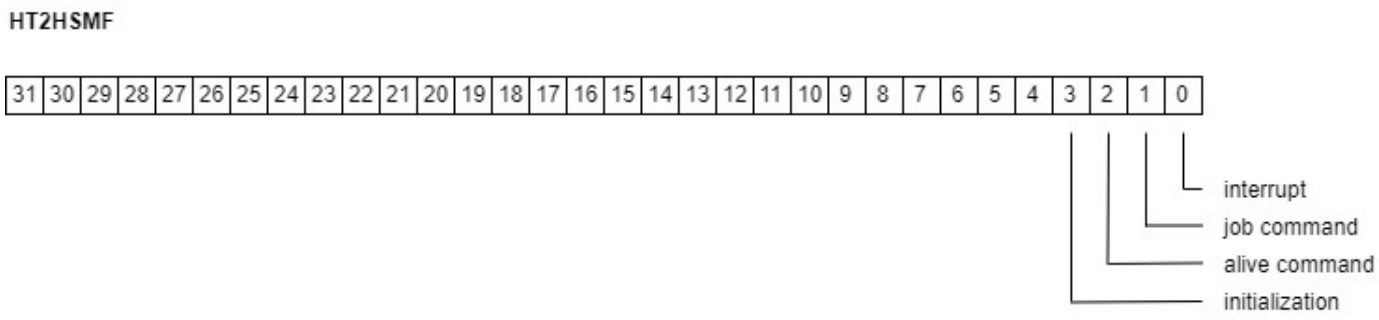


Figure 4.7. HT2HSMF_bitmap

- Host to HSM status register HT2HSMS

The HT2HSMS is mainly served in two ways:

- To set the pointer to the HT2HSM shared crypto job data memory during initialization phase
- To set the channel ID when a job command is triggered

The host will set the pointer in the HT2HSMS bridge register pointing to the crypto job data memory that is shared by the host and the HSM subsystem. On the opposite side, the EB zentur HSM Firmware stores the address of the shared memory based on the input from the host.

Besides the exchange of the shared memory address, the HT2HSMS is used to set the channel ID. The EB zentur HSM Firmware retrieves the channel ID from HT2HSMS, in case the corresponding job command has been set in HT2HSMF. If the activation in the HSM subsystem is succeeded, the channel ID dependent busy flag and the SHE busy flag is marked in HSM2HTS.

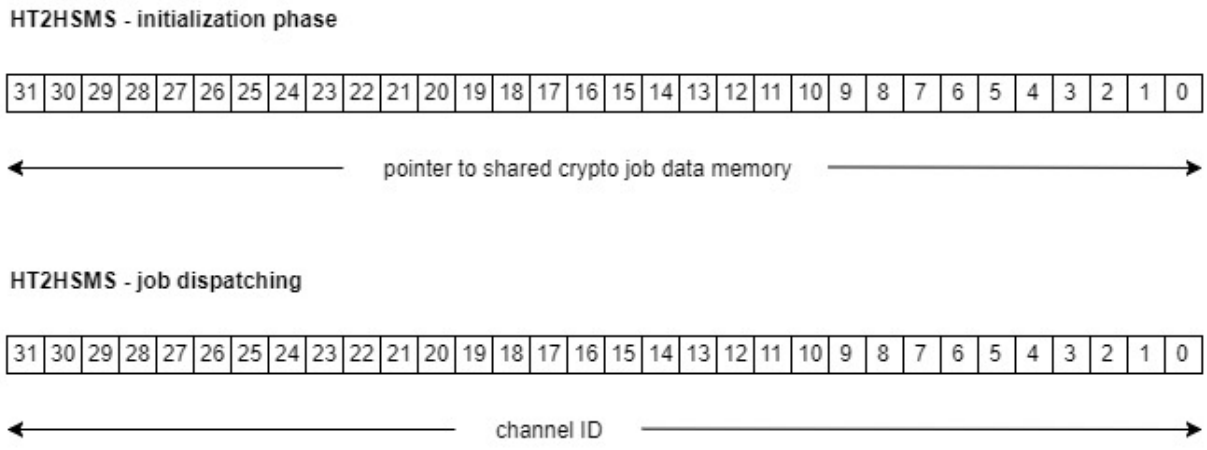


Figure 4.8. HT2HSMS_bitmap

- Host to HSM interrupt enable register `HT2HSMIE`

  The interrupts for the communication with the host are enabled by the EB zentur HSM Firmware during its initialization procedure.

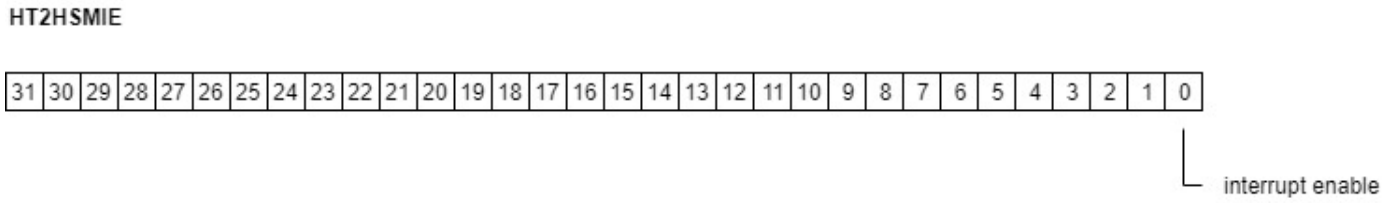


Figure 4.9. HT2HSMIE_bitmap

4.3.2.2.2. HSM to host registers

There are four SFRs for the communication from HSM system to the host.

- HSM to host flag register `HSM2HTF`

  The HSM to host flag register `HSM2HTF` is not used right now.

- HSM to host status register `HSM2HTS`

  The HSM to host status register informs the host about current status settings. The `HSM2HTS` register is used for different purposes:

  - HSM alive status
  - SHE status
  - Error information
  - HSM initialization status

- Channel dependent status

When a job is triggered by the host and the task has been successfully activated in the HSM subsystem, a channel ID dependent busy flag and a common SHE busy flag is set by the EB zentur HSM Firmware in `HSM2HTS`. The EB zentur HSM Firmware resets the channel dependent busy flag in `HT2HSMS` register as soon as the job is finished. The common SHE busy flag is not reset until the last job is finished. On the host side, the status of the channel dependent busy flag is checked either during the channel polling procedure to check the channel result for errors or just before a new command is dispatched to make sure the corresponding channel is not busy anymore.

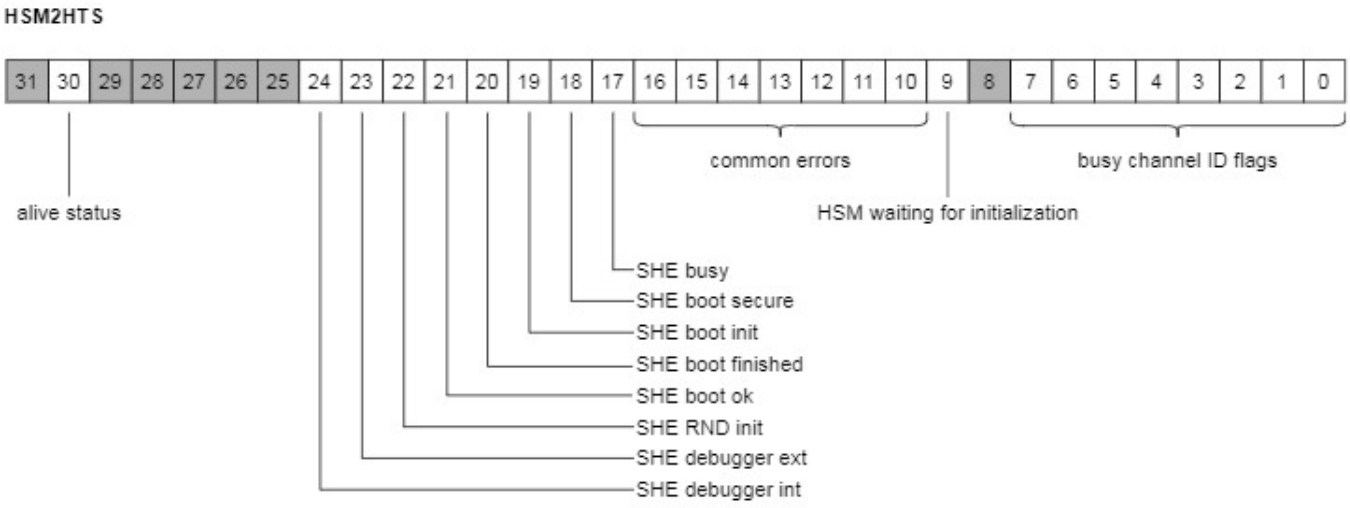


Figure 4.10. HSM2HTS_bitmap

- HSM to host interrupt enable register `HSM2HTIE`

  The interrupt enable register `HSM2HTIE` is not used so far.

- HSM to host interrupt select register `HSM2HTIS`

  The interrupt select register `HSM2HTIS` is not used so far.

4.3.3. Job operation

There are two job processing modes supported by EB zentur HSM Firmware:

- Asynchronous job processing
- Synchronous job processing

All API function calls which trigger a crypto operation on the HSM subsystem are designed to run asynchronously from the host software. It is up to you whether asynchronous or synchronous job processing is exported, in which the asynchronous mode is the preferred solution.

4.3.3.1. Asynchronous job processing

To trigger a crypto operation on the HSM subsystem, the user can do that asynchronously from the host software. In that case, function calls are posted to the EB zentur HSM Firmware and immediately return to the caller, whereas the crypto operation is ongoing on the HSM subsystem. There is no job finished interrupt by HSM subsystem back to the host, and it is up to you to poll the status of the job on a regular basis.

4.3.3.2. Synchronous job processing

Another way to trigger a crypto operation on the HSM subsystem is to run synchronously from the host software. Function calls are posted to the EB zentur HSM Firmware but do not return immediately to the caller. Instead, whereas the crypto operation on HSM subsystem is ongoing and executing, the calling function is waiting for the job to be completed for a certain time, `EB_HSM_TO_CMD_MS`, see Chapter 5, “Module references“ for more details. The status of the job processing is constantly polled, whereas the timeout `EB_HSM_TO_CMD_MS` is running. Either the function returns with a timeout error, which means that the job processing did not finish in time, or it returns that a general error happened on the HSM subsystem that is also reported back to the calling function. Best case, the function returns within a specified max time and no error is reported by HSM.

4.4. Use cases

This section lists basic use cases for the EB zentur HSM Firmware. It shows the interaction between the host, the HSM bridge, and the EB zentur HSM Firmware and illustrates the dynamic behavior of the software.

4.4.1. HSM Setup

The HSM setup is done in advance of the HSM initialization and is needed to register the system services. The system services to be registered are given by the wrapper functions for target system services, see Section 4.2.4, “Integrating EB zentur HSM Firmware”.

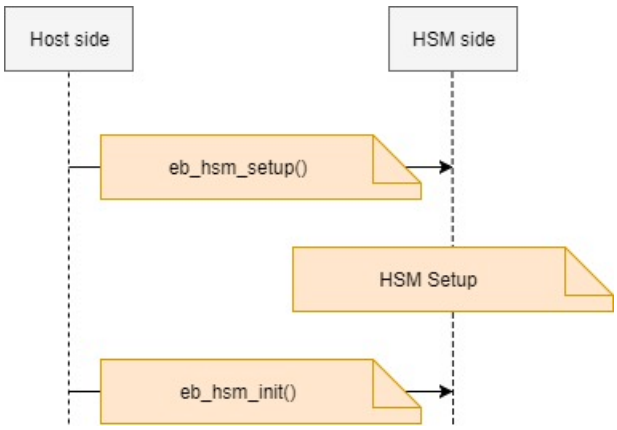


Figure 4.11. HSM Setup

4.4.2. Locking/Unlocking HSM

Locking and unlocking the HSM hardware is a common procedure that is executed for each EB zentur HSM Firmware API call.

After calling an EB zentur HSM Firmware API, a lock for the mutual exclusive access to the HSM is acquired to lock the resource, see Figure 4.12, “HSM Lock”. This is achieved by the host calling `eb_hsm_lock()`. This checks if the hardware resource is busy or free. The callout to the function for calling the MCAL abstraction layer must be registered during the setup of HSM, see Section 4.4.1, “HSM Setup”. EB zentur HSM Firmware then stores the mutual exclusive (mutex) status for the access to the HSM. The mutex status is reused later during unlocking procedure. As long as the required resource is busy, the resource cannot be locked. That means that the system remains in a busy waiting state. The lock procedure has been successful if the required resource is free.

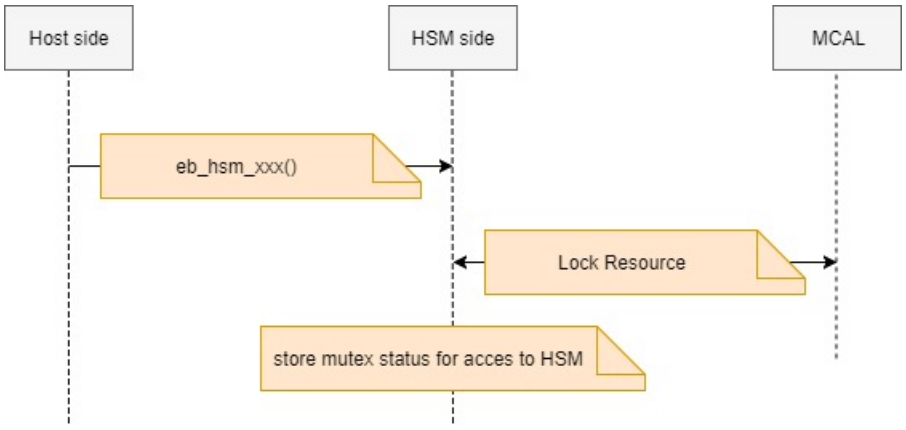


Figure 4.12. HSM Lock

After a job is dispatched, the HSM resource unlocking procedure is performed, see Figure 4.13, “HSM Unlock”. This is achieved by the host calling `eb_hsm_unlock()` which releases the lock for the mutual exclusive access

to the HSM resource. The callout to the function for calling the MCAL abstraction layer must be registered during the setup of HSM, see Section 4.4.1, “HSM Setup”.

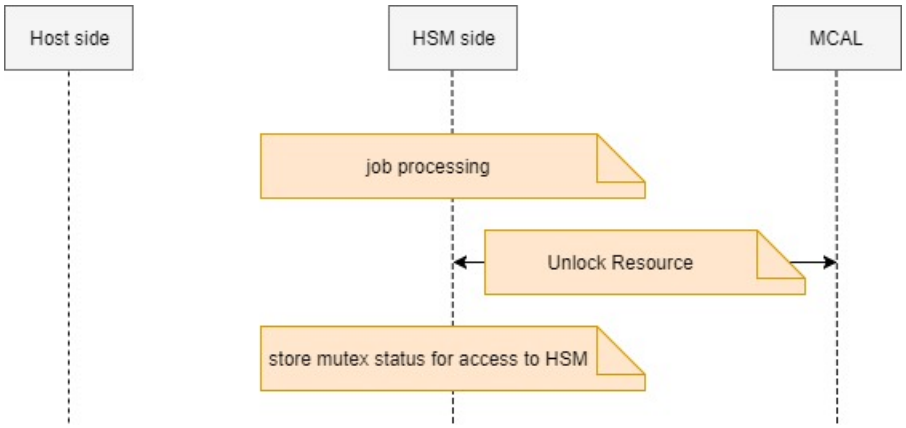


Figure 4.13. HSM Unlock

4.4.3. Initializing EB zentur HSM proxy and Firmware

The EB zentur HSM Firmware and the proxy layer is initialized by the host calling the API `eb_hsm_init()`. The message sequence chart in Figure 4.14, “Initialize HSM Proxy / Firmware” depicts the subsequent relationship.

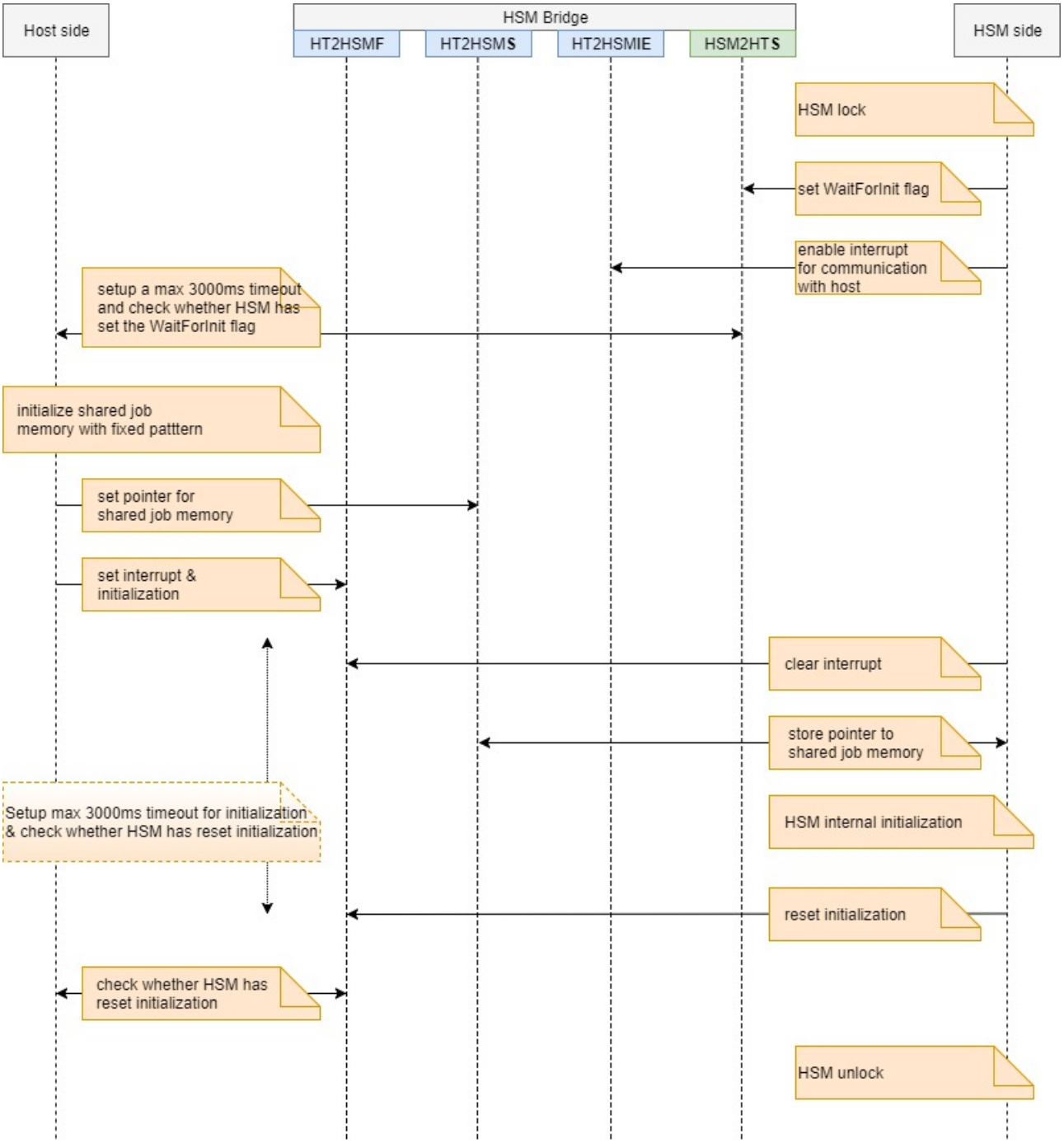


Figure 4.14. Initialize HSM Proxy / Firmware

A lock for the mutual exclusive access to the HSM is acquired to lock the resource. As long as the required resource is busy, the resource cannot locked and the system remains in a busy waiting state. The common procedure of HSM locking and unlocking is shown in Section 4.4.2, “Locking/Unlocking HSM”.

For TC23x, at this point in time, a special handling is needed due to the fact that for TC23x the PFlash is in use. The problem is that a concurrent access to the PFlash from host and from HSM can happen. This problem is valid in following procedures:

- Initialization
- Load key
- Key generate
- Debug activate
- Secure boot
- Secure boot memory block add/update/verify

Therefore interrupts and traps must be disabled:

- Disabling host interrupts:
  The current status of hardware interrupts is stored in order to restore them again at a later point in time. All hardware interrupts are disabled by clearing the Interrupt Enable flag in the interrupt control register (ICR). This is needed so no other process tries to access the host flash command interface.
- Disabling traps:
  The PMU controls the Flash memory and the Boot ROM. All Flash registers are part of the PMU, including the Flash Configuration register (FCON) and the Margin Control register for the PFlash (MARP). To avoid running into Bus Errors and traps some mandatory settings in registers are needed. Reading from a busy PFlash bank causes a Bus Error that must be suppressed or avoided. This is achieved by setting the STALL bit in the FCON. Although uncorrectable error traps are disabled by a system reset, the traps are enabled again by the startup software in Boot ROM. Therefore, the uncorrectable error traps must be disabled again.

The host must wait until the HSM has set a handshake flag in the `HSM2HTS` Bridge register. The flag indicates that HSM is waiting for initialization. A timeout is activated giving the HSM specific time to set the handshake flag. During this time, and as long as the handshake flag is not set by the HSM, the host communicates with the operating system to get the exact time to meet the timeout requirement.

When HSM has finally set the handshake flag in the `HSM2HTS` Bridge register within the required time, the host initializes the job data memory (which is shared between host and HSM) with a fixed pattern. A pointer to the shared job data memory is given to the Proxy Bridge register `HT2HSMS`. The interrupt bit and the initialization bit is set in the Proxy Bridge register `HT2HSMF` which serves as a trigger for the HSM to finalize the initialization procedure. HSM resets the interrupt bit in Proxy Bridge register `HT2HSMF`.

As the initialization bit in Proxy Bridge register `HT2HSMF` has been set by the host, the HSM starts to run its initialization command procedure. After the HSM internal initialization procedure, the initialization Bit in the Proxy Bridge register `HT2HSMF` is reset. The HSM is given `EB_HSM_TO_INIT_MS` (Chapter 5, “Module references“) to finish its tasks to set the initialization Bit at the end. The value of the timeout `EB_HSM_TO_INIT_MS` must be tuned according to the startup times measured in the integrated setup. Consequentially the HSM resource

must be unlocked again. The common procedure of HSM unlocking is shown in Section 4.4.2, “Locking/Unlocking HSM”.

A special handling for TC23x is needed at the end of the HSM initialization procedure: the hardware interrupts and the trap handling are enabled again in reverse order.

4.4.4. Dispatching a job

A Job is dispatched by calling one of the HSM Proxy APIs, see Chapter 5, “Module references“. The message sequence chart in Figure 4.15, “Dispatching a job” depicts the subsequent relationship.

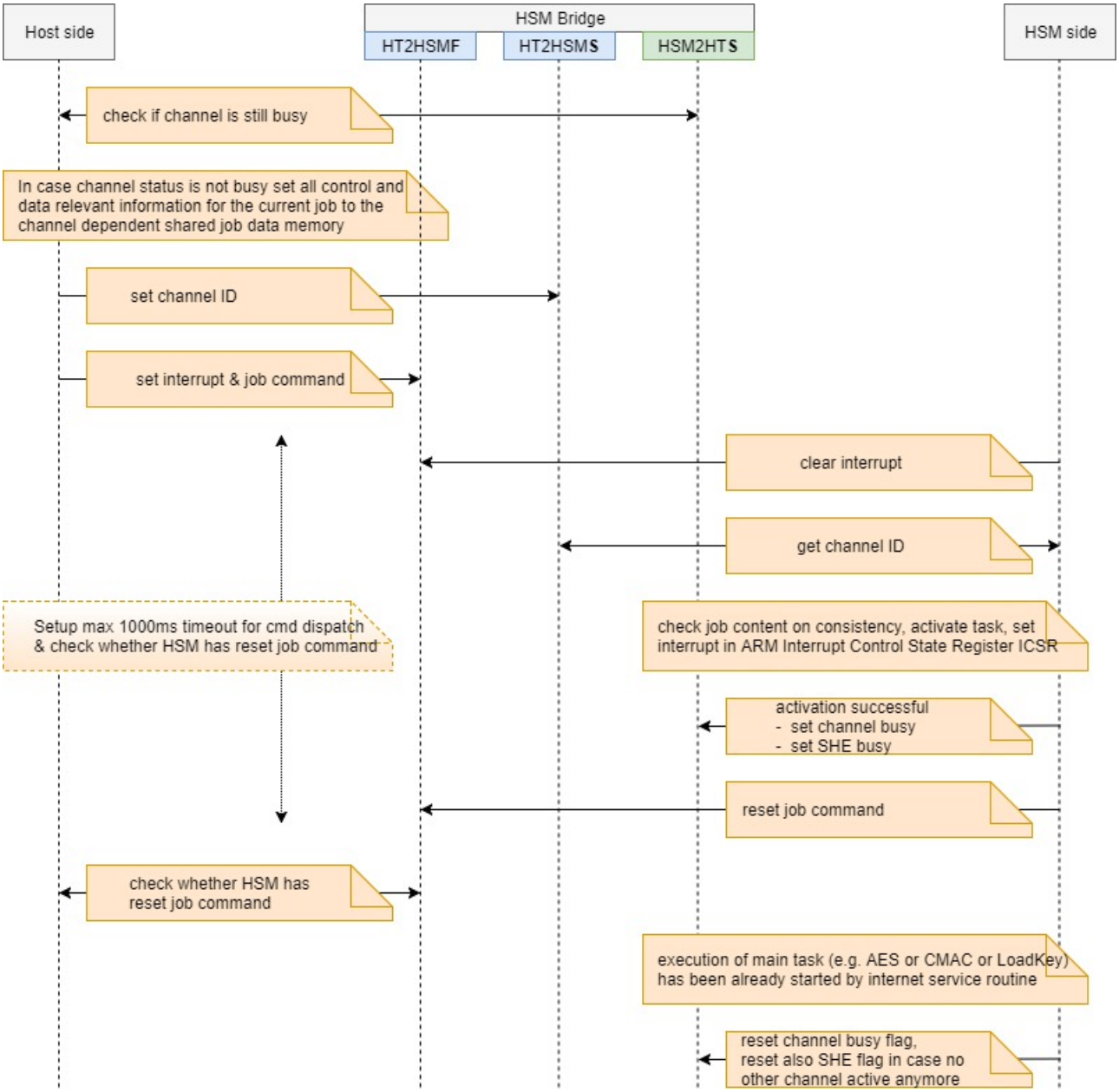


Figure 4.15. Dispatching a job

It is assumed that a certain API is called from host side to trigger a crypto job. At first – and this is common for all primitives – the resource must be locked for the mutual exclusive access to the HSM. As long as the required resource is busy, the resource cannot be locked and the system remains in a busy waiting state. The common procedure of HSM locking is shown in Section 4.4.2, “Locking/Unlocking HSM”.

A service is always called for a certain channel, and it must be ensured the status of the corresponding channel is not busy. To ensure this, the Proxy Bridge register `HSM2HTS` is checked for the channel status flag. If channel status returns a busy state, an error is returned to the user and the resource is unlocked again. In a successful

case (i.e. channel is not in busy state but in ready state) all control and data relevant information for the current job is written to the channel dependent shared job data memory.

Finally, the channel dependent job is dispatched. In this context, the current system time is stored as the start tick for timeout supervision. HSM is informed about a dispatched job by setting the channel ID in bridge status register `HT2HSMS` and the interrupt bit and job command bit in bridge flag register `HT2HSMF`. As a consequence, the HSM clears the interrupt bit and the channel ID is retrieved in order to fetch right input data from the shared job data memory. To indicate to the host that the job is in an execution phase and activation has been successful, the corresponding channel busy flag and the SHE busy flag is set in `HSM2HTS`. During this time, the status is checked from the host side i.e. the dispatch routine remains in a state waiting for HSM to clear the job command bit and the supervision timeout must be kept. When the job on HSM side is done, the corresponding channel bit in bridge register `HT2HSMF` is reset by HSM. That means the channel is not busy anymore. The SHE busy flag is also reset. With this result, the job dispatch procedure is considered to be finished.

After the finished job dispatch procedure, the hardware resource is unlocked again to be free for the next job to be processed by HSM. The common procedure of HSM unlocking is shown in Section 4.4.2, “Locking/Unlocking HSM”.

5. Module references

5.1. Application programming interface (API)

5.1.1. Type definitions

5.1.1.1. eb_hsm_callout_GetCounterValue

Purpose	Callout function signature of get current timer ticks.
Type	`StatusType(*)(TickRefType)`

5.1.1.2. eb_hsm_callout_GetElapsedValue

Purpose	Callout function signature of get elapsed timer ticks.
Type	`StatusType(*)(TickRefType, TickRefType)`

5.1.1.3. eb_hsm_callout_IntDisable

Purpose	Callout function signature for interrupt disabling.
Type	`void(*)(void)`

5.1.1.4. eb_hsm_callout_IntRestore

Purpose	Callout function signature for interrupt restoration.
Type	`void(*)(void)`

5.1.1.5. eb_hsm_callout_Lock

Purpose	Callout function signature for resource locking.
Type	`StatusType(*)(void)`

5.1.1.6. eb_hsm_callout_Ticks2Ms

Purpose	Callout function signature of conversion from timer ticks to real time.
Type	`TickType(*)(TickType)`

5.1.1.7. eb_hsm_callout_Unlock

Purpose	Callout function signature for resource unlocking.
Type	`StatusType(*)(void)`

5.1.1.8. eb_hsm_callouts_t

Purpose	Contains callouts for system services to the EB-HSM zentur Proxy.	
Type	struct	
Members	`eb_hsm_callout_GetCounterValue fp_getCounterValue`	
	`eb_hsm_callout_GetElapsedValue fp_getElapsedValue`	
	`eb_hsm_callout_Ticks2Ms fp_ticks2Ms`	
	`eb_hsm_callout_Lock fp_lock`	
	`eb_hsm_callout_Unlock fp_unlock`	
	`eb_hsm_callout_IntDisable fp_intDisable`	
	`eb_hsm_callout_IntRestore fp_intRestore`	

5.1.2. Macro constants

5.1.2.1. EB_HSM_TO_CMD_MS

Purpose	General timeout for command execution in milliseconds.
Value	(4000U)

5.1.2.2. EB_HSM_TO_DISPATCH_MS

Purpose	Timeout for command dispatching in milliseconds.
Value	(1000U)

5.1.2.3. EB_HSM_TO_INIT_MS

Purpose	Timeout for initialization in milliseconds.
Value	(3000U)

5.1.3. Objects

5.1.3.1. eb_hsm_callouts

Purpose	
Type	eb_hsm_callouts_t

5.1.4. Functions

5.1.4.1. eb_hsm_advance_life_cycle

Purpose	Change the life cycle state to next one.	
Synopsis	`comm_rc_t` **`eb_hsm_advance_life_cycle`** `( uint32 channelId , comm_processing_t procType );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	`procType`	asynchronous or synchronous processing
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_BUSY`	Channel is still busy
	`COMM_INVALID_CHANNEL`	Channel is not within COMM_MAX_-NUM_CHANNELS
	`COMM_IGNORE_UPCOMING_REQUESTS`	Rejecting any request after the successful "stop HSM" command execution

	COMM_NO_NEXT_STATE	No next state available, see supported states in description
	COMM_GENERAL_ERROR	Advancing next security level failed
Description	This API provides functionality to change a security state level. Calling this API causes a state transition from a current state to next life cycle state. Supported life cycles / security levels: ► Init: All services allowed ► Secure Level 1 Loading of root certificates not allowed anymore	

5.1.4.2. eb_hsm_aes

Purpose	Execute services of the Advanced Encryption Standard.	
Synopsis	`comm_rc_t eb_hsm_aes ( uint32 channelId , comm_processing_t procType , uint32 keyId , comm_aes_modes_t aes_mode , comm_aes_mode_t * modePtr );`	
Parameters (in)	channelId	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	procType	Asynchronous or synchronous processing
	keyId	Retrieved HSM key slot ID from driver internal keyID
	aes_mode	AES modes ECB_ENC, ECB_DEC, CBC_ENC, CBC_DEC, GCM_ENC or GCM_DEC
	modePtr.ecb.src	Pointer to input message: plaintext for encryption, ciphertext for decryption
	modePtr.ecb.nr_block	Number of 16 byte blocks to process: 1 for ECB_ENC and ECB_DEC
	modePtr.cbc.iv	Pointer to init vector
	modePtr.cbc.src	Pointer to input message: plaintext for encryption, ciphertext for decryption
	modePtr.cbc.nr_block	Number of 16 byte blocks to process: inputLength / 16 for CBC_ENC and CBC_DEC
	modePtr.gcm.iv	Pointer to init vector
	modePtr.gcm.aad	Pointer to aad, can be null

	modePtr.gcm.aad_len	aad length, can be zero
	modePtr.gcm.src	Pointer to input message: plaintext for encryption, ciphertext for decryption
	modePtr.gcm.src_len	Total length of the input message
	modePtr.gcm.tag	In decryption the tag value generated in an encryption
	modePtr.gcm.tag_len	Tag length - see COMM_AES_TAG_LENGTHS
Parameters (out)	modePtr.ecb.dst	Pointer to output message: ciphertext for encryption, plaintext for decryption There must be nr_block * 16 bytes available for writing
	modePtr.cbc.dst	Pointer to output message: ciphertext for encryption, plaintext for decryption There must be nr_block * 16 bytes available for writing
	modePtr.gcm.dst	Pointer to output message: ciphertext for encryption, plaintext for decryption
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No errors
	COMM_GENERAL_ERROR	HSM not locked for processing
	COMM_BUSY	Channel is still busy
	COMM_INVALID_CHANNEL	Channel is not within COMM_MAX_NUM_CHANNELS
	COMM_IGNORE_UPCOMING_REQUESTS	Rejecting any request after the successful "stop HSM" command execution
Description	This API provides a function that allows cryptographic operations: Symmetric Encryption, Symmetric Decryption	

5.1.4.3. eb_hsm_alive

Purpose	Check availability of HSM.	
Synopsis	comm_rc_t **eb_hsm_alive** (void);	
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No errors

	COMM_GENERAL_ERROR	Alive flag did not toggle
	COMM_TIMEOUT	Alive procedure takes longer than EB_-HSM_TO_CMD_MS
Description	This function checks whether HSM is alive or dead. It reads the alive flag and dispatches the corresponding command. It waits a certain time till HSM toggles the alive bit and returns the result of the operation which must be executed within a certain time.	

5.1.4.4. eb_hsm_boot_failure

Purpose	Amend info about failed host boot and impose sanctions.	
Synopsis	comm_rc_t **eb_hsm_boot_failure** (uint32 channelId , comm_processing_t procType);	
Parameters (in)	channelId	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	procType	Asynchronous or synchronous processing
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No errors
	COMM_GENERAL_ERROR	HSM not locked for processing
	COMM_BUSY	Channel is still busy
	COMM_INVALID_CHANNEL	Channel is not within COMM_MAX_-NUM_CHANNELS
	COMM_NO_SECURE_BOOT	Secure boot did not run before or secure boot process already completed
	COMM_IGNORE_UPCOMING_REQUESTS	Rejecting any request after the successful "stop HSM" command execution
Description	This API provides a function to mark the secure boot process failed at a later stage than the initial run of secure boot after HSM start-up. This API can be invoked only once if initial result from secure boot was OK. It imposes the same sanctions as if the initial run of secure boot after HSM start-up would detect a failure. It finishes the secure boot process.	

5.1.4.5. eb_hsm_boot_ok

Purpose	Amend info about successful host boot and finish the secure boot process.

Synopsis	comm_rc_t **eb_hsm_boot_ok** (uint32 channelId , comm_processing_t procType);	
Parameters (in)	channelId	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	procType	Asynchronous or synchronous processing
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No errors
	COMM_GENERAL_ERROR	HSM not locked for processing
	COMM_BUSY	Channel is still busy
	COMM_INVALID_CHANNEL	Channel is not within COMM_MAX_NUM_CHANNELS
	COMM_NO_SECURE_BOOT	Secure boot did not run before or secure boot process already completed
	COMM_IGNORE_UPCOMING_REQUESTS	Rejecting any request after the successful "stop HSM" command execution
Description	This API provides a function to confirm successful host boot at a later stage than the initial run of secure boot after HSM start-up. This API can be invoked only once if initial result from secure boot was OK. It finishes the secure boot process.	

5.1.4.6. eb_hsm_cancel

Purpose	Cancel HSM command.	
Synopsis	comm_rc_t **eb_hsm_cancel** (uint32 channelId);	
Parameters (in)	channelId	The ID of the channel to interrupt. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No error.
	COMM_[XXX]	Error. See comm_rc_t error codes.
Description	This API provides a function to interrupt a running command. This API is not the SHE user accessible function CMD_CANCEL but only to wait till the job is finished	

5.1.4.7. eb_hsm_debug_activation

Purpose	Activate HSM internal debugging facilities.	
Synopsis	`comm_rc_t eb_hsm_debug_activation ( uint32 channelId , comm_processing_t procType , uint8 * authorization );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	`procType`	Asynchronous or synchronous processing
	`authorization`	authorization
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_GENERAL_ERROR`	HSM not locked for processing
	`COMM_BUSY`	Channel is still busy
	`COMM_INVALID_CHANNEL`	Channel is not within COMM_MAX_-NUM_CHANNELS
	`COMM_IGNORE_UPCOMING_REQUESTS`	Rejecting any request after the successful "stop HSM" command execution
Description	This API	

5.1.4.8. eb_hsm_export_pub_key

Purpose	Execute Public Key Retrieval.	
Synopsis	`comm_rc_t eb_hsm_export_pub_key ( uint32 channelId , comm_processing_t procType , uint16 keyId , uint8 * pubKeyPtr , uint32 * pubKeyLenPtr );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	`procType`	Asynchronous or synchronous processing
	`keyId`	id refers to the slot where the key pair is stored
Parameters (out)	`pubKeyPtr`	pointer to public key
	`pubKeyLenPtr`	pointer to the length of public key
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors

	COMM_GENERAL_ERROR	HSM not locked for processing
	COMM_BUSY	Channel is still busy
	COMM_SMALL_BUFFER	The buffer in proxy is too small to receive coverage data from HSM
	COMM_KEY_INVALID	If key id is not valid from the available asymmetric private keys
	COMM_KEY_EMPTY	If there is no key generated and stored on requested key id
	COMM_INVALID_CHANNEL	Channel is not within COMM_MAX_-NUM_CHANNELS
Description	This API provides a function to get public keys (asymmetric public key).	

5.1.4.9. eb_hsm_export_ram_key

Purpose	Export the RAM key.	
Synopsis	`comm_rc_t` **`eb_hsm_export_ram_key`** `( uint32 channelId , comm_processing_t procType , uint8 * m1 , uint8 * m2 , uint8 * m3 , uint8 * m4 , uint8 * m5 );`	
Parameters (in)	channelId	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	procType	Asynchronous or synchronous processing
Parameters (out)	m1	Target unique identifier
	m2	New Key with another Key in M2
	m3	Calculated MAC over previous data
	m4	Info which Key is updated
	m5	MAC calculated over M4
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No errors
	COMM_GENERAL_ERROR	HSM not locked for processing
	COMM_BUSY	Channel is still busy
	COMM_INVALID_CHANNEL	Channel is not within COMM_MAX_-NUM_CHANNELS
	COMM_IGNORE_UPCOMING_REQUESTS	Rejecting any request after the successful "stop HSM" command execution

Description	This API provides a function to export the RAM key. The function exports the RAM_KEY into a format protected by SECRET_KEY. The key can be imported again by using CMD_LOAD_KEY.

5.1.4.10. eb_hsm_fw_rollback

Purpose	Roll back FW image.	
Synopsis	`comm_rc_t eb_hsm_fw_rollback ( comm_processing_t procType , const uint8 * FwuDataPtr , uint32 FwuDataLength );`	
Parameters (in)	`procType`	Asynchronous or synchronous processing
	`FwuDataPtr`	Pointer to FW update data ASN.1 DER encoded.
	`FwuDataLength`	Length of the FW update data [bytes].
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_INVALID_CHANNEL`	Channel is not the highest priority channel
	`COMM_[XXX]`	Other errors from sub-functions, see comm_rc_t error codes.
Description	This API provides a function to rollback the recent FW image. For security reasons an authentication is needed. Therefore the signed FWU data profile is used.	

5.1.4.11. eb_hsm_fw_update_data

Purpose	Send FW image data (block).	
Synopsis	`comm_rc_t eb_hsm_fw_update_data ( comm_processing_t procType , const uint8 * FwImageDataPtr , uint32 FwImageLength );`	
Parameters (in)	`procType`	Asynchronous or synchronous processing
	`FwImageDataPtr`	Pointer to FW image data start, requires 16-byte alignment.
	`FwImageLength`	Length of the FW image data (block).
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_INVALID_CHANNEL`	Channel is not the highest priority channel

	COMM_FWU_WRITE_IMAGE	Error on flashing FW image
	COMM_FWU_FW_IMAGE_LENGTH	Invalid FW length
	COMM_FWU_DECRYPT	FW image decryption failed
	COMM_[XXX]	Other errors from sub-functions, see comm_rc_t error codes
Description	This API provides a function to send FW image data to the HSM. In the case it is not doable to sent the whole FW image in a single call this API can be called multiple times (for block transfer) as required to transfer the complete FW image to HSM.	

5.1.4.12. eb_hsm_fw_update_finish

Purpose	Finish FW update process.	
Synopsis	comm_rc_t **eb_hsm_fw_update_finish** (comm_processing_t procType);	
Parameters (in)	procType	Asynchronous or synchronous processing
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No errors
	COMM_INVALID_CHANNEL	Channel is not the highest priority channel
	COMM_FWU_SIGNATURE_ALGO	Signature algorithm not supported
	COMM_FWU_SIGNATURE	Signature mismatch
	COMM_FWU_FW_VERSION	Invalid FW version
	COMM_FWU_FW_PARTITION	No valid FW partition found in PFLASH
	COMM_FWU_FD_WRITE_BOOT_MAC	PFLASH data: Error on writing boot MAC
	COMM_FWU_FD_WRITE_FW_LENGTH	PFLASH data: Error on writing FW length
	COMM_FWU_FD_WRITE_FW_VERSION	PFLASH data: Error on writing FW version
	COMM_FWU_FD_WRITE_FW_STATUS	PFLASH data: Error on writing FW status
	COMM_[XXX]	Other errors from sub-functions, see comm_rc_t error codes
Description	This API provides a function the complete a FW update process.	

5.1.4.13. eb_hsm_fw_update_start

Purpose	Start FW update process.

Synopsis	`comm_rc_t` **`eb_hsm_fw_update_start`** `( comm_processing_t procType , const uint8 * FwuDataPtr , uint32 FwuDataLength );`	
Parameters (in)	`procType`	Asynchronous or synchronous processing
	`FwuDataPtr`	Pointer to FW update data ASN.1 DER encoded.
	`FwuDataLength`	Length of the FW update data [bytes].
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_BUSY`	Other channels are still busy
	`COMM_INVALID_CHANNEL`	Channel is not the highest priority channel
	`COMM_GENERAL_ERROR`	ASN.1 DER data profile not valid, or data profile signature wrong
	`COMM_FWU_CIPHER_KEY`	Invalid ciphering key
	`COMM_FWU_CIPHER_ALGO`	Ciphering algorithm not supported
	`COMM_FWU_ERASE_PARTITION`	Error on erasing FW update partition
	`COMM_FWU_FW_PARTITION`	No valid FW partition found in PFLASH
	`COMM_FWU_FW_IMAGE_LENGTH`	Invalid FW length
	`COMM_FWU_FW_VERSION`	Invalid FW version
	`COMM_FWU_SD_FW_IMAGE_LENGTH`	DER data: Error on decoding Fw image length object
	`COMM_FWU_SD_SIGN_VALUE`	DER data: Error on decoding signature object
	`COMM_FWU_SD_SIGN_DATA`	DER data: Error on decoding data signature object
	`COMM_FWU_SD_SIGN_ALGO`	DER data: Error on decoding signature algorithm object
	`COMM_FWU_SD_SIGN_KEY_ID`	DER data: Error on decoding signature key ID object
	`COMM_FWU_SD_FW_CIPHER_INFO`	DER data: Error on decoding cipher object
	`COMM_FWU_SD_FW_VERSION_INFO`	DER data: Error on decoding FW version object
	`COMM_FWU_SD_SIGN_IMAGE`	DER data: Error on decoding image signature object
	`COMM_[XXX]`	Other errors from sub-functions, see comm_rc_t error codes

Description	This API provides a function to start a FW update process.

5.1.4.14. eb_hsm_gen_rnd

Purpose	Generate random number.	
Synopsis	`comm_rc_t` **`eb_hsm_gen_rnd`** `( uint32 channelId , comm_processing_t procType , uint8 * rnd );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	`procType`	Asynchronous or synchronous processing
Parameters (out)	`rnd`	output pointer for the generated random number. Must have space for at least 16 bytes available.
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_GENERAL_ERROR`	HSM not locked for processing
	`COMM_BUSY`	Channel is still busy
	`COMM_INVALID_CHANNEL`	Channel is not within COMM_MAX_-NUM_CHANNELS
	`COMM_IGNORE_UPCOMING_REQUESTS`	Rejecting any request after the successful "stop HSM" command execution
Description	This API provides a function that generates a vector of 128 random bits.	

5.1.4.15. eb_hsm_get_bl_version

Purpose	Retrieve HSM bootloader version and status information in partitions 0 and 1.	
Synopsis	`comm_rc_t` **`eb_hsm_get_bl_version`** `( uint32 channelId , comm_processing_t procType , comm_BL_VersionInfoType * blVersionInfo );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0.. (COMM_MAX_NUM_CHANNELS - 1).
	`procType`	Asynchronous or synchronous processing
Parameters (out)	`blVersionInfo`	Pointer to buffer where version and status information of bootloader is stored

Return Value	Result of operation of type comm_rc_t	
	COMM_NO_ERROR	Version is correctly read
	COMM_[XXX]	Version is not read correctly

5.1.4.16. eb_hsm_get_challenge

Purpose	Provide a challenge for a challenge response protocol.	
Synopsis	comm_rc_t **eb_hsm_get_challenge** (uint32 channelId , comm_processing_t procType , uint8 * challenge , comm_challenge_mode_t challengeMode);	
Parameters (in)	channelId	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	procType	Asynchronous or synchronous processing
	challengeMode	challenge use case (e.g. Debug challenge or Stop-HSM challenge)
Parameters (out)	challenge	challenge
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No errors
	COMM_GENERAL_ERROR	HSM not locked for processing
	COMM_BUSY	Channel is still busy
	COMM_INVALID_CHANNEL	Channel is not within COMM_MAX_NUM_CHANNELS
	COMM_IGNORE_UPCOMING_REQUESTS	Rejecting any request after the successful "stop HSM" command execution
Description	This API provides a function that provides a challenge for the selected challenge mode	

5.1.4.17. eb_hsm_get_fw_version

Purpose	Retrieve HSM Firmware version and status information in partitions 0 and 1.
Synopsis	comm_rc_t **eb_hsm_get_fw_version** (uint32 channelId , comm_processing_t procType , comm_FW_VersionInfoType * activeFirmwareInfo , comm_FW_VersionInfoType * passiveFirmwareInfo);

Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0.. (COMM_MAX_NUM_CHANNELS - 1).
	`procType`	Asynchronous or synchronous processing
Parameters (out)	`activeFirmwareInfo`	Pointer to buffer where version and status information of active partition is stored
	`passiveFirmwareInfo`	Pointer to buffer where version and status information of passive partition is stored. NULL pointer for passiveFirmwareInfo is accepted, if information is not needed.
Return Value	Result of operation of type comm_rc_t	
	`COMM_NO_ERROR`	Version is correctly read
	`COMM_[XXX]`	Version is not read correctly

5.1.4.18. eb_hsm_get_id

Purpose	Return UID of the device.	
Synopsis	`comm_rc_t eb_hsm_get_id ( uint32 channelId , comm_processing_t procType , const uint8 * challenge , uint8 * id , uint8 * sreg , uint8 * mac );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	`procType`	Asynchronous or synchronous processing
	`challenge`	Challenge
Parameters (out)	`id`	Identifier
	`sreg`	status register
	`mac`	CMAC[MASTER_KEY] (challenge | id | sreg)
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_GENERAL_ERROR`	HSM not locked for processing
	`COMM_BUSY`	Channel is still busy
	`COMM_INVALID_CHANNEL`	Channel is not within COMM_MAX_-NUM_CHANNELS
	`COMM_IGNORE_UPCOMING_REQUESTS`	Rejecting any request after the successful "stop HSM" command execution

Description	This API provides a function to return the unique identifier (UID) of the device. It returns also the value of the status register protected by a MAC over a challenge and the data.

5.1.4.19. eb_hsm_get_life_cycle

Purpose	Return current life cycle state.	
Synopsis	`comm_rc_t` **`eb_hsm_get_life_cycle`** `( uint32 channelId , comm_processing_t procType , comm_life_cycle_state_t * life_cycle_state_ptr );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	`procType`	asynchronous or synchronous processing.
Parameters (out)	`life_cycle_state_ptr`	Current life cycle state.
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_BUSY`	Channel is still busy
	`COMM_INVALID_CHANNEL`	Channel is not within COMM_MAX_NUM_CHANNELS
	`COMM_IGNORE_UPCOMING_REQUESTS`	Rejecting any request after the successful "stop HSM" command execution
	`COMM_GENERAL_ERROR`	Failure of reading the life cycle value
Description	This API provides functionality to return a current life cycle / security level. Life cycle state must be handled as uint8 value. Supported life cycles / security states: ► Init: All services allowed ► Secure Level 1 Loading of root certificate not allowed anymore	

5.1.4.20. eb_hsm_get_load_key_id_and_range

Purpose	Calculate key ID and key range.
Synopsis	`comm_rc_t` **`eb_hsm_get_load_key_id_and_range`** `( comm_keyid_t key_id , uint8 * she_key_id , comm_key_range_t * KeyRange );`

Parameters (in)	`key_id`	The EB HSM key ID to compute the Key ID or Auth ID of.
Parameters (out)	`she_key_id`	The key ID to use for the M1.
	`KeyRange`	The computed key range.
Return Value	Result of the operation	
	`COMM_NO_ERROR`	No errors
	`COMM_KEY_INVALID`	The value of key_id is outside of range.
Description	This function maps the EB HSM key ID to the key ID required with the M1 message of the memory update protocol (cf. to SHE Specification V1.1 ch. 9). Furthermore it calculates the value for the KeyRange parameter required by eb_hsm_load_key().	

5.1.4.21. eb_hsm_get_result_channel

Purpose	Get channel corresponding result.	
Synopsis	`comm_rc_t` **`eb_hsm_get_result_channel`** `( uint32 channelId );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
Return Value	Result of operation including error code	
	`COMM_INVALID_CHANNEL`	If channelId is out of range
	`COMM_NO_ERROR`	No errors
	`COMM_xxx`	Error code
Description	This function gets the channel result from HSM Proxy bridge HSM2HTS.	

5.1.4.22. eb_hsm_get_she_status

Purpose	Get the status of SHE.	
Synopsis	`comm_rc_t` **`eb_hsm_get_she_status`** `( uint8 * sheStatus );`	
Parameters (in)	–	-
Parameters (out)	`sheStatus`	SHE status
Return Value	Result of HSM2HTS error codes	
	`COMM_CHANNEL_xxx`	error codes stored in HSM2HTS
Description	This API provides a function that retrieves the content of the status register as specified by the SHE Functional Specification. Initially the bridge register HSM2HTS is	

	checked whether any HSM error is already observed. Then the SHE status relevant info from bridge register HSM2HTS is feedback as SHE status.

5.1.4.23. eb_hsm_hash

Purpose	Generate HASH.	
Synopsis	`comm_rc_t eb_hsm_hash ( uint32 channelId , comm_processing_t procType , comm_hash_modes_t mode , const uint8 * iptr , uint8 * optr , uint32 ilen );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	`procType`	Asynchronous or synchronous processing
	`mode`	HASH SHA mode
	`iptr`	pointer to the message
	`optr`	pointer to hash
	`ilen`	message length in bytes
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_GENERAL_ERROR`	HSM not locked for processing
	`COMM_BUSY`	Channel is still busy
	`COMM_INVALID_CHANNEL`	Channel is not within COMM_MAX_NUM_CHANNELS
	`COMM_IGNORE_UPCOMING_REQUESTS`	Rejecting any request after the successful "stop HSM" command execution
Description	This API provides a function to which takes an input or message and returns a fixed-size alphanumeric string. NOTE: This functionality is NOT released in RFM version 1.4	

5.1.4.24. eb_hsm_init

Purpose	Initialize HSM.
Synopsis	`comm_rc_t eb_hsm_init ( void );`

Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_GENERAL_ERROR`	In case of general error
	`COMM_TIMEOUT`	If the initialization procedure takes longer than EB_HSM_TO_INIT_MS
Description	This function initializes the HSM and the HSM proxy layer. It returns the result of the operation. The API is called in advance to any cryptographic service and must be executed within a certain time.	

5.1.4.25. eb_hsm_integ_CalloutsGetRef

Purpose	Getter for accessing the EB-HSM callout routines.
Synopsis	`eb_hsm_callouts_t * eb_hsm_integ_CalloutsGetRef ( void );`
Return Value	Pointer to the callouts buffer.
Description	To be provided with EB-HSM proxy API eb_hsm_setup().

5.1.4.26. eb_hsm_integ_CryptoJobDataGetRef

Purpose	Getter for accessing interface buffer.
Synopsis	`CryptoJobTypeCacheAligned * eb_hsm_integ_CryptoJobDataGetRef ( void );`
Return Value	Pointer to job entry buffer.
Description	To be provided with EB-HSM proxy API eb_hsm_setup().

5.1.4.27. eb_hsm_integ_GetCounterValue

Purpose	Wrapper for the system service to read the current timer ticks.	
Synopsis	`StatusType eb_hsm_integ_GetCounterValue ( TickRefType pCntVal );`	
Parameters (out)	`pCntVal`	The current tick value of the counter
Return Value	Result of the operation.	

	`0`	No errors
	`!0`	Operation failed
Description	To be provided as callout service routine with EB-HSM proxy API eb_hsm_setup().	

5.1.4.28. eb_hsm_integ_GetElapsedValue

Purpose	Wrapper for the system service to get the number of ticks between the current tick value and a previously read tick value.	
Synopsis	`StatusType` **`eb_hsm_integ_GetElapsedValue`** `( TickRefType pCntVal , TickRefType pElpsdVal );`	
Parameters (in,out)	`pCntVal`	in: the previously read tick value of the counter out: the current tick value of the counter
Parameters (out)	`pElpsdVal`	The difference to the previous read value
Return Value	Result of the operation	
	`0`	No errors
	`!0`	Operation failed
Description	To be provided as callout service routine with EB-HSM proxy API eb_hsm_setup().	

5.1.4.29. eb_hsm_integ_IntDisable

Purpose	Wrapper for the system service that disables HW interrupts.
Synopsis	`void` **`eb_hsm_integ_IntDisable`** `( void );`
Description	Required for programming the ENDINIT protection registers. To be provided as callout service routine with EB-HSM proxy API eb_hsm_setup().

5.1.4.30. eb_hsm_integ_IntRestore

Purpose	Wrapper for the system service that restores the interrupt status.
Synopsis	`void` **`eb_hsm_integ_IntRestore`** `( void );`

Description	Required for programming the ENDINIT protection registers. To be provided as callout service routine with EB-HSM proxy API eb_hsm_setup().

5.1.4.31. eb_hsm_integ_Lock

Purpose	Wrapper for the system service that provides resource locking to synchronize mutual exclusive access to HSM.	
Synopsis	`StatusType` **`eb_hsm_integ_Lock`** `( void );`	
Return Value	Result of the operation	
	`0`	No errors
	`!0`	Operation failed
Description	To be provided as callout service routine with EB-HSM proxy API eb_hsm_setup().	

5.1.4.32. eb_hsm_integ_TICKS2MS

Purpose	Wrapper for the system service to convert the elapsed timer ticks to real time in milliseconds.	
Synopsis	`TickType` **`eb_hsm_integ_TICKS2MS`** `( TickType elpsdVal );`	
Parameters (in)	`elpsdVal`	The elapsed ticks
Return Value	The elapsed time in milliseconds	
Description	To be provided as callout service routine with EB-HSM proxy API eb_hsm_setup().	

5.1.4.33. eb_hsm_integ_Unlock

Purpose	Wrapper for the system service that provides resource unlocking to synchronize mutual exclusive access to HSM.	
Synopsis	`StatusType` **`eb_hsm_integ_Unlock`** `( void );`	
Return Value	Result of the operation	
	`0`	No errors
	`!0`	Operation failed

Description	To be provided as callout service routine with EB-HSM proxy API eb_hsm_setup().

5.1.4.34. eb_hsm_key_gen

Purpose	Execute Key Generation.	
Synopsis	`comm_rc_t eb_hsm_key_gen ( uint32 channelId , comm_processing_t procType , comm_key_gen_type_t cryptoKeytype , uint16 keyId , uint8 * pubKeyPtr , uint32 * pubKeyLenPtr );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	`procType`	Asynchronous or synchronous processing
	`cryptoKeytype`	cryptographic key algorithm (e.g., COMM_KEY_GEN_ED25519, COMM_KEY_GEN_ECCNIST_P256,...)
	`keyId`	id refers to the slot where the expected private key or symmetric key will be stored
Parameters (out)	`pubKeyPtr`	pointer to public key (only relevant for asym. key generation)
	`pubKeyLenPtr`	pointer to the length of public key (only relevant for asym. key generation)
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_GENERAL_ERROR`	HSM not locked for processing
	`COMM_BUSY`	Channel is still busy
	`COMM_SMALL_BUFFER`	The buffer in proxy is too small to receive coverage data from HSM
	`COMM_ALGO_NOT_SUPPORTED`	Requested key type is not supported for generation
	`COMM_INVALID_CHANNEL`	Channel is not within COMM_MAX_NUM_CHANNELS
Description	This API provides a function to generate keys (symmetric key or asymmetric key pair). In case of symmetric key, the key is stored in a secure memory. In case of asymmetric public key, the key is stored in a buffer (pointer pubKeyPtr to such buffer must be passed). Symmetric key generation is supported in all platforms but asymmetric key generation support depend on the chipset.	

5.1.4.35. eb_hsm_load_asym_priv_key

Purpose	Asymmetric private key update.	
Synopsis	comm_rc_t **eb_hsm_load_asym_priv_key** (uint32 channelId , comm_processing_t procType , comm_asym_keyid_t keyId , uint8 * KeyPtr , uint16 KeyLength);	
Parameters (in)	channelId	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	procType	Asynchronous or synchronous processing
	keyId	comm_asym_keyid_t
	KeyPtr	Pointer to the key data
	KeyLength	Length of the key in bytes
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No errors
	COMM_GENERAL_ERROR	HSM not locked for processing
	COMM_BUSY	Channel is still busy
	COMM_INVALID_CHANNEL	Channel is not within COMM_MAX_NUM_CHANNELS
Description	This API provides a function to trigger an asymmetric private key update.	

5.1.4.36. eb_hsm_load_asym_pub_key

Purpose	Update asymmetric public key.	
Synopsis	comm_rc_t **eb_hsm_load_asym_pub_key** (uint32 channelId , comm_processing_t procType , comm_asym_keyid_t keyId , uint8 * KeyPtr , uint16 KeyLength);	
Parameters (in)	channelId	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	procType	Asynchronous or synchronous processing
	keyId	comm_asym_keyid_t
	KeyPtr	Pointer to the key data
	KeyLength	Length of the key in bytes

Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_GENERAL_ERROR`	HSM not locked for processing
	`COMM_BUSY`	Channel is still busy
	`COMM_INVALID_CHANNEL`	Channel is not within COMM_MAX_-NUM_CHANNELS
	`COMM_MEMORY_FAILURE`	Flash writing fails.
	`COMM_ROOT_CERT_LOCKED`	Root certificate locked due to security level.
Description	This API provides a function to trigger the update of an asymmetric public key or certificates. Note that a root certificate can be loaded only if the "life cycle" state is init. See eb_hsm_impl_get_life_cycle and eb_hsm_impl_advance_life_cycle.	

5.1.4.37. eb_hsm_load_key

Purpose	Load and update a cipher key.	
Synopsis	`comm_rc_t eb_hsm_load_key ( uint32 channelId , comm_processing_t procType , comm_key_range_t KeyRange , uint8 * m1 , uint8 * m2 , uint8 * m3 , uint8 * m4 , uint8 * m5 );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	`procType`	Asynchronous or synchronous processing
	`KeyRange`	SHE key range or SHE+ (extended) key range
	`m1`	Target unique identifier (UID, key ID, auth key ID)
	`m2`	Encrypted key data
	`m3`	MAC of m1 and m2 calculated with auth key
Parameters (out)	`m4`	Info which Key is updated
	`m5`	MAC calculated over M4
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_GENERAL_ERROR`	HSM not locked for processing

	COMM_BUSY	Channel is still busy
	COMM_INVALID_CHANNEL	Channel is not within COMM_MAX_-NUM_CHANNELS
	COMM_KEY_WRITE_PROTECTED	The new key cannot be written due to protected key slot
	COMM_IGNORE_UPCOMING_REQUESTS	Rejecting any request after the successful "stop HSM" command execution
Description	This API provides a function that allows to update a AES key with a SHE key container. M1, M2 and M3 are the messages relevant to key loading to be able to load keys onto the SHE enabled ECU. M1, M2 and M3 contain the target UID (Unique IDentifier), the new Key encrypted with another Key in M2 and a MAC calculated over this data in M3. The receiving ECU replies with the messages M4 and M5 which provide digital evidence that the new Key has been loaded properly. M4 contains information which key is updated. M5 is a MAC calculated over M4.	

5.1.4.38. eb_hsm_load_plain_key

Purpose	Load and upate a plain key.	
Synopsis	`comm_rc_t` **`eb_hsm_load_plain_key`** `( uint32 channelId , comm_processing_t procType , const uint8 * KeyPtr , uint16 KeyLength );`	
Parameters (in)	channelId	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	procType	Asynchronous or synchronous processing
	KeyPtr	pointer to raw plaintext key data
	KeyLength	key length in bytes. Must be 16 as only 128-bit AES keys are supported.
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No errors
	COMM_GENERAL_ERROR	HSM not locked for processing
	COMM_BUSY	Channel is still busy
	COMM_INVALID_CHANNEL	Channel is not within COMM_MAX_-NUM_CHANNELS
	COMM_IGNORE_UPCOMING_REQUESTS	Rejecting any request after the successful "stop HSM" command execution
Description	This API provides a function that updates the SHE AES RAM KEY in plaintext format.	

5.1.4.39. eb_hsm_mac

Purpose	Execute MAC generation or MAC verification.	
Synopsis	`comm_rc_t eb_hsm_mac (uint32 channelId , comm_processing_t procType , uint8 algoFamily , uint8 algoMode , comm_keyid_t keyId , comm_service_mode_t mode , const uint8 * msgPtr , uint8 * macPtr , uint32 msgLength , uint32 * macLengthPtr , comm_verify_status_t * verStatus , comm_key_range_t keyRange);`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	`procType`	Asynchronous or synchronous processing
	`algoFamily`	algorithm family (AES, SHA256, SIPHASH)
	`algoMode`	algorithm mode (CMAC, HMAC, SIPHASH24, SIPHASH48)
	`keyId`	retrieved HSM key slot ID from driver internal keyID
	`mode`	MAC generation or MAC verification
	`msgPtr`	pointer to source message
	`msgLength`	message length in bytes
	`macLengthPtr`	MAC length pointer in bits. (For CMAC verification with BOOT_MAC_KEY the length must be 128 bits.) (For SipHash only allowed value is 64 bits) (For HMAC this ignored)
	`keyRange`	COMM_KEY_RANGE_VKMS if VKMS key is used. COMM_KEY_RANGE_UNUSED for other keys.
Parameters (out)	`macPtr`	pointer to store the calculated MAC to. Must have length of at least 16 bytes.
	`verStatusPtr`	verification status pointer (pass/fail)
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_GENERAL_ERROR`	HSM not locked for processing
	`COMM_BUSY`	Channel is still busy

	COMM_INVALID_CHANNEL	Channel is not within COMM_MAX_-NUM_CHANNELS
	COMM_INVALID_JOB_DATA	Invalid macLength for BOOT_MAC_KEY
	COMM_IGNORE_UPCOMING_REQUESTS	Rejecting any request after the successful "stop HSM" command execution
Description	This API provides a function to generate or to verify a message authentication code.	

5.1.4.40. eb_hsm_mem_block_add

Purpose	Adds the cmac value of the new memory block.	
Synopsis	`comm_rc_t eb_hsm_mem_block_add ( uint32 channelId , comm_processing_t procType , const uint8 * block_address_ptr , uint32 block_size , comm_mem_block_id_t block_id , comm_keyid_t key_id );`	
Parameters (in)	channelId	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	procType	Asynchronous or synchronous processing
	block_address_ptr	Start address of the new memory block.
	block_size	A block size in bytes.
	block_id	A block ID where a cmac value is stored.
	key_id	A symmetric ID used for cmac calculation.
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No errors
	COMM_GENERAL_ERROR	HSM not locked for processing
	COMM_KEY_EMPTY	If there is no key generated and stored on requested key id
	COMM_BUSY	Channel is still busy
	COMM_INVALID_CHANNEL	Channel is not within COMM_MAX_-NUM_CHANNELS
	COMM_INVALID_JOB_DATA	Wrong block ID or the block is already used
	COMM_PARAM_MISALIGNED	Start address alignment is not correct
	COMM_IGNORE_UPCOMING_REQUESTS	Rejecting any request after the successful "stop HSM" command execution

Description	This API provides a function to add a new memory block over which a CMAC value is calculated and stored to HSM. A block add can be used once only and it is meant to be used within a production. After addition an existing block can be updated with eb_hsm_mem_block_update.

5.1.4.41. eb_hsm_mem_block_update

Purpose	Calculate the CMAC value of the new memory block.	
Synopsis	`comm_rc_t eb_hsm_mem_block_update ( uint32 channelId , comm_processing_t procType , const uint8 * block_address_ptr , uint32 block_size , comm_mem_block_id_t block_id , comm_keyid_t key_id );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	`procType`	Asynchronous or synchronous processing
	`block_address_ptr`	Start address of the new memory block.
	`block_size`	A block size in bytes.
	`block_id`	A block ID where a cmac value is stored.
	`key_id`	A symmetric ID used for CMAC calculation.
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_GENERAL_ERROR`	HSM not locked for processing
	`COMM_KEY_EMPTY`	If there is no key generated and stored on requested key id
	`COMM_KEY_INVALID`	The key id is not valid
	`COMM_BUSY`	Channel is still busy
	`COMM_INVALID_CHANNEL`	Channel is not within COMM_MAX_NUM_CHANNELS
	`COMM_INVALID_JOB_DATA`	Wrong block ID
	`COMM_INVALID_AUTHORIZATION`	Authorization verification failed
	`COMM_INVALID_AUTHENTICATION`	Authentication failed
	`COMM_PARAM_MISALIGNED`	Start address alignment is not correct
	`COMM_IGNORE_UPCOMING_REQUESTS`	Rejecting any request after the successful "stop HSM" command execution

Description	This API provides secure way to update an existing memory block of a certain block-ID. The memory block update init must be called before this function with same parameters.

5.1.4.42. eb_hsm_mem_block_update_init

Purpose	Init the calculation of the CMAC value of the new memory block.	
Synopsis	`comm_rc_t eb_hsm_mem_block_update_init ( uint32 channelId , const uint8 * block_address_ptr , uint32 block_size , comm_mem_block_id_t block_id , comm_keyid_t key_id );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	`block_address_ptr`	Start address of the new memory block.
	`block_size`	A block size in bytes.
	`block_id`	A block ID where a cmac value is stored.
	`key_id`	A symmetric ID used for CMAC calculation.
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_GENERAL_ERROR`	HSM not locked for processing
	`COMM_KEY_EMPTY`	If there is no key generated and stored on requested key id
	`COMM_KEY_INVALID`	The key id is not valid
	`COMM_BUSY`	Channel is still busy
	`COMM_INVALID_CHANNEL`	Channel is not within COMM_MAX_NUM_CHANNELS
	`COMM_INVALID_JOB_DATA`	Wrong block ID
	`COMM_INVALID_AUTHENTICATION`	Authentication failed
	`COMM_INVALID_AUTHORIZATION`	Authorization failed
	`COMM_PARAM_MISALIGNED`	Start address alignment is not correct
	`COMM_IGNORE_UPCOMING_REQUESTS`	Rejecting any request after the successful "stop HSM" command execution
Description	This API provides initialization of the updating of a memory block. It has to be called before the actual memory block update due to authentication reasons.	

5.1.4.43. eb_hsm_mem_block_verify

Purpose	Verifies the block.	
Synopsis	`comm_rc_t eb_hsm_mem_block_verify ( uint32 channelId , comm_processing_t procType , comm_mem_block_id_t block_id , comm_verify_status_t * ver_status_ptr );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	`procType`	Asynchronous or synchronous processing
	`block_id`	A block ID where a cmac value is stored.
Parameters (out)	`ver_status_ptr`	Verification status (pass/fail)
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_GENERAL_ERROR`	HSM not locked for processing
	`COMM_MEMORY_FAILURE`	No memory block available
	`COMM_BUSY`	Channel is still busy
	`COMM_INVALID_CHANNEL`	Channel is not within COMM_MAX_NUM_CHANNELS
	`COMM_INVALID_JOB_DATA`	Wrong block ID
	`COMM_IGNORE_UPCOMING_REQUESTS`	Rejecting any request after the successful "stop HSM" command execution
Description	This API provides a function to verify a given block of which CMAC value is stored to HSM.	

5.1.4.44. eb_hsm_poll_channel

Purpose	Check channel status.	
Synopsis	`comm_chan_stat_t eb_hsm_poll_channel ( uint32 channelId );`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
Return Value	Result of channel status	
	`COMM_CHANNEL_READY`	channel is ready and be used

	COMM_CHANNEL_BUSY	channel is still busy or channelId is out of valid range
Description	This API provides a function to check the status of the corresponding channel in HSM Proxy bridge HSM2HTS.	

5.1.4.45. eb_hsm_resumeHsmExecution_fromFlash

Purpose	Resume HSM execution from Flash.	
Synopsis	`comm_rc_t eb_hsm_resumeHsmExecution_fromFlash ( void );`	
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No errors
	COMM_GENERAL_ERROR	if resume API is called without calling stop API before
	COMM_BUSY	If any process is ongoing in HSM, it returns COMM_BUSY
Description	This API provides a function to resume HSM execution from Flash.	

5.1.4.46. eb_hsm_rnd_extend_seed

Purpose	Request to extend the RNG seed.	
Synopsis	`comm_rc_t eb_hsm_rnd_extend_seed ( uint32 channelId , comm_processing_t procType , const uint8 * entropy );`	
Parameters (in)	channelId	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	procType	Asynchronous or synchronous processing
	entropy	128bit entropy
Return Value	Result of operation including error code	
	COMM_RNG_SEED	SHE error code ERC_RNG_SEED
	COMM_NO_ERROR	No errors
	COMM_GENERAL_ERROR	HSM not locked for processing
	COMM_BUSY	Channel is still busy
	COMM_INVALID_CHANNEL	Channel is not within COMM_MAX_NUM_CHANNELS

	COMM_IGNORE_UPCOMING_REQUESTS	Rejecting any request after the successful "stop HSM" command execution
Description	This API extends the seed of the PRNG by compressing the former seed value and the supplied entropy into a new seed which will be used to generate the following random numbers.	

5.1.4.47. eb_hsm_rnd_init

Purpose	Initialize the seed of RNG.	
Synopsis	comm_rc_t **eb_hsm_rnd_init** (uint32 channelId , comm_processing_t procType);	
Parameters (in)	channelId	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	procType	Asynchronous or synchronous processing
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No errors
	COMM_GENERAL_ERROR	HSM not locked for processing
	COMM_BUSY	Channel is still busy
	COMM_INVALID_CHANNEL	Channel is not within COMM_MAX_NUM_CHANNELS
	COMM_IGNORE_UPCOMING_REQUESTS	Rejecting any request after the successful "stop HSM" command execution
Description	This function initializes the Random Number Generator and derives a key for the PRNG.	

5.1.4.48. eb_hsm_secure_boot

Purpose	Initiation of the secure boot process.	
Synopsis	comm_rc_t **eb_hsm_secure_boot** (uint32 channelId , comm_processing_t procType , const uint8 * start_address , uint32 size);	
Parameters (in)	channelId	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	procType	Asynchronous or synchronous processing

	start_address	pointer to the start address of the data to be verified
	size	length of the data to be verified, in bytes
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No errors
	COMM_GENERAL_ERROR	HSM not locked for processing
	COMM_BUSY	Channel is still busy
	COMM_INVALID_CHANNEL	Channel is not within COMM_MAX_-NUM_CHANNELS
	COMM_NO_SECURE_BOOT	Secure boot already run after power cylce/ reset or BOOT_MAC_KEY not loaded
	COMM_KEY_NOT_AVAILABLE	BOOT_MAC_KEY or BOOT_MAC are debug protected
	COMM_[XXX]	Other errors from sub-functions, see comm_rc_t error codes.
	COMM_IGNORE_UPCOMING_REQUESTS	Rejecting any request after the successful "stop HSM" command execution
Description	This API provides a function to start the secure boot process. The function can only be called for the first initialization of the secure boot or for an update procedure to renew the start address and the data size to be verified. Note that before calling it a BOOT_MAC_KEY has to be loaded.	

5.1.4.49. eb_hsm_setup

Purpose	Register system services to EB-HSM Proxy.	
Synopsis	`comm_rc_t eb_hsm_setup ( const eb_hsm_callouts_t * p_-callouts , CryptoJobTypeCacheAligned * p_cryptoJobData );`	
Parameters (in)	p_callouts	Pointer to the callouts that provide necessary system services to the EB-HSM zentur SW.
	p_cryptoJobData	Pointer to static crypto job data shared memory between host and HSM.
Return Value	Result of operation including error code	
	COMM_NO_ERROR	No errors
	COMM_GENERAL_ERROR	At least one function pointer is NULL

5.1.4.50. eb_hsm_sign

Purpose	Signature generation and verification.	
Synopsis	`comm_rc_t eb_hsm_sign (uint32 channelId , comm_processing_t procType , comm_asym_keyid_t keyId , uint8 algoFamily , uint8 algoMode , comm_service_mode_t mode , uint8 operationMode , const uint8 * msgptr , const uint8 * signptr , uint32 msgLength , uint32 * signLengthPtr , comm_verify_status_t * ver_status);`	
Parameters (in)	`channelId`	Corresponding channel ID. Valid range: 0 .. COMM_MAX_NUM_CHANNELS - 1.
	`procType`	Asynchronous or synchronous processing
	`keyId`	retrieved HSM key slot ID from driver internal keyID
	`algoFamily`	used algorithm family for signature generation/verification
	`algoMode`	used algorithm mode for signature generation/verification
	`mode`	Signature generation, verification
	`operationMode`	Operation mode
`msgptr`	pointer to message	
`msgLength`	source length in bytes	
Parameters (in,out)	`signptr`	pointer to signature
	`signLengthPtr`	destination length in bytes
Parameters (out)	`ver_status`	verification status (pass/fail)
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_GENERAL_ERROR`	HSM not locked for processing
	`COMM_BUSY`	Channel is still busy
	`COMM_INVALID_CHANNEL`	Channel is not within COMM_MAX_NUM_CHANNELS
Description	This API provides a function to generate or to verify a signature.	

5.1.4.51. eb_hsm_stopHsmExecution_fromFlash

Purpose	Stop HSM execution from Flash.

Synopsis	`comm_rc_t` **`eb_hsm_stopHsmExecution_fromFlash`** `( uint8 * authorization );`	
Parameters (in)	`authorization`	authorization response calculated from challenge
Return Value	Result of operation including error code	
	`COMM_NO_ERROR`	No errors
	`COMM_GENERAL_ERROR`	HSM not locked for processing
	`COMM_BUSY`	If any process is ongoing in HSM, it returns COMM_BUSY
	`COMM_IGNORE_UPCOMING_REQUESTS`	Rejecting any request after already successful "stop HSM" command execution
Description	This API provides a function to stop all HSM execution from Flash.	

6. Appendix

This section lists open issues and restrictions to be considered when using EB zentur HSM Firmware.

6.1. Restrictions

The following features are not part of the delivery:

- Job handling via parallel channels (operations) with and without different priorities
- Usage of the same cryptographic primitive on multiple channels in parallel
- ECDSA signature generation/verification using NIST curves P-224 and P-192
- HSM2HTF does not support interrupt handling from HSM towards host
- The SHE user accessible function CMD_CANCEL

For TC23x, special handling is needed due to the fact that the PFlash is in use. A concurrent access to the PFlash from host system and from HSM system can happen. This is a problem for following procedures:

- initialization
- key loading
- debug activation

Therefore, in order to avoid concurrent access to the PFlash, the host interrupts are disabled for TC23x. Beyond that, reading from a busy PFlash bank can cause a bus error. In order to avoid a bus error and running into traps, the STALL bit in the flash configuration register FCON is set. Although uncorrectable error traps are disabled by a system reset, the traps are enabled again by the start-up software in Boot ROM. Therefore the uncorrectable error traps must be disabled again.

A special handling is needed for the FWU feature. Due to the fact that the EB zentur HSM Firmware accesses the PFlash during the FWU procedures. Thus a concurrent access to the PFlash from the host and from the HSM system can happen. This applies to the following procedures:

- initialization of the EB zentur HSM Firmware. See Section 5.1.4.24, “eb_hsm_init”
- start of the FWU procedure. See Section 5.1.4.13, “eb_hsm_fw_update_start”
- programming of the EB zentur HSM Firmware image. See Section 5.1.4.11, “eb_hsm_fw_update_data”
- finalization of the FWU procedure. See Section 5.1.4.12, “eb_hsm_fw_update_finish”
- rollback of the recent version of the EB zentur HSM Firmware. See Section 5.1.4.10, “eb_hsm_fw_rollback”

A concurrent access to the PFlash can cause a bus error. In order to avoid a bus error the STALL bit in the flash configuration register FCON is set during the procedures mentioned above. The host interrupts are disabled during the set/reset of the STALL bit itself.

6.2. XORShift128 algorithm

The XORShift128 algorithm is used

- to activate the stop HSM command
- to authorize read and write access for Secure Event Logging feature

This section describes the complete flow of the XORShift128 algorithm, including:

- Section 6.2.1, “Parameters involved in the XORShift128 algorithm”
- Section 6.2.2, “Overview of steps involved in the XORShift128 algorithm”
- Section 6.2.3, “Test vectors for the XORShift128 algorithm”

6.2.1. Parameters involved in the XORShift128 algorithm

- Challenge:
  - Size: 16 bytes
  - Stored in little-endian format, i.e. the least significant byte (LSB) is stored at the lowest address. For details, see Section 3.8.1, “Prerequisites”.
- UID:
  - Size: 16 bytes
  - Stored in little-endian format, i.e. the LSB is stored at the lowest address. For details, see Section 3.8.1, “Prerequisites”.
- ModifiedUID:
  - Size: 16 bytes
  - The UID to be used in the calculation shall discard the value of the status register, which is returned in the 16th byte (MSB) of the UID. Hence, as part of the XORShift128 algorithm, both HSM and host generate the ModifiedUID by changing the value of the MSB, i.e. the 16th byte, of the UID to 0xFF. See Figure 6.1, “Preparing initial word states for the XORShift128 algorithm” for better understanding.
- XORedUID (Challenge ^ ModifiedUID):
  - Size: 16 bytes
  - The XOR operation is performed byte by byte between Challenge and ModifiedUID. See Figure 6.1, “Preparing initial word states for the XORShift128 algorithm” to understand how the XOR operation is carried out.
- Word states (`a,b,c,d`):
  - Size of each state: 4 bytes

- The generated value XORedUID is divided among 4 word states (`a,b,c,d`). See Figure 6.1, “Preparing initial word states for the XORShift128 algorithm” to understand how these word states (`a,b,c,d`) are formed.

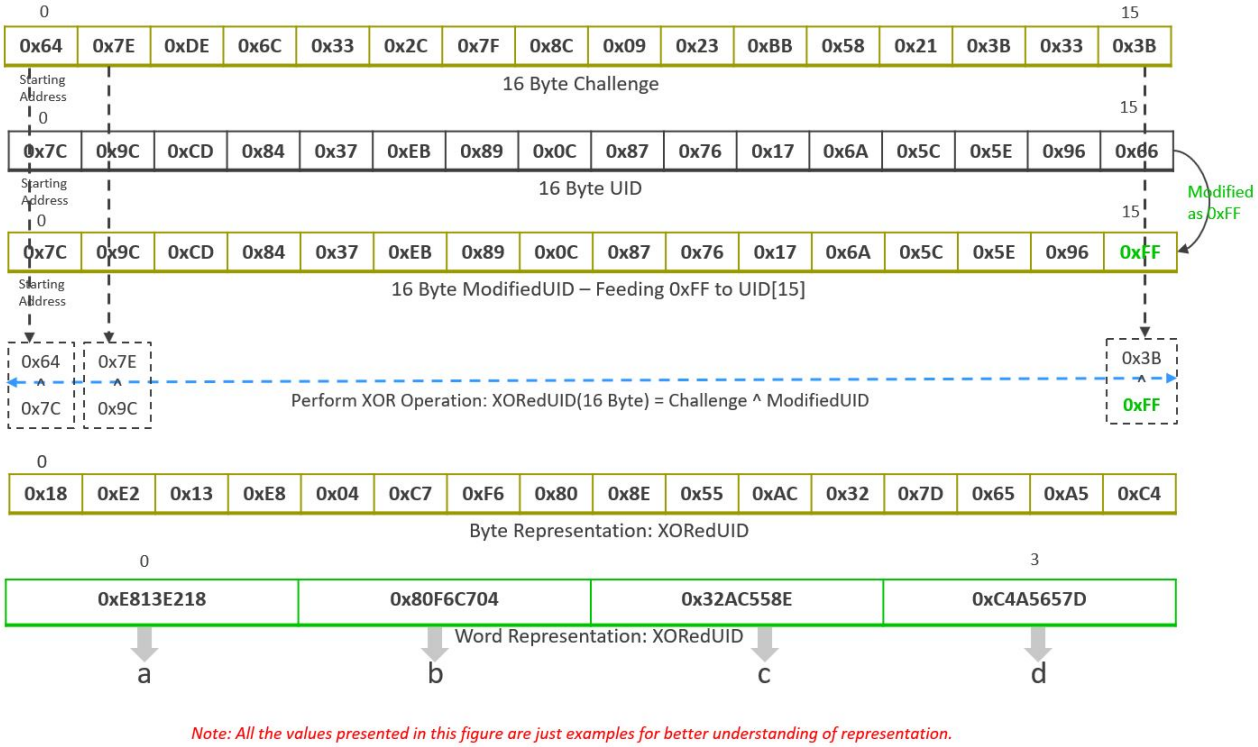


Figure 6.1. Preparing initial word states for the XORShift128 algorithm

6.2.2. Overview of steps involved in the XORShift128 algorithm

For the calculation of the response, the XORShift128 algorithm takes the following steps:

- Get the normal challenge value (16 bytes) from the HSM (see Section 3.8.1, “Prerequisites”)
- Read the UID (16 bytes) from the HSM (see Section 3.8.1, “Prerequisites”)
- Generate the ModifiedUID (16 bytes) by ignoring the sreg byte and setting the MSB of the received UID as 0xFF
- Perform the XOR operation byte by byte between the challenge and the ModifiedUID
- Apply the XORShift128 algorithm four times. The initial state of the algorithm must be set as the 16 bytes output of the XOR operation (see the previous step). The flowchart of four iterations of the XORShift128 algorithm is shown in Figure 6.2, “Four iterations of the XORShift128 algorithm”. Each iteration of the XORShift128 algorithm is implemented according to the example implementation described on https://en.wikipedia.org/wiki/Xorshift.

- The 4 bytes output (word state `a`) of the last iteration in the previous step is the response. The output is stored in little-endian format.

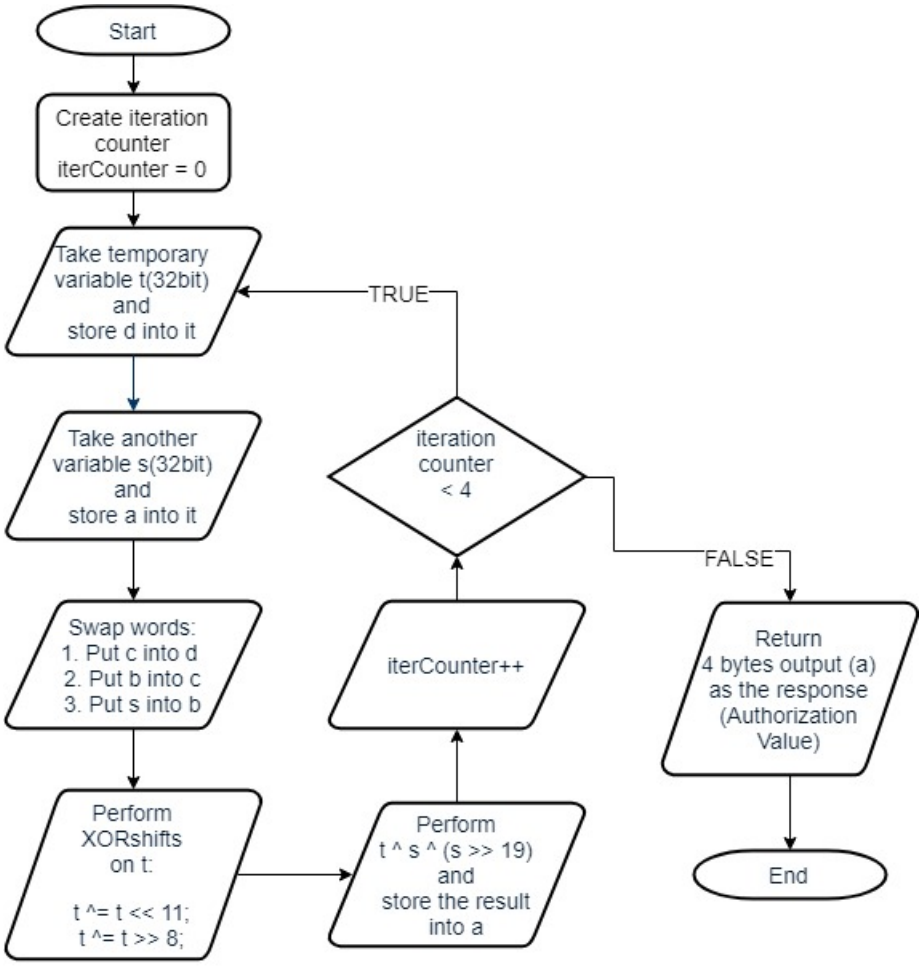


Figure 6.2. Four iterations of the XORShift128 algorithm

The response calculated using this algorithm is stored in little-endian format. Two test vectors are provided to verify the implementation at the host end (see Section 6.2.3, “Test vectors for the XORShift128 algorithm”).

6.2.3. Test vectors for the XORShift128 algorithm

Two test vectors are provided here. The test vectors are only to verify the implementation of the XORShift128 algorithm which the host must also implement:

- Challenge and UID as input variables
- ModifiedUID (see Section 6.2.1, “Parameters involved in the XORShift128 algorithm”) shall be generated as part of the XORShift128 algorithm implementation.

- The response is the word state `a` (as the output of the last iteration of the XORShift128 algorithm). It shall be in little-endian format. Here, the response is shown in the word representation.

List of test vectors:

- Test Vector-1:
  - Challenge : 0x64, 0x7E, 0xDE, 0x6C, 0x33, 0x2C, 0x7F, 0x8C, 0x09, 0x23, 0xBB, 0x58, 0x21, 0x3B, 0x33, 0x3B
  - UID : 0x7C, 0x9C, 0xCD, 0x84, 0x37, 0xEB, 0x89, 0x0C, 0x87, 0x76, 0x17, 0x6A, 0x5C, 0x5E, 0x96, 0x66
  - Response : 0x16AED7A9
- Test Vector-2:
  - Challenge : 0xA4, 0x71, 0x21, 0xAC, 0x77, 0x29, 0x72, 0x80, 0x19, 0x27, 0x2B, 0x38, 0x20, 0x7B, 0x77, 0x7B
  - UID : 0x82, 0x12, 0x20, 0x87, 0x38, 0xCB, 0x81, 0x02, 0x88, 0x87, 0x18, 0x7A, 0x52, 0x5C, 0x17, 0x7A
  - Response : 0x4CF3B3FE

Bibliography

[SHE_Spec] *SHE specification v1.1*, SHE specification 2009-04-01_SHE_Functional_Spec_v1_1_rev439.pdf

[AURIX_TC2xx_TS] , *1st Generation AURIX TC2xx Hardware Security Module HSM_TS_V1.0_Rev1.4.pdf*

[AURIX_TC3xx_TS] , *2nd Generation AURIX TC3xx Hardware Security Module HSM_TS_V2.0_Rev2.0.pdf*

[FWU_data_profile] *FWU data profile, Version 1.0, 2019-08-22 20190822_FWU_data_profile.pdf*